DefectoFix
An interactive defect fix logging tool

M. Muzaffar Hameed, M. Zeeshan ul Haq
This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 40 weeks of full time studies.

Contact Information:
Author(s):
Muhammad Muzaffar Hameed
Address: Folkparksvagen 18:18, 372 40, Ronneby, Sweden
E-mail: muzaffar@muzaffar.net

Muhammad Zeeshan ul Haq
Address: Folkparksvagen 18:18, 372 40, Ronneby, Sweden
E-mail: m.zeeshanulhaq@gmail.com

University advisor(s):
Dr. Robert Feldt
Department of Software Engineering

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25
**ABSTRACT**

Despite the large efforts made during the development phase to produce fault free system, most of the software implementations still require the testing of entire system. The main problem in the software testing is the automation that could verify the system without manual intervention. Recent work in software testing is related to the automated fault injection by using fault models from repository. This requires a lot of efforts, which adds to the complexity of the system. To solve this issue, this thesis suggests DefectoFix framework.

DefectoFix is an interactive defect fix logging tools that contains five components namely Version Control Syssem (VCS), source code files, differencing algorithm, Defect Fix Model (DFM) creation and additional information (project name, class name, file name, revision number, diff model). The proposed differencing algorithm extracts detailed information by detecting differences in source code files. This algorithm performs comparison at sub-tree levels of source code files. The extracted differences with additional information are stored as DFM in repository. DFM(s) can later be used for the automated fault injection process.

The validation of DefectoFix framework is performed by a tool developed using Ruby programming language. Our case study confirms that the proposed framework generates a correct DFM and is useful in automated fault injection and software validation activities.

**Keywords:** Software validation, static source code analysis, defect fix logging, automated fault injection.
ACKNOWLEDGEMENTS

In the name of Allah who is the most gracious and merciful. We are thankful to our creator (Allah) who blessed us with talent to do this thesis.

We are very thankful to our supervisor Dr. Robert Feldt for his guidance and patience at every step of this thesis, specially his e-Meetings and very quick response to our emails that help us a lot. Without his support and invaluable feedback, we could not be able to complete this thesis. We acknowledge his contributions to enhance our knowledge on the subject. We are also very thankful to our friends Vickey Kamlesh Wadhwani and Aleem Ahmad for sparing their time to review our thesis.

We cannot forget to thanks our grandparents, parents, brothers, sisters and other relatives who always pray for our success. Their love always remains the key source of our motivation. We dedicate our thesis to our respective families.
Table of Contents

ABSTRACT ............................................................................................................................. I

ACKNOWLEDGEMENTS ......................................................................................................... II

TABLE OF FIGURES .............................................................................................................. VI

TABLE OF TABLES ................................................................................................................ VII

1 INTRODUCTION .................................................................................................................. 1
  1.1 BACKGROUND .................................................................................................................. 1
  1.2 AIMS AND OBJECTIVES ................................................................................................. 3
  1.3 RESEARCH QUESTIONS ................................................................................................. 3
  1.4 EXPECTED OUTCOME ................................................................................................. 3
  1.5 RESEARCH METHODOLOGY ....................................................................................... 4
  1.6 THESIS OUTLINE .......................................................................................................... 4

2 PRELIMINARIES .................................................................................................................. 7
  2.1 WHAT ARE DEFECTS? .................................................................................................... 7
  2.2 DEFECT CLASSIFICATION .......................................................................................... 7
    2.2.1 Orthogonal Defect Classification (ODC) ................................................................. 9
    2.2.2 IEEE Defect Classification ................................................................................... 10
  2.3 REVISION CONTROL .................................................................................................... 11
    2.3.1 Concurrent Versions System .................................................................................. 12
    2.3.2 Subversion ........................................................................................................... 12
  2.4 OPEN SOURCE SOFTWARE .......................................................................................... 13
  2.5 DEFECT TRACKING TOOLS ......................................................................................... 14
  2.6 FAULT INJECTION ......................................................................................................... 15
    2.6.1 Fault Injection Techniques ....................................................................................... 16
  2.7 RUBY ............................................................................................................................. 16
  2.8 RUBY PARSER ............................................................................................................. 17
  2.9 ABSTRACT SYNTAX TREE ......................................................................................... 17
  2.10 S-EXPRESSION ............................................................................................................ 18
  2.11 MYSQL ........................................................................................................................ 19

3 A SURVEY OF DEFECT TRACKING TOOLS ..................................................................... 20
  3.1 BUGZILLA ....................................................................................................................... 20
    3.1.1 Limitations ............................................................................................................... 20
  3.2 BUGCRAWLER ............................................................................................................... 21
    3.2.1 Limitations ............................................................................................................... 21
  3.3 CODESTRIKER ............................................................................................................. 22
    3.3.1 Limitations ............................................................................................................... 22
  3.4 ITRACKER ....................................................................................................................... 22
    3.4.1 Limitations ............................................................................................................... 22
  3.5 JITTERBUG .................................................................................................................... 22
    3.5.1 Limitations ............................................................................................................... 23
  3.6 MANTIS .......................................................................................................................... 23
    3.6.1 Limitation ................................................................................................................ 23
  3.7 REQUEST TRACKER (RT) ............................................................................................. 23
    3.7.1 Limitations ............................................................................................................... 23
  3.8 ROUNDP ........................................................................................................................ 24
    3.8.1 Limitation ................................................................................................................ 24
  3.9 REDMINE ....................................................................................................................... 24
    3.9.1 Limitations ............................................................................................................... 24
  3.10 GNATS .......................................................................................................................... 24
4 DEFECTOFIX FRAMEWORK DESIGN ................................................................. 27

4.1 FRAMEWORK GOALS ..................................................................................... 27
4.2 OVERALL FRAMEWORK DESIGN ................................................................. 27
4.3 DEFECTOFIX FRAMEWORK PARTS ............................................................... 28
  4.3.1 Version Control System ........................................................................... 28
  4.3.2 Faulty and Correct Source Code Files ...................................................... 28
  4.3.3 Source Code Differencing Algorithm ...................................................... 28
  4.3.4 Defect Model Creation ........................................................................... 29
  4.3.5 Additional Information .......................................................................... 29
  4.3.6 Record Information in Defectofix Repository ......................................... 29
4.4 DEFECT FIX MODEL USAGE SCENARIO IN FAULT INJECTION ..................... 29
4.5 SOURCE CODE DIFFERENCING ALGORITHMS .......................................... 30
  4.5.1 Diff Algorithm ....................................................................................... 30
  4.5.2 Tree Diff Algorithm ................................................................................ 33
  4.5.3 JDiff Algorithm ..................................................................................... 34
5 DEFECTOFIX DIFFERENCING ALGORITHM .................................................. 36

5.1 ALGORITHM ...................................................................................................... 36
  5.1.1 Compare Repository Algorithm .............................................................. 37
  5.1.2 Register DFM Algorithm ....................................................................... 38
5.2 DEFECTOFIX EXPLANATION ....................................................................... 38
  5.2.1 Step 1: (diff of both files) ....................................................................... 38
  5.2.2 Step 2 .................................................................................................... 38
  5.2.3 Step 3 (Comparison in Defectofix Repository) ...................................... 41
5.3 DEFECTOFIX ALGORITHM EXPLANATION BY EXAMPLES ....................... 41
  5.3.1 Example No. 1 ....................................................................................... 41
  5.3.2 Example No. 2 ....................................................................................... 43
  5.3.3 Example No. 3 ....................................................................................... 46
6 DEFECTOFIX SYSTEM DESIGN .................................................................... 49

6.1 OVERVIEW OF DEFECTOFIX WORKING .................................................. 49
6.2 FRAGMENTATION OF SOURCE CODE FILES ........................................... 50
6.3 ABSTRACT SYNTAX TREE (AST) GENERATION ....................................... 50
6.4 EXTRACTION OF TREE LEVELS FROM AST ............................................ 51
6.5 CONVERTING TREE LEVELS IN ARRAYS ................................................ 51
6.6 NODE MATCHING OF LEVELS AND NODES ............................................. 51
6.7 DFM REGISTRATION ..................................................................................... 52
6.8 VIEW/EDIT/DELETE DEFECTS FROM DEFECTOFIX REPOSITORY .......... 52
7 EVALUATION OF DEFECTOFIX ..................................................................... 53

7.1 TOOLS ............................................................................................................. 53
  7.2 CASE STUDY SUBJECTS ............................................................................ 54
  7.2.1 Redmine .................................................................................................. 55
  7.2.2 puituniverse .......................................................................................... 55
  7.2.3 h2o-ruby ................................................................................................ 55
7.3 PROCEDURE & RESULTS FOR CASE STUDY SUBJECTS ............................. 55
  7.3.1 Simplified Redmine Bug (SRB) ............................................................... 56
  7.3.2 Redmine ................................................................................................ 60
  7.3.3 puituniverse .......................................................................................... 62
  Figure 7.10 Correct AST levels of collideable.rb (puituniverse) ....................... 65
  7.3.4 h2o-ruby ................................................................................................ 65
7.4 RESULT ANALYSIS .......................................................................................... 67
8 DISCUSSION ........................................................................................................... 69
## Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview of Research Methodology</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>AST of Code</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>Graphical representation of S-Expression</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>DefectoFix Framework</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>AST Tree</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>AST (S-Expression) Tree of Faulty Source Code</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>AST (S-Expression) Tree of Correct Source Code</td>
<td>45</td>
</tr>
<tr>
<td>5.4</td>
<td>AST (S-Expression) Tree of Class “C” of faulty and correct source code</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>AST (S-Expression) Tree of Class “D” of faulty and correct source code</td>
<td>48</td>
</tr>
<tr>
<td>6.1</td>
<td>DefectoFix System Design</td>
<td>49</td>
</tr>
<tr>
<td>6.2</td>
<td>Fragmentation of source codes</td>
<td>50</td>
</tr>
<tr>
<td>6.3</td>
<td>Generation of Abstract Syntax Tree</td>
<td>51</td>
</tr>
<tr>
<td>6.4</td>
<td>Generation of Tree levels</td>
<td>51</td>
</tr>
<tr>
<td>7.1</td>
<td>Faulty AST levels of acts_as_event.rb (SRB)</td>
<td>58</td>
</tr>
<tr>
<td>7.2</td>
<td>Correct AST levels of acts_as_event.rb (Redmine)</td>
<td>58</td>
</tr>
<tr>
<td>7.3</td>
<td>Diff Model outputs, when DefectoFix applied on acts_as_event.rb (SRB)</td>
<td>59</td>
</tr>
</tbody>
</table>
TABLE OF TABLES

Table 2.1 Defect Types with Description ................................................................. 7
Table 2.2 Severity Levels of Defects with Description (El Emam et al., 1998) ........... 8
Table 2.3 Severity Levels of Defects with Description ........................................... 8
Table 2.4 Code and its S-Expression ....................................................................... 18
Table 3.1 Goals of Defect Tracking Tools ................................................................. 25
Table 5.1 Faulty and correct source code file .......................................................... 41
Table 5.2 Abstract Syntax Tree of the code ............................................................... 42
Table 5.3 Faulty and correct source code file .......................................................... 43
Table 5.4 Abstract Syntax Tree of the code ............................................................... 44
Table 5.5 Faulty and Correct source code file .......................................................... 46
Table 5.6 Abstract Syntax Tree of the code ............................................................... 47
Table 7.1 Faulty and Correct Source Code for acts_as_event.rb (SRB) ................. 56
Table 7.2 A defect fix model with sample ............................................................... 59
Table 7.3 Faulty and Correct Source Code for acts_as_event.rb (Redmine) ........... 60
Table 7.4 Faulty and Correct Source Code for collideable.rb (puituniverse) ......... 63
1 **INTRODUCTION**

This chapter provides the brief introduction to the thesis. In Section 1.1, motivation for the thesis is described with reference to relevant studies and background knowledge. The aims and objectives are discussed in Section 1.2 of this chapter. The research questions and expected outcomes of this thesis are mentioned in Section 1.3 and 1.4 respectively. Section 1.5 describes the research methodology in brief that will further be explained in Chapter 3. Section 1.6 provides outline of each chapter of the thesis.

1.1 **Background**

Making errors and then learning from them is part of human nature and it affects also software developers. They often know what the error is, how to resolve it and how to learn from these errors in software projects. They use several tools to do these tasks like defect tracking tools etc. When software developers work on a software project, they have to manage defect which they find in the project. They can use spreadsheets in the start of the software project (when there are few errors) but as the project makes progress, the number of defect also increases in quantity. With several developers and/or sources of defect reports, a spreadsheet will not do this task efficiently. At this stage, spreadsheets cannot fulfill this task and there is a need of some tools like defect tracking tools, which are useful in such situations. Many software projects reach this point, especially during testing and deployment phases when users tend to find an application’s defect. A defect tracking tool is a database of defect reports whose front end facilitates actions such as filing new defect reports, changing the state to reflect the progress of the work done to address the defect, and generating reports on the defect data (Tejas Software Consulting, 2008). The basic reason behind using these defect tracking systems is to accomplish the better quality in software products. Quality has become a main issue in software industry today. Defect tracking systems give developers a unique and clear view about user’s everyday product experiences (Serrano et al., 2005). These experiences are useful in achieving the good quality in software products, which is a valuable factor for the company to become successful in the industry. In order to achieve quality in software projects, different techniques are used like defect prevention, defect detection, reusability etc. There are also several process improvement methods like Capability Maturity Model Integration (CMMI), ISO (TickIT), SPICE and Bootstrap etc. are used to attain the quality in products.

The number of defect is commonly used to measure software quality. Both defect counts and defect-fix times are important factor for defect related analysis (Kim et al. 2006). Defect-fix time can be determined by identifying defects introducing changes and corresponding defect fixes (Kim et al. 2006). The defect resolution process within the software engineering lifecycle requires engineers to go through defect analysis, fixing, and updating mechanism. The majority of these activities are performed within a single Integrated Development Environment (IDE). However, the last phase of defect management requires the software engineer to go for two additional and separate systems - the configuration management system to enter changed comments and the defect tracking system for status and comment updates (Russell et al., 2006). Tools can be created to detect defects in software projects and they warn developer about defect’s existence. However, with a large universe of potential defects, real world history should be evaluated to determine where and how to search for potential errors. In research to devise tools to detect potential defects there
has been significant difficulty in correlating defect reports and source code (Russell et al., 2006).

Applied Testing and Technology’s web site (ApTest, 2008) offers a list of more than 88 defect-tracking tools. In those tools, some are commercial and some are open source. In order to review the defect tracking tools regarding this topic, authors have focused on open source softwares like Bugzilla, Abuky, Buggit, BugRat, BugTrack, GNATS, JitterBug, Scrab, Roundup, ITracker. In Bugzilla, to track a particular defect, it must be able to locate. The Product, Component, Version, Status, and Reporter fields are related to tracking, whereas the Summary, Status Whiteboard, Keywords, Severity, Attachments, and Dependency fields are related to fixing it. These fields contain the data that must input when reporting a defect and the data that helps to filter searches. Bugzilla’s reports are also another way to find information and it describes the current state of the defect, and charts, which describe an application’s state over time. As Bugzilla is a web application, so users interact only with its HTML pages. It works only with MySQL (Remillard, J. 2005). ITracker is also an issue-tracking system designed by Jason Carroll in 2002 to support multiple projects with independent user bases. Its features resemble Bugzilla’s. The main difference is that it is platform independent (because it’s a J2EE application) and database independent (Remillard, J. 2005). Abuky is a system for tracking defect and aiding the developer to fix them, written in Java with JSP as web interface (Serrano et al., 2005). Buggit manages defect and features throughout the software development process. Testers, developers, and managers can all benefit greatly from the use of Buggit. Buggit provides an unlimited number of central, multi-user databases, each capable of handling multiple concurrent users across the development team (Serrano et al., 2005). BugRat is free Java software that provides a sophisticated, flexible defect reporting and tracking system (Serrano et al., 2005). JitterBug is a web based defect tracking system. It was originally developed by the Andrew Tridgell to handle defect tracking, problem reports and queries from Samba users. JitterBug operates by receiving defect reports via email or a web form. Authenticated users can then reply to the message, move it between different categories or add notes to it. In some ways JitterBug is like a communal web based email system (Serrano et al., 2005). These are few of many defect logging tools which resemble to the author’s prototype model based defect logging tool.

Fault injection is a technique for improving the coverage of a test by introducing faults in order to test code. There are different ways of using fault injection technique like modifying the program’s source code or changing the state of machine of an executing program. Artificial errors can be injected into programs source code to estimate the number of remaining faults during testing (Bieman et. al., 1995). Software Implemented Fault Injection (SWIFI) techniques can be categorized into two types: Compile-Time Injection and Runtime Injection. In Compile-Time Injection technique, source code is modified to inject simulated faults into a system whereas in Runtime Injection techniques, software trigger has to inject a fault into a running software system. Faults can be injected via a number of physical methods and triggers can be implemented in a number of ways. Although these types of faults can be injected by hand, the possibility of introducing an unintended fault is high, so tools exist to parse a program automatically and insert faults. Five commonly used fault injection tools are Ferrari, FTAPE, Doctor, Orchestra, Xception and Grid-FIT (Wikipedia, 2008A). Some of these fault injection techniques will be used to inject faults in CVS source code files to validate the author’s prototype model based defect logging tool.
CVS and Bugzilla are two popular applications used to respectively manage source code and to track defects. In quality assessment approaches, information obtained by merging data extracted from problem reporting systems (such as Bugzilla) and versioning systems (such as Concurrent Version System (CVS)) is widely used. Changes in the source code are identified by means of file name and line numbers, referred as location by site (Ayari et. al., 2007). Reporting systems and versioning systems permits the integration of the information extracted from source code, defect reports and CVS change logs. It has been observed that available integration heuristics are unable to recover thousands of traceability links. Furthermore, Bugzilla classification mechanisms do not enforce a distinction between different kinds of maintenance activities. Obtained evidence suggests that a large amount of information is lost; it is assumption that to take benefit from CVS and problem reporting systems, more systematic issue classifications and more reliable traceability mechanisms are needed (Ayari et. al., 2007). In order to avoid the loss of valuable information and taking the full benefit from CVS source code files and problem reporting system, authors are motivated to design a model which will be useful in building the rich models of defect and fixes, which might ultimately overcome those problems.

1.2 Aims and Objectives
The basic aim behind this proposal is to design a model which can be used for extracting the information from the source code files. This framework should be used to build models of defect and their fixes. Following are the set of objectives that needed to be achieved.

- Exploring the existing defect fixing/tracking tools and their goals.
- To use explored defect fix models to build a database that can be mined for future defect avoidance and fix support.
- To design a model for extracting information that will help in building richer models of defect and fixes.
- To develop a prototype model of an interactive tool for logging defect fixes.
- To utilize the proposed tool to detect changes in the source/version control system.

1.3 Research Questions
Following are the research questions that need to be addressed:

- What are goals of existing defect fixing/tracking tools and their limitations?
- How to overcome the limitations in existing defect fixing/tracking tools?
- Design and development a tool to answer the limitations in existing defect tracking tools?

1.4 Expected Outcome
The following outcomes are expected after finalizing this research:

- A model for extracting information from the source code files that will help in building rich models of defect and fixes.
- DefectoFix, a defect fix logging will be developed using wxRuby.
- Evaluation of the designed framework
1.5 Research Methodology

Mixed research approach will be adopted to conduct this research. The research will be carried out in multiple phases as shown in Figure 1.1. In the initial phase, detailed review of literature and existing defect tracking tools will be done to understand the limitations in earlier defect logging tools. It will also help to select the effective procedure for designing a model for logging and fixing information. This procedure will lead to build a rich model of defect and fixes. In second phase, an interactive GUI tool will be developed using wxRuby to implement the prototype model. In third and last phase, a case study will be performed in which some defects will be injected in the source code file to validate the prototype model using the developed tool.

![Figure 1.1 Overview of Research Methodology]

1.6 Thesis Outline

This section provides the chapter outline of the thesis.

Chapter 2 “Preliminaries”, provides an overview about the general terms like Defects, Defects classification, and Ruby, which are used during the thesis. In Section 2.1 description about the defects is presented. Defect classification and revision control system are described in Section 2.2 and 2.3 respectively. Open source software and different types of defect tracking tools are described in Section 2.4 and 2.5 respectively. Section 2.6 describes about the fault injection techniques. Description about Ruby and Ruby parser is presented in Section 2.7 and 2.8 respectively. In Section 2.9 Abstract Syntax tree (AST) is described. S-expression is described in Section 2.10. In Section 2.11 description about is presented MySQL.

Chapter 3 “A Survey of Defect Tracking Tools”, provides the goals and limitation of the different types of defect tracking tools. Section 3.1 explains the goals and limitations
of Bugzilla. Bugcrawler’s goal and limitations are presented in Section 3.2. Section 3.3 discusses Codestriker’s goals and limitations. ITracker’s goals and limitations are explained in Section 3.4. JitterBug is explained with its goals and limitations in Section 3.5. Section 3.6 presented the goals and limitations of Mantis. Request Tracker’s goals and limitations are explained in Section 3.7. Section 3.8 discusses the goals and limitations of Roundup. Section 3.9 describes the goals and limitations of Redmin. GNATS goals and limitations are explained in Section 3.10. Section 3.11 presented the goals and limitations of DiTrack. CVSTrack’s goals and limitations are discussed in Section 3.12. Section 3.13 contains the summary table of goals of existing defect tracking tool.

Chapter 4 “Defectofix Framework Design”, describes the DefectoFix Framework. The framework will analyze the differences between the faulty and correct source code files, and store them in DefectoFix repository. Furthermore in the chapter different types of differencing algorithms, like diff, and JDiff, are described. Section 4.1 explains the goals of DefectoFix framework design. Overall design of the DefectoFix is presented in Section 4.2. Section 4.3 explores the various parts of the DefectoFix framework. Different types of file differencing algorithms are explained in Section 4.4.

Chapter 5 “Defectofix Differencing Algorithm”, describes the DefectoFix algorithm in detail. Furthermore, five different examples are selected to validate the DefectoFix algorithm. In Section 5.1 DefectoFix algorithm is presented. Explanation of DefectoFix algorithm is discussed in Section 5.2. Section 5.3 describes five different types of examples.

Chapter 6 “How Defectofix Works?”, describes the overall internal working of DefectoFix. Section 6.1 explains the working of DefectoFix. Fragmentation of source code files is presented in Section 6.2. Section 6.3 describes the Abstract Syntax Tree (AST) Generation. The extraction of Tree levels from AST is explained in Section 6.4. Section 6.5 describes the process of converting Tree levels in arrays using make_array() procedure. Section 6.6 describes the node matching of levels and nodes using NodeMatch() procedure. Defect registration process is explained in Section 6.7. Section 6.8 describes the validation process of DefectoFix through repository. View/Edit/Delete Defects from Defectofix repository is describe in Section 6.9.

Chapter 7 “Evaluation of Defectofix”, explains the evaluation of DefectoFix and it is divided into following subsections. The tools which are used in the development of DefectoFix are given in Section 7.1. Section 7.2 contains description about Case study. Description about the performed procedure for the evaluation of DefectoFix is presented in Section 7.3. Section 7.4 Section is about threats to validity of the conducted case study. Results and results analysis are given in Section 7.5 and 7.6 respectively.

Chapter 8 “Discussion”, discusses the findings after evaluation of DefectoFix. Section 8.1 gives the findings from our research work. Threats to validity to our research work are explained in Section 8.2. Section 8.3 describes the experience by the user while conducting this research.
Chapter 9 “Epilogue”, contains epilogue of the thesis. Section 9.1 contains conclusion of the thesis. Answering research questions are given in Section 9.2. Section 9.3 and 9.4 contains contribution to thesis and future work respectively.
2 PRELIMINARIES

This chapter provides an overview about the general terms like Defects, Defects classification, and Ruby, which are used during the thesis. In Section 2.1 description about the defects is presented. Defect classification and revision control system are described in Section 2.2 and 2.3 respectively. Open source software and different types of defect tracking tools are described in Section 2.4 and 2.5 respectively. Section 2.6 describes about the fault injection techniques. Description about Ruby and Ruby parser is presented in Section 2.7 and 2.8 respectively. In Section 2.9 Abstract Syntax tree (AST) is described. S-expression is described in Section 2.10. In Section 2.11 description about is presented MySQL.

2.1 What are Defects?

The term defect is commonly used to describe software problems and is closely related to the terms failure, fault, and error. According to the standard IEEE definitions (IEEE standard glossary of software engineering terminology, 1990), a failure occurs when program behavior deviates from user expectations, a fault is an underlying cause within a software program that leads to certain failures, and an error is a missing or incorrect human action that injects certain faults into the product. Failures, faults, and errors are often collectively referred to as defects, and defect handling deals with recording, tracking, and resolving these defects. Robert Grady of Hewlett-Packard stated in 1996 that “software defect data is the most important available management information source for software process improvement decisions,” and that “ignoring defect data can lead to serious consequences for an organization’s business” (Grady, 1996).

2.2 Defect Classification

The classification of software defects plays an important role in product improvement. The choices of defect types have evolved over the time. The idea is to capture distinct activities in fixing a defect which help to improve the quality of the process and product. Thus, there are only so many distinct things possible when fixing a defect. Defects vary from situation to situation. According to (Wilson et al., 2001), there are ten standard defect types, which are listed in Table 2.1 with its description. The defect type is based upon the semantics of the defect correction. The defect types are independent of the software product or development process used. The defect types span all software development life cycle phases, while at the same time each type is associated with a particular development activity.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation</td>
<td>comments, messages</td>
</tr>
<tr>
<td>Syntax</td>
<td>spelling, punctuation, typos, instruction formats</td>
</tr>
<tr>
<td>Build, Package</td>
<td>change management, library, version control</td>
</tr>
<tr>
<td>Assignment</td>
<td>declaration, duplicate names, scope, limits</td>
</tr>
<tr>
<td>Interface</td>
<td>procedure calls and references, I/O, user formats</td>
</tr>
<tr>
<td>Checking</td>
<td>error messages, inadequate checks</td>
</tr>
<tr>
<td>Data</td>
<td>structure, content</td>
</tr>
</tbody>
</table>
Some defects are very critical and other might be less important. According to (El Emam et al., 1998), there are five level of defect severity which are listed in Table 2.2.

Table 2.2 Severity Levels of Defects with Description (El Emam et al., 1998)

<table>
<thead>
<tr>
<th>Severity Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>The defect results in completely failure of a software system, subsystem, or software unit (program or module) within the system.</td>
</tr>
<tr>
<td>Major</td>
<td>The defect results in completely failure of a software system, subsystem, or of a software unit (program or module) within the system. There is no way to make the failed component(s), however there are acceptable processing alternatives which will yield the desired result.</td>
</tr>
<tr>
<td>Average</td>
<td>The defect does not result in a failure, but causes the system to produce incorrect, incomplete, or inconsistent results, or the defect impairs the systems usability.</td>
</tr>
<tr>
<td>Minor</td>
<td>The defect does not cause a failure and the desired processing results are easily obtained by working around the defect.</td>
</tr>
<tr>
<td>Exception</td>
<td>The defect is the result of non-conformance to a standard, is related to the aesthetics of the system, or is a request for an enhancement. Defects at this level may be deferred or even ignored.</td>
</tr>
</tbody>
</table>

In addition to the defect severity level defined above, defect priority level can be used with severity categories to determine the immediacy of repair. A five-level repair priority scale has also been used in common testing practices which are described in Table 2.3 (El Emam et al., 1998).

Table 2.3 Severity Levels of Defects with Description

<table>
<thead>
<tr>
<th>Severity Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resolve Immediately</td>
<td>Further development and/or testing cannot occur until the defect has been repaired. The system cannot be used until the repair has been affected.</td>
</tr>
<tr>
<td>Give High Attention</td>
<td>The defect must be resolved earlier because it is impairing development and/or testing activities. System use will be severely affected until the defect is fixed.</td>
</tr>
<tr>
<td>Normal Queue</td>
<td>The defect should be resolved in the normal course of development activities. It can wait until a new build or version is created.</td>
</tr>
<tr>
<td>Low Priority</td>
<td>Article I. The defect should be fixed but after serious defect has been resolved.</td>
</tr>
<tr>
<td>Defer</td>
<td>The defect repair can be put of indefinitely. It can be resolved in a future major system revision or not resolved at all.</td>
</tr>
</tbody>
</table>
Some other researchers have also worked on defect types. They classified these defects according to their analysis, thus called defect classification. According to (El Emam et al., 1998), classifications of a software error or fault based on an evaluation of the degree of impact that error or fault on the development or operation of a system. Further, it is necessary to know clearly the relationships of cause-effect or input-output during the modeling of software development process as an observable and controllable system. Many methods for the classification of software defects have been defined by various industry experts and software researchers (Bridge et al. 1997). Some of these various defect classifications are described in the following sections.

2.2.1 Orthogonal Defect Classification (ODC)

Orthogonal Defect Classification (ODC) is a method for classifying and analyzing software defects. ODC makes an essential improvement in the current technology to assist the software engineering decisions via analysis and measurement. This can be accomplished by exploiting software defects that occur throughout the software. ODC extracts the information from defects in efficient manner, like converting what is semantically very rich into a few important measurements on the product and the process. These measurements provide a grip on which sophisticated analysis and methodologies are developed. With the low cost and mathematical traceability of quantitative methods, it is possible to replace some details and expansiveness of qualitative analysis. (Lyu, M. R. 1996)

ODC classifies each defect based upon the logic of the defect correction and links the defect distribution to the development progress and maturity of the product. ODC is based upon the principle that different types of defects are normally discovered during different phases of the software development life cycle and that too many defects of the wrong type are discovered during a particular phase that may indicate a problem. ODC bridges the gap between casual analysis and statistical defect modeling. ODC also links the defects distribution to the maturity and development of the product. It provides measurement paradigm called as in-process that extracts key properties from defects and enable measurement of cause-effect relationships as opposed to a simple taxonomy of defects for descriptive purpose. ODC improves the technology for in-process measurement for the process of software development. (Bridge et al. 1997)

In almost any of the software development process, ODC creates a powerful software engineering measurement. This can happens by extracting information which is contained in software defects and something quite available in any software development process. ODC is commonly uses for cost reduction, process diagnostics quality improvement, schedule management, etc. Mainly ODC means that categorization of a software defect into classes that collectively point to the part of the development process that needs attention; it is mostly like characterizing a point in Cartesian system of orthogonal axis by its (x, y, and z) coordinates. The activities in software development process are broadly divided into design, test, and code; each organization might have its variation. This case can also happen that various releases may be developed in parallel while the process stages in several instances may overlap. Process stage can be carried out by different people and most of the time different organizations. For widely applicability of the defect classification, there must be consistency of classification scheme between the stages of defect. It is ideal that the defect classification should be independent of the specifics of a product or organization. If the classification is both independent of the product and consistent across phases, it is likely to
be process invariant and can eventually yield relationships and models that are very useful. Therefore the system has at least three requirements which is said to be good measurement system and that allows learning from experience and provides a means of communicating experience between projects: (Chillarege et al. 1992)

- Orthogonality
- Consistency across phases
- Uniformity across products

2.2.1.1 ODC Defect Attributes

The defect classification is not based on opinion like where it was injected but what was known about the defect (defect type or trigger). To provide measurement of software defects, ODC uses two attributes, Defect Type and Defect Trigger. (Bridge et al. 1997 and Chillarege et al. 1992)

2.2.1.1.1 Defect Type

The defect type attributes is designed to measure the progress of a product through development process. It basically identifies what is corrected and where it is associated with the different stages of the development process. Therefore, from different stages in the process a set of defects is classified to an orthogonal set of attributes that should bear the signature of this stage in it distribution. The alert signals points to the stage of the process that needs attention provides by the departure from the distribution. Therefore, the defect type provides feedback on the development process. (Chillarege et al. 1992)

ODC uses eight categories for defect i.e. Interface, Function, Checking, Algorithm, Build/Package/Merge, Assignment, Documentation, and Timing/Serialization (Chillarege et al. 1992). The eight categories which ODC uses for defect type are independent of the development process or software product used. The defect type categories cross all phases of software development life cycle, whereas at same instance each category is linked associated with a particular development activity. (Bridge et al. 1997)

2.2.1.1.2 Defect Trigger

The defect trigger attribute is designed to provide a measure of the effectiveness of the verification process. Defect triggers capture the circumstance that helps in finding the defect. The information that produces the trigger, measures the aspects of completeness of a verification stage. The verification stages could be the testing of code or the inspection and review of a design. These data can eventually provide feedback on the verification process. (Chillarege et al. 1992)

2.2.2 IEEE Defect Classification

According to (IEEE guide to classification for software anomalies, 1996), the classification process is a series of activities, which started with the recognition of defect. This process is divided into four sequential steps i.e. recognition, investigation, action, and disposition. The recognition step occurs when a defect is found. When a defect is recognized, its supporting attributes are recorded to identify the defect and the environment in which it occurred. The recognition step followed by is investigation. This investigation shall be of sufficient depth either to identify all known related issues to the particular defect or to indicate that the defect requires no action. A plan of action shall be established based on the
results of the investigation. The action includes all activities necessary to resolve the immediate defect. Following the completion of either all required resolution actions or at least identification of long term correction actions, each defect shall be disposed of.

2.3 Revision Control

Revision Control is also known as Version Control System (VCS) or Source Control or Source Code management (SCM). It is the management of multiple revisions of the same unit of information. In Revision Control, changes to the source code documents are identified by incrementing an associated number or letter code. This letter code is called "revision number" or simply "revision" and it is associated historically with the person making the change. Most revision control software can be used as "delta compression", in which only the differences between successive versions of files are kept. This allows more efficient storage of many different versions of files. Normally, revision control systems use a centralized model, where all the revision control functions are performed on shared server. If two developers try to change the same file at the same time may end up overwriting each other's work. This problem is addressed in centralized revision control systems by "File locking". Most version control systems allow multiple developers to be editing the same file at the same time. The first developer to "check-in" changes to the central repository will be able to change (Wikipedia, 2007B).

Distributed Revision Control Systems (DRCS) use peer-to-peer approach. In DRCS, each peer's working copy of codebase is a bona-fide repository. In DRCS, synchronization is conducted by exchanging patches from peer to peer. There are two types DRCS: open and closed. Open systems are those systems which provide some combination of interoperability, portability, and open software standards where as closed systems are more traditional and used to refer theoretical scenarios (Kalid, 2007).

There are number of instances of open source Version Control Systems in client server architecture available (Zooko, 2008). These include:

- Concurrent Versions System (CVS)
- CVSNT
- Fossil
- OpenCVS
- Subversion
- Vesta

Instances of open source, Version Control Systems in distributed model are (Zooko, 2008):

- GNU arch
- ArX
- Monotone
- Bazaar
- Mercurial
- Codeville
- Darcs
- SVK
- Aegis
- DCVS
- LibreSource
2.3.1 Concurrent Versions System

Concurrent Version System (CVS) is an important component of Source Configuration Management (SCM). It is used to record the history of sources files, and documents. It is also known as Concurrent Versioning System. It is an open source version control system. Instead of saving every version of every file, CVS stores only the differences between the versions. CVS also helps to work in a group on the same project: CVS merges the work when each developer has done its work. CVS uses client server architecture, where, a server stores the current version(s) of the project and its history and client connect to the server in order to check out a complete copy of the project, work on this copy and then later send their changes. CVS allows several developers to work on the same project concurrently. Each developer can edit files within his/her own working copy of the project and sending their modification to the server. (Akadia Information Technology, 2000)

CVS allows the isolation of the changes onto a separate line of development, which is called as a branch. When files are changed on a branch, the changes do not appear on the main trunk. Later changes can move from one branch to another branch (or the main trunk) by merging. The CVS repository stores a complete copy of all the files and directories which are under version control. CVS can access a repository by a variety of means. It might be on the local computer, or it might be on a computer across the room or across the world. A single project managed by CVS is called a module. A CVS server stores the modules by managing its repository. In CVS, acquiring a copy of a module is called checking out. The checked out files serve as a working copy, sandbox or workspace. Changes to the working copy will be reflected in the repository by committing them. (Akadia Information Technology, 2000)

2.3.2 Subversion

Subversion (SVN) is a version control system, which is used to maintain current and historical versions of files such as source code, web pages, and documentation. It is very popular in open source community. Some well known projects that use Subversion include: Apache Software Foundation, KDE, GNOME, Free Pascal, GCC, Python, Ruby, Samba and Mono. SVN is particularly designed for remote users and it has unreserved checkouts, similar to CVS. There are many clients are possible with SVN like command line client, Windows GUI client etc. SVN is apache-based, so it is high performance, scalable and secure. It can be used for peer review, testing, community feedback and contributions.

Subversion has four basis steps:

- Check out a “working copy”
- Make any edits
- Merge changes from server
- Commit changes

2.3.2.1 Subversion Features

The features of Subversion to our best knowledge are given below:

- The Subversion repository implements virtual versioned filesystem that tracks tree structures over time. Files and directories are versioned.
• A commit either goes into the repository completely or not at all (Ben, 2002).
• The Subversion network server is Apache, and client and server speak WebDAV to each other (Ben, 2002).
• A binary diffing algorithm is used to store and transmit deltas in both directions, regardless of whether a file is of text or binary type, which improve the access of the network (Ben, 2002).
• Each file or directory has an invisible hash table attached (Ben, 2002).
• Subversion has no historical baggage; it was designed and implemented as a collection of shared C libraries with well defined APIs. This makes Subversion extremely maintainable and usable by other applications and languages (Ben, 2002).
• Subversion is released under an Apache/BSD-style, open-source license (Ben, 2002).

Authors are motivated to use Subversion in their thesis, as Ruby (an OOP language, which will be used to implement the DefectoFix) is using Subversion and there are some drawbacks with CVS like:

• CVS can only track file contents, not tree structures. As a result, the user has no way to copy, move or rename items without losing history. Tree rearrangements are always ugly server-side tweaks (Ben, 2002).
• CVS uses the network inefficiently (Ben, 2002).
• CVS codebase is the result of layers upon layers of historical "hacks". This makes the code difficult to understand, maintain or extend (Ben, 2002).

2.4 Open Source Software

Open source software is computer software for which the human-readable source code is made available under a copyright license that meets the Open Source Definition (DiBona et al., 1999). This permits users to use, change, and improve the software, and to redistribute it in modified or unmodified form. It is often developed in a public, collaborative manner. Open source software is the most prominent example of open source development and often compared to user generated content i.e. the contents/code which is coded by the users.

Open source is a set of principles and practices on how to write software for which the source code will be openly available. The Open Source Definition, which was created by DiBona et al. (DiBona et al., 1999) and is currently, maintained by the Open Source Initiative, adds additional meaning to the term, one should not only get the source code but also have the right to use it. If it is denied then the license is categorized as a shared source license.

Under the Open Source Definition, licenses must meet ten conditions in order to be considered open source licenses (DiBona et al., 1999).

The software can be freely given away or sold.

1. The source code must either be included or freely obtainable.
2. Redistribution of modifications must be allowed.
3. Licenses may require that modifications are redistributed only as patches.
4. No one can be locked out.
5. Commercial users cannot be excluded.
6. The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.
7. The program cannot be licensed only as part of a larger distribution.
8. The license cannot insist that any other software it is distributed with must also be open source.
9. License must be a technology neutral.

In 1997, Eric S. Raymond (Raymond, 1999) suggests a model for developing Open Source Software (OSS) known as the Bazaar model. Gregorio Robles (Robles, 2004) suggests that software developed using the Bazaar model should exhibit the following patterns:

- Users should be treated as co-developers.
- The first version of the software should be released as early as possible so as to increase one’s chances of finding co-developers early.
- New code should be integrated as often as possible so as to avoid the overhead of fixing a large number of defects at the end of the project life cycle.
- There should be at least two versions of the software. There should be a buggier version with more features and a more stable version with fewer features.
- The general structure of the software should be modular allowing for parallel development.
- There is a need for a decision making structure, whether formal or informal, that makes strategic decisions depending on changing user requirements and other factors.

Most well-known OSS products follow the Bazaar model. These include projects such as Linux, Netscape, Apache, the GNU Compiler Collection, and Perl to mention a few.

2.5 Defect Tracking Tools

Defect tracking is a critical component to a successful software quality effort. Database is a major component of any defect tracking tool, as it records the facts about known defects. These facts may include the time when a defect was reported, its severity, the erroneous program behavior, and details on how to reproduce the defect; as well as the identity of the person who reported it and any programmers who may be working on fixing it. In defect tracking tools, at first software developers create a protocol entry whenever they detect a new defect in a software document. The entry starts with a timestamp recording. When the software developer localizes, understands, and repairs the defect, an additional time stamp and descriptive information completes the entry. The description can include the defect’s exact location, its type according to a fixed classification, the phase when it was most probably created, a hypothesis as to why it was, and possibly a verbal description. After
practicing this technique, and when using a compact format, recording becomes simpler and faster if the software developers perform it with a tool that records the time stamps and do some simple consistency checking. In the defect data analysis phase, software developers collect the defects into related groups according to appropriate criteria. These groups are then analyzed to find out the most frequent mistake to determine why they make those mistakes. Defect data tabulations, which are categorized by work phase, defect type, repair cost, and so on, are created automatically and are used to aid analysis. (Prechelt, 2001)

Control Version System (CVS) and defect tracking systems contain very useful information for various software engineering tasks. Information extracted from these systems is crucial in modeling quality characteristics such as error proneness. CVS is a simple system to record and coordinate changes, manage releases and versions. CVS is the backbone of open-source development and they are also widely adopted by the industry (Ayari et al., 2007). Defect tracking systems help to understand which parts of the system are affected by problems (D'Ambros et al, 2007). The change history of a software project contains a rich collection of code changes that record previous development experience. Changes that fix defect are especially interesting, since they record both the old faulty code and the new correct code (Kim, 2006).

Defect tracking tools are created to detect defect and to warn developers. But it is very difficult to determine where and how to search for potential errors. In research to devise tools to detect potential defects there has been significant difficulty in correlating defect reports and source code. Current research in capturing source location metrics associated with defects largely relies on complex data mining of the code repositories (Williams et al., 2005). The information obtained by data mining of code repositories, while fixing defects, provides higher level of granularity than what can be obtained by evaluating the code repository metadata. By integrating the control version system, the IDE, and the defect database, the classes, methods and functions can be logged in the defect database. Because the association between defect type, cause, description and location are stored in one database schema, reporting metrics is significantly more efficient and less costly (Russell et al., 2006).

## 2.6 Fault Injection

It may take a very long time for some errors to occur. Fault injection attempts to speed up this process by injecting faults into a running system. Fault-injection is the insertion of faults or errors into a computer system in order to determine its response. It is an effective method for validating existing fault-tolerant systems, and observing how systems behave in the presence of faults. Fault injection tests fault detection, fault isolation, and reconfiguration and recovery capabilities. Faults can be introduced in source code, trap/exception, time-out, trace mode, middleware and computational reflection. There are two categories of faults that can compose the fault load i.e. internal faults and external faults. The targets of fault injection are component, module/object, subsystem and system. Internal faults are introduced by developers by pre-runtime injections (mutation operators) and runtime injections. External faults caused by human interaction, operating system, hardware, other software system (Voas et al, 1998).
There are different ways of using fault injection technique like modifying the program’s source code or changing the state of machine of an executing program. Artificial errors can be injected into programs source code to estimate the number of remaining faults during testing (Bieman et al., 2000). Faults can be injected via a number of physical methods and triggers can be implemented in a number of ways. Although these types of faults can be injected by hand, the possibility of introducing an unintended fault is high, so tools exist to parse a program automatically and insert faults. Five commonly used fault injection tools are Ferrari, FTAPE, Doctor, Orchestra, Xception and Grid-FIT (Wikipedia, 2008A).

2.6.1 Fault Injection Techniques

Fault injection approaches can be classified into hardware-implemented and software-implemented fault injection (SWIFI). Software fault-injection techniques are attractive because they don’t require expensive hardware. Furthermore, they can be used to target applications and operating systems, which is difficult to do with hardware fault injection. Software-implemented fault injection can be further classified into simulation-based fault injection and prototype-based (Larsson et al., 2006).

2.6.1.1 Simulation-based Fault Injection

Most often low-cost, simulation-based fault injection techniques are used for evaluation. Simulation-based fault injection assumes that errors or failures occur according to predetermined distribution. It is useful for evaluating the effectiveness of fault-tolerant mechanisms and a system’s dependability. However, it requires accurate input parameters, which are difficult to supply. (Larsson et al., 2006)

2.6.1.2 Prototype-based Fault Injection

On the other hand, testing a prototype, allows evaluating the system without any assumptions about system design, which yields more accurate results. In prototype-based fault injection, faults are injected into the system to study system behavior in the presence of faults and performance loss. To do prototype-based fault injection, faults are injected either at the hardware level or at the software level to monitor its effects. (Voas et al, 1998)

Although the software approach is flexible, it has its shortcomings as well like it cannot inject faults into locations that are inaccessible to software, and the poor time-resolution of the approach may cause problems for short latency faults, such as bus and CPU faults, the approach may fail to capture certain error behavior, like propagation. Software injection methods can be categorized on the basis of when the faults are injected: during compile-time or during runtime. In compile-time injection, the program instruction must be modified before the program image is loaded and executed. Whereas in runtime injection a mechanism is needed to trigger fault injection. (Voas et al, 1998)

2.7 Ruby

This study is focused on the defects finding in dynamic languages. One such dynamic language is Ruby (Thomas, 2004), which has continued to grow in popularity in recent years. Ruby was created in 1994 by Yukihiro Matsumoto and publically available in 1995. Ruby is very popular in Japan. It is basically mixture of Perl, Smalltalk, Eiffel, Ada, and Lisp (Ruby, 2008). It is extremely flexible and easy to use language and contains pretty nil features as now days other programming languages contain. Ruby is purely object-oriented language, which means that everything is an object, including primitive types such
as bytes, integers, Booleans and chars. Ruby is duck-typing i.e. objects are described by what they can or can not do, instead of being associated to a specific type. So the interpreter doesn't care what type the data is, but only if it can be used in given context. Another characteristic of Ruby is its reflective ability, which means that much information about the code itself is accessible during runtime. Its execution speed is very fast as compared to other languages. The program written in one environment can run mostly in all other environment without any change in code because of it’s highly portability. One of main advantage is that, it is available free. These features makes Ruby an attractive programming language. Ruby is available in many other forms like wxRuby (for GUI), Ruby on Rail (for web) etc (Ruby, 2008).

2.8 Ruby Parser

Ruby Parser (RP) has been developed purely in Ruby language (utilizing racc—which does by default use a C extension) with easy to use interface. The output generated from RP is represented as symbolic expression (s-expression) using ruby’s arrays and base types, which is same as output generated by Parse Tree. RP is totally free to use, modify, and publish. The output of the line “2+2” by RP is given below in form of s-expression (Ryan, 2007):

```
RubyParser.new.parse "2+2"
=> s(:call, s(:lit, 2), +, s(:array, s(:lit, 2)))
```

2.9 Abstract Syntax Tree

Abstract Syntax Tree (AST) is the representation of source code as a tree of nodes representing constants or variables (leaves) and operators or statements (inner nodes). AST’s are at abstract level so they do not contain the entire information of source code like spacing, comments, parenthesis etc. AST is nearly direct translation of the grammar (Stephen, 2003). AST’s are called as n-ary trees, because each node holds pointers and token to its next sibling and first child (Stephen, 2003).

AST is also called as ”parse tree”, because they are obviously used to develop parsers (Jeff, 2004). An AST is output of a parser, and forms the input to semantic analysis and code generation (Cunningham, 2006). A parser converts the syntax into corresponding AST. AST mainly contains the semantic information of the soured code. There are two steps to generate AST. First step, lexer (like flex) is used in order to determine tokens (sequence of character that makes words in the language). Second step, parser (like bison) is used to structurally group the words in order to generate corresponding AST (Jeff, 2004). The AST of the given below code is shown in Figure 2.1 (Jeff, 2004).

```
Code:
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
x := a + b
}
```

![Figure 2.1 AST of Code](chart.png)
2.10 S-Expression

S-expressions are a data structure for representing complex data. They are a variation on LISP S-expressions. The basic data structures of LISP are lists (object composed of atoms and/or other lists) and atoms (smallest indivisible element). Mainly S-expression is consists of both lists and atoms. In context of LISP all data and program are S-expression. Figure 2.2 explains the S-expression graphically (Marcelo, 2004). Table 2.4 shows the S-expression of the code.

![Graphical representation of S-Expression](image)

Table 2.4 Code and its S-Expression

<table>
<thead>
<tr>
<th>Code</th>
<th>S-Expression</th>
</tr>
</thead>
</table>
| class TestClass  
  def method1(a,b)  
    sum = a + b  
    puts sum  
  end  
  def method2(*c)  
    l = c.length  
    puts l  
  end | s(class, :TestClass, s(:const, :Object), s(:defn, :method1, s(:scope, s(:block, s(:args, :a, :b), s(:lasgn, :sum, s(:call, s(:lvar, :a)), :+, s(:array, s(:lvar, :b)))), s(:fcall, :puts, s(:array, s(:lvar, :sum))))), s(:defn, :method2, s(:scope, s(:block, s(:args, :"*c"), s(:lasgn, :l, s(:call, s(:lvar, :c)), :length)), s(:fcall, :puts, s(:array, s(:lvar, :l)))))) |

The term S-expression also known as sexp (S stands for symbolic) refers to a convention for representing semi-structured data in human-readable textual form. S-expressions are much known for their use in the Lisp family of programming languages. It is probably best known in the context of LISP2. S-expression is a notation for presenting tree structures in a linear text enclosed in parenthesis, containing either atom elements or further s-expressions. If we talk in the light of Parse Tree so the first element of s-expression defines the meaning of whole expression (McCarthy, 1959). LISP has purely symbol-oriented structure. S-expression can be either an OctetString or lists of s-expressions. A sample s-expression is given below (Rivest, 1997):

```
(tables "mno" (#04# HGRA ))
```

The above s-expression is a list of length three:
1. OctetString “tables”
2. OctetString “mno”
3. Sub-list containing two elements
   - hexadecimal constant #04#
   - base-64 constant [HGRA] (which is same as "mno")
2.11 MySQL

MySQL is based on SQL (Structured Query Language) database server. SQL is one of most popular database for web applications and due to its high robustness and speed. MySQL is based on client/server architecture. It has lots of usage in web applications like PHP and Ruby on Rails. It has lots of remarkable features which make him so famous. Few of them are given below (MySQL, 2008):

- The data (like password etc) transfer on the net are encrypted.
- It can handle huge database.
- Mostly all types of ODBC (Open Database Connectivity) can functions.
- No memory leakage. It has been tested on commercial bases memory leakage detector (purify)
- **Isamchk** utility is used to check and repair table very fast.
- It has very strong cross-platform support with multiple storage engines.

There is a detailed survey of defect tracking tools in the next chapter, which highlight the goals and limitation of existing defect tracking tools. In the end, a summary of major goals of defect tracking tools is presented.
3 A SURVEY OF DEFECT TRACKING TOOLS

Defect tracking is a methodology used by software developers to collect reports of defects or "bugs" in software programs. Defect tracking allows developers to further refine software design by making continual changes or upgrades to the product in order to better serve the customer base.

Applied Testing and Technology’s web site (ApTest, 2008) offers a list of more than 88 defect tracking tools. In those tools, some are commercial and some are open source. Furthermore, few of them are web-based and few are desktop applications. All those tools have different goals and tradeoffs. We have selected few tools to evaluate DefectoFix. The selections of the tools are on basis of the following attributes:

- The tools which are open source
- The tools which can run on multiple platforms
- The tools which can store information in multiple database formats like MySQL etc.
- The tools which can integrate with revision control systems like CVS, Subversion etc.
- The tools which extract information from the source code
- The tools which fix defects at source code level
- The tools whose common goal is to facilitate the users in finding defects
- The tools which are web-based application or desktop application.

This chapter provides the goals and limitation of the different types of defect tracking tools. Section 3.1 explains the goals and limitations of Bugzilla. Burcrawler’s goal and limitations are presented in Section 3.2. Section 3.3 discusses Codestricker’s goals and limitations. ITracker’s goals and limitations are explained in Section 3.4. JitterBug is explained with its goals and limitations in Section 3.5. Section 3.6 presented the goals and limitations of Mantis. Request Tracker’s goals and limitations are explained in Section 3.7. Section 3.8 discusses the goals and limitations of Roundup. Section 3.9 describes the goals and limitations of Redmin. GNATS goals and limitations are explained in Section 3.10. Section 3.11 presented the goals and limitations of DITrack. CVSTrack’s goals and limitations are discussed in Section 3.12. Section 3.13 contains the summary table of goals of existing defect tracking tool. Our findings about these tools are described in Section 3.14.

3.1 Bugzilla

Bugzilla is one of most popular web-based defect-tracking tool. It is written in Perl. It allows individuals or groups of developers to keep track of outstanding defects in their product effectively. Bugzilla is mainly focused on dependency tracking and graphing, milestone tracking, detailed defect reporting (including component selection), resource description, developer assignment, granular priority description and attachment capabilities. Bugzilla is maintaining revision history using path file. It stored data in MySQL server and also integrated with CVS. (Remillard, 2005)

3.1.1 Limitations

The limitations for Bugzilla to our best findings are listed below:
• Bugzilla is a big project and full of features but it requires a lot of up-front planning and setup. Developers will have no problems using it, but end users find it a little too idiosyncratic (Remillard, 2005).
• Bugzilla is a web application, so users interact only with its HTML pages (Johnson et al., 2003).
• Bugzilla’s design principles state that it should support commercial databases, but it works only with MySQL, a popular open source database (Serrano et al., 2005).
• Bugzilla is not useful for formal inspection due to its inability to track comments individually (Remillard, 2005).
• There are number of security flaws in Bugzilla that can put users of the defect tracking tool at risk of cross site scripting, data manipulation and data exposure attacks (Naraine, 2006).
• Bugzilla does not sanitize various fields when embedded in certain HTML headline tags. This can be exploited to execute arbitrary HTML and script code in a user’s browser (Naraine 2006).
• In Bugzilla, when attachments are viewed in “diff” mode, it could let unauthenticated users to read the description of all attachments (Naraine 2006).
• During the export process of defects to XML format, the “deadline” field is visible to users who are not members of the “timetrackinggroup” group, which can be exploited to gain knowledge of potentially sensitive information. Also this could allow a malicious user to pass a URL to an administrator and make the administrator delete or change something that he or she had not intended to delete or change (Naraine, 2006).
• Unpatched versions of Bugzilla allow users to perform certain sensitive actions via HTTP GET and POST requests without verifying the user’s request properly. This can exploited to modify, delete or create defects (Naraine, 2006).
• Bugzilla is not user friendly and it is very difficult for the users to use it without some basic training (Naraine, 2006).

3.2 BugCrawler

BugCrawler is a language independent tool which supports software evolution and reverse engineering. It is easy to use and good GUI. It is designed to simply insert logs and run crawls. It is based on combination of software metrics and interactive visualization. It integrates structural information computed from the source code with evolutionary information retrieved from CVS log files and problem reports. BugCrawler has predefined views, which create user-defined visualization. It supports multi-layered projects. BugCrawler supports the analysis of the evolution of software systems by showing them under different perspectives. It provides visualizations targeted at answering reverse engineering questions. (D’Ambros et al, 2007)

3.2.1 Limitations

The limitations for BugCrawler to our best findings are listed below:
• BugCrawler has predefined views, which confuse users often.
• It focused more on reverse engineering.
• It should have more easy views with customization option to the users.
• It should also have more generalized structure rather than specialized to reverse engineering.

3.3 Codestriker
Codestriker is an open source web application which supports online code reviewing. Traditional document reviews are supported, as well as reviewing differences generated by an SCM (Source Code Management) system and plain unidifferent patches. There is a plug-in architecture for supporting other SCMs and issue tracking systems. Codestriker is useful for reviewing as it minimizes the paper work and ensures that issues, comments and decisions are recorded in a database. It provides comfortable workspace for actually performing code inspections. It has a good support for formal inspections with metrics and for inspection meetings. (Remillard, 2005)

3.3.1 Limitations
The limitations for Codestriker to our best findings are listed below:
• Codestriker is limited to review pure text file only. It cannot review documents that require formatting, tables, or images (Remillard, 2005).
• Codestriker sends a lot of emails. It doesn't allow users to customize their email preferences (Remillard, 2005).
• Codestriker doesn't support checklists (Remillard, 2005).

3.4 ITracker
ITracker is an issue-tracking system. It is designed to support multiple projects with independent user bases. It is a J2EE application with platform and database independent. It supports multiple versions and components, detailed issue histories, and email notifications. (Serrano et al., 2005)

3.4.1 Limitations
The limitations for ITracker to our best findings are listed below:
• ITracker required Java runtime for work (Mantis, 2006).
• ITracker project, which is an open source project, has been terminated so there is no further fixes and supports for the users of this tool (Mantis, 2006).

3.5 JitterBug
JitterBug (Tridgell, A., 2002) is a web based defect tracking system. JitterBug operates by receiving defect reports via email or from a web form. JitterBug is designed to track defects found in samba, which is an application that allows file sharing between Windows and UNIX platforms. It is written in C language, and it runs as a CGI with a built-in email client. There is no database required in it and all defects are kept as flat files. Defects are reported and updated via email or web forms. JitterBug is very useful for small group deployments. Each user has a configurable web environment, which is well organized and easy to use. Authenticated users can then reply to the message, move it between different categories or add notes to it. In some ways JitterBug is like a communal web based email system. (Viega, J. et al., 2002)
3.5.1 Limitations
The limitations for Jitterbug to our best findings are listed below:

- Although Jitterbug is a freely available, open source defect tracking system, but it is only available for Linux platform (Martin, 2006).
- Jitterbug has some problems with input handling. Because of this, an attacker may be able to gain unauthorized access to vulnerable systems (Martin, 2006).
- The project of JitterBug is not actively maintained by its developers, so there are problems to get support for this project (Martin, 2006).
- JitterBug items are sorted by ID (default), which is a meaningless field. Sorting by ID puts the issues in order by ascending submission date, which throw out recent issues far away at the bottom of the list (Jones, 2008).

3.6 Mantis
Mantis is an open source project which is written in PHP scripting language. It works with MySQL and PostgreSQL databases. It can be installed on Windows, Linux, Mac OS, OS/2, and other operation systems. Mantis allows contributing changes in core package, which help in reimplementation of changes after upgrade. The basic goal of Mantis is to produce and maintain lightweight, simple defect tracking system (Ito, 2002).

3.6.1 Limitation
The limitation for ITracker to our best findings is listed below:

- Most of the corporate users uses Oracle database. A ticket engine like Mantis is fine with MySQL but as soon as corporate users start talking integration and security, at this stage MySQL cannot compete with Oracle database (Mantis, 2006).

3.7 Request Tracker (RT)
Request Tracker (RT) is designed for custom extension. It was developed in Perl language. RT offers command-line, web, and email interface to the defect tracking tool. It uses SQL as back end, with MySQL being the default database. It offers customizable scripts that allow modifying workflow notification and resolution behavior. It can run on multiple platforms. Defect histories are supported by RT. RT is an industrial-grade ticketing system. Using RT, a group of people intelligently and efficiently can manage requests submitted by a community of users. RT is used by systems administrators, customer support staffs, NOCs, developers and even marketing departments at over a thousand sites around the world. (Rich, 2008)

3.7.1 Limitations
The limitations for RT to our best findings are listed below:

- In RT, end users cannot tracker work order, or they are unable to find out history of work order, if they deleted work order emails (Jay Lee et al., 2006).
- As RT was developed in Perl and Perl based installation can be difficult to install or upgrade. These setting remains stable until the user don’t touch anything though (Jay Lee et al., 2006).
• RTFM, which is a knowledgebase module of RT, it needs lot of work and improvements (Lee, J. et al., 2006).
• Searching mechanism is complicated in RT (Lee, J. et al., 2006).
• RT mostly uses emails, which can get very long and thus difficult to follow when users quote text in replies (Lee, J. et al., 2006).

3.8 Roundup

Roundup is an open source defect tracking tool, which has command-line, web-based, and email interfaces. Roundup focuses on issue assignments. The command line tool of Roundup can export and import data from the tracker and do other tasks. It can be customized to achieve higher level of detail. It is based on the winning design from Ka-Ping Yee in the Software Carpentry "Track" design competition (Jones, 2008). One of Roundup's strengths is its support for many different databases. In Roundup, all of the database access is abstracted through a hyperdb, which handles the actual database communication. It is easy to install and configure. It is platform independent. (Richard, 2008)

3.8.1 Limitation

The limitation for Roundup to our best finding is listed below:

• Roundup user interface is complex and user takes lot of time to understand it (Richard, 2008).

3.9 Redmine

Redmine is a flexible project management web application. It is written in Ruby on Rails framework. It is a cross-platform and cross-database application. It supports multiple projects and has a flexible role based access control system. It is equipped with Gantt chart and calendar. It supports SCM integration like, SVN, CVS, Mercurial, Bazaar and Darcs. (Redmine, 2006)

3.9.1 Limitations

The limitations for Redmine to our best findings are listed below:

• According to Redmine official website, Redmine is in beta status yet and improved regularly.
• Its installation is complex and user feels problem during installation.

3.10 GNATS

GNATS is a portable defect tracking system which runs on UNIX-like operating systems. It easily handles thousands of problem reports. It has been in wide use since the early 90s, and can do most of its operations over e-mail. There are several front end interfaces of GNATS exist, including command line, emacs, and Tcl/Tk interfaces. There are also a number of Web (CGI) interfaces written in scripting languages like Perl and Python. It allows easy use and provides good flexibility. (Gray et al., 2003)

3.10.1 Limitation

The limitations for GNATS to our best findings are listed below:

• GNATS is limited to UNIX like operating systems. It should platform independent.
• Not user friendly due to multiple interfaces, users can confuse while operating GNATS.

3.11 DITrack

DITrack is a free, open source, lightweight, distributed defect tracking system. It is implemented in Python and runs in UNIX (BSD, Linux, MacOS) and Windows environment. The goal of this tool is to provide instantly a solution to a user to start tracking issues i.e. without requirements for complex backend infrastructure. Currently DITrack uses Subversion as its distributed file system backend. (Skvortsov et al., 2006)

3.11.1 Limitations

The limitation for DITrack to our best finding is listed below:

• DITrack is newly built project and yet to be mature. It needs more optimization.

3.12 CVSTrac

CVSTrac is a Web-based issue tracking tool that integrates with the CVS version control system. CVSTrac includes a CVS repository browser. (Tsai, 2003)

3.12.1 Limitations

The limitations for CVSTrac to our best knowledge are listed below:

• Searching facility in CVSTrac is not efficient (Tsai, 2003).

• There is no pre-built distribution of CVSTrac for MAC OS X users (Tsai, 2003).

3.13 Summary of Goals of Defect Tracking Tools

The main goal of all the tools which authors have described above, are given in Table 3.1 shows.

<table>
<thead>
<tr>
<th>Tools</th>
<th>Design Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>It is focused on dependency tracking, milestone tracking, detailed defect reporting, resource description, developer assignment, granular priority description and attachment capabilities.</td>
</tr>
<tr>
<td>BugCrawler</td>
<td>Simple to use is the main objective of this tool. It is specially designed for multi-layered projects.</td>
</tr>
<tr>
<td>Codestriker</td>
<td>Its main goal is online code reviewing. It is design to integrate with many source code control systems, including CVS, Subversion, Clearcase, Visual Source Safe.</td>
</tr>
<tr>
<td>ITracker</td>
<td>The main goal of this project is to design an issue-tracking system that will support multiple projects with independent user bases.</td>
</tr>
<tr>
<td>JitterBug</td>
<td>It is designed to track defects found in Samba which is an application that allows file sharing between Windows and UNIX platforms.</td>
</tr>
<tr>
<td>Mantis</td>
<td>It is designed to produce and maintain lightweight, simple defect tracking system, which can run on any platform and database.</td>
</tr>
<tr>
<td>Tool</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Request Tracker (RT)</td>
<td>It allows custom extension, which is the main goal of Request Tracker. It offers customizable scripts that allow modifying the workflow notification and resolution behavior.</td>
</tr>
<tr>
<td>Roundup</td>
<td>It is specially designed to allow issue assignment on multiple databases.</td>
</tr>
<tr>
<td>Redmine</td>
<td>This tool is developed to support multiple projects and to have a flexible role based access control system.</td>
</tr>
<tr>
<td>GNATS</td>
<td>It is developed to allow problem report management and communication with users via various means.</td>
</tr>
<tr>
<td>DITrack</td>
<td>The goal of this tool is to provide a solution that would allow one to start tracking issues almost instantly, i.e. without requirements for complex backend infrastructure.</td>
</tr>
<tr>
<td>CVSTrack</td>
<td>This tool is designed to integrate with the CVS version control system.</td>
</tr>
</tbody>
</table>
4 DEFECTOFIX FRAMEWORK DESIGN

This chapter describes the DefectoFix Framework. The framework will analyze the differences between faulty and correct source code files, and store the obtained defect fix model in a DefectoFix repository for future usage like fault inject process. Section 4.1 explains the goals of DefectoFix framework design. Overall design of the DefectoFix is presented in Section 4.2. Section 4.3 explores the various parts of the DefectoFix framework. DEFECT fix model usage scenario is explained in Section 4.4. Different types of file differencing algorithms are explained in Section 4.5.

4.1 Framework Goals

The major goals behind the design of DefectoFix framework are:

- To develop a defect fix logging tool that will collect more information and thus avoid loss of defect information while fixing source code defects.
- This framework will be able to utilize various version control systems to keep track of various source code files by recording their differences and their extracted models.
- This framework will act as a foundation for developing an easy to use open source defect fix logging tool, which can run on every operating system.
- To develop a defect fix logging tool, that can used to extract differences between faulty and correct source code files.
- A framework can be used with any other programming language by changing the core of differencing algorithm.

4.2 Overall Framework Design

The overall design of DefectoFix framework is explained in Figure 4.1. Various parts of DefectoFix framework are explained in detail in the following sections.
4.3 DefectoFix Framework Parts

DefectoFix framework consists of the following parts.

1. Version Control System (e.g. SVN, CVS)
2. Faulty and correct source code file(s)
3. Source code differencing algorithm
4. Defect Model Creation
5. Additional Information
6. Record information in DefectoFix repository

Below is the detailed explanation of every part of DefectoFix framework.

4.3.1 Version Control System

This part works as an input to DefectoFix framework. The Version Control System (VCS) contains various versions of the source code files. Our proposed framework will checkout source code files (faulty and correct source code file) or from the information of log messages of VCS for analysis purpose.

4.3.2 Faulty and Correct Source Code Files

There are two input files for the framework i.e. faulty and correct source code files, but in general, there can be more than two input files. Faulty file is an incorrect version of correct source code file that may contain a number of defects. The defects might be like reference problems in variable, change of data structure of variable or the body of the method is changed etc. Where-as in a correct source code file these problems do not exists or they are resolved. Both faulty and correct source code files will be extracted from VCS.

4.3.3 Source Code Differencing Algorithm

A source code differencing algorithm is used to find out the differences between the faulty and correct source code files. The basic purpose of this algorithm is to find out detailed information about defects in source code files. We have developed our own differencing algorithm (DefectoFix Differencing Algorithm) for this purpose. The algorithm is explained in detail in chapter no.5. However, we have studied number of related algorithm, which are discussed in the end of this chapter.

Our differencing algorithm uses a parser for static code analysis purpose of faulty and correct source code files. The parser allows us to perform the analysis at the syntax level so that full information of the fault is extracted. This is in contrast to text-, word- or line-based differencing algorithms. The syntax of the source code is treated as a tree level in Abstract Syntax Tree (AST) and compared with each other in order to find out faulty nodes. The faulty tree levels will be stored with their faulty children in the repository because tree level with children will provide full information about fault location. The obtained information (all faulty and correct tree nodes) will be passed on to the next part of framework i.e. Defect Model Creation (DFC).
4.3.4 Defect Model Creation

It is a process that generalizes the extracted diff information (obtained from the previous step) into a model of the defect fixes. The extracted Defect Model can later be used for various purposes e.g. fault injection in source code files.

4.3.5 Additional Information

The additional information about the defect will be attached in this part. This additional information can come either from the VCS, from (filtered out info) from the diff or by allowing the developer to add additional information to the model to further characterize the defect.

4.3.6 Record Information in DefectoFix Repository

The extracted defect model information with additional information, obtained in the previous step is stored in DefectoFix repository (e.g. MySQL). If in the future, the similar enough fault(s) is/are found, the DefectoFix can be used to inform about previous situations and possible solutions. The information about differing instances of the same defect, which is matched by a single defect model, will be saved in the repository.

4.4 Defect Fix Model Usage Scenario in Fault Injection

The Defect Fix Model (DFM) is extracted from DefectoFix framework by comparing faulty and correct source codes. It can then be used for various areas of software verification and validation. Fault injection can be one of these areas, where DFM can be useful. In automated fault injection, the extracted DFM from the repository would be injected in the correct source code for further validation and verification processes. By doing so, we make faulty source code out of correct source code. These two files are then compared to find problems among these files. Figure 4.2 explains the whole process of fault injection in source code files for validation processes.

![Figure 4.2 Defect Fix Model Usage Scenario in Fault Injection](image-url)
4.5 Source Code Differencing Algorithms

In order to find differences in source code files by tree level/node comparison procedure, DefectoFix is needed to use some sort of differencing algorithm, which can extract maximum information of fault in the source code files. There are number of existing differencing algorithms for the same purpose; however each of them has their own design goals. A customized differencing algorithm is required to fulfill the task because DefectoFix operates in tree level/node comparison procedure for object oriented languages, which lacks in previous differencing algorithms. In order to develop own customized differencing algorithm, a detailed study of related differencing algorithm is required. It is very difficult to explain all the related differencing algorithms, therefore only few of differencing algorithms are discussed in detailed below. Our proposed differencing algorithm is explained in detail along with examples in next chapter.

4.5.1 Diff Algorithm

Diff can identify variations between versions of source code files to detect modifications. It is helpful to detect addition, deletion and replacement of code within a source code file. This algorithm is available in number of variations but the basic functionality is always the same i.e. line-by-line comparison of text files (Thinkage, 2000).

diff3 is another shape of diff algorithm, which is used to show differences among three files (Wikipedia, 2008C). The unified format (or unidiff), is another flavor of diff algorithm. It inherits the technical improvements made by the context format, but produces a smaller diff with old and new text presented immediately adjacent (Wikipedia, 2008C).

Wdiff, is another diff algorithm shape. It makes easy to see the words or phrases that changed in a text document, especially in the presence of word-wrapping or different column widths.

diff algorithm assumes that each input file will have some sections that are the same as all the other files. The sections that are common in all files allow diff to resynchronize its comparisons. We can say that, diff looks for blocks of identical text in order to break the input files into corresponding sections, then compares the corresponding sections for differences (Thinkage, 2000).

4.5.1.1 Example

For example, suppose that File1 (faulty source code file) and File2 (correct source code file) are different versions of the same program source code. The marker blocks are made up of the lines that are the same in both files. In between the marker blocks are lines that are different in the two files. diff identifies the marker blocks first in order to "sync up" the structures of the files that are being compared. In this way, diff compares corresponding sections of the two files. diff searches out all possible marker blocks in the files that are being compared then shift out the differences that occur in between those markers. In practice, however, this would require too much memory and processor time, especially with large files. Therefore, diff starts at the beginning of each file and looks for marker blocks that are "reasonably large". The sections of text between these large marker blocks are called "difference sections" and the large marker blocks are called "major" marker blocks. Once diff has identified a difference, diff scans the section progressively for
smaller marker blocks, until diff has divided the section into lines that are the same in all files and lines that are different (Thinkage, 2000).

4.5.1.2 How diff Algorithm Works?

diff algorithm is based on line by line comparison. So in order to compare a line from one file with a line from another, diff begins by extracting column sections from each. Column sections are compared individually. After a line has been broken into column sections, each section is reduced according to the “Ignore=” options”. Every string of one or more Ignore= characters is replaced by a single special character that shows where characters have been ignored. Strings of ignored characters are discarded. (Thinkage, 2000)

As an example of how this works, suppose that we are using Ignore= “” to ignore blank characters. Then all of the following are equivalent (Thinkage, 2000):

"a b"
"a   b"
"a   b  

All of these are reduced to "aXb", where X stands for the special character which shows where characters were ignored. Notice that the final result is NOT the same as "ab". Ignore=“” does not discard characters in the middle of strings; it simply reduces multiple occurrences to a single character (Thinkage, 2000).

Note that leading characters are not discarded. For example,

" a  b"
" a   b"
" a   b  

Both reduce to "XaXb" (where X stands for the special character). This is not the same as "aXb".

In comparing column sections, empty columns compare equal to one another. For example, suppose there is a column section running from columns 73-80; if diff is comparing two lines that are both shorter than 73 characters, the 73-80 column section is empty for both lines. Therefore, the lines are considered equal in that column section. The same holds for partial column sections. For example, if the column section runs from 73-80, but both input lines end at column 74, diff only compares the characters in columns 73 and 74. If these characters are equal, the input lines are considered equal in that column section (Thinkage, 2000).

There are four possible output formats, depending on the settings of the “Matching” and “Differences” options. If we specify both -Matching and -Differences, we just get a summary of the number of differences between the files involved. If we specify +Matching and -Differences, we get the lines that are found in all the files being compared. Groups of matching lines are separated into blocks: the first group of lines that are common to all files, the next group of lines that are common to all files, and so on. Blocks are separated by the primary separator (Thinkage, 2000). By default, the separator is
But this can be changed with the “SeParator= option”. The line containing the primary separator also contains file names and line numbers to show where the block of matching lines appears in each file, as in

-------- file1,10 file2,14 file3,5

If we specify +Differences and -Matching, we get the lines where the files differ from each other. The output is divided into blocks. Each block shows one or more lines where the input files differed. Blocks are divided into subsections, with each subsection showing the contents of one or more files (Thinkage, 2000). Here is a typical example of a block:

---------------- file1,line# file3,line#

some lines in file1
"a same line"

---------- file2,line# file4,line#

the equivalent (changed) lines in file2
"a same line, as above"

.....

The first line in the block begins with the primary separator string. The default primary separator is "--------" but you can specify a different string with the SeParator= option. On the same line is a set of file names with line numbers. All of these files have the same data in this part of the file. In the example above, "file1" and "file3" have the same contents in this part of the file (Thinkage, 2000).

In summary, the block begins with a primary separator line and consists of subsections which each begin with a secondary separator line. Each subsection within the block gives a set of lines that appear in one or more files. These subsections show where the groups of files differ. If we specify +Matched and +Differences, we get a complete listing of where the files match and where they are different. Blocks of matching lines are marked (on the end) with primary separators, without file names. Blocks of differing lines are marked in the same way as in the previous output format.

4.5.1.3 Shortcomings of diff Algorithm

The shortcomings to our best knowledge about diff algorithm are as follows:

- It is now clear from all above explanation that diff algorithm is performing comparison on the basis of “line-by-line”.
- Authors needs more fault explanation, rather than just comparing one line with other line.
So this algorithm does not fulfill our requirements therefore it gives us good idea, how we can compare two files with each other.

4.5.2 Tree Diff Algorithm

A tree differencing algorithm is used to extract detailed information from the source code files. This algorithm is an improved form of existing algorithm by Chawathe et al. This algorithm extracts changes by finding both a match between the nodes of compared two abstract syntax trees and a minimum edit script that can transform one tree into the other given the computed matching. This results in identification of fine-grained change types between program versions according to taxonomy of source code changes. According to authors, tree diff algorithm approximates the minimum edit script 45 percent better than the original extraction approach by Chawathe et al.

Old techniques and tools are valuable but suffer from the low quality of information available for changes. For example, the information, in particular for source code, is stored by versioning systems (CVS or Subversion). They keep track of changes by storing the text line added and/or deleted from a particular file. Structural changes in the source code are not considered at all. Tree diff algorithm's main focus is on structural changes in source code. Many approaches are able to narrow down changes to the method level, but fail in further qualifying changes, such as the addition of method invocation in the else branch of an if-statement. Furthermore, a classification of changes according to their impact on other source code entities is missing, which is very important to improve the quality of software evaluation results. Source code can be represented as abstract syntax trees (ASTs) tree differencing can be used to extract detailed change information. This approach is useful because exact information on each entity and statement is available in an AST (Beat, 2007).

Tree diff algorithm uses the bigram string similarity to match source code statements (such as method invocation, condition statements, and so forth) and the subtree similarity to match source code structures (such as if statement or loops). In order to improve the matching, tree diff has used a best match algorithm for all leaf nodes and inner node similarity weighting. To overcome mismatch propagation in small subtrees, dynamic thresholds are used for subtree similarity (Beat, 2007).

The identification of changes occur in different versions of a program is one of the main issue in software evolution analysis. Change in software is a vital part of software’s development life cycle. Now days there are lot of tools and techniques has been developed to aid software engineers in maintaining and evolving huge complicated software systems. Mostly the information for source code is stored by different types of versioning systems such as, Subversion or CVS. These systems do not consider structural changes in the source code. They mainly deal with storing the text lines added or/and deleted from a specific file. However source code can be represented as abstract syntax trees (ASTs). To extract detailed change information tree differencing can be used (Beat, 2007).

4.5.2.1 Shortcomings of Tree diff Algorithm

This algorithm is quite near to our idea of differencing. We are also focused to identify structural changes in the source codes files. As Tree diff algorithm is based on
Chawathee et al., as our DefectoFix algorithm is the next step of Tree diff algorithm. The shortcomings to our best knowledge about Tree diff algorithm are as follows:

- The algorithm is still limited in finding the appropriate number of operations. This limitation is also mentioned by the authors of Tree diff.
- The best match approach may match reoccurring statements that are not at the same position in the method body.
- The declaration changes, in particular the parameter ordering changes, are also an implication of the small tree problem.
- This technique is not suitable for object-oriented programming concepts.

Authors have tried to overcome these shortcomings in their proposed differencing algorithm. There is more detailed information being stored in it. Further MySQL repository is used in order to store differences. Authors selected MySQL as it is open source and very robust for validation processes.

4.5.3 JDiff Algorithm

JDiff is a technique for comparing object-oriented programs that identifies both differences and correspondences between two versions of a program. The technique is based on a representation that handles object-oriented features. Earlier techniques are limited in their ability to detect changes in programs because they provide purely textual differences and do not consider changes in program behavior indirectly caused by textual modifications (Apiwattanapong, 2007).

JDiff is basically extending an existing differencing algorithm. JDiff algorithm takes two versions of the files as input i.e. faulty version and correct version. Two other parameters are also taken as inputs which are used for node level matching. One parameter is maximum lookahead, which is used during the attempt of matching nodes in methods. Other parameter is used to determine the similarity of two hammocks (Apiwattanapong, 2007).

At completion, the algorithm outputs a set of pairs in which the first element is a pair of nodes and the second element is the status—either “modified” or “unchanged.” The algorithm also returns sets of pairs of matching classes (C), interfaces (I), and methods (M) in both original and modified program. JDiff algorithm performs its comparison first at the class and interface levels, then at the method level, and finally at the node level (Apiwattanapong, 2007).

JDiff algorithm begins its comparison at the class and interface levels. The algorithm matches classes that have the same fully-qualified name. JDiff counts the possibility of interacting with the user while matching classes and interfaces to improve the differencing results. After matching classes and interfaces, JDiff compares, for each pair of matched classes or interfaces, their methods. The algorithm first matches each method in a class or interface with the method with the same signature in another class or interface. Then, if there are unmatched methods, the algorithm looks for a match based only on the name (Apiwattanapong, 2007).

JDiff uses the set of matched method pairs to perform matching at the node level. First, the algorithm considers each pair of matched methods and builds ECFGs. JDiff uses hammocks and hammock graphs for its comparison of two methods. Hammocks are single-
entry, single-exit subgraphs and provide a way to impose a hierarchical structure on the ECFGs that facilitates the matching. JDiff also defines hammock matching algorithm, which is based on the earlier diff algorithm proposed by Laski and Szermer, which transform two graphs into their respective isomorphic graphs (Apiwattanapong, 2007).
5 **DefectoFix Differencing Algorithm**

DefectoFix algorithm is a combination of three approaches node comparison, file comparison and fault detection in behavior comparisons. These approaches used in are extracted from Apiwattanapong, 2007; Beat, 2007; Thinkage, 2000. DefectoFix algorithm takes as input the faulty version of source code (R’) and corrected version of source code (R). At the completion, the algorithm outputs a set of pairs (P) in which the first element is a pair of nodes (N) of faulty sub-tree and the second element is the status, either “modified” or “unchanged”. The algorithm also returns set of pairs of matching classes (rC) and methods (rM) from R and R’. In the end, DefectoFix stores rC, rM and rN in its repository for future comparison.

The DefectoFix algorithm takes faulty source code and correct source code to break them into different code fragments based on classes. If there is no class used in particular source code file, it will be treated as a file with single code fragment. The DefectoFix algorithm performs its comparison starting from class level, then method level and in the end at node level (if needed). The algorithm first compares each class in with the like-named class in R’ and produces a set of class pairs (C). For each pair of classes, DefectoFix then matches methods in the class in R with methods having, the same signature in the class in R’. The result is a sub-tree where fault/change exists. This sub-tree along with additional information (Defect Fix Model Id, Project Named, File Name, Class Name, File Revision Number and Sub Tree Level) makes Defect Fix Model (DFM) which is later on stored in repository for future use like fault injection process.

5.1 **Algorithm**

The algorithm “DefectoFix” finds the defects from the source code after comparing corrected source code file with faulty source code file. Input and output are as follows:

**Input:**
- Faulty source code file (R)
- Corrected source code file (R’)
- Maximum Lookahead (LH)

(Both R and R’ will be retrieved from VCS repository, R is 1.0 version where as R’ is 1.1 version)

**Output:**
- Set of <class, class> rC
- Set of <method, method> rM
- Set of <level, level> rL
- Set of <<node, node>, status> rN
- Diff information of all defects found i.e. faulty and corrected statements
- Storing of rC, rM, rL, rN with additional information in DefectoFix repository for future comparison

**Declare:**
- Node n, n´
- Stack ST, ST´

**Begin:**
- DefectoFix

**Step 1: Diff R and R’**
Checking both files has the same source code or not
   if Diff value equal to True then
       Exit
   else move to Step 2
   end if

Step 2: Compare most similar classes in R and R’, add matched class pairs to C

   for each pair <c, c’> in C do
     create AST, T and T’ for classes c and c’
     extraction of nodes having same parents
     push(ST, node), push(ST’, node’)
     compare methods; add matched method pairs to M
     N = N’ U NodeMatch(n, n’, LH, ST, ST’)
   end for

   if (CompRep(C, M, N)) //Check whether same defect already exists?
     print “Fault already exists”
     Exit
   else
     dbReg(C, M, N, N’) //Register defect in repository
   end if

   Exit

End: DefectoFix

5.1.1 Compare Repository Algorithm

The algorithm “Compare Repository” will compare classes, methods, nodes and defect statements (faulty and corrected) from the DefectoFix repository.

Declare: C represents class
   M represents Method
   N represent Nodes
   pn represent parent node of faulty class
   pn’ represents parent node of correct class

Begin: Compare Repository

Boolean CompRep(C, M, N)

Compare C, M and N with DefectoFix repository

   for each pair <c, c’> in C do
     for each pair <pn, pn’> in same parent do
       compare <pn, pn’> with DefectoFix repository with previous versions
       if <pn, pn’> Exists then
         return True
       Exit
       else
         return false
       end if
     end for
end for

5.1.2 Register DFM Algorithm

The algorithm "Register DFM" will save the values of class C, method M, faulty nodes N and correct nodes N' along with additional information in DefectoFix repository.

Declare:  
C represents class  
M represents method  
N represent faulty nodes  
N' represents correct nodes  
AI represents additional information

Begin:  
Register DFM

dbReg(C, M, N, N', AI)  
save in DefectoFix repository  
End

5.2 DefectoFix Explanation

Consider the faulty source code file as R and the corrected source code file as R'. If we have to inspect the output of DefectoFix, then run it on R and R'. Below is the step by step explanation of DefectoFix algorithm parts.

5.2.1 Step 1: (diff of both files)

DefectoFix algorithm, at first step, takes both files R and R' as an input (these files can also be checked out from VCS repository). In this step, DefectoFix will compare both source code files (faulty and correct) using diff utility. If the diff value is “True”, then it means, both files have the same source code and there is no difference in them. So at this point, DefectoFix algorithm will exit from next steps. If Diff value is not “True”, then DefectoFix algorithm will move to step 2.

5.2.2 Step 2

In step 2, code fragments of both faulty and correct source code files are generated. These fragments are generated on the basis of keyword “class”. In this way, all the classes in the source code files (faulty and correct) are separated in the form of fragments. If there is no class in the source code (faulty and correct), then the source code in that file will be treated as a single code fragment. After separating these source codes files in class fragments, Abstract Syntax Tree (AST) of all fragments are generated. AST are generated to get tree structure of the source code file. It helps in node to node comparison of source code files and also extracts detailed information about particular node from the source code files.

5.2.2.1 Class Level Matching

DefectoFix begins its comparison at the class levels. It matches classes that have the most similar names. Matching classes in R and R' are added to C. Classes in R that do not appear in R' are “deleted classes”, whereas classes in R’ that do not appear in R are “added classes”.

38
5.2.2.2 Method Level Matching

After matching classes, DefectoFix compares, for each pair of matched classes, their methods. The algorithm first matches each method in a class with the method with the same signature in another class. Then, if there are unmatched methods, the algorithm looks for a match based only on the name. This matching accounts for cases in which parameters are added to (or removed from) an existing method, and increases the number of matches at the node level. Pairs of matching methods are added to M. Like the approach used for classes, methods in R that do not appear in set M are “deleted” methods, whereas methods in R’ that do not appear in set M are “added” methods.

5.2.2.3 Node Level Matching

DefectoFix uses the set of matched levels to perform matching at the node level. First, the algorithm considers each pair of matched methods (l, l’) in L. Next, it calls procedure NodeMatch(), passing n, n’, LH, and S as parameters. NodeMatch() identifies differences and correspondences between nodes in T and T’, and returns N, the set of matched nodes and corresponding labels (“modified” or “unchanged”). Finally, DefectoFix returns C, M, L and N. In the next section, we discuss the AST, the representation that we use to perform node matching and how we process them. Finally, we present and explain our node-matching algorithm, NodeMatch().

5.2.2.3.1 AST Graphs

When comparing two methods m and m’, the goal of our algorithm is to find, for each statement in m, a matching (or corresponding) statement in m’, based on the method structure. Thus, the algorithm requires a modeling of the two methods that (1) explicitly represents their structure, and (2) contains sufficient information to identify differences and similarities between them. Although Control Flow Graphs (CFGs) can be used to represent these control structure of methods but they do not suitable to model many object oriented constructs. For this purpose, we have used AST graphs. This approach is promising because exact information on each entity and statement is available in an AST.

5.2.2.3.2 Node Match Algorithm

NodeMatch() takes an input n and n’, LH, indicating the maximum Lookahead, and S, a threshold for deciding whether two nodes are similar enough to be considered a match. The algorithm outputs the detail information about the defect found in faulty source code, having information like faulty statement, correct statement and diff value etc. The obtained nodes with their minimal sub-tree and additional information are stored with label “modified” in the DefectoFix repository.

The algorithm starts matching nodes in the AST graphs by performing a Depth-First pair wise traversal of T and T’. A Depth-First traversal of a tree always starts at the root of the tree. Thus, in start of algorithm, a pair of start nodes is added to stack ST, which will use as a work-list. Main while loop (NodeMatch algorithm is described below) extracts one node pair in all iteration from the stack and checks whether the two nodes match. The body of the loop first checks whether any node of the loop in the current pair is already matched. The algorithm takes care that an already matched node must not be considered again. In this case, the algorithm continues by considering the next pair in the work-list.
To compare two nodes, NodeMatch algorithm uses CompNode(c, c', N) procedure where c and c' are the two nodes to compare. N is the set of matching nodes. If the two nodes c and c' are matched (i.e. CompNode returns “True”), they are stored in a variable “match” as a pair and later added to the set of matched nodes with label “unchanged”. Otherwise, NodeMatch tries to find a match for c by examining c’s descendent up to specified maximum lookahead.

At first “match” is initialized to “null”, and lookahead sets L and L' are initialized to contain only the current nodes. The algorithm then executes for loop until a match is found or depth d reaches the maximum lookahead LH. When the algorithm terminates, all nodes in the faulty version that are not in any pair (i.e. they have not been matched to any other node) are considered “deleted” node. Similarly all nodes in the corrected version that are not in any pair are considered “added nodes”.

**Procedure:** NodeMatch (C, M, N, LH, ST, ST')

**Input:**
- Nodes in faulty version n having same parent
- Node in corrected version n' having same parent
- Maximum lookahead LH

**Output:**
- Set of pairs ((node, node'), label) contains faults

**Declare:**
- Current nodes c and c'
- Lookahead nodes L and L'
- num,j,k are local variables
- Pair(node, node') match

**Begin:**

```plaintext
If ST.size >= ST'.size then
  for num=1 to ST.size by 1
    If ST.size == ST'.size
      p=pop(ST), p'=pop(ST')
      If p != p' then
        Do store(P, P', "modified")
        Set k=k+2
      End if
    Else
      p=pop(ST), p'=pop(ST')
      If p != p' then
        Do store(P, P', "modified")
        Set k=k+2
      Else if
        Set j=j+1 //Correct Lookahead (LH)
      End if
    End if
  End for
Else
  for num=1 to ST'.size by 1
    p=pop(ST), p'=pop(ST')
    If p != p' then
      Do store(P, P', "modified")
      Set k=k+2
    Else if
      Set j=j+1 //Faulty Lookahead (LH)
    End if
End if
```

---

40
5.2.3 Step 3 (Comparison in DefectoFix Repository)

This comparison of existing DFM from repository is useful for various applications of DefectoFix framework. Fault injection process is one of a usage scenario for DefectoFix framework, where faults are injected in correct source code files by DefectoFix repository. For comparison with DefectoFix repository, it begins from class level. The algorithm compares classes in the DefectoFix repository. If the names of classes are same then next step will perform, otherwise (class name does not match), class name would be added into the repository as rC. After class comparison the DefectoFix starts comparing method for each pair of matched classes. The algorithm compares method with the stored methods in the DefectoFix repository. If the name of class and method are matched with the repository class (rC) and method (rM) then it will start comparing at node level otherwise it will store the class and method in repository. DefectoFix uses the set of matched method pairs (M) to perform comparison at the node level. DefectoFix is taking variables and statements as a Node. First, the algorithm considers each pair of matched M and compares every node. The node pair values (N) are compared with the stored node rN in the DefectoFix repository. In this way, all Node pair values are extracted and compare with the node rN.

5.3 DefectoFix Algorithm Explanation by Examples

In order to fully understand the proposed differencing algorithm and DFM, we have explained different scenarios which are discussed in the following sections.

5.3.1 Example No. 1

In this example, a fault is produced by changing the spelling of local variable. The lengths of both source code files are equal. Both source code files have one method. Consider R is the faulty source code and \( R' \) is a correct source code as shown in Table 5.1.

<table>
<thead>
<tr>
<th>Faulty Source Code (R)</th>
<th>Correct Source Code (R')</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. class C</td>
<td>1. class C</td>
</tr>
<tr>
<td>2. def m(a1, a2)</td>
<td>2. def m(a1, a2)</td>
</tr>
<tr>
<td>3. sum = a1 + arg2</td>
<td>3. sum = a1 + arg2</td>
</tr>
<tr>
<td>4. return a2 / sum</td>
<td>4. return a2 / sum</td>
</tr>
<tr>
<td>5. end</td>
<td>5. end</td>
</tr>
<tr>
<td>6. end</td>
<td>6. end</td>
</tr>
</tbody>
</table>

5.3.1.1 Code Fragmentation

DefectoFix algorithm will generate the code fragments as given below.

<table>
<thead>
<tr>
<th>Faulty Source Code Fragments</th>
<th>Correct Source Code Fragment</th>
</tr>
</thead>
</table>
5.3.1.2 Abstract Syntax Tree Generation

The generated Abstract Syntax Tree (AST) for each code fragment is shown in Table 5.2.

<table>
<thead>
<tr>
<th>Faulty Source Code</th>
<th>Correct Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>s[:class, :C, nil, s[:scope, s[:defn, :m, s[:scope, s[:block, s[:args, :a1, :a2], s[:lasgn, :sum, s[:call, s[lvar, :a1], +, s[array, s[:call, :arg2)]], s[:return, s[:call, s[lvar, :a2], /, s[array, s[lvar, :sum]]]]]]]]</td>
<td>s[:class, :C, nil, s[:scope, s[:defn, m, s[:scope, s[:block, s[:args, :a1, :a2], s[:lasgn, :sum, s[:call, s[lvar, :a1], +, s[array, s[:call, :arg2]]], s[:return, s[:call, s[lvar, :a2], /, s[array, s[lvar, :sum]]]]]]]</td>
</tr>
</tbody>
</table>

5.3.1.3 AST in Tree (Graph) Format

Figure 5.1 shows the graph for AST of code fragments for node wise information.

5.3.1.4 Applying NodeMatch()

DefectoFix algorithm calls NodeMatch() procedure for level and node comparison of both faulty and correct source codes. We assume that increment in lookahead (LH) is 1. In the first iteration over the main loop, the algorithm extracts tree level 1 from stack ST and tree level 10 from ST' as can be seen in Figure 5.1. In the second iteration of loop tree level 2 would be popped from the stack ST and tree level 11 would be popped from the stack ST'. This process continues until we reached to the last level of stack ST or ST' depending on the greater in size stack. The algorithm compares the two levels by calling NodeMatch(1, 10, LH), which compares the level labels and return “True” if they are similar, else “False”.

42
DefectoFix algorithm compares faulty sub-tree level with correct code sub-tree level and if the sub-tree levels are not equal then DefectoFix algorithm will save both sub-tree levels along with respective sub-tree nodes in the DefectoFix repository. DefectoFix algorithm returns set of classes, methods, sub-tree levels, nodes and additional information with their respective status. The DFM extracted in this process is given below. It will be stored in DefectoFix repository.

<table>
<thead>
<tr>
<th>Defect Fixed Model Id</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Name</td>
<td>Example 1</td>
</tr>
<tr>
<td>File Name</td>
<td>faulty.rb &amp; correct.rb</td>
</tr>
<tr>
<td>Class Name</td>
<td>C</td>
</tr>
<tr>
<td>Defect Fix Type</td>
<td>Misspelled Argument/Variable</td>
</tr>
<tr>
<td>Defect Fix Message</td>
<td>In an old argument rename (from argX to aX form of arguments) we had missed converting all argument references to the new format.</td>
</tr>
<tr>
<td>File Revision Number</td>
<td>01</td>
</tr>
<tr>
<td>Sub Tree Level</td>
<td>9, 18</td>
</tr>
<tr>
<td>Faulty AST</td>
<td>s(array, s(vcall,:arg2))</td>
</tr>
<tr>
<td>Correct AST</td>
<td>s(array, s(lvar,:a2))</td>
</tr>
<tr>
<td>Diff Model</td>
<td>s(array, s(vcall,:arg2)) &gt;&gt; s(array, s(lvar,:a2))</td>
</tr>
</tbody>
</table>

5.3.2 Example No. 2

In this example, multiple faults are exists in faulty source code files. There are two different types of faults, first one is the change of data structure of a variable and second one is about missing operation “type conversion of variable”. The lengths of both source code files are not equal. Both source code files have same number of methods (i.e. two methods). Consider R is the faulty source code and R’ is a correct source code as shown in Table 5.7.

<table>
<thead>
<tr>
<th>Faulty Source Code (R)</th>
<th>Correct Source Code (R’)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. class C</td>
<td>1. class C</td>
</tr>
<tr>
<td>2. data=10</td>
<td>2. @@data=10</td>
</tr>
<tr>
<td>3. def m()</td>
<td>3. def m()</td>
</tr>
<tr>
<td>4. data1=data</td>
<td>4. data1=@@data</td>
</tr>
<tr>
<td>5. m2(data1)</td>
<td>5. m2(data1)</td>
</tr>
<tr>
<td>6. end</td>
<td>6. end</td>
</tr>
<tr>
<td>7. def m2(data)</td>
<td>7. def m2(data)</td>
</tr>
<tr>
<td>8. result=10+data2</td>
<td>8. data2=data.to_i()</td>
</tr>
<tr>
<td>9. result=result + data</td>
<td>9. result=10+data2</td>
</tr>
<tr>
<td>10. puts result</td>
<td>10. result=result + @@data.to_i()</td>
</tr>
<tr>
<td>11. end</td>
<td>11. end</td>
</tr>
<tr>
<td>12. end</td>
<td>12. end</td>
</tr>
</tbody>
</table>

5.3.2.1 Code Fragmentation

DefectoFix algorithm will generate the code fragments as given below.

<table>
<thead>
<tr>
<th>Faulty Source Code Fragment</th>
<th>Correct Source Code Fragment</th>
</tr>
</thead>
</table>

43
5.3.2.2 Abstract Syntax Tree (AST) Generation

The generated Abstract Syntax Tree (AST) for each code fragment is shown in Table 5.8.

<table>
<thead>
<tr>
<th>Faulty Source Code (R)</th>
<th>Correct Source Code (R')</th>
</tr>
</thead>
<tbody>
<tr>
<td>s(:class, :C, nil, s(:scope, s(:block, s(:lasgn, :data, s(:lit, 10)), s(:defn, :m, s(:scope, s(:block, s(:args), s(:lasgn, :data1, s(:vcall, :data)), s(:fcall, :m2, s(:array, s(:lvar, :data1)))), s(:defn, :m2, s(:scope, s(:block, s(:args, :data), s(:lasgn, :result, s(:call, s(:lit, 10)), +, s(:array, s(:lvar, :result))), s(:call, s(:array, s(:lvar, :data)))), s(:fcall, :puts, s(:array, s(:lvar, :result)))))), s(:lasgn, :result, s(:call, s(:array, s(:lvar, :data2))), s(:lasgn, :result, s(:call, s(:array, s(:lvar, :data)))), s(:fcall, :puts, s(:array, s(:lvar, :result))))))</td>
<td>s(:class, :C, nil, s(:scope, s(:block, s(:cvdecl, :data, s(:lit, 10))), s(:defn, :m, s(:scope, s(:block, s(:args), s(:lasgn, :data1, s(:vvar, :data)), s(:fcall, :m2, s(:array, s(:lvar, :data1)))), s(:defn, :m2, s(:scope, s(:block, s(:args, :data), s(:lasgn, :result, s(:call, s(:lit, 10)), +, s(:array, s(:lvar, :result))), +, s(:array, s(:lvar, :data)))), s(:fcall, :puts, s(:array, s(:lvar, :result))))))</td>
</tr>
</tbody>
</table>

5.3.2.3 AST in Tree (Graph) Format

Figure 5.5 and 5.6 shows the AST representation of both faulty and correct source code in tree format.
5.3.2.4 Applying NodeMatch()

The NodeMatch() procedure will follow same process as explained in previous example (Section 5.3.1.4). The Levels which will be compared during this example are shown in Figure 5.5 and 5.6.

The DFM extracted in this process is given below. It will be stored in DefectoFix repository.

<table>
<thead>
<tr>
<th>Defect Fixed Model Id</th>
<th>Project Name</th>
<th>File Name</th>
<th>Class Name</th>
<th>Defect Fix Type</th>
<th>Defect Fix Message</th>
<th>File Revision Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>002</td>
<td>Example 2</td>
<td>faulty.rb &amp; correct.rb</td>
<td>C</td>
<td>Wrong data structure used, missing one operation</td>
<td>We have missed to convert variable in required data structure and data.</td>
<td>01</td>
</tr>
</tbody>
</table>
5.3.3 Example No. 3

In this example, both faulty and correct source code files contain two classes. Therefore, code fragmentation procedure will be in action. There are two simple faults are produced by changing the spelling of local variable in both classes. Consider R is the faulty Ruby source code and R’ is a corrected Ruby source code. Consider R is the faulty source code and R’ is a correct source code as shown in Table 5.9.

<table>
<thead>
<tr>
<th>Faulty Source Code (R)</th>
<th>Correct Source Code (R’)</th>
</tr>
</thead>
<tbody>
<tr>
<td>class C</td>
<td>class C</td>
</tr>
<tr>
<td>def m(a1, a2)</td>
<td>def m(a1, a2)</td>
</tr>
<tr>
<td>sum=0</td>
<td>sum = a1 + arg2</td>
</tr>
<tr>
<td>sum = a1 + arg2</td>
<td>return a2 / sum</td>
</tr>
<tr>
<td>return a2 / sum</td>
<td>end</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>8.</td>
</tr>
<tr>
<td></td>
<td>class D</td>
</tr>
<tr>
<td></td>
<td>def m(a1, a2)</td>
</tr>
<tr>
<td></td>
<td>sum = a1 + [b2]</td>
</tr>
<tr>
<td></td>
<td>return a2 / sum</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>11. end</td>
</tr>
</tbody>
</table>

5.3.3.1 Code Fragmentation

DefectoFix algorithm will generate the code fragments for both classes (C and D) as given below.

Faulty Source Code Fragment 1

```ruby
class C
def m(a1, a2)
  sum=0
  sum = a1 + arg2
  return a2 / sum
end
end
```

Correct Source Code Fragment 1

```ruby
class C
def m(a1, a2)
  sum = a1 + arg2
  return a2 / sum
end
```

Faulty Source Code Fragment 2

```ruby
class D
def m(a1, a2)
  sum = a1 + b2
  return a2 / sum
end
end
```

Correct Source Code Fragment 2

```ruby
class D
def m(a1, a2)
  sum = a1 + arg2
  return a2 / sum
end
end
```
5.3.3.2 **Abstract Syntax Tree (AST) Generation**

The generated Abstract Syntax Tree (AST) for each code fragment is shown in Table 5.10.

Table 5.6 Abstract Syntax Tree of the code

<table>
<thead>
<tr>
<th>Faulty Source Code (R)</th>
<th>Correct Source Code (R')</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class C</strong> s(:class, :C, nil, s(:scope, s(:defn, :m, s(:scope, s(:block, s(:args, :a1, :a2), s(:lasgn, :sum, s(:lit, 0)), s(:lasgn, :sum, s(:call, s(:ivar, :a1), :+:, s(:array, s(:vcall, :arg2)))), s(:return, s(:call, s(:ivar, :a2), :/, s(:array, s(:ivar, :sum)))))))))</td>
<td><strong>Class C</strong> s(:class, :C, nil, s(:scope, s(:defn, :m, s(:scope, s(:block, s(:args, :a1, :a2), s(:lasgn, :sum, s(:call, s(:ivar, :a1), :+:, s(:array, s(:vcall, :arg2)))), s(:return, s(:call, s(:ivar, :a2), :/, s(:array, s(:ivar, :sum)))))))))</td>
</tr>
</tbody>
</table>

| **Class D** s(:class, :D, nil, s(:scope, s(:defn, :m, s(:scope, s(:block, s(:args, :a1, :a2), s(:lasgn, :sum, s(:call, s(:ivar, :a1), :+:, s(:array, s(:vcall, :b2)))), s(:return, s(:call, s(:ivar, :a2), :/, s(:array, s(:ivar, :sum)))))))) | **Class D** s(:class, :D, nil, s(:scope, s(:defn, :m, s(:scope, s(:block, s(:args, :a1, :a2), s(:lasgn, :sum, s(:call, s(:ivar, :a1), :+:, s(:array, s(:vcall, :arg2)))), s(:return, s(:call, s(:ivar, :a2), :/, s(:array, s(:ivar, :sum)))))))) |

5.3.3.3 **AST in Tree (Graph) Format**

Figure 5.7 shows AST (S-Expression) representation in Tree format of Class “C” of faulty and correct source code.

Figure 5.8 shows AST (S-Expression) representation in Tree format of Class “D” of faulty and correct source code.
5.3.3.4 Applying NodeMatch()

The NodeMatch() procedure will follow the same process as explained in previous example (Section 5.3.1.4) for both code fragments (Class C and Class D). The Levels which will be compared during this example are shown in Figure 5.7 and 5.8.

The DFM extracted in this process is given below. It will be stored in DefectoFix repository.

---

**Defect Fixed Model Id** 003

**Project Name** Example 3

**File Name** faulty.rb & correct.rb

**Class Name** C, D

**Defect Fix Type** Wrong data structure used

**Defect Fix Message** We have missed to convert variable in required data structure and data structure of variable “data” is not valid.

**File Revision Number** 01

**Sub Tree Level** 9, 18

**Faulty AST**

s(array, s(vcall, :arg2))

**Correct AST**

s(array, s(lvar, :a2))

**Diff Model**

s(lasgn, :sum, s(lit, 0)), s(lasgn, :sum, s(call, s(lvar, :a1), :+:, s(array, s(vcall, :arg2)))), s(return, s(call, s(lvar, :a2), /=, s(array, s(lvar, :sum))))

```
>>> s(lasgn, :sum, s(call, s(lvar, :a1), :+:, s(array, s(vcall, :arg2)))), s(return, s(call, s(lvar, :a2), /=, s(array, s(lvar, :sum))))

s(array, s(vcall, :arg2))
```

---

**Figure 5.5 AST (S-Expression) Tree of Class “D” of faulty and correct source code**
6 DEFECTOFIX SYSTEM DESIGN

This chapter describes the system design (working) of DefectoFix tool in detail. The major parts of DefectoFix system are listed below:

- Extraction of faulty and correct source code files from VCS repository
- Fragmentation of files on the basis of classes.
- Generation of Abstract Syntax Tree (AST)
- Extracting tree levels from AST and converting them in Array format
- NodeMatch() for level and nodes comparison
- Storing DFM in DefectoFix repository
- Other DefectoFix repository database operations such as View, Edit & Delete etc.

Section 6.1 explains the working of DefectoFix. Fragmentation of source code files is presented in Section 6.2. Section 6.3 describes the Abstract Syntax Tree (AST) Generation. The extraction of Tree levels from AST is explained in Section 6.4. Section 6.5 describes the process of converting Tree levels in arrays using make_array() procedure. Section 6.6 describes the node matching of levels and nodes using NodeMatch() procedure. Defect registration process is explained in Section 6.7. Section 6.8 describes the validation process of DefectoFix through repository. View/Edit/Delete Defects from Defectofix repository is describe in Section 6.9.

6.1 Overview of DefectoFix Working

Figure 6.1 displays the overall working structure of the DefectoFix. Both faulty and correct source code files are input to the DefectoFix. There are three possibilities for this input. First, these input files (faulty and correct source code files) can be attached by the user using DefectoFix GUI interface. Secondly it can be extracted from Version Control System (VCS like SVN etc.) directly. At this stage connection to SVN is not established, therefore it is not shown in the Figure 6.1. Thirdly, by “Fault Injection” process. In this process, correct file will extract from SVN repository or by user and faulty file will be made by injecting some fault in the selected correct source code file (by user or from SVN). The third possibility is used to validate the basic working of DefectoFix and the usability of the extracted models.

![Figure 6.1 DefectoFix System Design](image)

Both faulty and correct source code files are passed to the process of code fragmentation. In this process, both files are divided further into sub-files based on their
“class” keyword. This will help in the node comparison method. Ruby parser will parse the generated code fragments and forms the Abstract Syntax Tree (AST) for each code fragment, which becomes input to NodeMatch() procedure. The NodeMatch() procedure is then used to find out faults by comparing every node of faulty and correct source code files. The below section describes each part of DefectoFix in detail.

6.2 Fragmentation of Source Code Files

It is the initial process of DefectoFix tool. A code fragment can be composed of structures such as code block, individual statements, expressions, and mixture of these structures. To find differences in the source code files, user will input both faulty and correct source code files as shown in Figure 6.2. Both files will be extracted from the SVN repository (future work) or input by the user. code_fragment() procedure is used to attain fragments of the source codes. Code fragmentation is done on the basis of the keyword “class” i.e. for every class, it will make separate fragment. After fragmentation, they will be saved in separate file with prefix “fragment_”.

6.3 Abstract Syntax Tree (AST) Generation

The Abstract Syntax Tree (AST) of the source code files is generated from the fragmented source code file. Ruby Parser is used to generate AST of fragmented code. DefectoFix tool stores AST in the form of S-Expression. S-Expression contains detail description of every node which will be used in next steps. sexp() procedure will be used to generate AST as shown in Figure 6.3. AST is used to extract detailed information of every node in the source code. Therefore detail explanation about the fault of particular node is stored in DefectoFix repository.

---

Figure 6.2 Fragmentation of source codes
6.4 Extraction of Tree Levels from AST

The extraction of sub-tree levels from AST of the source code will be performed by treelevel() procedure as shown in Figure 6.4. Extraction of sub-tree level is helpful to find differences in particular node so that full sub-tree can be stored. The full-sub tree is used for complete explanation of the fault in the source code. Ruby regular expressions are used for extraction of tree levels. The extracted tree levels are stored in separate files with prefix “ast_”.

6.5 Converting Tree Levels in Arrays

In the next step of DefectoFix, nodes are required in the form of arrays for tree levels. It helps DefectoFix tool to perform comparison between, class level, method level, tree level and node level respectively. make_array() procedure will be used for this purpose. The NodeMatch() algorithm has been developed to perform the actual comparison between these arrays.

6.6 Node Matching of Levels and Nodes

The essential requirements for finding differences in a source code, is carried out in this step. NodeMatch() takes array as a parameter as shown in Figure 6.6. DefectoFix will perform comparison between faulty and correct source code files, if difference is found then it will be saved with additional information in DefectoFix repository, else it will exit. There are three variations of NodeMatch() procedure, each one is triggered on the true value of particular condition. The first variation occurs when the length of faulty and correct source
code is same. The second variation works when the length of faulty source code is less than correct source code. The last variation triggers when the length of the faulty source code is greater than correct source code.

Figure 6.6 Applying NodeMatch & Registering Defect

### 6.7 DFM Registration

User interface is provided for DFM registration process. User is facilitated to add additional information for a particular defect/difference. This information is stored in DefectoFix repository as DFM and used for different purposes like fault injection. The DFM registration form consists of different fields such as DFMid, Project Name, Class Name, File Name, File Revision Number, Sub Tree Level, Faulty Statements, Correct Statements and Diff Model. The DFM is the actual model and real findings of DefectoFix tool. DFM registration form is connected to MySQL database and stores the information in DefectoFix repository.

### 6.8 View/Edit/Delete Defects from DefectoFix Repository

These are the basic features of any database repository. These features enable users to view, edit or delete existing Defects in the DefectoFix repository.
7 Evaluation of DefectoFix

During our evaluation, we come across different problems such as the problem of locating suitable case study subjects and finding multiple versions of the subject. Often Open Source Software (OSS) are readily accessible in multiple versions. But most of OSS are not very well equipped with code-level description of defect fixes that makes evaluation easier if VCS check-in consists only of a single bug fix with a comment on what it was. The code-level description of defect fixes is necessary in order to finding the exact differences in the source code files. DefectoFix may not be robust enough to operate on the selected subjects. Some times, the time required to ensure adequate robustness may be prohibitive. To qualitatively evaluating the proposed framework, we have implemented a prototype using Ruby language called DefectoFix. Broadly stated, our goal for this case study was to obtain meaningful information about the effectiveness of the proposed model for finding differences in source code files and later then connecting it to the bug and fix reports. An important goal was to carry out the evaluation on real-world problems. The reminder of this section describes our case study evaluation of the DefectoFix.

This chapter is divided into following subsections. The tools which are used in the development of DefectoFix are given in Section 7.1. Section 7.2 contains description about case study. Description about the performed procedure for the evaluation of DefectoFix is presented in Section 7.3. Results and results analysis are given in Section 7.4 and 7.5 respectively.

7.1 Tools

The implemented tool, “DefectoFix” uses static analysis to extract code properties. A parser is required for static analysis. As there are so many parsers (basically for any programming language) available out there, the only important thing is the selection of programming language. The main criteria for the selection of programming language is object oriented support because the earlier differencing algorithms lack the object oriented support, or if support is available, it is very limited. Based on these reasons, we have selected the Ruby programming language. Because Ruby is an open source programming language therefore it will be easy to find a object oriented subject for the case study. Another aim of this thesis is to explore the new emerging object oriented programming language. That’s why Ruby seemed to be a better candidate for DefectoFix implementation and preferred over Java programming language.

There are multiple options available in Ruby. The tool can either be developed for web technologies using Ruby on Rails (RoR) or desktop technologies using wxRuby and fxRuby. But we have decided to build prototype of our framework using desktop version of Ruby i.e. wxRuby. The main aim of this thesis is to verify the working of proposed framework. There are many reasons for the selection of wxRuby. wxRuby provides good support for the building of prototypes as compared to other versions of Ruby. Another reason is that there is a lot of online help available on wxRuby as compared to fxRuby or other flavours of Ruby. wxRuby also gives native look to the applications, which is not available in fxRuby. It is clear from the above discussion that wxRuby is the best available option for the building of DefectoFix prototype.
As we already mention in previous section that we needed a parser because our model is based on the comparison of Abstract Syntax Tree (AST). Therefore, we needed a parser to construct the AST from the source code files. And for that purpose, we used Ruby Parser as it is open source and freely available. In our framework, we also needed a repository to store the extracted differences and their additional information for future usage. MySQL is selected for the role of DefectoFix repository. MySQL is also open source and freely available and we have selected it due to its robustness and open source nature.

It is worth mentioning that the proposed model (source code differencing algorithm) is easily applicable in any other programming language as long as there is a parser available for it. However, the intuition says that programming languages must be similar to the object oriented language and it might work only on limited type of differences. An exploration about the model analysis in different programming language other than Ruby is out of scope of this thesis.

There are four major components of DefectoFix prototype. These components are GUI component, logical component, database component and utility component. GUI component is consisted of wxRuby classes which are used to build the graphical environment of the tool. Logical component is comprises of classes, which perform basic logical functionality of the tool. These classes are FindBug, RegisterDFM, EditDFM & DeleteDFM. The core procedures of DefectoFix can be found in FindBug.rb. These procedures are node_match(), code_fragment(), sexp(), find_diff(), make_array(), put_line_number(), tree_split and file_exist(). RegisterDFM, EditDFM and DeleteDFM classes contains procedure which are providing interaction with DefectoFix repository. Database component consisted of MySQL and its operations. Ruby parser and fileutils are the part of utility components. Both faulty and correct files are manually linked with DefectoFix.

7.2 Case Study Subjects

The choice of case study subjects are extremely important while evaluating the new model for finding differences in source code file and connecting them to bug and fix reports. While implementing the DefectoFix, we have used medium sized source codes written in Ruby programming language. Our model is based on static analysis, the subject programs are biased in this sense. The implemented prototype might show different results for the subject program because of its static nature. As our analysis is carried out on medium sized programs, therefore, it may create hindrance in the generalization of results.

For the case study, we have analyzed some open source projects that are publicly available and contain enough information on bug fix reports. Our decision about the selection of subjects depends on the documentation of defects, the repository extraction performance and actively maintained by its developers. We have selected “Redmine” as first subject because it fulfills all the mentioned requirements. For the second subject, we have selected “puituniverse” and for the third subject, we selected, “h2o-ruby” because they also fulfill the basic requirement for the selection of subject.
7.2.1 Redmine

According to the website of Redmine (Redmine, 2006), “Written using Ruby on Rails framework, it is cross-platform and cross-database”. Its first release (0.1.2) was announced on 25th June 2006. The current stable release (0.7.3) was announced on 6th July 2008. According to their issue tracker (Redmine, 2006), there are total 612 bugs found in which 513 are resolved by the developers and 99 are still open. For the analysis purpose, we have selected a bug from revision 0.7.2 (11th July, 2008) that was resolved in revision 0.7.3 (12th July, 2008). The reason behind the selection of this subject is that it consist two types of bugs; the first one is the change of data structure i.e. an array data structure is replaced by hash data structure. The second bug is invalid reference of old data structure in right hand side value of assignment statement in self.event_option. It is referring to the old data structure instead of the new data structure. These types of faults are very common and these can help in making a model that can be utilized for automated fault injection process. The complete detail about these bugs are explained in Section 7.3.1 & 7.3.2.

7.2.2 puituniverse

According to github (github, 2008A), “puituniverse is the ruby game programming tutorial, and an easy Ruby 2D game engine & multiverse”. For the analysis purpose, we have selected a bug from revision 100640 (20th April, 2008) and it was resolved in revision 100644 (22nd April, 2008). The reason behind the selection of this bug is that it involves fix in the references of different variables. By executing DefectoFix on this type of bug, we can say that the proposed model can fix this type of bugs by replacing the faulty code level values with the corresponding correct code level values. The complete detail about these bugs are explained in Section 7.3.3.

7.2.3 h2o-ruby

As mentioned on github (github, 2008B), “h2o is template engine in ruby. For the analysis purpose, we have selected a bug from revision 100755 (3rd August, 2008) and it was resolved in revision 100748 (31st July, 2008). DefectoFix will find out this documented bug using differencing algorithm and later on connect it to the bug reports, where it can be document in repository.

The selection of h2o-ruby is slightly different from previous two subjects in the sense that the selected bug of h2o-ruby will help in order build a model, which can be used to re-inject faults in the source codes. In this subject, we injected faults in source code in order to find the injected faults by executing source code in DefectoFix. In this subject, we interested in different areas of faults. These faults lie in the areas of “reference problems” and replacement of “method definition”. The complete detail about these bugs are explained in Section 7.3.4.

7.3 Procedure & Results for Case Study Subjects

As mentioned in previous section, we have analyzed three different types of bugs from three open source projects “Redmine”, “puituniverse” and “h2o-ruby”. The section 7.3.1 explains the analysis procedure and results for the “SRB” and section 7.3.2 presents the analysis procedure and results for the ”Redmine”. The analysis and results of “puituniverse” are explained in section 7.3.3 and the analysis and results of “h2o-ruby” are described in Section 7.3.4.
7.3.1 Simplified Redmine Bug (SRB)

7.3.1.1 Procedure for SRB

“Redmine” is used for project management of web applications written using Ruby on Rails framework. Two revisions (041208 & 071208) of this project were selected for analysis of DefectoFix tool based on the proposed framework. To explain the defect fix model, extracted from “Redmine” project, in a better way in this thesis, a simplified version of Redmine was used. In this simplified version, the defect was distilled down so that it could be expressed in as few lines of code as possible. The resulting code is shown in table 7.1. We call this simplified version of “Redmine” as “Simplified Redmine Bug (SRB)”. The defect fix model which is extracted from this subject will be used in automated fault injection method in later section.

The selected source files were reduced to contain two changes. As shown in Table 7.1, there are two major changes in the corrected source code; first change is the conversion of “options” array into “default_options” hash, and the second change is the assignment of data structure into “self.event_options”.

<table>
<thead>
<tr>
<th>Faulty Source Code</th>
<th>Correct Source Code</th>
</tr>
</thead>
</table>
| module ClassMethods
| options[:datetime] ||= created_on
| self.event_options = options
| end |
| module ClassMethods
| default_options = {:datetime => :created_on}
| self.event_options = default_options.merge(options)
| end |

Table 7.1 Faulty and Correct Source Code for acts_as_event.rb (SRB)

For the analysis purpose, we have applied DefectoFix on acts_as_event.rb (revision 0.7.2) and on acts_as_event.rb (revision 0.7.3). In file acts_as_event.rb (revision 0.7.2) uses “options” was defined as a “Ruby Array” and this was changed in acts_as_event.rb (revision 0.7.3) by “default_options” which was used as a “Ruby Hash”. There is also another change in these files i.e. assignment of data structure in “self.event_options”. Therefore, we have linked acts_as_event.rb (revision 0.7.2) as a faulty source code input file and acts_as_event.rb (revision 0.7.3) as correct source code input file in DefectoFix tool.

DefectoFix tool applied Ruby parser to the generated code fragments and that creates AST from these code fragments. These AST files were then used by DefectoFix tool to make stack of node level of faulty and correct source code files. In the final step, DefectoFix tools compared the stack (ST) generated from faulty source code fragments with corresponding stack generated (ST') from correct source code fragments. ST denotes the stack of faulty source code fragment and ST’ represents the stack of correct source code fragments.

DefectoFix tool compared two stacks by calling its NodeMatch() procedure for node level code comparison. For comparison of two stacks level, we needed increment variable lookahead and by default, its value is 1. In the first iteration over the main loop, the DefectoFix pops and compares first tree level from stack ST (revision 0.7.2) with first tree
level from ST’ (revision 0.7.3) of “acts_as_event.rb” file. The subsequent nodes of faulty AST level would be compared with corresponding nodes of correct AST level.

In the second iteration, second tree level would be popped from ST and ST’ and compared. Subsequent nodes of level 2 would be compared in similar fashion as of level 1. This process continues until we reached to the last level of stack ST or ST’ depending on the size of stack. The tool compared the two levels by calling its NodeMatch() procedure and it compares the level nodes. NodeMatch() return “True” if tree levels are similar and return “False”, if they are not similar. If “False” is returned then that faulty and corrected minimal sub tree of AST saved in repository with all available valuable information. The final step of DefectoFix algorithm continues until all the stacks generated from the faulty and correct revisions of Redmine were not processed. Next section explains the detailed procedure for fault injection in source code file.

This saved sub-tree is a part of defect fix model, which can be used in a future for finding changes in the source code files. This model consists of two parts, the first part of model consist of sub trees of faulty AST and correct source code AST, where change exists. The second part of model consists of additional information that includes Defect Fix Model Id (DFMId), Project Name (PN), File Name (FN), Class Name (CN), File Revision Number (FRN) and Sub Tree Level (STL). The additional information in a model can later be used in automated fault injection process using VCS and repository.

7.3.1.2 SRB Results

The results of SRB are evaluated in two steps. In first step, we have extracted defect fix model from SRB and in second step, the extracted defect fix model was used in automated fault injection process for the validation of DefectoFix tool. The other reason for fault injection is to validate the proposed framework and DefectoFix tool for the automated fault injection process.

7.3.1.2.1 Defect Fix Model Extraction

In this step, we have extracted defect fix model(s) from “act_as_event.rb” of SRB project. Defect fix model extraction starts by comparing AST of faulty and correct source code files using sub tree level and node level comparisons. Figure 7.3 and 7.4 shows AST of faulty and correct source code files respectively. In these figures, the most important thing to understand is numbering scheme of tree levels. The values assigned to levels and sub levels follow a particular format. The root node is labeled as “s” and its children are labeled as “s1, s2,.., sN”. In the similar way, if “s1” has more children, then it would be marked as “s11, s12,.., s1N”. We have used this standard for the marking of sub tree and nodes of AST in all subjects.
DefectoFix tool compares sequentially levels of generated AST from faulty source code to the generated AST from the correct source code. That means first level “s” of faulty AST is compared to the first level “s” of correct AST. This process will continue until all the nodes of faulty and correct AST are not traversed. DefectoFix tool stores the faulty sub tree and correct sub tree in repository, if two compared sub trees contain different nodes.

The results of this subject (SRB) shows that sub tree “s322” & “s32342” of faulty source AST are different than the corresponding sub tree of correct source code AST as shown in figure 7.3 and 7.4. “s322” sub tree of faulty AST shows that “Ruby Array” was used as a data structure and it was replaced by “Ruby Hash” in correct AST. Furthermore, DefectoFix identified “reference problem” in the sub tree “s32342” of faulty and correct AST. In “s32342”, faulty sub tree refers to an old data structure (Ruby Array) and it was replaced by new data structure (Ruby Hash) in correct sub tree “s32342”. DefectoFix tool reports “s322” & “s32342” faulty and correct sub tree in the output form (as shown in figure 7.5). The output form of DefectoFix also contains input fields for the additional information of defect fix model. There are two methods to input this additional information; first one is manually by user, and second one is by getting these values from log files of VCS. Once both of these parts of defect fix model are acquired, then it can be stored as a defect fix model in repository.
Table 7.2 A defect fix model with sample

<table>
<thead>
<tr>
<th>DFMId</th>
<th>PN</th>
<th>CN</th>
<th>FN</th>
<th>FRN</th>
<th>STL</th>
<th>FaultyAST</th>
<th>CorrectAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>SRB</td>
<td>Redmine</td>
<td>acts_as_event.rb</td>
<td>2.3</td>
<td>1223</td>
<td>s(:vcall, :options)</td>
<td>s(:call, s(:lvar, :default_options), :merge, s(:array, s(:vcall, :options)))</td>
</tr>
</tbody>
</table>

Figure 7.3 Diff Model outputs, when DefectoFix applied on acts_as_event.rb (SRB)

7.3.1.2.2 Fault injection in SRB

In the second step, faults would be injected in a correct source code file from repository in order to validate the working of DefectoFix tool based on the proposed framework. This step will also help us to use defect fix model repository for automated fault injection process. During fault injection process, a correct source code file from VCS would be extracted and faults are automatically injected in it by using defect fix repository. To inject faults, DefectoFix tool needs additional information from correct source code file. This additional information includes DFMId, PN, FN, CN, FRN and STL. Based on this additional information a matching defect fix model would be extracted from the repository and injected in a correct source code file to make it faulty source code file.
As we already described earlier in section 7.4.1.1 that DefectoFix tool successfully identifies the faults in the “act_as_event.rb”. These identified faults are stored as defect fix model in repository. For fault injection process, “act_as_event.rb” file is extracted from VCS and defect fix model(s) is/are injected in it from the DefectoFix repository to make it faulty source code file. The defect fix model(s) is/are injected based on the additional information of defect fix model and correct source code file. The additional information of correct source code file would be compared against the additional information of defect fix models in order to inject defect fix model. In this process, we have found that DefectoFix tool successfully identified the injected defect fix model. This shows that the proposed framework can be used to extract defect fix models that can later be used for fault injection process.

7.3.2 Redmine

7.3.2.1 Procedure for Redmine

In this section, we will describe the analysis procedure for the “Redmine”. The two revisions selected for the analysis for this subject are revision 041208 and revision 071208. The file acts_as_event.rb (revision 041208) was used as faulty source code file and acts_as_event.rb (revision 071208) was used as correct source code file. The remaining analysis procedure remains same as described for the “SRB” in section 7.3.1. As shown in Table 7.3, there are two major changes in the corrected source code; first change is the conversion of “options” array into “default_options” hash, and the second change is the assignment of data structure into “self.event_options”.

Table 7.3 Faulty and Correct Source Code for acts_as_event.rb (Redmine)

<table>
<thead>
<tr>
<th>Faulty Source Code</th>
<th>Correct Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>module ClassMethods</td>
<td>module ClassMethods</td>
</tr>
<tr>
<td>def acts_as_event(options = {})</td>
<td>def acts_as_event(options = {})</td>
</tr>
<tr>
<td>return</td>
<td>return</td>
</tr>
<tr>
<td>if self.included_modules.include?(Redmine::Acts::Event::InstanceMethods)</td>
<td>if self.included_modules.include?(Redmine::Acts::Event::InstanceMethods)</td>
</tr>
</tbody>
</table>
7.3.2.2 Results for Redmine

When we have analyzed the results for this subject, we found two different scenarios. The first scenario could be the different number of levels in faulty and correct AST that need to be compared. In first scenario, the number of levels is more in the faulty AST than the correct AST. Second scenario is the reverse of the first scenario. In this subject, the first scenario was true. In this scenario, DefectoFix tool compares sequentially levels of AST generated from faulty source code to the AST generated from the correct source code. AST levels are labeled in similar fashion as mentioned in section 7.4.1. The levels are also compared in the same way of SRB.

![Faulty AST levels of acts_as_event.rb (Redmine)](image1)

![Correct AST levels of acts_as_event.rb (Redmine)](image2)
The results of this subject (Redmine) show that sub tree s1111122, s1111123, s1111124, s1111125 and s1111128 of faulty source AST are different than the corresponding levels of correct source code AST as shown in figure 7.5 & 7.6. DefectoFix tool reports in output form that sub tree s1111122, s1111123, s1111124, s1111125 and s1111128 of faulty AST should be replaced with s1111122, s1111123, s1111124, s1111125 and s1111128 of correct AST. The defect fix model obtained by applying DefectoFix is shown in Figure 7.7. The total number of levels in faulty AST is higher than correct AST, as shown in Figure 7.5 & 7.6. The remaining levels are extra in faulty AST and would be marked as “not needed” by the DefectoFix. The complete AST for faulty and correct source code is presented in Appendix A in a table format.

If look in detail in Appendix A, we can identify that level s1111122 in faulty AST of “Redmine” shows that faulty code contains “Ruby Array” data structure. Where-as by looking at correct AST of “Redmine”, we identify that level s1111122 nodes are replaced by new nodes at level s1111122, which is a “hash” in correct source code. So in defect fix model, “Ruby Array” data structure is replaced by the “Ruby Hash” data structure in correct revision in “Redmine”. Now if we look for another fault in the faulty source code, we can identify that sub tree s1111123, s1111124 and s1111125 in faulty AST is replaced by level s1111123, s1111124 and s1111125 of correct AST respectively. The extracted defect fix model for “Redmine” is shown in figure 7.7.

![Figure 7.7 Diff Model outputs, when DefectoFix applied on acts_as_event.rb (Redmine)](image)

### 7.3.3 puituniverse

#### 7.3.3.1 Procedure for puituniverse

In this section, we will describe the analysis procedure for the “puituniverse”. The two revisions selected for the analysis for this subject are revision 100640 and revision 100640. The file collideable.rb (revision 100640) was used as faulty source code file and
collideable.rb (revision 100644) was used as correct source code file. The remaining analysis procedure is same as described for the “Redmine”. There are four reference problems in the two revisions of collideable.rb as shown in Table 7.4.

<table>
<thead>
<tr>
<th>Faulty Source Code</th>
<th>Correct Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>def bottom() y.to_f + height.to_f; end</td>
<td>def bottom() y.to_f + height.to_f; end</td>
</tr>
<tr>
<td>def top= value</td>
<td>def top= value</td>
</tr>
<tr>
<td>y = value</td>
<td>@y = value</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>def bottom= value</td>
<td>def bottom= value</td>
</tr>
<tr>
<td>y = value - height</td>
<td>@y = value - height</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>def left= value</td>
<td>def left= value</td>
</tr>
<tr>
<td>x = value</td>
<td>@x = value</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>def right= value</td>
<td>def right= value</td>
</tr>
<tr>
<td>x = value - width</td>
<td>@x = value - width</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

### 7.3.3.2 puituniverse Results

We have selected “puituniverse” as a third subject for our case study. AST generated by the DefectoFix tool for collideable.rb (revision 100640) and collideable.rb (revision 100644) contains same levels. When we executed DefectoFix on collideable.rb (revision 100640) & (revision 100644), it identifies the differences of “instance variable” in these two files. The diff model output obtained by applying DefectoFix is shown in Figure 7.8 below:
Figure 7.8 Model outputs, when DefectoFix applied on collideable.rb (puituniverse)

As we explain earlier in the Section 7.4.1 that DefectoFix compares nodes of different sub tree and if it finds change in any node of two files, then that complete sub tree is replaced by the corresponding sub tree in the correct AST. As shown in Figure 7.7 & 7.8 that two nodes of s3322 and S5522 are changed in both AST. DefectoFix identifies this change and report defect fix model as shown in Fig above that s3322 and s5522 of faulty AST should be replaced by s3322 and s5522 of correct AST. The extracted defect fix model for “puituniverse” is shown in figure 7.8.

Figure 7.9 Faulty AST levels of collideable.rb (puituniverse)
7.3.4 h2o-ruby

7.3.4.1 Procedure for h2o-ruby

This section presents the analysis procedure for the third subject “h2o-ruby”. The only difference between the analysis procedure of “Redmine” and “h2o-ruby” was the input source code files to DefectoFix tool. In “Redmine”, we have two different revisions of files for the input of faulty and correct source code files. But in the “h2o-ruby”, we have only one file “h2o.rb” and the second file is generated by injecting faults in “h2o.rb”. The file with injected fault was named as “faulty_h2o.rb” and correct file was named as “correct_h2o.rb”. The injected faults are shown in Table 7.5.

Table 7.5 Faulty and Correct Source Code for h2o.rb (h2o-ruby)

<table>
<thead>
<tr>
<th>Faulty Source Code</th>
<th>Correct Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Template</td>
<td>class Template</td>
</tr>
<tr>
<td>attr_reader :context</td>
<td>attr_reader :context</td>
</tr>
<tr>
<td>def initialize (filename, options = {})</td>
<td>def initialize (filename, env = {})</td>
</tr>
<tr>
<td>@file = Pathname.new(filename)</td>
<td>@file = Pathname.new(filename)</td>
</tr>
<tr>
<td>@parser = Parser.new(@file.read, @file)</td>
<td>@parser = Parser.new(@file.read, @file)</td>
</tr>
<tr>
<td>env[:search_path] = @file.dirname</td>
<td>env[:search_path] = @file.dirname</td>
</tr>
<tr>
<td>@nodelist = Template.load(@file, env)</td>
<td>@nodelist = Template.load(@file, env)</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>def render (context = {})</td>
<td>def render (context = {})</td>
</tr>
<tr>
<td>@nodelist.render( @context, output_stream)</td>
<td>@nodelist.render( @context, output_stream)</td>
</tr>
<tr>
<td>output_stream</td>
<td>output_stream</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>def self.parse source</td>
<td>def self.parse file, env</td>
</tr>
<tr>
<td>parser = Parser.new(source, file, env)</td>
<td>file = env[:search_path] + file if file.is_a? String</td>
</tr>
<tr>
<td>parser.parse</td>
<td>parser.parse</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

65
7.3.4.2 h2o-ruby Results

“h2o-ruby” was selected as a third subject for our case study. AST generated by the DefectoFix tool for h2o.rb (faulty_h2o.rb) and h2o.rb (correct_h2o.rb) contains same levels. When we executed DefectoFix on h2o.rb (faulty_h2o.rb) & (correct_h2o.rb), it identifies the differences of injected faults in these two files. The Diff model output obtained by applying DefectoFix is shown in Figure 7.10 below:

![Diff Model output](image)

7.10 Diff Model outputs, when DefectoFix applied on faulty_h2o.rb and correct_h2o.rb

Figure 7.11 & 7.12 depicts that levels of sA12, sA14 and SE111 are changed in both AST. DefectoFix identifies this change and suggest “Diff output” as shown in Fig above that sC111 and sE111 of faulty AST should be replaced by sC111 and sE111 of correct AST. By detailed analysis of Diff Model output, one can identify & document the fault in the repository.

![Faulty AST levels](image)

Figure 7.11 Faulty AST levels of faulty_h2o.rb (h2o-ruby)
7.4 Result Analysis

In the above section, we have presented the results based on the data collected from case study. In this section, we will present result analysis of the proposed framework based on the data collected from the case study.

We have already described that the proposed framework uses new approach to identify faults with additional available information in source code files. It was also different from other differencing models in a sense that it uses static analysis rather than dynamic analysis and works on node level analysis for object oriented programming languages. If DefectoFix identifies any change in the nodes of any sub-tree of faulty AST, then output form “Diff Model” reports that the faulty sub-tree should be replaced by the corresponding sub-tree of correct AST. This sub-tree would be saved with additional information in a defect fix repository and we called this sub-tree and additional information as a “Defect Fix Model (DFM)”

For the validation of our proposed model, we have conducted a case study on different subjects to show authenticity of the proposed framework. From the results of the case study, we want to evaluate the working of DefectoFix based on the proposed framework. The results from three different subjects show that DefectoFix successfully identified the faults and then stored these faults with additional information as defect fix model(s) in a repository. The other purpose of the case study is to evaluate the extracted defect fix model(s) for automated fault injection process. From results obtained in section 7.3.1.2.2, we can see that extracted defect fix model works for the fault injection process. Thus we can say that extracted defect fix model(s) can be integrated in automated fault injection tools for the purpose of software validation.

There are some limitations of DefectoFix tool and the proposed model. The proposed model suggests the complete sub-tree change even the fault exists in a single node of that sub-tree. If faulty source code file and correct source code file are similar but their order is different, this situation is not handled by our proposed framework. The proposed framework is validated for the limited scenarios. We have tested our framework for the

Every research has some negative aspects with its positive aspects. It is similar in our case. We have proposed the framework that can extract defect fix model(s) by applying differencing algorithm of DefectoFix in the source code files. DefectoFix needs more testing before it can be applied for different kind of projects, consisting of different types of faults. And once, these improvements have been finalized then this tool could be proved very lucrative in the area of defect finding using efficient differencing algorithm.
8 DISCUSSION

The evaluation of DefectoFix, as described in previous chapters, shows that the DefectoFix implementation of the differencing algorithm is capable of finding differences from the source code files and stored them as DFM in repository for further use.

Section 8.1 gives the findings from our research work. Threats to validity to our research work are explained in Section 8.2. Section 8.3 describes the experience by the user while conducting this research.

8.1 Findings

We will discuss findings from our research work in two categories namely strengths and limitations of the DefectoFix based on the proposed framework. The rationale behind discussing our findings in two categories is that it becomes easy for us to identify where our idea works well and where it needs improvements. In Section 8.1.1 we will explain the strengths of the DefectoFix based on proposed model and Section 8.1.2 describes the limitations of the DefectoFix based on the proposed model.

8.1.1 Strengths

The result of case study shows that DefectoFix correctly find the differences between two revisions of the subjects “Redmine”, “puituniverse” and “h2o-ruby”. On the basis of obtained results we can say that the extract model of level and node comparison can easily be applied to any other code that contains different types of bugs. This is because, every AST level of faulty source code is being compared with corresponding AST level of correct source code. However, more testing on different subjects is needed to improve the validity of DefectoFix tool.

DefectoFix also correctly identified the faults that are injected in the source code file of “SRB” in order to validate the tool. These results reflect the usefulness of our proposed framework and also validate the implementation of the tool. The results of this fault injection enabled us to say that the extracted specified model (DFM) can be re-used for fault injection in other source codes that shares few similarities among them.

Another strength of proposed framework is that it is general and can be implemented using any other object oriented language by only changing the parser. The rest of the procedure will remain same with all other object oriented languages. Furthermore, the tool’s GUI is native in different platforms because of wxRuby and it also follows the guidelines from the Human Computer Interaction (HCI) as stated in (Usability Guidelines, 2007). This resulted in user friendly GUI.

8.1.2 Limitations

There are some limitations of DefectoFix tool based on the proposed framework which is discussed below.

- If there is some difference found in the source code, DefectoFix algorithm replaces complete level/sub-level even that change exists in a single node. This problem can be observed in Section 7.5.2.
In the current implementation of DefectoFix tool, it is restricted for two 
source code files as input i.e. faulty and correct source code files.

DefectoFix tool does not show the exact place/location/line number of 
the differences exist in faulty and correct source code file. For this reason, we 
have connected the “Diff output” with the bug and fix reports. This bug 
and fix report will be manually entered by the user.

Code fragmentation of the DefectoFix is not working according to the 
desired functionality. The main motivation behind the code fragmentation 
is that it can separate source code in different fragments based on classes. 
Currently, DefectoFix read source code file and whenever it finds “class” 
keyword, it starts making fragments. We identified a problem in this 
approach that if “class” keyword is used in comments, then DefectoFix 
didn’t recognize it as a comment and creates new code fragment.

If both faulty and correct source code files have exactly same source code 
but their order of line numbers are different, so in that case DefectoFix 
algorithm do not work. It needs to be fixed in the future versions of 
DefectoFix.

Sometimes, the presentation structure of the output is disturbed because of 
the availability of excessive output data. Two buttons (Register and 
Cancel) in the output form are disappeared because of this problem.

The result of the evaluation cannot be generalized as DefectoFix is only 
tested with limited scenarios. DefectoFix tool might behave differently 
with other scenarios, which can impact the effectiveness of the proposed 
framework.

At the moment, DefectoFix tool is working with the source code files that 
are manually attached by the user. However, the tool linkage with version 
control system is still incomplete; hence it is not validated with Version 
Control System. It might be possible that the difference in input file path 
may affect the results from the Defectofix tool.

8.2 Threats to Validity

We have conducted a case study, there are two types of threat exists in such kind 
of research methodology. The first one is “general validity threat” and other threat could be 
“internal validity threats”.

As we have analyzed only three real world project that may cause validity threat in 
our research. And we have identified this threat before starting the evaluation step on the 
tool. So we have analyzed one more example “SRB” which also shows the usage of DFM 
models in fault injection process, in addition to real world project. This strategy helped us to 
mitigate the validity threat to some extent.

Threat to internal validity mostly concerns with possible errors in our algorithm 
implementation and analyzing tool (DefectoFix) that could affect the outcome. To overcome 
these threats, we have validated the implementation of DefectoFix tool on known examples 
and performed several sanity checks.
The fact that the proposed framework is applied only on differences retrieved from three open source projects. So it has raised some concerns about the interpretation and generality of the results. More and better controlled empirical evaluations will be required before we can claim the overall effectiveness of the proposed framework and tool with confidence. This step is anticipated in future work.

8.3 User Experience

As we have experience of software development and during the development of DefectoFix tool, based on the proposed framework, we have analyzed that the DefectoFix can be used as a valuable assistant in code review meetings. During code review meetings, usually a few chosen code excerpts are analyzed. The implemented tool can also be worked in the similar manner.

The inner workings of the framework are not bounded to any specific code characteristics that are known in advance and may leads to failures. The prototype does not look for analogies of patterns defined in advanced in a given code base. The user can instead train the prototype to identify the faults specific to his/her domain of interest. This can be achieved only with annotation-based static analyzers that require invasive code modifications or writing compiler extensions (as described by Engler et al., 2000, for instance). Using this tool, none of them are required.

This chapter explains the strengths and limitations of the DefectoFix tool, based on the proposed framework. DefectoFix tool overcomes the limitations of existing tools by providing more useful information about the differences and providing more user friendly GUI. But with its strengths, there are some limitations of DefectoFix tool, such as limited testing and no integration with version control repositories. Overall, DefectoFix tool has more benefits than its limitations and as a software developer, we think that it can be used in different phases of software development life cycle, especially in code review meetings.
9 **EPILOGUE**

The epilogue contains conclusion, answers research questions, contribution and the future work.

### 9.1 Conclusion

A vast study of various existing defect fixing and logging tools shows that a large amount of information is lost during fixing and logging of defects. This thesis presents the solution how to gather and use the information which is missed out while committing the bug fixes in Version Control System (VCS) by developers. We have described the design and implementation of DefectoFix framework for this purpose. This framework is able to extract differences in source code. In order to gather maximum missing information, a new differencing algorithm is proposed which performs comparison of source code files at nodes levels and also perform static analysis of source code files. The results of extraction of valuable information are saved as Defect Fix Model (DFM).

The prototype of the framework also provides an interactive form, where the developer has an opportunity to enter his/her comments regarding those particular problems along with differences in source code files. Thus, these differences and developer information forms the Defect Fix Model (DFM), which can be used for various purposes. Automated fault injection is one of its application.

The DefectoFix framework has been evaluated for detecting the faults in different source code files by making DFM. These DFM(s) include additional information such as project name, class name, file name, revision number and diff model. These DFM(s) were validated by checking them for the identification of injected faults in source code files. The results show the effectiveness of the DefectoFix framework for testing, verification and validation through static code analysis.

### 9.2 Research Questions Revisited

In the beginning of the thesis three research questions were raised in order to guide this research work. These questions are now revisited to see how the outcome of the research helped to answer them.

1. *What are the goals of existing defect fixing/tracking tools and their limitations?*

   The survey of existing defect fixing and logging tools is conducted and given in this thesis in order to find out their goals and limitations. Tools like Bugzilla are designed as an issue-tracking system which supports multiple projects with independent user bases. Some tools like CodeStriker are developed for online code reviewing. A brief overview of goals of various tools can be found in Table 3.1

   The survey of defect tracking tools shows that a large amount of information is lost during defect fixing and logging. Furthermore, few tools such as Bugzilla are large and rich in features but they are complex and difficult to understand. These complex tools are not user friendly, easy to use, and also require user training. To overcome the loss of information such
as Version Control System (VCS) comments, information from log files, project information, class information, file information, file revision information and diff model information, we have designed a framework, which is useful in building the richer models of defect fixing and logging.

2. How to overcome the limitations in existing defect tracking tools?

We have developed a tool based on DefectoFix framework to overcome the loss of valuable information (VCS comments, information from log files, project information, class information, file information, file revision information and diff model information) in the defect finding process. This tool extracts maximum available information of faults found using AST and node comparison, thus saving valuable information. The other major limitation with existing tools is the user friendliness. We tried to overcome this problem with two possible solutions; we have integrated Human Computer Interaction (HCI) in our tool and we have tried to provide GUI in our tool which support native look in different platforms.

3. Designing and development of a tool to answer the limitations in existing defect fixing/tracking tools?

In the second research question, we have identified two common limitations in most of the tools which are loss of valuable information and complex GUI. To address the problem of the loss of valuable information, we have designed and implemented DefectoFix framework that extracts defect fix information using node comparison from AST of faulty source code files. For constructing AST from source code files, we have used Ruby Parser. The generated AST findings are used for the differences from faulty source code files using node comparison. These faults and their additional information are stored as DFM in the repository (MySQL).

We address the GUI issue by implementing GUI of this tool with the help of Human Computer Interaction (HCI) guidelines. This tool is implemented using Ruby (wxRuby). Ruby is a platform independent and object oriented language. Thus it provides native GUI look on different platforms like MS Windows and Linux operating systems etc.

9.3 Contributions

• Summary of literature in the area of defect tracking tools and differencing algorithms.
• New algorithm for finding differences in the source code files.
• Design and implementation of a framework that is able to find differences in source code files and stores them as defect fix model that can be used with automated fault injection process. It also implements the algorithm for finding differences in source code files.
• An implemented tool that integrates the proposed algorithm.
• A case study that evaluates the DefectoFix tool.
9.4 Future Work

In this study we have proposed DefectoFix framework for locating faults in source code files. However, during its implementation and evaluation, the number of various improvements has been added in “todo list”. The most important items from the “todo list” are as follows:

- The tool should be capable of automated injection of faults using Defect Fix Model from repository.
- The tool should be evaluated for more examples which are closed to the real world problems.
- To integrate this tool with Version Control System so that it can be used by the software industry. This will result in better validation of the proposed framework and DefectoFix tool.
- At the moment, this tool can only be used with Ruby source code files. But in future, we want the tool to work for source code of different object oriented languages.
- Immediate follow-up work includes making our proposed tool and technique publicly available under some open source license, for instance, GNU Public License (GPL). This will allow other researchers to try the proposed framework. Based on the received feedback, new further trends might be considered.
- Last but not the least is writing a paper with the intention to be published in an appropriate workshop or a journal.
REFERENCES

Books and Papers


Web Resources


https://zooko.com/revision_control_quick_ref.html
APPENDIX A: AST LEVEL CHARTS

acts_as_event.rb (revision 0.7.2) Redmine

```
s :module, :ClassMethods, sA
sA :scope, sA1
sA1 :defn, :acts_as_event, sA1
sA11 :scope, sA11
sA111 :block, sA1111
sA1111 :args, :options, sA11111, sA11112
sA11111 :block, sA111111
sA111111 :assign, :options, sA1111111
sA1111111 :hash

sA11112 :if, sA111121, sA111122, sA111123, sA111124, sA111125, sA111126, sA111127, sA111128, sA111129
sA111121 :call, sA1111211, :include?, sA1111212, nil
sA1111211 :self,

sA111122 :array, sA1111212
sA1111212 :colon2, sA11112121, :InstanceMethods
sA11112121 :colon2, sA111121211, :Event
sA111121211 :colon2, sA1111212111, :Acts
sA1111212111 :const, :Redmine
sA11112112 :return,

sA1111222 :op_asgn1, sA1112221, sA11112222, "||", sA1111223
sA1111221 :ivar, :options

sA1111222 :array, sA11112221
sA11112221 :lit, :datetime

sA1111223 :lit, :created_on
sA111123 :op_asgn1, sA1111231, sA1111232, "||", sA1111233
sA1111231 :ivar, :options
sA1111232 :array, sA11112321
sA11112321 :lit, :description

sA1111233 :lit, :description
sA111124 :op_asgn1, sA1111241, sA1111242, "||", sA1111243
sA1111241 :ivar, :options

sA1111242 :array, sA11112421
sA11112421 :lit, :author

sA1111243 :lit, :author
sA111125 :op_asgn1, sA1111251, sA1111252, "||", sA1111253
sA1111251 :ivar, :options

sA1111252 :array, sA11112521
sA11112521 :lit, :url

sA1111253 :hash, sA11112531, sA11112532
sA11112531 :lit, :controller

sA11112532 :str, "welcome"

sA111126 :op_asgn1, sA1111261, sA1111262, "||", sA1111263
sA1111261 :ivar, :options

sA1111262 :array, sA11112621
sA11112621 :lit, :type

sA1111263 :call, sA11112631, :dasherize
sA11112631 :call, sA111126311, :underscore

sA111126311 :call, sA1111263111, :name
sA1111263111 :self,

sA111127 :fcall, :cattr_accessor, sA1111271
```
acts_as_event.rb (revision 0.7.3) Redmine

acts_as_event.rb (revision 0.7.3) Redmine
```
| sA11112311 | :array, sA11112311 |
| sA11112311 | :lit, :event_options |
| sA111124   | :attrasgn, sA1111241, :event_options=, sA1111242 |
| sA1111241   | :self |
| sA1111242   | :array, sA11112421 |
| sA11112421  | :call, sA111124211, :merge, sA111124212 |
| sA111124211 | :ivar, :default_options |
| sA111124212 | :array, sA11112421 |
| sA1111242121 | :ivar, :options |
| sA111125   | :fcall, :send, sA1111251 |
| sA1111251   | :array, sA11112511, sA11112512 |
| sA11112511  | :lit, :include, |
| sA11112512  | :colon2, sA111125121, :InstanceMethods |
| sA111125121 | :colon2, sA1111251211, :Event |
| sA1111251211 | :colon2, sA11112512111, :Acts |
| sA11112512111 | :const, :Redmine |
```