Analyzing Software Bugs in Code Clones

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Judith Francisca Islam

Permission to Use

Abstract

An extensive area of research in Software Engineering for the last two decades has been finding duplicate source code in the code base. The act of duplicating source code or copy-pasting source code in software systems is known as code cloning. Reusing source code or code cloning is a very common practice in software development. Code cloning can save time and cost. On the other hand, if a code fragment contains a bug then duplicating that source code will also duplicate the bug. Thus, in the literature there is a common belief that code cloning is responsible for spreading bugs in software systems. The primary goal of our study is to validate this common belief. To achieve this goal, we performed four major studies and analyzed the characteristics of bug-proneness of code clones. First, we aimed to understand bug-proneness between clone and non-clone code. Afterwards, we explored bug-proneness of micro-clones which are smaller in size (from 1 line of code to 4 lines of code) in our second study. Later, we focused on bug-replication of regular code clones and micro-clones in our third and fourth studies. Our result supports the intuition from the literature and we found that code clones are more bug-prone than non-clone code. Moreover, we found that it is important to emphasize bug-prone code clones so that it will help us in clone management and better maintenance of software systems. We also proposed a possible solution to identify and manage these bug-prone code clones and showed some future paths of this research.

Acknowledgements

I am highly grateful to the Almighty God for everything and helping me to come to this far achievement of my life. I would like to express my heartiest gratitude to my respected supervisor Dr. Chanchal K. Roy for his constant guidance, advice, encouragement and extraordinary patience during this thesis work. Without his support this work would have been impossible.

I would like to thank Dr. Kevin A. Schneider, Dr. Mark Keil, Dr. Ian McQuillan, Dr. Banani Roy and Dr. Khan Wahid for their participation on my supervising committee, including their advisement and evaluation of my work and thesis. I would like to thank my external examiner Dr. Yoshiki Higo for his evaluation of my thesis.

I thank to all the present and past members of Software Research Lab for their support and motivations. Specially, I am thankful to Dr. Manishankar Mondal for his valuable comments and suggestions during my research.

I want to thank all of my co-authors of my published papers including Chanchal K. Roy, Kevin A. Schneider, Manishankar Mondal, Iman Keivanloo, Mohammad Mamun Mia and Jeffrey Svajlenko. Also, thanks to all of those anonymous reviewers who reviewed my papers and provided valuable suggestions.

I am thankful to all the staff members of the Department of Computer Science who's continuous support helped me to reach out my degree goals.

I would like to thank Department of Computer Science and College of Graduate and Postdoctoral Studies (CGPS) for their generous support and providing me financial help, award, scholarship, retroactive medical leave which eased my path of this long journey.

I would like to remember my six miscarried babies, two of them were boys (Noah and Michael) and the rest were gender unknown, who I believe are my key strength to work hard during this PhD. I express my gratitude to my better half, Mohammad Mamun Mia who's love and patience helped me to complete this thesis. I am also extremely thankful to my parents (Carmel and Patrick), my sisters (Jane and Jessica) and their family, my brothers (Giles and Justus) and their family, and my parents-in-law (Sarufa and Alam), my brother-in-law (Masum) for supporting me all the way to this end goal.

For those that I have not precisely mentioned here, thank you for being a part of this thesis and helping me for becoming a person and a researcher.

I dedicate this thesis to my beloved mother, Ruth Carmel Gomes, whose tremendous support and inspiration helped me to grow as a strong woman, to my father, Patrick Gomes, whose enormous motivation and encouragement inspired me to be independent, and to my life partner, Mohammad Mamun Mia, whose constant support and sacrifices made it possible to complete this work successfully.

Contents

Permission to Use **i**

Abstract **ii**

Acknowledgements **iii**

Contents **v**

List of Tables **ix**

List of Figures **xi**

List of Abbreviations **xiii**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Definition 2
1.3 Addressing the Research Problem 3
1.3.1 Decomposing our Research behind Addressing the Problem 4
1.3.2 Study 1: Comparing Bug-proneness in between Clone and Non-Clone Code 4
1.3.3 Study 2: Finding and Comparing Bug-proneness in between Regular and Micro-Clones 4
1.3.4 Study 3: Identifying and Analyzing Bug-replication in Code Clones 5
1.3.5 Study 4: Detecting and Comparing Bug-replication in between Regular and Micro-Clones 5
1.4 Related Publications 5
1.5 Co-authorship 6
1.6 Outline of the Thesis 7

2 Background **8**
2.1 Code Clones 8
2.2 Types of Clones 8
2.2.1 Type 1 Clone 9
2.2.2 Type 2 Clone 9
2.2.3 Type 3 Clone 10
2.2.4 Type 4 Clone 10
2.3 Clone Detection Tools and Techniques 11
2.4 Impact of Code Clones 12
2.4.1 Clone Code is Better 13
2.4.2 Non-clone Code is Better 13
2.4.3 Neutral Results 13
2.5 Software Bugs 14
2.5.1 Bug Report 14
2.5.2 Bug Localization 14
2.6 Bug Detection Tools and Techniques 15
2.6.1 FindBugs 16
2.6.2 PMD 17
2.6.3 Comparison between Bug Detection Tools 17
2.7 Software Testing 17
2.7.1 Testing Scientific Software 18
2.7.2 GUI Testing 19
2.7.3 Test Automation 19

2.7.4 Test Data/Case Generation 20
2.7.5 Test Driven Development 20
2.8 Software Debugging 21
2.9 Analysis of Bug Detection Tools Used for Comparing Clone and Non-Clone Code 21

3 Software Bugs in Clone and Non-Clone Code 23
3.1 Introduction 23
3.2 Background 25
3.2.1 Bug-fix Commits 25
3.2.2 Severe Bugs 25
3.2.3 Bug-proneness Detection Technique 25
3.3 Experimental Steps 26
3.4 Experimental Results 27
3.4.1 *RQ 1: What percentage of files get affected because of clone and non-clone bug-fix commits?* 29
3.4.2 *RQ2: How often do bug-fix changes occur to the clone and non-clone code?* 33
3.4.3 *RQ 3: Is there any difference between the severity of the bugs occurring in clone and non-clone code?* 36
3.4.4 *RQ 4: Which types of severe and non-severe bugs can occur in code clones?* 37
3.5 Related Work 39
3.6 Limitations 40
3.7 Summary 41

4 Software Bugs in Micro Clones 42
4.1 Introduction 42
4.2 Background 44
4.2.1 Micro Clones 44
4.3 Experimental Steps 44
4.4 Experimental Results 46
4.4.1 *RQ 1: Do micro-clones contain more bug-fix changes than regular code clones?* 46
4.4.2 *RQ 2: Are the bug-fix changes consistent in micro and regular code clones?* 49
4.4.3 *RQ 3: What percentage of files get affected for fixing bugs in micro and regular clones?* 51
4.4.4 *RQ 4: Do micro-clones contain more severe bugs compared to regular code clones?* . . 54
4.5 Manual Analysis 55
4.6 Related Work 56
4.7 Limitations 58
4.8 Summary 59

5 Bug Replication in Code Clones 60
5.1 Introduction 60
5.2 Background 61
5.2.1 Similarity Preserving Co-change (SPCO) 61
5.2.2 Late Propagation in Code Clones 62
5.2.3 Late Propagation in Literature 62
5.3 Experimental Steps 64
5.4 Replicated Bugs in Code Clones 66
5.4.1 Identifying Replicated Bugs in Code Clones 66
5.4.2 Steps in Identifying Replicated Bugs in Code Clones 67
5.4.3 Manual Analysis of the SPCOs in the Bug-fix Commits 67
5.5 Experimental Results and Analysis 67
5.5.1 *RQ 1: What percentage of the clone fragments in different clone-types takes part in bug-replication?* 68
5.5.2 *RQ 2: What is the extent of bug-replication in the buggy clone classes of different types of clones?* 71

5.5.3 *RQ 3: What percentage of the bugs that were experienced by the code clones of different clone-types are replicated bugs?* . . . 74
5.5.4 *RQ 4: Which code statements are highly related to bug-replication?* . . . 75
5.6 Related Work . . . 78
5.7 Limitations . . . 78
5.8 Summary . . . 79

6 Bug Replication in Micro Clones **80**
6.1 Introduction . . . 80
6.2 Experimental Steps . . . 81
6.3 Experimental Results and Analysis . . . 84
6.3.1 *RQ 1: What percentage of the clone fragments in Regular and Micro-clones takes part in bug-replication?* . . . 84
6.3.2 *RQ 2: What is the extent of bug-replication in the buggy clone classes of Regular code clones and Micro-clones?* . . . 86
6.3.3 *RQ 3: What percentage of bugs experienced by regular and micro-clones are replicated bugs?* . . . 88
6.3.4 *RQ 4: Are the replicated bugs in micro-clones more likely to be severe than the replicated bugs in regular clones?* . . . 91
6.4 Related Work . . . 93
6.5 Limitations . . . 94
6.6 Summary . . . 94

7 Empirical Study on Bug Replication in Regular and Micro Clones **96**
7.1 Introduction . . . 96
7.2 Experiment Steps . . . 97
7.2.1 Preliminary Steps . . . 99
7.3 Analysis of Experimental Results . . . 100
7.3.1 First research question (RQ 1): What percentage of the clone fragments in Regular and Micro-clones takes part in bug-replication? . . . 100
7.3.2 Second research question (RQ 2): What is the extent of bug-replication in the buggy clone classes of Regular code clones and Micro-clones? . . . 104
7.3.3 Third research question (RQ 3): What percentage of bugs experienced by regular and micro-clones are replicated bugs? . . . 108
7.3.4 Fourth research question (RQ 4): Are the replicated bugs in micro-clones more likely to be severe than the replicated bugs in regular clones? . . . 112
7.3.5 Fifth research question (RQ 5): What is the percentage of line coverage of replicated clone fragments in both regular and micro-clones? . . . 116
7.4 Related Work . . . 122
7.5 Threats to Validity . . . 122
7.6 Conclusion . . . 123

8 Conclusion **124**
8.1 Summary . . . 124
8.2 Threats to Validity . . . 126
8.3 Future Research Direction . . . 126
8.3.1 Buggy Micro-clones . . . 126
8.3.2 Context aware bugs in clones . . . 126
8.3.3 Increasing the number of subject systems . . . 127
8.3.4 Considering Type 4 code clones . . . 127
8.3.5 Automatic tool support for recommendation . . . 127
8.3.6 Clone detection tool for micro-clones . . . 127
8.3.7 Bug tracking . . . 127
8.3.8 ML in buggy clone research . . . 128

References **129**

Appendix A Detection of Replicated Bugs in Regular and Micro-Clones **141**

List of Tables

2.1 An Example of Bug Report [195, 4] 16
2.2 Bug Detection Tools and Techniques 18
2.3 Comparing Clone and Non-Clone Code using Bug Detection Tools 22

3.1 Research Questions 25
3.2 Subject Systems 26
3.3 NiCad Settings for three types of clones 27
3.4 Number of Files that have Changed Clone Code found in Bug-fixing Commits 28
3.5 Number of Files that have Changed Non-Clone Code in Bug-fixing Commits 30
3.6 Mann-Whitney-Wilcoxon Test Result for RQ1 31
3.7 Number of Bug-fix Commits affecting Clone Code 32
3.8 Number of Bug-fix Commits affecting Non-Clone Code 34
3.9 Mann-Whitney-Wilcoxon Test Result for RQ2 35
3.10 Most frequent change types of severe and non-severe bugs in code clones 38

4.1 Research Questions 44
4.2 Subject Systems 45
4.3 Number of Bug-fix commits that have Changed Clone Fragments in Micro-clones and Regular Clones 48
4.4 Mann-Whitney-Wilcoxon Test Result for RQ1 48
4.5 Number of Consistent Changes in Regular and Micro-Clones 52
4.6 Number of Files that Changes in Regular and Micro-Clones 53
4.7 Mann-Whitney-Wilcoxon Test Result for RQ2, RQ3 and RQ4 54
4.8 Number of Severe Bugs in Regular and Micro-Clones 55
4.9 Most Frequent Change-Types During Fixing Bugs in Micro Clones 57

5.1 Research Questions 62
5.2 Subject Systems 64
5.3 NiCad Settings for three types of clones 65
5.4 Number of SPCOs experienced by different types of code clones during bug-fix commit operations 68
5.5 Number of clone genealogies related to bug-replication 69
5.6 Number of Bugs that were Replicated in Clones 76
5.7 Most frequent change-types during fixing replicated bugs in code clones 77

6.1 Research Questions 82
6.2 Subject Systems 82
6.3 NiCad Settings for Detecting Regular and Micro-clones 83
6.4 Number of Clone Fragments that Experienced Bug-replication in Regular and Micro-Clones . 86
6.5 Extent of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones . . 88
6.6 Number of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones . . 91
6.7 Number of Severe Replicated Bugs that Experienced by code clones in Regular and Micro-Clones 93
6.8 Mann-Whitney-Wilcoxon Test Result for RQ1, RQ3 and RQ4 93

7.1 Research Questions 98
7.2 Subject Systems 98
7.3 Parameters of Clone Detection Tools 99
7.4 Number of Clone Fragments that Experienced Bug-replication in Regular and Micro-Clones for NiCad clone detection tool 101
7.5 Number of Clone Fragments that Experienced Bug-replication in Regular and Micro-Clones for ConQat clone detection tool 103

7.6 Number of Clone Fragments that Experienced Bug-replication in Regular and Micro-Clones for iClones clone detection tool . . . 104
7.7 Extent of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for NiCad clone detection tool . . . 105
7.8 Extent of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for ConQat clone detection tool . . . 106
7.9 Extent of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for iClones clone detection tool . . . 106
7.10 Number of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for NiCad clone detection tool . . . 111
7.11 Number of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for ConQat clone detection tool . . . 111
7.12 Number of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for iClones clone detection tool . . . 111
7.13 Number of Severe Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for NiCad clone detection tool . . . 114
7.14 Number of Severe Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for ConQat clone detection tool . . . 115
7.15 Number of Severe Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for iClones clone detection tool . . . 115
7.16 Total Number of Lines of Code for NiCad clone detection tool . . . 118
7.17 Total Number of Lines of Code for ConQat clone detection tool . . . 118
7.18 Total Number of Lines of Code for iClones clone detection tool . . . 119
7.19 Mann-Whitney-Wilcoxon Test Result for RQ1, RQ3, RQ4 and RQ5 for NiCad, ConQat, and iClones clone detection tools . . . 121

List of Figures

3.1 Percentage of the number of files that have changed clone and non-clone fragments. 28
3.2 Percentage of bug-fix commits that have changed clone and non-clone fragments. 32

4.1 The execution flow diagram of the experimental steps in detecting bug-fix commits in regular and micro-clones 45
4.2 Percentage of bug-fix commits that have changed clone fragments in regular and micro-clones. 46
4.3 This is an example of consistent changes in micro-clones 49
4.4 Percentage of bug-fix commits that have consistent changes of clones in regular and micro-clones. 51
4.5 Percentage of files that get affected by bug-fix commits in regular and micro-clones. 53
4.6 Percentage of severe bugs in bug-fix commits in regular and micro-clones. 56

5.1 Similarity Preserving Co-change (SPCO) example 63
5.2 The execution flow diagram of the experimental steps in detecting replicated bugs in code clones 66
5.3 Percentage of clone fragments related to bug-replication with respect to all clone fragments. . 70
5.4 Percentage of clone fragments related to bug-replication with respect to all clone fragments related to bug-fix. 72
5.5 Extent of bug-replication in the buggy clone classes. 74
5.6 Percentage of replicated bugs in different types of code clones. 76

6.1 The execution flow diagram of the experimental steps in detecting replicated bug-fix commits in Regular and Micro clones 83
6.2 Percentage of clone fragments that related to bug-replication in regular and micro-clones. . . 85
6.3 Extent of bug-replication in the buggy clone classes for regular and micro-clones. 89
6.4 Percentage of replicated bugs in regular and micro-clones. 90
6.5 Percentage of severe replicated bugs in regular and micro-clones. 92

7.1 Percentage of clone fragments that related to bug-replication in regular and micro-clones for NiCad clone detection tool. 102
7.2 Percentage of clone fragments that related to bug-replication in regular and micro-clones for ConQat clone detection tool. 102
7.3 Percentage of clone fragments that related to bug-replication in regular and micro-clones for iClones clone detection tool. 103
7.4 Extent of bug-replication in the buggy clone classes for regular and micro-clones for NiCad clone detection tool. 107
7.5 Extent of bug-replication in the buggy clone classes for regular and micro-clones for ConQat clone detection tool. 107
7.6 Extent of bug-replication in the buggy clone classes for regular and micro-clones for iClones clone detection tool. 108
7.7 Percentage of replicated bugs in regular and micro-clones for NiCad clone detection tool. . . . 109
7.8 Percentage of replicated bugs in regular and micro-clones for ConQat clone detection tool. . . 110
7.9 Percentage of replicated bugs in regular and micro-clones for iClones clone detection tool. . . 110
7.10 Percentage of severe replicated bugs in regular and micro-clones for NiCad clone detection tool. 113
7.11 Percentage of severe replicated bugs in regular and micro-clones for ConQat clone detection tool. 113
7.12 Percentage of severe replicated bugs in regular and micro-clones for iClones clone detection tool. 114
7.13 Percentage of line coverage of replicated bugs compared to non-replicated bugs in code clones for regular and micro-clones for NiCad clone detection tool. 119
7.14 Percentage of line coverage of replicated bugs compared to non-replicated bugs in code clones for regular and micro-clones for ConQat clone detection tool. 120

7.15 Percentage of line coverage of replicated bugs compared to non-replicated bugs in code clones for regular and micro-clones for iClones clone detection tool. 120

A.1 A GUI interface used for detection of replicated bugs in code clones 142
A.2 A GUI interface used for detection of replicated bugs using NiCad clone detection tool 143
A.3 A GUI interface used for detection of replicated bugs using ConQat clone detection tool . . . 144
A.4 A GUI interface used for detection of replicated bugs using iClones clone detection tool . . . 145
A.5 Result showing percentage of replicated bugs (RQ3) for Carol subject system using ConQat clone detection tool 146

List of Abbreviations

LOC	Lines of Code
SPCO	Similarity Preserving Co-change
MWW	Mann-Whitney-Wilcoxon
RQ	Research Question
FC	number of Files that have Clone code
FCC	number of Files that have Changed Clone code
PFCC	Percentage of number of Files that have Changed Clone code
OPFCC	Overall Percentage of number of Files that have Changed Clone code
FNC	number of Files that have Non-clone code
FCNC	number of Files that have Changed Non-clone code
PFCNC	Percentage of number of Files that have Changed Non-clone code
OPFCNC	Overall Percentage of number of Files that have Changed Non-clone code
CC	number of Commits affecting Clone code
BCC	number of Bug-fix Commits affecting Clone code
PBCC	Percentage of Bug-fix Commits affecting Clone code
OPBCC	Overall Percentage of Bug-fix Commits affecting Clone code
CNC	number of Commits affecting Non-clone code
BCNC	number of Bug-fix Commits affecting Non-clone code
PBCNC	Percentage of Bug-fix Commits affecting Non-clone code
OPBCNC	Overall Percentage of Bug-fix Commits affecting Non-clone code
MC	Minimum number of Characters per clone line
CG	number of Clone Genealogies
CGBF	number of Clone Genealogies related to Bug-Fix
CGBR	number of Clone Genealogies related to Bug-Replication
PCGBR	Percentage of number of Clone Genealogies related to Bug-Replication with respect to all clone genealogies
OPCGBR	Overall Percentage of number of Clone Genealogies related to Bug-Replication with respect to all clone genealogies
PCGBRBF	Percentage of Clone fragments related to Bug-Replication with respect to all clone fragments related to Bug-Fix
ECC	Eligible Clone Class
CF	number of Clone Fragments in the eligible clone class (ECC)
CFRB	number of Clone Fragments that contained Replicated Bugs
EBR	Extent of Bug-Replication for a particular ECC
OEBR	Overall Extent of Bug-Replication
BFC	Bug-Fix Commit
NBC	Number of Bugs experienced by the code Clones
NBRC	Number of Bugs that were Replicated in the code Clones
BFCR	The number of distinct bug-fix commits where regular and/or micro-clones experienced similarity preserving co-changes (SPCOs)
PCFRB	Percentage of number of Clone Fragments that contained Replicated Bugs
OPCFRB	Overall Percentage of number of Clone Fragments that contained Replicated Bugs
PEBR	Percentage of Extent of Bug-Replication for a particular ECC
OPEBR	Overall Percentage of Extent of Bug-Replication for a particular ECC
PNBRC	Percentage of Number of Bugs that were Replicated in the code Clones
OPNBRC	Overall Percentage of Number of Bugs that were Replicated in the code Clones
NSBRC	Number of Severe Bugs that were Replicated in the code Clones

PNSBRC	Percentage of Number of Severe Bugs that were Replicated in the code Clones
OPNSBRC	Overall Percentage of Number of Severe Bugs that were Replicated in the code Clones
LOCCF	Total lines of code of all clone fragments
LOCCFRB	Total lines of code of all replicated clone fragments
PLC	Percentage of Line Coverage
OPLC	Overall Percentage of Line Coverage

1 Introduction

Copy-pasting source code is a common phenomenon in the daily life cycle of software development. Copying a piece of code from a code base and pasting it with (nearly similar) or without (exact similar) any modification to another or the same code base is known as code cloning [157, 151]. Two code fragments which are clones of each other are called clone pairs. A group of similar code fragments forms a clone class. Usually, the most common reasons behind code cloning is faster software development, cost reduction, program comprehension etc. For the last two decades a huge number of research [33, 35, 60, 58, 66, 88, 91, 94, 82, 95, 109, 24, 126, 128, 186, 102, 171, 84] have been done on the impacts of code clones in software systems. However, there is still debate among researchers on whether code cloning is good or bad. Many of the existing studies [33, 58, 66, 91, 94, 82, 95] on code clones show that clones have positive impacts on software development. On the other hand, a good number of studies [35, 88, 109, 60, 24, 126, 128, 102, 171, 84] showed the negative impacts of code clones. Major negative impacts are instability [126], late propagation [35], unintentional inconsistencies [60], hidden bug-propagation [24] etc. Intuitively, if a code fragment contains a bug, duplicating this piece of buggy code will increase the number of bugs in the software system. There is a number of studies [35, 131] which show that there is a direct relation between code clones and software bugs. Despite negative impacts of code clones, we can not avoid duplicate code since it is a part of the software development life cycle. Focusing on this, we suggest detecting and managing software bugs in code clones. This method will help us to take the benefit of code cloning while reducing or removing the software bugs from the code bases. Also, it is suspected that cloning is responsible for replicating bugs [157]. If developers copy and paste buggy code, the original bug may replicate throughout the code base. In this situation, we aimed to reveal and analyze the bug-replication in code clones. Furthermore, some recent studies [40, 191, 135] introduced a new term *micro-clones* which is also a new avenue of code clone research. This area is yet to be explored and none of the previous studies was conducted pointing on finding software bugs in micro-clones. Moreover, none of the studies in literature explored bug-replication in code clones. Focusing on this circumstance, we decide to reveal these area a bit more.

1.1 Motivation

In literature we have found that software bugs can introduce severe problems to software systems. There are many real world examples of incidents which have occurred due to software bugs. In 2007 a drastic train incident occurred in Tokyo, Japan due to a bug in their software system [202]. This affected 662 railway stations where more than 4,000 automatic gate machines failed to start-up. Approximately, 2.6 million passengers suffered from this incident. The program vendor took half a day to find the single line defect in the source code. Unfortunately, six days later from the same vendor another start-up failure occurred in more than 100 fare adjustment machines in 65 railway stations in Tokyo, Japan affecting 400 passengers. Upon later investigation, it was found that both incidents were similar type of source code and defects except that they had different data format. If the program vendor could have found the similar type of defective source code (buggy clone code) before the second incident, it would be possible to prevent the second incident.

Another recent vital incident occurred due to software bugs and lack of software maintenance i.e. Boeing-737 crashes. Ethiopian Airlines [20, 21, 22] (157 people died) and Lion Air Indonesia [19] (189 people died). After investigation defect was found in a software named Maneuvering Characteristics Augmentation System (MCAS). Finding software bugs and fixing them is a mandatory task in everyday software development cycle.

Considering these incidents, finding software bugs in the code base is a non-trivial task. Finding defects and duplicate defects of code is a regular software maintenance issue. Proper maintenance of the software can prevent such severe type of incidents as well as poor structure of a software system.

1.2 Problem Definition

Research Problem

Which type of source code is more bug-prone and more responsible for bug replication in system?

In this research problem *type of source code* refers to either clone code (i.e. Type 1, Type 2, Type 3), non-clone code, micro-clone code, or regular clone code. Before investigating bug-replication between these different type of source code, we need to identify and analyze bug-proneness of these source code. From the beginning of code clone research it has been a great debate whether cloning is good or bad. To mitigate this debate an enormous number of studies [78, 80, 82, 95, 147, 127, 129, 163, 164, 126] have been done on contrasting clone and non-clone code. Among these studies a considerable number of studies [164, 163, 78, 80] distinguished clone and non-clone code using bug detection tools. A common constraint of using bug detection tools is, they have considered tool generated bug reports which has only one snapshot of the last revision of each subject system. Also, the input data of their study might be biased by the bug detection tool. Considering bugs reported during the evolution of a software system through thousands of commits could improve the quality of study. Also, none of these studies [164, 163, 78, 80] considered multiple programming languages. Three studies [164, 163, 78] considered the Java programming language and one study [80] considered the C programming language. Including multiple programming languages would be more realistic research. Other common perspectives that are applied for measuring clone and non-clone code are stability, vulnerability, age or change-proneness, bug-proneness and dispersion of changes in code, software metrics, and change pattern etc. Krinke [82, 95] performed two comparative studies between clone and non-clone code, one is based on stability [82] and the other one is based on age [95]. Rahman et al. [147]

compared clone and non-clone code from the bug-proneness perspective. These three studies [82, 95, 147] found that clone code is better than non-clone code since clone code is more stable and less defect prone than non-clone code. Mondal et al. [127, 129] performed two studies, one is based on dispersion of code changes [127] and other one is based on change-proneness [129] to compare between clone and non-clone code. Both of Mondal's studies [127, 129] found that non-clone code is better than clone code. In another study Mondal et al. [126] compared clone and non-clone code by observing change patterns. Saini et al. [163] compared clone and non-clone code using software metrics. These two studies [126, 163] found that there is no significant difference between clone and non-clone code. None of the above studies compared clone and non-clone code from the bug-proneness perspective except Rahman et al. [147]. However, they [147] considered monthly snap-shots (i.e., revisions) of their systems and thus, they have the possibility of missing buggy commits. Focusing on these drawbacks of existing studies, we address the above mentioned problem through our research.

1.3 Addressing the Research Problem

To address the problem stated in previous Section 1.2, first we intend to compare the bug-proneness of clone code with non-clone code. It is important to know which category of code i.e. clone or non-clone code introduce bugs more in the systems. By knowing this we can focus on that category of code during software maintenance. After investigating, we revealed that code clones are more bug-prone than non-clone code. Since clone code are found to be more bug-prone than non-clone code, we explore further on bug-proneness of clone code. We analyze bug-fixing commits in micro-clones (micro-clone is defined in Section 4.2.1) and regular code clones. Since none of the previous research considered micro-clones, we explored it in our study. We realize that micro-clones are more bug-prone than regular code clones. We suggest to emphasize micro-clones over regular clones during clone management and software maintenance. Previous research suspected code clones are responsible for replicating software bugs in the software systems [157]. Considering this fact, we analyze bug-replication of different types of code clones (Type 1, Type 2, and Type 3) and realize that Type 2 and Type 3 code clones have higher tendencies of having replicated bugs than Type 1 clones. Also, it is very common for replicating bugs through code cloning. We suggest that Type 2 and Type 3 clones are more important for clone management. Since micro-clones are more bug-prone than regular code clones, we further analyze bug-replication in micro-clones and regular code clones. We observe that micro-clones are equally important as regular code clones for making clone management decisions. Thus, from the bug-replication perspective, we should consider both micro-clones and regular code clones during software maintenance tasks like refactoring or clone tracking. Overall, we performed 205 hours of manual analysis for all of the studies in this thesis.

1.3.1 Decomposing our Research behind Addressing the Problem

We decompose our research to address the research problem stated in Section 1.2 into four studies. These four studies are followings.

- **Study 1:** *Comparing bug-proneness in between clone and non-clone code* [76, 74].
- **Study 2:** *Comparing bug-proneness in between regular and micro-clones* [73].
- **Study 3:** *Identifying and analyzing bug-replication in code clones* [72].
- **Study 4:** *Detecting and comparing bug-replication in between regular and micro-clones* [75, 77].

A brief description of each of these studies has been given in the following subsections.

1.3.2 Study 1: Comparing Bug-proneness in between Clone and Non-Clone Code

In literature, few studies [35, 131] show the direct relationship between software bugs and code clones. We are interested in comparing between clone and non-clone code from the perspective of software bugs. To achieve this goal we have conducted an empirical study on clone and non-clone code comparing the bug-proneness. Sajnani et al. [164] compared between clone and non-clone code using bug patterns. They found that clone code contains less problematic bug patterns than non-clone code. Certain points of this paper [164] motivates us to do further research related to bug-proneness of clone and non-clone code. For instance, they have used bug reports which are reported by the bug detection tool FindBugs [5]. We proposed to use bug reports which are reported by software developers during software evolution of a software system. These include compile time errors and real time bug reports. Additionally, they have considered only Java programming language whereas we have considered two programming languages, C and Java. Considering these issues we performed a comparative study on bug-proneness in between clone and non-clone code.

1.3.3 Study 2: Finding and Comparing Bug-proneness in between Regular and Micro-Clones

As code clones appear to be more bug-prone than non-clone code, we wanted to further investigate whether regular clones or micro-clones are more bug-prone during evolution. Our results from the previous study [76, 74] indicate that we should emphasize clone code over non-clone code during software maintenance. Finding the result of our first study [74], we are inspired to investigate further on bug-proneness of clone code. We have found that in literature for the last two decades, clone research ignored small clone fragments which consist of minimum 1 to maximum 4 lines of code (LOCs). All of them considered clone fragments which consist of at least 5 LOCs or more, which we call *regular clones*. These motivate us to investigate further on small clones or *micro-clones* from the perspective of bug-proneness. We performed a comparative study on bug-proneness between micro-clones and regular code clones. To the best of our knowledge, no other study has been performed before our research on bug-proneness of micro-clones.

1.3.4 Study 3: Identifying and Analyzing Bug-replication in Code Clones

As we found code clones are more bug-prone than non-clone code during evolution, we wanted to investigate the role that bug-replication in code clones plays in the higher bug-proneness of code clones. If a particular code fragment contains a bug and a programmer copies that code fragment to several other places in the code-base without the knowledge of the existing bug, the bug in the original fragment gets replicated. Fixing such replicated bugs may require increased maintenance effort and cost for software systems. However, although cloning is suspected to be responsible for replicating bugs, there is no study on the evaluation of bug-replication through cloning. Such a study can provide us helpful insights for minimizing bug-replication as well as for prioritizing code clones for refactoring or tracking. Focusing on this we conducted an in-depth empirical study regarding bug-replication in the code clones of the major clone-types i.e. Type 1, Type 2, and Type 3.

1.3.5 Study 4: Detecting and Comparing Bug-replication in between Regular and Micro-Clones

As regular code clones have a tendency of containing replicated bugs, we wanted to investigate whether micro-clones also have such a tendency. Since our third study [72] was on bug-replication in regular code clones, we already explored that bug replication through code cloning is a common phenomenon during software maintenance and evolution. Thus, we were motivated to examine the bug-replication in both micro-clones and regular code clones and then compare between them. We substantially extend this study [75] by adding two more clone detection tools and one more research question. The extended study [77] contains revised data set and confirms that the previous study [75] was not influenced by the clone detection tool. To the best of our knowledge, none of the previous research has been performed to identify and analyze bug-replication in micro-clones.

1.4 Related Publications

This section contains the list of publications that came out of this thesis research. I am the primary author in each of these publications. The research behind each publication was conducted under the supervision of Dr. Chanchal K. Roy.

Refereed Journal Contributions

- **Judith F. Islam** and Chanchal K. Roy "An Empirical Study on Bug-replication in Regular and Micro-Clones", Journal of Systems and Software (JSS), 2020 (under review).

- **Judith F. Islam**, Manishankar Mondal, Chanchal K. Roy, Kevin A. Schneider, "Comparing Software Bugs in Clone and Non-Clone Code: An Empirical Study", International Journal on Software Engineering & Knowledge Engineering (IJSEKE), Vol. 27 (09n10): 1507-1527, November 2017.

Refereed Conference Contributions

- **Judith F. Islam**, Manishankar Mondal, Chanchal K. Roy, Kevin A. Schneider "Comparing Bug Replication in Regular and Micro Code Clones", In Proceedings of the 27th IEEE/ACM International Conference on Program Comprehension (ICPC), pp. 81-92, Montreal, Canada, May 2019.

- **Judith F. Islam**, Manishankar Mondal, Chanchal K. Roy, "A Comparative Study of Software Bugs in Micro-clones and Regular Code Clones", In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 73-83, Hangzhou, China, February 2019.

- **Judith F. Islam**, Manishankar Mondal, Chanchal K. Roy, Kevin A. Schneider, "A Comparative Study of Software Bugs in Clone and Non-Clone Code", In Proceedings of the 29th International Conference on Software Engineering & Knowledge Engineering (SEKE), pp. 436-443, Pittsburgh, USA, July 2017 (Best Paper Award).

- **Judith F. Islam**, Manishankar Mondal, Chanchal K. Roy, "Bug Replication in Code Clones: An Empirical Study", In Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 68-78, Osaka, Japan, March 2016.

1.5 Co-authorship

The research presented in this thesis is my own and this research has been conducted under supervision of Dr. Chanchal K. Roy. I have cited extensively those works which are not my idea and techniques. A number of my publications presented in this thesis were performed collaboratively. I am sole contributor in data generation, implementation, analysis of the results, and writing of the papers. The co-authors helped in discussing the ideas, correcting the flaws in methodology and write-ups.

In all my publications [76, 74, 73, 72, 75, 77], I have used the SPCP-Miner [137] tool to perform the first few steps of my experiments. Manishankar Mondal was the original author of SPCP-Miner [137] tool. I had to extend further steps substantially to complete my studies. Dr. Kevin A. Schneider contributed by providing suggestions for improvements and proof-reading of the papers. Other than these contributions of the co-authors of my papers, the entire work presented in this thesis is the result of my own research performed under the supervision of Dr. Chanchal K. Roy.

1.6 Outline of the Thesis

The chapters of this thesis are organized as follows.

- Chapter 1 introduces research problem and shortly describes our research to address those problems.
- Chapter 2 describes the essential background knowledge on code clone and software bugs.
- In Chapter 3, we described our first research study on comparing bug-proneness of clone code and non-clone code.
- We described our second study in Chapter 4 where we analyzed bug-proneness of micro-clones and regular code clones.
- Chapter 5 presents our third research study on bug-replication of different types of clone code.
- Chapter 6 provides our fourth research study which analyzes bug-replication in micro-clones and regular code clones.
- Chapter 7 is a significant extension of our previous study in Chapter 6. We compared replicated bugs between regular and micro-clones using three clone detection tools along with a revised data set in this chapter.
- Finally, Chapter 8 concludes this thesis along with some plausible future paths in this field of research.

2 Background

We demonstrate the basic terminology on clone code in this chapter. We also elaborate on some background knowledge regarding both positive and negative impacts of clone code, and several clone detection tools and techniques. The rest of this chapter are organized as follows. Section 2.1 describes some terminology of clone code, different types of code clones are defined in Section 2.2, some state-of-the-art clone detection tools are described in Section 2.3, impacts of code clones are given in Section 2.4, Section 2.5 describes some background knowledge on software bugs, several bug detection tools and techniques are described in Section 2.6, software testing, verification and validation are illustrated in Section 2.7, different processes of software debugging in literature are discussed in Section 2.8, and this chapter is concluded by analyzing bug detection tools which are used for comparing between clone and non-clone code in Section 2.9.

2.1 Code Clones

First, we define the most common terminologies for code clone research in this section.

- **Code Fragment** A series of source code lines between the start line and end line is referred to as a code fragment. A code fragment indicated by the start line and end line of the source code.
- **Code Clone** Two code fragments are defined as clone code if they are similar with each other to some given extent.
- **Clone Pair** Two code fragments form a clone pair if they are similar enough according to a given definition of similarity.
- **Clone Class** Two or more code fragments which are all similar to each other to a given extent, is a clone class.

2.2 Types of Clones

There are four types of clones. We conduct our analysis considering both exact (Type 1) and near-miss clones (Type 2 and Type 3 clones) [157, 151]. The clone-types are defined below.

- **Type 1 Clones** If two or more code fragments in a particular code-base are exactly the same disregarding the comments, white space, and indentations, these code fragments are called exact clones or Type 1 clones of one another.

- **Type 2 Clones** Type 2 clones are syntactically similar code fragments in a code-base with the only variation in renaming of identifiers and/or changing of data types.

- **Type 3 Clones** Type 3 clones are syntactically similar, but can differ by additions, deletions, or modifications of lines. Type 3 clones are also known as *gapped* clones.

- **Type 4 Clones** Type 4 clones are possibly syntactically dissimilar code fragments that implement the same functionality. Type 4 clones are also known as *semantic clones*.

For better perception, we illustrate in the following subsections individual example of each code clone type.

2.2.1 Type 1 Clone

Listing 2.1 and 2.2 is an instance of Type 1 code clones. Listing 2.2 is a clone code of Listing 2.1. The only change in this instance is, Comment1 and Comment2 of code 1 are given in a different line in code 2. Comment1' and Comment2' of code 2 are the replacement of Comment1 and Comment2. Thus both the code fragments, code 1 and code 2 from Listing 2.1 and Listing 2.2 are exact copy of each other. This is defined as a Type 1 code clone.

Listing 2.1: code 1 (Type 1)

```
if (a >= b) {
      c = d + b; // Comment1
      d = d + 1;}
else
      c = d - a; //Comment2
```

Listing 2.2: code 2 (Type 1)

```
if (a>=b) {
      // Comment1'
      c=d+b;
      d=d+1;}
else // Comment2'
      c=d-a;
```

2.2.2 Type 2 Clone

Listing 2.3 and 2.4 is an example of a Type 2 code clone. Here all the identifiers' names of code 1 in Listing 2.3 are changed in code 2 of Listing 2.4. Other than changing identifiers' names there is no other syntactical changes. Code 2 of Listing 2.4 is a Type 2 code clone of code 1 in Listing 2.3.

Listing 2.3: code 1 (Type 2)

```
if (a >= b) {
      c = d + b; // Comment1
      d = d + 1;}
else
       c = d - a; //Comment2
```

Listing 2.4: code 2 (Type 2)

```
if (m >= n)
      { // Comment1'
      y = x + n;
      x = x + 1; //Comment3
      }
else
      y = x - m; //Comment2'
```

2.2.3 Type 3 Clone

Type 3 clones are also known as *gapped clones*. Listing 2.5 and 2.6 is an instance of a Type 3 code clone. Code 1 of Listing 2.5 and code 2 of Listing 2.6 are identical except that one new statement i.e. *e = 1;* has been added to code 2 of Listing 2.6. Any addition, deletion or modification of any statement can occur in Type 3 code clone.

Listing 2.5: code 1 (Type 3)

```
if (a >= b) {
      c = d + b; // Comment1
      d = d + 1;}
else
      c = d - a; //Comment2
```

Listing 2.6: code 2 (Type 3)

```
if (a >= b) {
  c = d + b; // Comment1
  e = 1;//This statement is added
  d = d + 1; }
else
     c = d - a; //Comment2
```

2.2.4 Type 4 Clone

In a Type 4 code clone pair, one code fragment is semantically identical with other code fragment. Type 4 clones are also known as *semantic clones*. Listing 2.7 and 2.8 is an example of Type 4 code clones. They perform identical functionality. Here, code 1 in Listing 2.7 and code 2 in Listing 2.8 both are finding the factorial value of a given number, but in different way. Among the four types of code clones, Type 4 code clones are the most challenging for both detecting and analyzing. Very few code clone detection tools can find Type 4 code clones. More research is needed on Type 4 code clones to manage clones.

Listing 2.7: code 1 (Type 4)

```
int i, j=1;
      for (i=1; i<=VALUE; i++)
            j=j*i;
```

Listing 2.8: code 2 (Type 4)

```
int factorial(int n) {
  if (n == 0)
    return 1 ;
  else
    return n * factorial(n-1);
}
```

2.3 Clone Detection Tools and Techniques

A great number of tools for clone detection exist. The most prominent tools are CP-Miner [104, 105], Deckard [83], CCFinder [89], NiCad [153, 49]. Deckard [83] is a tree based code clone detection tool. The algorithm of Deckard has the flexibility to work with different sizes of code fragments and it is language independent. Characteristic vectors are their novel technique for detecting similar trees. This clone detection tool has been used in context-based detection of buggy clones [83]. They have compared their clone detection tool [83] with CloneDR [38] and CP-Miner [104, 105]. They consider both clone quantity (number of detected clones) and clone quality (number of false clones) as the features to compare with other tools and found that Deckard is better than both CloneDR and CP-Miner. Deckard detects more clones than both CloneDR and CP-Miner. They have noted that CP-Miner does not work if the distance metric is more than one. The only risk in their research is they inspected 100 randomly selected rcAN sets reported by Deckard and this might give different results for different sampling selection. A number of clone detection tools emphasized the detection of incremental code clones such as iClones [59], Hummel's Index-Based [70], and Higo's PDG-based [65] clone detection tools. Few studies [48, 190, 28, 140] focused on detection of Type 3 clones also known as near-miss clones or gapped clones. Murakami et al. [140] proposed a tool named CDSW to detect gapped clones using the Smith-Waterman algorithm.

A lightweight clone detection tool, named NiCad [153, 49] is provided using parser based textual line comparison by Roy and Cordy. An in depth comparison of the quality of a number of clone detection tools from different aspects is presented by Roy et al. [156]. Solanki et al. [169] recently performed a short comparative study on different code clone detection tools and techniques. Very few code clone detection tools can detect clones across multiple programming languages. Such a tool LICCA was recently proposed by Vislavski et al. [192]. LICCA is integrated with the SSQSA platform and is dependent on high-level representation of source code. A PDG based clone detection tool, Scorpio, has been proposed by Higo et al. [62, 63] which can detect both contiguous and non-contiguous clones.

To improve the existing clone detection tools, Svajlenko et al. [174] intended a shuffling framework which makes the clone detection tool scalable with high recall (later, they extended their study [173]). A token-based large scale inter-project clone detection tool, SourcererCC, proposed by Sajnani et al. [165]. An optimized inverted-index has been used to detect code clones in code blocks. They have used a heuristic filter based on the order of tokens, so that the size of the index, number of code-block comparisons and token comparisons have been reduced drastically. The authors of this paper compared their tool with CCFinderX [89], Deckard [83], iClones [59] and NiCad [49] to measure the precision manually. To calculate recall they have used two benchmarks, BigCloneBench [172, 177] and a Mutation/Injection-based framework [154, 155, 182, 181]. SourcererCC clone detection tool has high precision and recall value and it is scalable to large scale database.

One of the most recent clone detection tools is CloneWorks [179], [178] presented by Svajlenko and Roy which is able to handle big data by scaling and faster than any previous tools. Wang et al. [193] describes large-gap clones as clones having a large number of edits. Their token based tool, CCAligner, can detect large-gap clones as well as general types of clones with high precision and recall compared to other state-of-the art clone code detectors. They have designed a e-mismatch index and used asymmetric similarity coefficient for similarity measurement for 10MLOC input. Livieri et al. [108] used distributed CCFinder (D-CCFinder) to analyze code clones in very large scale.

In literature, a number of clone benchmarks exist using manual [96, 44, 52, 159] and automatic [203, 99, 27] validation to evaluate clone detection tools. Svajlenko and Roy [175, 176] evaluated eleven clone detection tools using four benchmark frameworks including their own. Their framework BigCloneEval [180] can automatically evaluate and compare clone detection tools.

2.4 Impact of Code Clones

It has been a controversial debate for the last two decades among researchers and software developers as to whether or not code clones are harmful for software systems. Initial research indicated code clones as the main reason for producing bugs in the source code of a system. However, later researchers showed that consistent code cloning can be helpful to reduce the clone maintenance cost and effort. A substantial number of studies [33, 35, 60, 58, 66, 88, 91, 94, 82, 95, 109, 24, 126, 128, 186, 102, 171, 84] have been conducted on discovering the impact of cloning on software maintenance. While a number of studies [33, 58, 66, 91, 94, 82, 95] have revealed some positive sides of code cloning, there is strong empirical evidence [35, 88, 109, 60, 24, 126, 128, 102, 171, 84, 198, 110] of negative impacts of code clones too. These negative impacts include higher instability [126], late propagation [35], and unintentional inconsistencies [60]. Existing studies [35, 131] show that code clones are related to bugs in the code-base.

To mitigate the debate on clone and non-clone code an enormous number of studies have been done on contrasting clone and non-clone code. Most common perspectives that are applied for measuring clone and non-clone code are stability, vulnerability, age or change-proneness, bug-proneness and dispersion of changes in code.

2.4.1 Clone Code is Better

Krinke [82, 95] performed two comparative studies between clone and non-clone code. The first study [82] has been conducted to compare stability of clone and non-clone code. Working on five software systems they have revealed that cloned code is more stable than non-cloned code only after excluding the deletion of cloned code. The second study [95] investigated clone and non-clone code from an age perspective. They found that cloned code is on average older than non-cloned code. This paper [95] has supported the first findings [82] in a sense that the longer the code remains unchanged the more stable it is. In other words, change-proneness of code makes them unstable. Hotta et al. [66, 67] found that clone code are less frequently modified compared to non-clone code and implied that clone code are more stable. Rahman et al. [147] found in their study that clone code is less bug-prone than non-clone code.

2.4.2 Non-clone Code is Better

On the contrary to the previous section, clone code could be detrimental in many cases. These are highlighted in many research works such as Mondal et al. [127] who measured dispersion of changes in code by extracting method genealogies using a framework. More dispersed changes in a code base refers to instabilities of the system as well as requiring more maintenance efforts. They have found that changes in clone code is more dispersed than changes in non-clone code. Thus they suggested that cloned code requires more maintenance effort than non-cloned code. In other words, non-clone code is more stable than clone code. Later, this study has been extended [129] by increasing the number of subject systems. They have found statistically significant correlation between change dispersion and change-proneness, also known as instability of source code. Source code which have more change dispersion is more change-prone. They have revealed that clone code has more dispersed changes thus they are more change-prone i.e. instable. We have performed a comparative study [76] in between clone and non-clone code from the bug-proneness perspective. From this study [76], we have found that clone code appears to be more bug-prone than the non-clone code. Later, we have extended our study [74] by analyzing the types of severe bugs which occur in clone and non-clone code.

2.4.3 Neutral Results

Many studies found neutral results after comparing clone and non-clone code from diverse perspective. For instance, Saini et al. [163] found a statistically insignificant difference between quality of cloned and non-cloned code in granularity as the method level of open source Java projects. They have used 27 software metrics to measure the quality and this result is true for 24 metrics. Although, there is a difference in size of the methods between clone and non-clone code. According to their study, clone methods are 20% smaller than the non-clone methods on average. From another perspective, measuring the stability as well as changeability of source code, Mondal et al. [126] found that there is no consistent change patterns between clone and non-clone code.

2.5 Software Bugs

Faulty code or buggy code are very prevalent in software systems. Over the last few decades, a vast amount of research has been done on bug detection, bug triaging, bug localization and so on. Detecting bugs and fixing them immediately can minimize software maintenance costs drastically. When a bug is found in a software system a bug report is generated by the programmers. An extensive number of bugs are reported in large open source projects through Issue Tracking Systems (ITS) such as BugZilla [1], Mantis [12], JIRA [7] etc. In order to fix a bug, bug reports are distributed to the appropriate developer through a bug triaging process. To fix a bug, developers may need to make changes to files. These changes can be diverse, from small changes such as a simple typo correction, to a large number of file changes. Bug-fix changes can be made by editing, removing, renaming, and copying existing files and/or adding new files to the project repository. Some background knowledge on software bugs is discussed in this section, along with bug reports and bug localization.

2.5.1 Bug Report

A bug report is submitted by a user which contains the description of the problem that they are dealing with [195]. The major components of a bug report are bug id, a short description of the problem, a detailed description of the problem, and a date when bug is reported. Table 2.1 shows an example of a bug report. Here, bug ID is a unique number which is used to identify the bug and is also used in version control systems to identify the commit which fixed that bug. The summary and description of the bug report helps to understand the problem. Fixed files show the list of files that contain affected source code by the bug and later fixed. Bug reports are categorized in several levels (e.g. minor, regular, severe etc) to mark the importance of that report. Developers can prioritize bug reports using these categories. Intuitively, severe bugs or highly prioritized bugs need to be fixed urgently.

2.5.2 Bug Localization

After receiving a bug report, locating the bug in the code base for debugging is a time-consuming task. To locate the bug manually is often a painstaking task and not cost effective [206]. To leverage this problem, there are a number of automatic bug localization tools that exist, such as AmaLgam+ [195], BLUiR+ [161], BRtracer+ [199], BugLocator [206], and TFIDF-DHbPd [161]. Localizing a bug is a prerequisite of software debugging.

There are two basic categories of bug localization techniques. One is Spectrum-based, also known as dynamic technique, and the other one is Information Retrieval (IR)-based or static technique [100].

Spectrum-based Technique

Dynamic technique uses program spectra on passed and failed executions of programs. Then they compute the suspiciousness and rank the program elements. The most prominent spectra-based techniques are [86, 25, 112, 113, 114, 106, 107, 31, 32, 46, 204, 205].

Information Retrieval (IR)-based Technique

Information retrieval technique calculates the similarity between a bug report and the source code. A number of prominent IR-based approaches are found in the literature [149, 115, 101, 168, 206, 161, 194, 196, 201]. These techniques use different models to perform bug localization. To name few of them are Latent Dirichlet Allocation (LDA) [115], Latent Semantic Indexing (LSI) [145, 146], Latent Semantic Analysis (LSA), Smoothed Unigram Model (SUM) [149], rVSM [206] etc.

Bug Localization Process

Generally, there are fours steps of processing bug localization. These are corpus creation, indexing, query construction, and retrieval & ranking [206].

- **Corpus creation** In the first step, a lexical analysis is performed on each source code file. Removal of keywords, operators, separators and English stop words are performed. Splitting individual tokens into two separate words and stemming a token to find its root is done in this first step.
- **Indexing** In this step all the files are indexed so that one can find a particular file using a query and rank them.
- **Query Construction** Bug reports are used as the query of bug localization. To do this task, one needs to transform a bug report into a query. To construct the query, extraction of tokens from the bug report, removing stop words, and stemming is needed.
- **Retrieval and Ranking** In the last step, based on the relevance (textual similarity) of the query search result each retrieved file is ranked from the corpus.

2.6 Bug Detection Tools and Techniques

Finding software bugs in a subject system is important from a management and maintenance perspective so that the bugs can be fixed. Detecting bugs in software systems is a common phenomenon in software development cycle. It is mostly done automatically through the bug detection tools. A number of bug detection tools have been created. Some of the most prominent bug finding tools are FindBugs [5], [68], PMD [13], JLint [30], [8], Bandera [47] and ESC/Java [53]. Another recent approach [189] proposed an automatic solution for detecting *performance bugs.* They determined that by reducing the number of changes on a

Table 2.1: An Example of Bug Report [195, 4]

Bug ID	76138
Open Date	2004-10-12 21:53:00
Summary	**Ant editor** not following tab/space setting on shift right
Description	This is from 3.1 M2. I have **Ant**→**Editor**→Display tab width set to 2, "insert spaces for tab when typing" checked. I also have **Ant**→**Editor**→Formatter→Tab size set to 2, and "Use tab character instead of spaces" _unchecked_. Now when I open a build.xml and try to do some indentation, everything works fine according to the above settings, except when I highlight a block and press tab to indent it. It's the tab character instead of 2 spaces that's inserted in this case.
Fixed Files	org.eclipse.ant.internal.ui.editor.**AntEditor**.java org.eclipse.ant.internal.ui.editor.**AntEditorSourceViewer Configuration**.java

commit, it can decrease the occurrences of performance bugs. On the other hand, Lu et al. [111] proposed an automatic method named MUVI to detect *semantic and concurrency bugs.* In another study, Shi et al. [167] implemented a bug detection tool called DefUse to automatically detect *concurrency bugs* as well as *memory and semantic bugs.* Below I have shortly described the two most prominent bug detection tools.

2.6.1 FindBugs

FindBugs [5] is the most widely used bug detection tool in Java. Hovemeyer and Pugh [68] observed bug patterns in Java application and libraries to detect serious bugs. They found that an automatic approach to detecting bugs can effectively prevent mistakes and incorrectly using language features. Later they [69] have improved their tool by finding more null pointer bugs. Ayewah et al. [34] evaluated output warnings of FindBugs in three categories i.e. false positives, trivial bugs and serious bugs. Recently, Al-Ameen et al. [26] proposed an improved version of FindBugs and comparing with PMD [13] showed that their approach can find more bugs than the previous version. However, FindBugs detects many false positives thus overlooks real bugs [184]. On the other hand, our approach does not rely on predefined bug patterns and works with real bug reports.

2.6.2 PMD

PMD is a source code analyzer which detects common programming flaws in Java. Additionally, it has the copy paste detector (CPD) which detects duplicate code in Java, C, C++, C#, Groovy, PHP, Ruby, Fortran, JavaScript, PLSQL, Apache Velocity, Scala, Objective C, Matlab, Python, Go, Swift and Salesforce.com Apex and Visualforce [13].

2.6.3 Comparison between Bug Detection Tools

Research has been conducted comparing different bug detection tools from different perspectives, such as performance. Ruter et al. [158] studied five bug detection tools, PMD, FindBugs, JLint, ESC/Java, Bandera and compared their performance and discussed the pros and cons of these five bug detection tools. Also, Ruter et al. provided a meta-tool which combines these tools' output to find the most warned Lines of Code (LOCs)/methods/classes. Another comparison has been done later in 2011 by Araújo et al. [29]. Araújo et al. found that 50% of reported warnings by FindBugs are relevant, whereas only 10% of reported warnings by PMD are relevant. On the other hand, Thung et al. [188] discussed false negative results of three bug finding tools, FindBugs, JLint and PMD. They have found that FindBugs and PMD have less false negatives than JLint whereas JLint warns about fewer LOCs than the other two bug finding tools. In recent research, Jindal et al. [85] studied the efficiency of PMD and FindBugs based on prediction of refactoring. They found that PMD outperformed FindBugs by 81% for detecting refactoring in three different projects. Recently, another method for finding bugs in source code as well as fixing them was proposed by Ray et al. [150]. Additionally, their technique can evaluate other bug detection tools like PMD and FindBugs. This paper defines the unpredictability of bugs as "unnaturalness of software" or "entropy". They have identified bug-fixing commits using heuristic search by keywords i.e. entropy, in the commit messages and they have reported that their method is 96% correct.

Table 2.2 summarizes all the state-of-the-art bug detection tools and techniques. FindBugs and JLint work on bytecode and the rest of the bug detection tools take source code as their input. A command Line interface is available for all of the bug detection tools and techniques. Most tools have other interfaces available, such as a GUI, an IDE, or Apache Ant (a software tool for automating the software build process). Both FindBugs and JLint check syntactically and use dataflow analysis, but other tools use model checking, heuristic search, and theorem proving.

2.7 Software Testing

To improve software quality and remove bugs from software systems, software testing, verification and validation are mandatory tasks. The process of applying metrics to deduce the quality of a product is known as software testing [185]. The software testing process can be defined using different models such as TMMi

Table 2.2: Bug Detection Tools and Techniques

SL	Tool	Input	Interface	Technology	Language
1.	FindBugs [5]	Bytecode	Command Line, GUI, IDE, Ant	Syntax, dataflow	Java
2.	PMD [13]	Source code	Command Line, GUI, IDE, Ant	Syntax	Java
3.	JLint [30], [8]	Bytecode	Command Line	Syntax, dataflow	Java
4.	Bandera [47]	Source code	Command Line, GUI	Model checking	Java
5.	ESC/Java [53]	Source code	Command Line, GUI	Theorem proving	Java
6.	Ray's Approach [189]	Source code	Command Line	Heuristic search	Java

[54], ISTQB [81]. Also, there is a standard model for software testing process defined in ISO/IEC 29119 [23]. Precise standards for software verification and validation are provided in [17, 18]. In the software development cycle the most expensive step is software testing which causes almost 50% of the total costs [92]. In 2002, the United States economy estimated software bugs cost \$59.5 billion per year, and an improved testing process could reduce the cost by \$22.5 billion per year [185]. Higher quality of the testing process ensures higher quality bug free software. A good testing process (either manual and/or automatic) could find a higher number of bugs in software systems.

There are different areas of software testing research; the major categories are described below.

2.7.1 Testing Scientific Software

Testing scientific software is a challenging task since the characteristics of scientific software is different and there is a cultural difference between scientists and the software engineering community [90]. Kanewala and Bieman [90] investigated this problem and performed a Systematic Literature Review (SLR) on 62 studies showing that systematic testing can be useful for detecting software faults (software bugs) and that code clone detection techniques could be used to improve the testing procedure. In the following year (2015), another study [50] on scientific software product line (SSPL) used for bioinformatics showed that similar software products can be used to predict the commonalities and differences in them. Later, common features can be reused to support new bioinformatics software development. According to Costa et al. [50], the users of SSPL are usually scientists who do not have much training in software development. Scientists often duplicate the existing application to fulfill their own requirements while building their frameworks [50]. So software product line (SPL) developers can help SSPL users as well as scientists to follow a model to specify the product line (PL) while making decisions according to their requirements. Both of these studies [90, 50] required code clone detection, and a systematic literature survey [90] showed the benefits of software testing to detect software bugs.

2.7.2 GUI Testing

Focusing on semantic functionalities Mariani et al. [120] proposed "Augusto (AUtomatic GUi Semantic Testing and Oracles)", which automatically generates test cases using previous knowledge on semantics of three common Application Independent Functionalities (AIFs) such as CRUD (creating, removing, updating, and deleting), AUTH (sign up, signing in and signing out from applications), SAVE (saving data in files and loading them) operations. Their empirical study on seven Applications Under Test (AUTs) shows that Augusto outperformed two other testing tools i.e. GUITAR [143] and ABT (AutoBlackTest) [118, 119]. At the same time, Pezzè et al. [144] performed a comparative study on state-of-the-art automated GUI testing tools which focus on major desktop applications. They studied seven (two Testars, three Guitars, ABT, and Augusto) GUI testing tools. They found that tools which used a random technique (e.g. Testar-Random and ABT) have higher coverage than those used structured technique. Moreover, they revealed that Augusto [120] detected most faults due to its precise automated oracle. In another research, Brooks and Memon [43] proposed an automated GUI testing which uses "usage profiles". They have used reverse engineering methods to extract the structure of the application. In a previous study Memon et al. [122] introduced Planning Assisted Tester for grapHical user interface Systems (PATHS) to automatically generate test cases for GUI applications using planning. They have created a hierarchical GUI model based on its structure. In our research, we have dealt with the bugs which are related to GUI or generated in GUI operations. However, we did not separate GUI related bugs with other bugs.

2.7.3 Test Automation

The process of creating a mechanically explicable representation of a manual test case is known as test automation [187]. Test automation is a comparatively new avenue of research in the field of software testing. Garousi and Mäntylä [56] performed a Multivocal Literature Review on test automation, showing that books on test automation started publishing in 2010 whereas all other testing books started publishing before 1992. They [56] suggested to follow a systematic empirical-validated decision-support approach during development to avoid crucial incorrect expenditures i.e. resources and efforts. In a previous study, Thummalapenta et al. [187] proposed a novel algorithm for automating tests from a sequence of natural language test steps describing actions on targets. Their algorithm uses backtracking to resolve ambiguities in the actions, step order, and targets, and was able to automate over 82% of 293 steps.

Whether automation of testing is necessary or not for different cases is a crucial decision for developers.

To help make the right decision Sahaf et al. [162] proposed a decision support simulation model. They used a system dynamics modeling technique, conducted action research with a software company in Calgary, Canada, and were able to improve the cost effective planning of selected test activities for that company. Recently, an automated testing tool (i.e., Sungari) was developed by Gao et al. [55] for blockchain-based decentralized applications. Their approach is novel in combining both front-end and back-end testing. They used random events to infer abstract relationships between browser events and blockchain smart contracts, generate a set of test cases base on a read-write graph, and use taint analysis to track data flow of the smart contracts. Harman et al. [61] proposed a tool named "FiFiVerify" (Find, Fix, Verify) which is a Search Based Software Testing (SBST) tool. Their tool automatically finds bugs, fixes them and also verifies the fixes. They focused on non-functional properties of testing such as execution time, security, usability, efficiency, QoS etc.

Currently, there are a number of test automation tools available for Android applications (apps). The most commonly used Android app testing tools are JUnit, Monkeyrunner, Robotium, Robolectric, etc [93]. Mao et al. [117] introduced Sapienz, a multi-objective search-based automatic testing tool only for Android applications. Their tool significantly outperforms two other state-of-the-art testing tools i.e. Dynodroid [183] and Google's Android Monkey [6]. Sapienz [117] compared three key features i.e. code coverage, fault detection, and fault-revealing sequence length with the other two tools. However, industries are still heavily (86%) practicing manual testing for Android apps [93].

2.7.4 Test Data/Case Generation

Generating test case scenarios or test data has an important role in the testing software procedure. The correctness of testing using both manual and automated methods might vary due to the quality of test data. A popular method of test data generation is using meta-heuristic search technique. A meta-heuristic search is a high level framework which uses a general heuristic search method to find solutions for complex problems without compromising computational costs [121]. Phil McMinn [121] performed a survey on test data generation using meta-heuristic search method for structural testing, functional and non-functional testing, and grey-box properties testing.

2.7.5 Test Driven Development

At present software development industries are using agile methods significantly. The test-driven development (TDD) process and refactoring code are the major key factors to success in agile methods. Nanthaamornphong and Carver [142] explored the effectiveness of TDD in scientific software. In their survey [142], they found that TDD has both advantages and disadvantages for scientific software development. Positive results are that TDD improves the quality of software and it reduces problems in early stages. On the other hand, negative results show that TDD does not perform well for all scientific software, and for parallel computing it is difficult to write good test cases.

2.8 Software Debugging

The process of detecting defects in software systems and resolving them can be defined as software debugging [170]. The debugging process is both time consuming and costly if it is done manually. However, the effectiveness of automated debugging is still not well understood. To shed light on this dichotomy, Böhme et al. [42] presented a database called DBGBench. They surveyed 12 professional software engineers using 27 real bugs to study how developer localize, diagnose and fix bugs. DBGBench [42] will help to evaluate the effectiveness to automate the process of bug localization, diagnose and fixing. The following year, another study [39] explored the behavior of programmers regarding debugging activities. Beller et al. [39] conducted an online survey of 176 developers' debugging decision. Based on the data of this online survey, they observed how 458 developers used the debugger. A built-in plugin named WatchDog 2.0 was provided to the developers [39]. They found that developers often preferred print statements over the automated debugger. Additionally, there was a lack of use of advanced debugging features and knowledge of debugging among the developers.

ENLIGHTEN is an interactive and feedback-driven fault localization tool proposed by Li et al. [103] to automate the debugging process. They have found that debugging using ENLIGHTEN performed substantially better compared to without using it, although they tested only four faults with 24 participants. Increasing the number of faults or bugs might change the outcome. Recently, Motwani et al. [139] investigated automatic program repair techniques as to whether these techniques are capable of repairing hard and important bugs or not. In [139], they used 409 real bugs written in C and Java as their benchmarks and used seven popular automated repair techniques, AE, GenProg, Kali, Nopol, Prophet, SPR, and TrpAutoRepair. They found that automatic repair techniques produce less patches when developers have to deal with a high volume of code or files or tests regarding the defect. Additionally, automated repair techniques are not particularly effective for fixing defects which required addition of loops, new function calls, or changing method signatures [139]. Last year (2019), an extensive survey on 108 papers was published as a journal paper by Gazzola et al. [57] regarding automatic software repair. They compared all the state-of-the-art techniques and tools of automated software repair and discussed the open challenges.

2.9 Analysis of Bug Detection Tools Used for Comparing Clone and Non-Clone Code

Many different studies have used bug detection tools for comparing clone and non-clone code. Shajnani et al. [164] performed a comparative study between clone and non-clone code to determine different bug patterns. They have used bug reports generated by the bug detection tool FindBugs. They have found that cloned code has less problematic bug patterns than non-cloned code. One of the caveats of their study is that they have considered bugs reported by FindBugs from only one snapshot of the last revision of the subject systems. Considering bugs reported during the evolution of a software system through thousands of commits

Table 2.3: Comparing Clone and Non-Clone Code using Bug Detection Tools

Cite	**Name of Tool(s)**	**Comparison**	**Feature**	**Language**
[164]	FindBugs	Cloned Code<Non-Cloned Code	Bug Patterns	Java
[78]	PMD	Cloned Code=Non-Cloned Code	Vulnerabilities of Source Code	Java
[80]	Flawfinder and Cppcheck	Cloned Code>Non-Cloned Code	Security Vulnerabilities of Source Code	C
"Cloned Code<Non-Cloned Code" means Cloned Code is less problematic than Non-Cloned Code according to the authors of that study.				

and bugs reported by software developers in several open source projects could improve the quality of the study. Also, their study was narrowed down to only one programming language, namely Java. Including other programming languages would be more useful and generalizable. Two years later they compared clone and non-clone code using various software metrics [163], discussed in detail in Section 2.4.

Another study [78] used the PMD (version 5.3.2) bug detection tool to find vulnerabilities in source code for comparison purposes. To compare between categories of clone and non-clone code, MD Islam et al. [78] examined vulnerabilities of source code of 97 open source software systems. They established that there is not much difference between clone and non-clone code. As a continuation of this research, the following year they observed security vulnerabilities along with their severity, comparing different types of clone and non-clone code [80]. They used one code clone detector, NiCad (version 3.5), and two security vulnerability detection tools, Flawfinder (version 1.3) and Cppcheck (version 1.76.1), to detect bugs in 8.7 million lines of code (LOC) from 34 subject systems. The authors of this paper reveal that code clones have higher severity of security risks than non-clone code. On the other hand, the density of vulnerabilities have no substantial difference between clone and non-clone code. This is similar to their previous result.

Table 2.3 shows the overall picture of different bug detection tools, and a comparison of clone and non-clone code along with their citations. Here, "greater than (>)" symbol in column "Comparison" in Table 2.3 refers to "more problematic than" and vice versa. For example, "Cloned Code<Non-Cloned Code" refers to cloned code being less problematic than non-cloned code according to the authors of that particular study. And "equal to (=)" refers to there being not much difference between clone code and non-clone code. Also, different studies worked on different perspectives, and these are given in column "Feature".

3 Software Bugs in Clone and Non-Clone Code

In this chapter, we describe our first study on comparing bug-proneness of clone and non-clone code. We conduct a comparative study on bug-proneness in clone code and non-clone code by analyzing commit logs. We investigate thousands of revisions of seven diverse subject systems. We also perform a Mann-Whitney-Wilcoxon (MWW) test to show the statistical significance of our findings. The idea is that if clone code is more bug-prone than non-clone code then we have to place additional emphasis on clone code during software maintenance tasks. More bug-prone code is more important since this may cause vulnerability in systems. We believe that our findings are important for better maintenance of software systems.

Code cloning is a recurrent operation in everyday software development. Whether it is a good or bad practice is an ongoing debate among researchers and developers for the last few decades. In this chapter, we conduct a comparative study on bug-proneness in clone code and non-clone code by analyzing commit logs. According to our inspection on thousands of revisions of seven diverse subject systems, the percentage of changed files due to bug-fix commits is significantly higher in clone code compared with non-clone code. In addition, the possibility of severe bugs occurring is higher in clone code than in non-clone code. Bug-fixing changes affecting clone code should be considered more carefully. Finally, our manual investigation shows that clone code containing *if-condition* and *if-else* blocks has a high risk of having severing bugs. Changes to such types of clone fragments should be done carefully during software maintenance. According to our findings, clone code appears to be more bug-prone than non-clone code.

The rest of this chapter are organized as follows. Section 3.1 introduces this chapter. A short description of background is given in Section 3.2, our experimental steps are described in Section 3.3, results of our experiment are given in Section 3.4, related studies are presented in Section 3.5, threats to validity of our experiment is given in Section 3.6 and finally Section 3.7 summarizes this chapter.

3.1 Introduction

If two or more code fragments in a software system's code-base are exactly or nearly similar to one another we call them code clones [157, 151]. A group of similar code fragments forms a clone class. Code clones are mainly created because of the frequent copy/paste activities of programmers during software development and maintenance [157].

A significant number of studies [33, 35, 60, 58, 66, 88, 91, 94, 82, 95, 109, 24, 126, 128, 186, 102, 171, 84] have been conducted on discovering the impact of cloning on software maintenance. While a number of studies [33, 58, 66, 91, 94, 82, 95] have revealed some positive sides of code cloning, there is strong empirical evidence [35, 88, 109, 60, 24, 126, 128, 102, 171, 84] of negative impacts of code clones too. These negative impacts include higher instability [126], late propagation [35], and unintentional inconsistencies [60]. Existing studies [35, 131] show that code clones are related to bugs in the code-base.

Several studies have showed that bugs have a great effect on code in software systems. Sajnani et al. [164] showed that cloned code has less problematic bug patterns than non-cloned code. They have used bugs reported by FindBugs [5] from just one snapshot of the last revision of the system. Here, we consider bugs reported during the evolution of a software system through thousands of commits. In their paper [164], they worked on bugs reported by an automated tool whereas we work on bugs reported by the developer. Moreover, they considered only the Java programming language whereas we work on two programming languages, C and Java. These issues motivate us to work on bug reports generated by developers to see the impact of bug-fix commits on both clone code and non-clone code. We consider bug-fixing commits reported by the developers from thousands of commits in open source projects.

To explore the effects of bug-fix changes between clone and non-clone code, we conduct a comparative study. We consider thousands of revisions of seven diverse subject systems written in two different programming languages (Java and C). We detect code clones from each of the revisions of a subject system using the NiCad [49] clone detector, analyze the evolutionary history of these code clones, and investigate whether and to what extent they contain bugs. To find non-clone bug-fix commits, we first identify all the commits that are related to fixing a bug. Among these bug-fix commits, we detect those which have clone code. We consider the remaining bug-fix commits as non-clone bug-fix commits. We automatically count the total number of files that contain changes in source code. Among these files, we detect those files which have changes in clone code. Omitting these files from the total files, we get the changes in non-clone code. Then we calculate percentages of changes for both clone and non-clone code. We found that the percentage of changed files containing clone code is significantly higher than non-clone code. We validate our findings using the Mann-Whitney-Wilcoxon (MWW) [11] test for three types of clones with non-clone code.

We investigate the four research questions listed in Table 3.1. We find that the percentage of files changed due to bug-fix commits is significantly higher in clone code compared with non-clone code. Moreover, the percentage of files that have changes in Type 1 and Type 2 clone code is higher than changes in Type 3 clone code (definitions of different types of clones are given in Section 2.2). These findings can be used for ranking of clone code. Also, the occurrence of severe bugs is more in clone code than non-clone code. Finally, we find that most of the severe bugs contain changes in *if-condition* statements. This information is helpful for clone code management.

Table 3.1: Research Questions

SL	Research Question
RQ 1	What percentage of files get affected because of clone and non-clone bug-fix commits?
RQ 2	How often do bug-fix changes occur to the clone and non-clone code?
RQ 3	Is there any difference between the severity of the bugs occurring in clone and non-clone code?
RQ 4	Which types of severe and non-severe bugs can occur in code clones?

3.2 Background

The basic background of code clones are given in Section 2. Here, we provide additional background knowledge for this chapter. We describe the bug-proneness detection technique which has been used in this chapter. This technique has been used in our rest of the three studies too.

3.2.1 Bug-fix Commits

In the version control systems (e.g. SVN or Git) developers perform commits to keep track of the changes that they made in the code base. Developers often identify reported bugs in the software systems and fix them. The commit that occurs to fix a reported bug is known as a bug-fix commit. To fix these bugs, changes may occur to clone code or non-clone code. We observe these changes both in clone and non-clone code to contrast between them in terms of their bug-proneness.

3.2.2 Severe Bugs

Severe bugs are the software defects which can make negative impacts on the quality of software. Severe bugs are conventionally denoted using a critical level in a bug report. Developers can define bug levels depending on the criteria of bugs while reporting a bug in open source projects.

3.2.3 Bug-proneness Detection Technique

For each subject system, we first retrieve the commit messages by applying the 'SVN log' command. A commit message describes the purpose of the corresponding commit operation. We automatically infer the commit messages using the heuristic proposed by Mockus and Votta [123] in order to identify those commits that occurred for the purpose of fixing bugs. Then we identify which of these bug-fix commits make changes to clone fragments. If one or more clone fragments are modified in a particular bug-fix commit, then it is an implication that the modification of those clone fragment(s) was necessary for fixing the corresponding bug. In other words, the clone fragment(s) are related to the bug. In this way we examine the commit operations

Table 3.2: Subject Systems. **LLR** = LOC in the Last Revision.

Systems	**Language**	**Domains**	**LLR**	**Revisions**
Ctags	C	Code Definition Generator	33,270	774
Camellia	C	Image Processing Library	89,063	170
Brlcad	C	Solid Modeling CAD	39,309	735
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Management	45,515	1545

of a candidate system, analyze the commit messages to retrieve the bug-fix commits, and identify those clone fragments that are related to the bug-fix. We also determine the number of changes that occurred to such a clone fragment in a bug-fix commit using the UNIX *diff* command.

The procedure that we follow to detect the bug-fix commits was also previously followed by Barbour et al. [35]. Barbour et al. [35] detected bug-fix commits in order to investigate whether late propagation in clones is related to bugs. They at first identified the occurrences of late propagations and then analyzed whether the clone fragments that experienced late propagations are related to a bug-fix. In our study we detect bug-fix commits in the same way, as they detected, however, our study is different in the sense that we investigate the bugs of different types of code clones. Also, Barbour et al. [35] did not investigate the most important clone type, Type 3. Generally, the number of Type 3 clones in a system is the highest among the three clone-types. We consider Type 3 clones in our bug-fix study.

3.3 Experimental Steps

We conduct our research on seven subject systems (three C and four Java systems). We consider these seven subject systems since these systems have variations in application domains, sizes, revisions and also have been used by our other studies. These subject systems are listed in Table 3.2 which were downloaded from the SourceForge online SVN repository [16]. In this table, the total number of revisions of each subject system is given along with the lines of code (LOC) in the last revision.

We perform the following steps for detecting fixed bugs: **(1)** Extraction of all revisions (as stated in Table 3.2) of each of the subject systems from the online SVN repository; **(2)** Detection and extraction of code clones from each revision by applying NiCad [49] clone detector; **(3)** Detection of changes between every two consecutive revisions using UNIX *diff* command; **(4)** Locating these changes to the already detected clones

Table 3.3: NiCad Settings for three types of clones

Clone Types	**Identifier Renaming**	**Dissimilarity Threshold**
Type 1	none	0%
Type 2	blindrename	0%
Type 3	blindrename	20%

of the corresponding revisions; and **(5)** Detection of bug-fix commit operations. For completing the first four steps we use the tool SPCP-Miner [137]. We described the detection of bug-fix commits in Section 3.2.3. In Section 3.4, we will describe how we detect bug-fix changes in clone and non-clone code.

We use NiCad [49] for detecting clones since it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [156, 155]. Using NiCad, we detect block clones including both exact (Type 1) and near-miss (Type 2, Type 3) clones of a minimum size of 10 LOC with 20% dissimilarity threshold (for Type 3 clones) and blind renaming of identifiers. NiCad settings for detecting three clone-types (Type 1, Type 2, and Type 3) are shown in Table 3.3. For different settings of a clone detector the clone detection results can be different and thus, the findings on bugs in code clones can also be different. Hence, selection of appropriate settings (i.e., detection parameters) is important. We used the mentioned settings in our research, because [175] Svajlenko and Roy show that these settings provide better clone detection results in terms of both precision and recall. Moreover, code clones with a minimum size of 10 LOC are more appropriate from maintenance perspectives [157, 152, 37]. Before using the NiCad outputs of Type 2 and Type 3 cases, we processed them so that:

1. Every Type 2 clone class that exactly matched any Type 1 clone class was excluded from Type 2 outputs.
2. Every Type 3 clone class that exactly matched any Type 1 or Type 2 class was excluded from Type 3 outputs.

We processed NiCad clone detection results in this fashion because we wanted to investigate bugs in three types of clones separately where the three sets are disjoint. Next we detect bugs using commit messages. The detailed procedure of our bug detection technique is described in Section 3.2.3.

3.4 Experimental Results

We mention our four research questions in Table 3.1. In this section we present our experimental results and analyze them to find the answers to our research questions (RQs).

Table 3.4: Number of Files that have Changed Clone Code found in Bug-fixing Commits. **FC** = Number of Files that have Clone code. **FCC** = Number of Files that have Changed Clone code. **PFCC** = Percentage of the Number of Files that have Changed Clone code

	Type 1			Type 2			Type 3		
Subject Systems	**FC**	**FCC**	**PFCC**	**FC**	**FCC**	**PFCC**	**FC**	**FCC**	**PFCC**
Ctags	12	4	33.33%	25	4	16%	117	12	10.25%
Camellia	11	2	18.18%	0	0	0%	56	6	10.71%
Brlcad	33	3	9.09%	5	1	20%	66	5	7.57%
jEdit	21338	117	0.54%	695	14	2.01%	12655	137	1.08%
Freecol	259	13	5.01%	178	11	6.17%	3425	83	2.42%
Carol	121	14	11.57%	138	14	10.14%	991	51	5.14%
Jabref	97	8	8.24%	81	7	8.64%	1143	26	2.27%

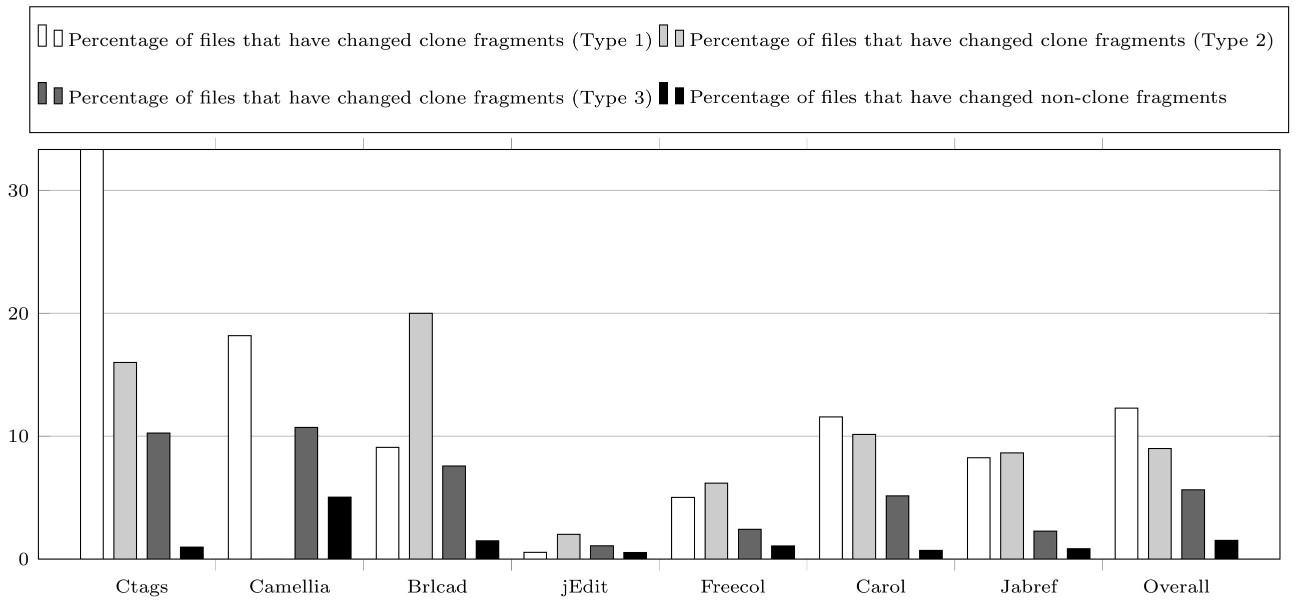


Figure 3.1: Percentage of the number of files that have changed clone and non-clone fragments.

3.4.1 *RQ 1: What percentage of files get affected because of clone and non-clone bug-fix commits?*

Motivation. It is important to know the percentage of affected files due to bug-fixing commits and compare between clone and non-clone code. More affected files means more changes in the system. More attention is needed when more changes occur. Knowing the information we can place emphasis on which type of code (clone or non-clone) are affecting the system more.

Methodology. To answer this research question, we automatically counted the total number of files that contain clone code in three different types of clones (Type 1, Type 2 and Type 3) and total number of files containing non-clone code. Also, we detect the total number of files that contain changed clone in three different clone types and the total number of files containing changed non-clone code. Then we calculate their percentages for individual clone types.

FC: This is the total number of files that have clone code. Columns with the heading FC in Table 3.4 represents the value.

FCC: This is the total number of files that contain changed clone code. This value is given in columns with the heading FCC in Table 3.4.

PFCC: To find out the percentage of the number of files that have changed clone code we use the following equation for all subject systems. In Table 3.4 columns with the heading PFCC show this value. Equation 3.1 shows the assessment of the percentages.

$$PFCC = \frac{100 \times FCC}{FC} \tag{3.1}$$

OPFCC: We also calculate overall percentage of files that have changed clone code using the following equation.

$$OPFCC_{T_i} = \frac{100 \times \sum_{all\ systems} FCC_{T_i}}{\sum_{all\ systems} FC_{T_i}} \tag{3.2}$$

Here, T_i represents different types of clones where $i \in \{1, 2, 3\}$.

Figure 3.1 shows the percentage PFCC and the overall percentage OPFCC for seven subject systems individually for each clone type. Here, we can see that Ctags, Camellia and Brlcad have a higher percentage than the rest of the subject systems (jEdit, Freecol, Carol and Jabref). jEdit has the lowest percentage among them all. However, we observe that overall percentage is decreasing from Type 1, Type 2 and Type 3 clone respectively. Table 3.4 describes the FC, FCC and PFCC for all the subject systems individually for each clone type (1, 2 and 3). We can see from this Table that only Camellia has no bug-fix commits related to clone Type 2.

Reducing the total number of files containing clone code in bug-fix commits from the total number of files in bug-fix commits, we get the total number of files that have only non-clone code. For answering RQ 1, we identify the total number of files that have changes in source code. From these files we identify the total

Table 3.5: Number of Files that have Changed Non-Clone Code in Bug-fixing Commits **FNC** = Number of Files that have Non-Clone code. **FCNC** = Number of Files that have Changed Non-Clone code. **PFCNC** = Percentage of the Number of Files that have Changed Non-Clone code.

Subject Systems	**FNC**	**FCNC**	**PFCNC**
Ctags	12318	120	0.97%
Camellia	972	49	5.04%
Brlcad	6978	104	1.49%
jEdit	2801	15	0.53%
Freecol	41442	444	1.07%
Carol	13302	94	0.70%
Jabref	39389	333	0.84%

number of files that have changes in clone code. The rest of the files made changes to non-clone code due to bug-fix commits.

FNC: This is the number of total files that have non-clone code. Columns with the heading FNC in Table 3.5 show the values.

FCNC: This is the number of total files which contain changes in non-clone code. We also include those files that do not contain clone code. All the columns of Table 3.5 with the heading FCNC show this value.

PFCNC: To calculate percentage of the number of files containing changed non-clone code we use the following equation. All the columns with the heading PFCNC in Table 3.5 represent this value. This is shown in equation 3.3.

$$PFCNC = \frac{100 \times FCNC}{FNC} \tag{3.3}$$

OPFCNC: We calculate the overall percentage of files containing changed non-clone code using the following equation.

$$OPFCNC = \frac{100 \times \sum_{all\ systems} FCNC}{\sum_{all\ systems} FNC} \tag{3.4}$$

PFCNCs of different clone types for each subject system are shown in Figure 3.1. Here, we can see that Camellia has the highest percentage among the subject systems. On the other hand, jEdit has the lowest percentage among all subject systems. Table 3.5 shows FNC, FCNC and PFCNC for all subject systems. We observe that percentage of changes due to bug-fix commits is higher in clone code than non-clone code. This result was expected because the total number of files containing non-clone code is much higher (almost three times) than clone code in every subject system.

Table 3.6: Mann-Whitney-Wilcoxon Test Result for RQ1

Clone Types	p-value	U value
Type 1	0.011719	8
Type 2	0.015714	9
Type 3	0.004574	5
Considering level of significance is 5%.		
For 5% two-tailed level, Critical value of U is 13		

The overall percentages (OPFCC) of files that have changes in Type 1 (12.28%) and Type 2 (8.99%) clones have more changes in files than Type 3 (5.63%) clones for bug-fix commits. Moreover, the percentage of files that have changes in clone code is higher than the percentage (OPFCNC) of files that have changes in non-clone code (1.52%).

Mann-Whitney-Wilcoxon (MWW) tests for RQ 1. We are interested to know whether the percentages of three clone types is significantly higher than non-clone code. First, we perform Mann-Whitney-Wilcoxon (MWW) test [11] with percentages of Type 1 clone and non-clone code. We consider the significance level is 5% for this test. According to the data critical U is 13. If the p-value is less than 0.05 and U value is less than 13 then the result is significant. Our result shows that percentage of Type 1 clone code is significantly higher than non-clone code. For two-tailed test we find the p-value of 0.011719 which is much lower than 0.05. In the same way, we perform the test for percentages of Type 2 clone code and Type 3 clone code with the non-clone code. We find the p-value of 0.015714 and 0.004574 for Type 2 clone and Type 3 clone respectively. Both percentages of Type 2 clone and Type 3 clone code are significantly higher than non-clone code. Thus, we can say that percentages of all three types of clones are significantly higher than non-clone code. We list our MWW test results in Table 3.6.

Answer to RQ 1. According to our experimental results, the percentage of the number of file changes in bug-fix commits to clone code in files with clones is higher than the percentage of the number of file changes in bug-fix commits to non-clone code in files that only have non-clone code. Also, in terms of overall percentage of files, Type 1 and Type 2 code clones (12.28% and 8.99%) have a higher percentage than Type 3 code clone (5.63%).

We observe that percentages of files containing changes due to bug-fix commits is higher in clone code than non-clone code. However, we still do not know what percentage of clone and non-clone code gets changed during bug-fix commits. Intuitively, percentages of bug-fix changes should be higher in clone code than non-clone code. To understand this we investigate our next research question.

Table 3.7: Number of Bug-fix Commits affecting Clone Code. **CC** = Number of Commits affecting Clone code. **BCC** = Number of Bug-fix Commits affecting Clone code. **PBCC** = Percentage of Commits that were applied for fixing Bugs in Clone code.

	Type 1			Type 2			Type 3		
Subject Systems	**CC**	**BCC**	**PBCC**	**CC**	**BCC**	**PBCC**	**CC**	**BCC**	**PBCC**
Ctags	14	3	21.42%	25	4	16%	59	11	18.64%
Camellia	8	1	12.5%	5	1	20%	26	5	19.23%
Brlcad	32	2	6.25%	7	1	14.28%	47	5	10.63%
jEdit	92	37	40.21%	24	10	41.66%	99	42	42.42%
Freecol	35	7	20%	36	10	27.77%	152	46	30.26%
Carol	41	8	19.51%	44	8	18.18%	112	22	19.64%
Jabref	48	6	12.5%	46	6	13.04%	149	23	15.43%

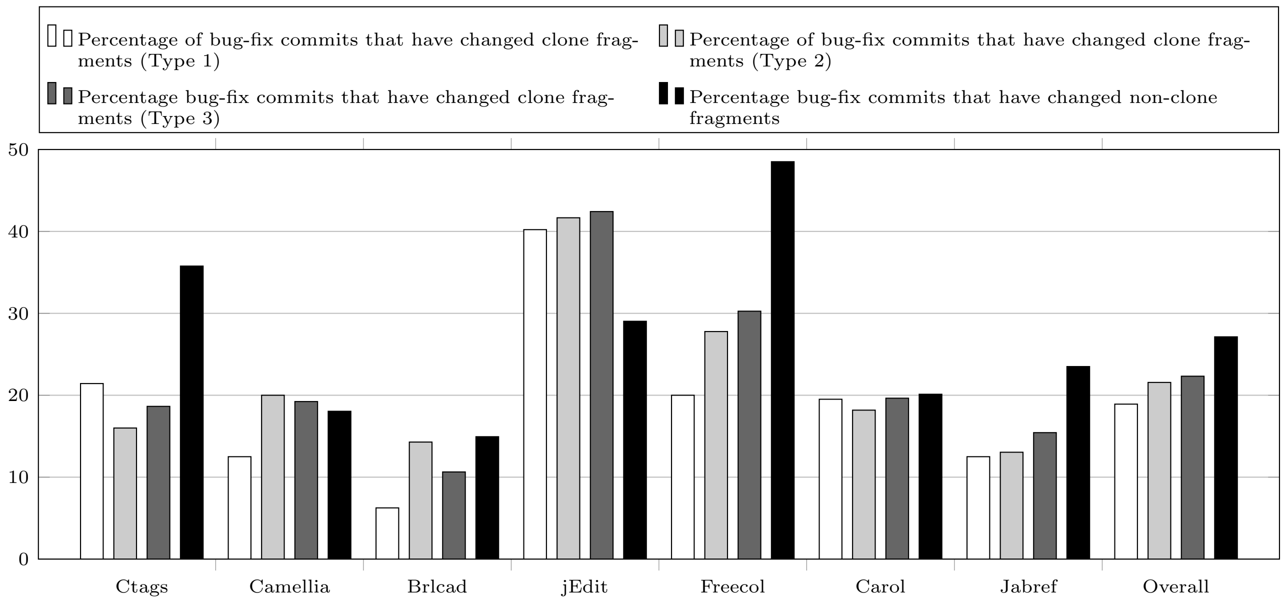


Figure 3.2: Percentage of bug-fix commits that have changed clone and non-clone fragments.

3.4.2 *RQ2: How often do bug-fix changes occur to the clone and non-clone code?*

Motivation. Though we have the answer of RQ 1 and hence we know the percentage of files changes in clone and non-clone code, but still we are not sure regarding how much these changes are influencing the system. It is important to know the frequency of the bug-fix changes in both clone and non-clone code. From a comparison between them, we can understand the impact of bug-fix changes. Intuitively, more importance should be given to the more frequent one. This will help us to manage clone code.

Methodology. We know the total number of commits of each subject system. As discussed in Section 3.3, we report the total number of commits that have changes in clone fragments. To answer the RQ 2 we automatically count the total number of bug-fix commits that contain changes in clone code. First, we find out the total number of bug-fix commits as described in Section 3.2.3. We automatically count the number of total bug-fix commits which have changed clone code. We deduct this number from the total number of bug-fix commits and found the total number of bug-fix commits which have changed non-clone code. In the following way we calculate the occurrences of bug-fix changes in clone and non-clone code.

CC: This is the total Number of Commits that made changes to Clone code. All the columns of Table 3.7 with the heading CC show this value.

BCC: This is the total Number of Bug-fix Commits that made changes to Clone code. Columns with the heading BCC in Table 3.7 show the values.

PBCC: Percentage of the Bug-fix Commits that made changes to Clone code. We calculate this for each subject systems and for three different types of clone i.e. Type 1, Type 2 and Type 3 clone code. All the columns with the heading PBCC in Table 3.7 represent this value.

We use the following equation for calculating the percentage.

$$PBCC = \frac{100 \times BCC}{CC} \tag{3.5}$$

We calculate the overall percentage of the bug-fix commits containing clone code using the following equation.

$$OPBCC_{T_i} = \frac{100 \times \sum_{all\ systems} BCC_{T_i}}{\sum_{all\ systems} CC_{T_i}} \tag{3.6}$$

Here, $OPBCC_{T_i}$ is the overall percentage of the clone code found in the bug-fix commits with respect to T_i type of clones ($i \in \{1, 2, 3\}$). Table 3.7 shows the value of CC, BCC and PBCC for seven subject systems and each type of clone individually. Figure 3.2 describes the percentage PBCCs of all subject systems along with the overall percentage OPBCC. We can see that jEdit has the highest percentage (over 40%) and Brlcad has the lowest percentage (less than 15%). However, in overall percentage Type 3 code clone has the higher percentage than Type 1 and Type 2 code clone.

Table 3.8: Number of Bug-fix Commits affecting Non-Clone Code. **CNC** = Number of Commits affecting Non-Clone code. **BCNC** = Number of Bug-fix Commits affecting Non-Clone code. **PBCNC** = Percentage of Commits that were applied for fixing Bugs in Non-Clone code.

Subject Systems	**CNC**	**BCNC**	**PBCNC**
Ctags	383	137	35.77%
Camellia	133	24	18.04%
Brlcad	589	88	14.94%
jEdit	31	9	29.03%
Freecol	672	326	48.51%
Carol	323	65	20.12%
Jabref	685	161	23.50%

We first identify the list of commits that made changes to the source code. From these commits we detect which commits made changes to clone code. The remaining commits in the list made changes to non-clone code. In the same way, we deduct the total number of bug-fix commits containing changes in clone code from the total number of bug-fix commits to find the number of changes in non-clone code bug-fix commits. Applying these findings we answer RQ 2.

CNC: This is the total Number of Commits that made changes to Non-clone code. This value is given in columns with the heading CNC in Table 3.8.

BCNC: This is the total Number of Bug-fix Commits that made changes to Non-clone code. Columns with the heading BCNC in Table 3.8 represent the value.

PBCNC: Percentage of the Bug-fix Commits that made changes to Non-clone code. In Table 3.8 columns with the heading PBCNC show this value.

Likewise equation 3.5 to compute the percentages we use equation 3.7.

$$PBCNC = \frac{100 \times BCNC}{CNC} \tag{3.7}$$

To see the overall percentage of the number of bug-fix commits that is related to non-clone code we use the following equation which is similar to the equation 3.6.

$$OPBCNC = \frac{100 \times \sum_{all\ systems} BCNC}{\sum_{all\ systems} CNC} \tag{3.8}$$

Here, $OPBCNC$ is the overall percentage of the bug-fix commits that contain non-clone code. The CNC, BCNC and PBCNC for all subject systems are shown in Table 3.8. Here, most of the percentages range from 15% to 35% (only the percentage of Freecol has more than 45%). Figure 3.2 depicts the percentage PBCNCs

Table 3.9: Mann-Whitney-Wilcoxon Test Result for RQ2

Clone Types	**p-value**	**U value**
Type 1	0.092892	16
Type 2	0.207578	20
Type 3	0.344562	23
Considering level of significance is 5%. For 5% two-tailed level, Critical value of U is 13		

and overall percentage OPBCNC of seven subject systems. Here, we observe that percentage of the bug-fix commits containing changed non-clone code is highest in Freecol system and lowest in Brlcad. The overall percentage is near 30% which is higher than clone code. Though the bug-fix changes occur more in non-clone code than clone code, the difference is not that particularly high.

The overall percentage (OPBCC) of Type 3 (22.32%) clone is higher than Type 1 (18.91%) and Type 2 (21.56%) clone. Obviously, Type 1 clone (has exact same code) and Type 2 clone (has renaming of identifiers or changing data type of identifiers) have less changes than Type 3 clone code (has addition, deletion or modification of code). We believe for this reason Type 3 clone code is affected more than Type 1 and Type 2 clone code. Also, the overall percentage (OPBCNC) for non-clone code (27.13%) is higher than clone code.

Mann-Whitney-Wilcoxon (MWW) tests for RQ 2. We performed an MWW test [11] to understand whether the difference between the percentages of clone and non-clone code is significant. The percentage of each type of clone is individually tested with non-clone code. We found the p-value of 0.092892, 0.207578 and 0.344562 for Type 1, Type 2 and Type 3 clone respectively. Calculated U value for Type 1, Type 2 and Type 3 clone code is 16, 20 and 23 respectively. Here, every p-value is greater than 0.05 (significance level is 5%) and every U value is greater than 13 (critical U value is 13). This indicates the result is not significant, which means we cannot conclude that the percentage of bug-fix commits having changed non-clone code is higher than for clone code. Table 3.9 describes the p-value and U value for all three types of clone code.

Answer to RQ 2. Comparing the overall percentage of bug-fix commits containing clone and non-clone code, we found that frequency of bug-fix commits is slightly higher in non-clone code (27.13%) than clone code (20.93%). Also, Type 3 clone code (22.32%) has a higher percentage of changes versus Type 1 and 2 clone code (18.91% and 21.56%) due to bug-fix commits.

We observe that bug-fix commits occur in non-clone code more often than clone code by 6.2%. From MWW test we find that the difference between percentages of clone and non-clone code is not significant.

3.4.3 *RQ 3: Is there any difference between the severity of the bugs occurring in clone and non-clone code?*

Motivation. In every single commit there is a message or comment written by the programmer which describes about the changes that they made from the previous commit. In case of bug-fix commits these messages describe the bug that occurs in the code base of the system. By reading a bug-fix commit message, we can understand whether a bug is severe or not. This message is helpful for debugging and understanding the scenario of the situation. To understand the severity of bugs and compare between clone and non-clone code, we automatically process the bug-fix commit massages followed by a manually inspection. It is important to give priority to more severe bugs while fixing them.

Why use commit messages. In each bug report there is an attribute named 'priority' which indicates different level of importance of that bug report. This attribute helps developers to decide which bug should be fixed first. This is possibly the easiest way to get an idea of the severity of a bug. However, we do not consider this attribute as a direct measure of bug severity. We manually checked that in most cases bug reporters do not assign this value and rather leave it as the default value, which usually describes a 'normal' level of severity. Thus, an incorrect evaluation will take place if we consider 'priority' as the scale of the bug severity itself. Similarly, Saha et al. [160] proved in their study that we cannot assume the leveling of 'priority' is always accurate. Their work [160] supports our decision about not considering the 'priority' of the bug report as the measurement of the severity. Considering this situation we decide to investigate commit messages instead of bug 'priority' of the bug report.

Methodology. We perform NLP-based preprocessing on bug-fix commit messages to reduce the noise. There are numerous types of data preprocessing, from which we use two types:

(1) Tokenization (removing punctuation, special characters, numbers)

(2) Stop words removal.

On the preprocessed commit messages, we automatically perform a heuristic search proposed by Lamkanfi et al. [97] in the bug-fix commit messages to identify severe bugs. Then we manually investigate the results for validation. We consider five subject systems (Ctags, Camellia, Brlcad, jEdit, Freecol) for this experiment. For the non-clone bug-fix commits we choose some random bug-fix commits which does not contain clone code. It is not feasible to check all the non-clone bug-fix commits by manual inspection. Hence, we keep the total number of non-clone bug-fix commits equal to the total number of clone bug-fix commits to maintain the data impartiality.

Lamkanfi et al. [97] suggested most significant terms for different components indicating severe and non-severe bugs. For example, 'fault', 'hang', 'freez', 'deadlock' etc. represent severe problems of the system. The words 'favicon', 'deprec', 'mnemon', 'outbox' etc. represent non-severe problems of the system. We take these terms as keywords to decide the severity of the bug. Though there are different levels of bug severity we consider only two categories for the simplicity. That is whether the bug is severe or not i.e. 'TRUE' or

'FALSE'. Here, 'TRUE' means the bug is severe (i.e. bug-fix commit messages containing the terms which represent severity) and 'FALSE' means the bug is non-severe (i.e. bug-fix commit messages containing the terms which represent non-severity). This is important for the bug triaging process since severe bugs need more care and prompt fixing.

We calculate the percentage of severe bugs in bug-fix commits for both clone and non-clone code. We observe that the existence of severe bugs in Camellia (both clone and non-clone), Brlcad (clone) and jEdit (non-clone) systems is zero (0%). We also observe that Freecol has highest percentage (85.71% for clone and 62.5% for non-clone code) of severe bugs in both clone and non-clone bug-fix commits compared to other subject systems. Percentages of severe bugs in the rest of the subject systems range from 50% to 67%. Ctags (non-clone) and Brlcad (non-clone) have the lowest percentage (50%) of severe bugs. Overall, clone code has a higher tendency of having severe bugs than non-clone code. The difference between the overall percentages of severe bugs of clone and non-clone code bug-fix commits is 17.46% which is highly significant. This findings imply that more importance should be given on clone code while fixing bugs for better software maintenance.

Answer to RQ 3. After careful inspection of each commit messages of the bug-fix commits for both clone and non-clone code, we found that clone code bug-fix commits have a higher percentage of severe bugs (overall percentage 71.63%) than the non-clone code (overall percentage 54.17%). This proves that occurrence of severe bugs is higher in clone code compared with non-clone code.

We observe that severity of bugs is higher in clone code than non-clone code. However, we find that some of the bug-fix commit messages are very short and it is not enough to describe the severity of the bug. Considering this constraint in the messages in bug-fixing commits, the result may vary in different cases. However, severe bugs should have the highest priority in software maintenance.

3.4.4 *RQ 4: Which types of severe and non-severe bugs can occur in code clones?*

Motivation. After answering RQ3 we know that the occurrence of severe bugs is higher in clone code than non-clone code. We wanted to better understand the severe bugs occurring in clone code. We investigated which type of severe bugs have the highest frequency of occurrence. This information will help to find out which type of severe bugs are causing more inconvenience to the systems so that the developers can be more careful about handling those type of severe bugs while fixing them.

Methodology. In Section 3.4.3, we described the procedures of identifying severe and non-severe bugs. To answer our 4th research question first we categorize all the bugs occurring in clone code. We consider five subject systems (Ctags, Camellia, Brlcad, jEdit, Freecol) for this experiment. We investigate the source code of each bug-fixing commit manually, totaling 337 bug-fix commits. We categorize the types of changes of source code due to these bug-fixing commits. Later we map severe and non-severe bugs with these categories

Table 3.10: Most frequent change types of severe and non-severe bugs in code clones

	Change Types	**Severe**	**Non-severe**
1	Addition of if-else blocks	5	4
2	Modification of if Condition	5	9
3	Deletion of if-else blocks	2	3
4	Modification of Parameters in the Called Method		6
5	Addition of Method Call	3	1
6	Replacement of Old Method Call by New Method Call	3	1
7	Deletion of Method Call		1

of code changes. In Chapter 5, we showed such categorization. However, we did not investigate the severity of bugs in that chapter.

Table 3.10 shows the results of our manual investigation of severe and non-severe bugs in clone code. Here, we observe that most of the severe bugs occurred due to the addition and modification of *if-condition* statements. We suggest developers to handle more carefully while copying or modifying an *if-condition.* On the other hand, the highest frequency was found for non-severe bugs in modification of *if-condition.* The second highest frequency for non-severe bugs was found for 'Modification of Parameters in the Called Method'. The other frequent change types shown in Table 3.10 are addition and deletion of a method call, deletion of if-else blocks and replacement of an old method call by a new method call. There are also some infrequent change types such as: replacement of the C preprocessor by a method call, addition, deletion or modification of loops.

We also observe that Type 3 clone code contains the highest number of severe bugs and Type 2 clone code contains few severe bugs. Type 1 clone code does not contain any severe bugs. On the other hand, all the three types of clone code contain non-severe bugs. Among them, Type 1 and Type 2 clone code contain a higher number of non-severe bugs than Type 3 clone code. This summarizes that we should place emphasis on Type 3 clone code while performing clone management operations.

Answer to RQ 4. According to our manual investigation, code clones containing *if-condition* and *if-else* blocks have a higher risk of severe bugs. These types of changes in clone code should be made carefully during software maintenance.

We observe the types of changes for fixing both severe and non-severe bugs in code clones. It is important to consider these types of changes during clone management. This will reduce software maintenance cost and effort in future.

3.5 Related Work

Shajnani et al. [164] performed a comparative study between clone and non-clone code for different bug patterns. They used bug reports generated by a tool, i.e., FindBugs whereas we worked on real bug reports that were reported by developers. In our previous study [72], it has been shown that cloning is responsible for replicating bugs. However, it does not show any comparison with non-clone code in that study.

Bug-proneness of code clones has been investigated by a number of existing studies. Li and Ernst [102] performed an empirical study on the bug-proneness of clones by investigating four software systems and developed a tool called CBCD on the basis of their findings. CBCD can detect clones of a given piece of buggy code. Li et al. [104, 105] developed a tool called CP-Miner which is capable of detecting bugs related to inconsistencies in copy-paste activities. Steidl and Göde [171] investigated finding instances of incompletely fixed bugs in near-miss code clones by investigating a broad range of features of such clones involving machine learning. Göde and Koschke [60] investigated the occurrences of unintentional inconsistencies to the code clones of three mature software systems and found that around 14.8% of all changes that occurred to the code clones were unintentionally inconsistent. Chatterji et al. [45] performed a user study to investigate how clone information can help programmers localize bugs in software systems. Jiang et al. [84] performed a study on the context based inconsistencies related to clones. They developed an algorithm to mine such inconsistencies for the purpose of locating bugs. Using their algorithm they could detect previously unknown bugs from two open-source subject systems. Inoue et al. [71] developed a tool called 'CloneInspector' in order to identify bugs related to inconsistent changes to the identifiers in the clone fragments. They applied their tool on a mobile software system and found a number of instances of such bugs. Xie et al. [200] investigated fault-proneness of Type 3 clones in three open-source software systems. They investigated two evolutionary phenomena on clones: (1) mutation of the type of a clone fragment during evolution, and (2) migration of clone fragments across repositories and found that mutation of clone fragments to Type 2 or Type 3 clones is risky.

None of the studies discussed above investigated bug-fix commits in code clones and non-clone code simultaneously. Mondal et al. [131] investigated bug-proneness of code clones. While the primary target of that study was to compare the bug-proneness of three clone-types, our target is to compare the bug-proneness of clone and non-clone code. Mondal et al. [131] did not investigate the bug-proneness of non-clone code in their study.

Rahman et al. [147] found that bug-proneness of cloned code is less than that of non-cloned code on the basis of their investigation on the evolution history of four subject systems using DECKARD [83] clone detector. Authors choose DECKARD [83] as clone detection tool over CCFinder [89] and CP-Miner [104, 105] since they found the performance of DECKARD is better than CCFinder and CP-Miner in their experiment. However, they considered monthly snap-shots (i.e., revisions) of their systems and thus, they have the possibility of missing buggy commits. They have calculated and found that on an average 3.3% of bugs have late propagation fixing with different staging snapshots. In our study, we consider all the snap-shots/revisions (i.e., without discarding any revisions) of a subject system from the beginning one. Thus, we believe that we are not missing any bug-fix commits. Moreover, our goal in this study is different. We investigate and compare the impacts of bug-fix commits of different types of code clones whereas they only focused on the bug-proneness of clones. Selim et al. [166] used Cox hazard models in order to assess the impacts of cloned code on software defects. They found that defect-proneness of code clones is system dependent. However, they considered only method clones in their study. We consider block clones in our study. While they investigated only two subject systems, we consider seven diverse subject systems in our investigation. Also, we investigate the bug-fix possibilities of different types of clones. Selim et al. [166] did not perform a type centric analysis in their study. A number of studies have also been done on the late propagation in clones and its relationships with bugs. Aversano et al. [33] investigated clone evolution in two subject systems and reported that late propagation in clones is directly related to bugs. Barbour et al. [35] investigated eight different patterns of late propagation considering Type 1 and Type 2 clones of three subject systems and identified those patterns that are likely to introduce bugs and inconsistencies to the code-base.

Focusing on this, we perform an in-depth investigation on a bug's impacts in code clones and non-clones in this research. Our experimental results are promising and provide useful implications for better understanding of the bug-proneness of clone and non-clone code.

3.6 Limitations

We used the NiCad clone detector [49] for detecting clones. While all clone detection tools suffer from the *confounding configuration choice problem* [197] and might give different results for different settings of the tools, the setting that we used for NiCad for this experiment are considered standard [153] and with these settings NiCad can detect clones with high precision and recall [156, 155, 175]. Thus, we believe that our findings on the bug-proneness of code clones are of significant importance.

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique proposed by Mocus and Votta [123] and also used by Barbour et al. [36]. The technique proposed by Mocus and Votta [123] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [36] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

The number of total subject system is not enough in our research to be able to generalize our findings regarding the comparative bug-fix changes of different types of clones. However, our candidate systems were of diverse variety in terms of application domains, sizes and revisions. Thus, we believe that our findings are important from the perspectives of clone and non-clone code.

3.7 Summary

In this chapter we conduct an in-depth comparative study of software bugs in both clone and non-clone code. For clone code we also consider three major types of clones, Type 1, Type 2 and Type 3. We investigated thousands of revisions of seven diverse subject systems. We also investigated bug-fix commit messages to measure the frequency of severe bugs in clone and non-clone code. From our examination, changes to files due to bug-fix commits is higher for clone code than for non-clone code. Additionally, changes to files due to bug-fix commits happens more in Type 1 and Type 2 code clones than in Type 3 code clones. In addition, the percentage of severe bugs is higher in clone code than non-clone code bug-fix commits. Finally, our manual investigation shows that certain types of code constructs in code clones has a higher risk of containing severe bugs. These are *if-condition* and *if-else* blocks. Changes to these types of code constructs in clone code should be prioritized during software evolution and maintenance. We believe that our findings on bug-fix commits are valuable for a better understanding of clone management such as ranking of clone code and software maintenance.

4 Software Bugs in Micro Clones

From the previous chapter we found that clone code is more bug-prone than non-clone code. After finding this result, we realize that we should place more emphasis on clone code over non-clone code since clone code are more likely to produce software bugs. To extend our first study, we conduct an empirical study on bug-proneness of micro-clones and regular code clones. A detailed description of micro-clones is given in Section 4.2.1.

Most of the existing studies on code clones ignore micro-clones where the size of a micro-clone fragment can be 1 to 4 LOC. In this chapter we compare the bug-proneness of micro-clones with that of regular code clones. From thousands of revisions of six diverse open-source subject systems written in three languages (C, C#, and Java), we identify and investigate both regular and micro-clones that are associated with reported bugs. Our experiment reveals that percentage of changed code fragments due to bug-fix commits is significantly higher in micro-clones than regular clones. The number of consistent changes due to bug-fix commits is significantly higher in micro-clones than regular clones. We also observe that a significantly higher percentage of files get affected by bug-fix commits in micro-clones than regular clones. Finally, we found that percentage of severe bugs is significantly higher in micro-clones than regular clones. We perform the Mann-Whitney-Wilcoxon (MWW) test to evaluate the statistical significance level of our experimental results. Our findings imply that micro-clones should be emphasized during clone management and software maintenance.

The rest of the chapter is organized as follows. Section 4.1 introduces this chapter. Section 4.2 contains the terminology, Section 4.3 discusses the experimental steps, Section 4.4 answers our research questions by presenting and analyzing the experimental results, Section 4.5 describes a rigorous manual analysis on micro-clones. Section 4.6 discusses the related work and compares with our study, Section 4.7 discusses possible threats to validity, and Section 4.8 concludes this study and discusses possible future work.

4.1 Introduction

Recurrent activities of copy-pasting code fragments is very common in everyday life of the software development cycle. The act of copying a piece of code and then pasting it without any modification (exactly similar) or with modifications (nearly similar) is known as code cloning [157, 151]. A group of similar code fragments constructs a clone class. Code clones are mainly created because of the frequent copy/paste activities of programmers during software development and maintenance. Besides copy/paste activities, there can be various other methods and reasons to create clone code [152]. Whatever may be the reasons behind cloning, code clones are significantly important from the perspectives of software maintenance and evolution [157].

During the last two decades code clone research was based on detecting, refactoring, tracking and managing code clones. Existing studies ignored code clones of small size i.e. 1 to 4 LOC stating that these clones are mostly unpromising. In a very recent study, Beller et al. [40] investigated these small code clones, referring to them as *micro-clones*. Also, Tonder et al. [191] worked on automatically detecting and safely removing micro-clones at a large scale. In another study, Mondal et al. [135] focused on the importance of micro-clones and found that during software maintenance and evolution 80% of all consistent updates occur in micro-clones. They have shown that the number of micro-clones is very high in software systems versus regular clones. However, they did not investigate bug-proneness of micro-clones. This motivates us to investigate buggy code in both micro and regular code clones. In order to explore the effects of bugs in micro-clones and regular code clones, we perform a comparative study, which to the best of our knowledge, is the first such comparative study.

We consider thousands of revisions of six diverse open source software systems written in three languages, Java, C# and C. For detecting code clones from each of the revisions of a subject system we use the NiCad clone detector [49]. We analyze the evolutionary history of both micro and regular code clones, and investigate whether they contain bugs and to what extent.

To investigate bug-proneness of regular and micro-clones, we observe bug-fix commits reported by the developers from thousands of commits in open source projects. The major findings of our research is as follows.

- The percentage of bug-fix changes occurring in micro-clones is significantly higher than the percentage of bug-fix changes in regular code clones.
- The number of consistent bug-fix changes is considerably higher in micro-clones than regular code clones. We found that total number of consistent bug-fix changes in micro-clones (4,118) is almost 6 times higher than regular code clones (728).
- Percentage of affected files due to bug-fix changes is significantly higher in micro-clones than regular clones. For micro-clones this percentage is 39.15%, whereas for regular clones the percentage is only 8.02%.
- Micro-clones can contain a significantly higher percentage of severe bugs compared to regular clones. From our inspection we found that micro-clones contain 16.98% more severe bugs than the regular code clones.

From these findings we can state that micro-clones draw our attention for clone management purpose, because they exhibit a high bug-proneness during evolution. In order to get rid of the negative impacts of micro-clones, we should track them for updating consistently. However, the existing clone trackers only consider regular code clones for tracking. Thus, it is important to investigate these clone trackers with a goal of making them capable of tracking micro-clones.

Table 4.1: Research Questions

SL	Research Question
RQ 1	Do micro-clones contain more bug-fix changes than regular code clones?
RQ 2	Are the bug-fix changes consistent in micro and regular code clones?
RQ 3	What percentage of files get affected for fixing bugs in micro and regular clones?
RQ 4	Do micro-clones contain more severe bugs compared to regular code clones?

4.2 Background

In this section we define necessary terminology to understand our second study.

4.2.1 Micro Clones

Micro-clones are smaller in size than the clone size of regular code clones. According to the literature [40, 191, 135], micro-clones can be of 4 LOC at most. The minimum size of a micro-clone fragment can be 1 LOC. In this paper we ignore those micro-clones which are part of regular clones. Thus we consider only true or pure micro-clones.

4.3 Experimental Steps

We conduct our research on six subject systems (two C, one C# and three Java systems). We consider these six subject systems since these systems have variations in application domains, sizes, and revisions. These subject systems are listed in Table 4.2 which were downloaded from the SourceForge online SVN repository [16]. In this table, the total number of revisions of each subject system is given along with the lines of code (LOC) in the last revision. Figure 4.1 shows the simple flow diagram of our work procedure for this study.

We perform the following steps for detecting fixed bugs: **(1)** Extraction of all revisions (as stated in Table 4.2) of each of the subject systems from the online SVN repository; **(2)** Detection and extraction of code clones from each revision by applying the NiCad [49] clone detector; **(3)** Detection of changes between every two consecutive revisions using diff; **(4)** Locating these changes to the already detected clones of the corresponding revisions; and **(5)** Detection of bug-fix commit operations. For completing the first four steps we use the tool SPCP-Miner [137]. We perform steps 1 to 5 for both regular and micro-clones. We will describe the detection of bug-fix commits later in this section. In Section 4.4 we will describe how we detect bug-fix changes in regular code clones and micro-clones.

We use NiCad [49] for detecting clones since it can detect code clones with high precision and recall [156, 155]. Using NiCad, we detect block clones (both exact and near-miss) of at least 10 lines with 20%

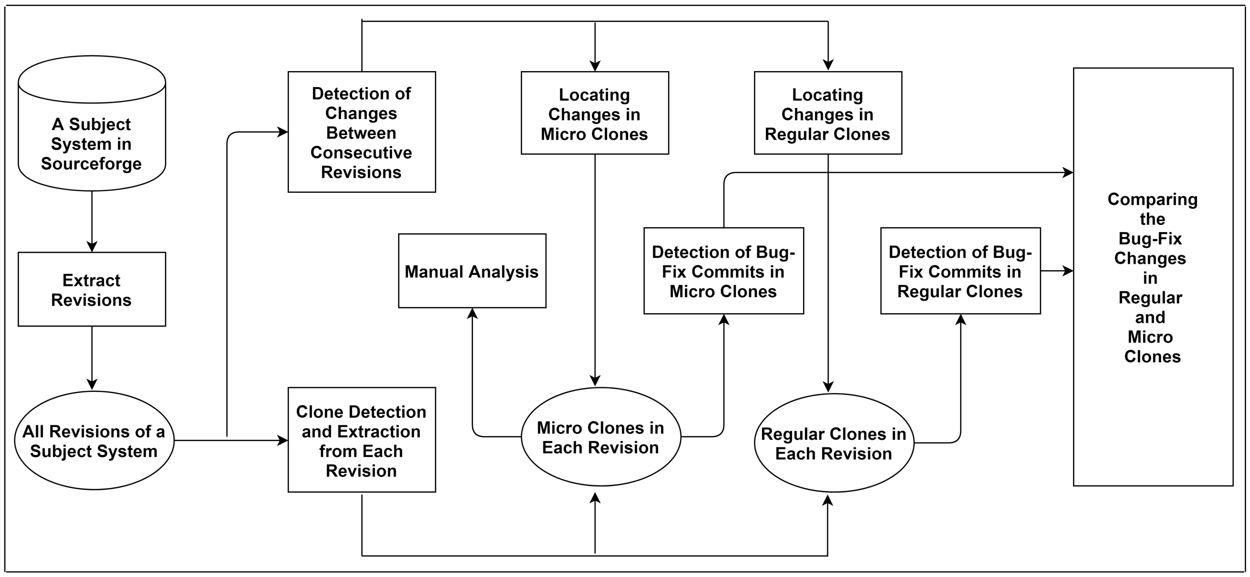

A Subject System in Sourceforge
Extract Revisions
All Revisions of a Subject System
Detection of Changes Between Consecutive Revisions
Clone Detection and Extraction from Each Revision
Locating Changes in Micro Clones
Locating Changes in Regular Clones
Manual Analysis
Detection of Bug-Fix Commits in Micro Clones
Detection of Bug-Fix Commits in Regular Clones
Micro Clones in Each Revision
Regular Clones in Each Revision
Comparing the Bug-Fix Changes in Regular and Micro Clones

Figure 4.1: The execution flow diagram of the experimental steps in detecting bug-fix commits in regular and micro-clones. The rectangles demonstrate the steps.

Table 4.2: Subject Systems

Systems	**Language**	**Domains**	**LLR**	**Revisions**
Ctags	C	Code Def. Generator	33,270	774
Brlcad	C	Solid Modeling CAD	39,309	735
MonoOSC	C#	Formats and Protocols	18,991	355
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Management	45,515	1545
LLR = LOC in the Last Revision				

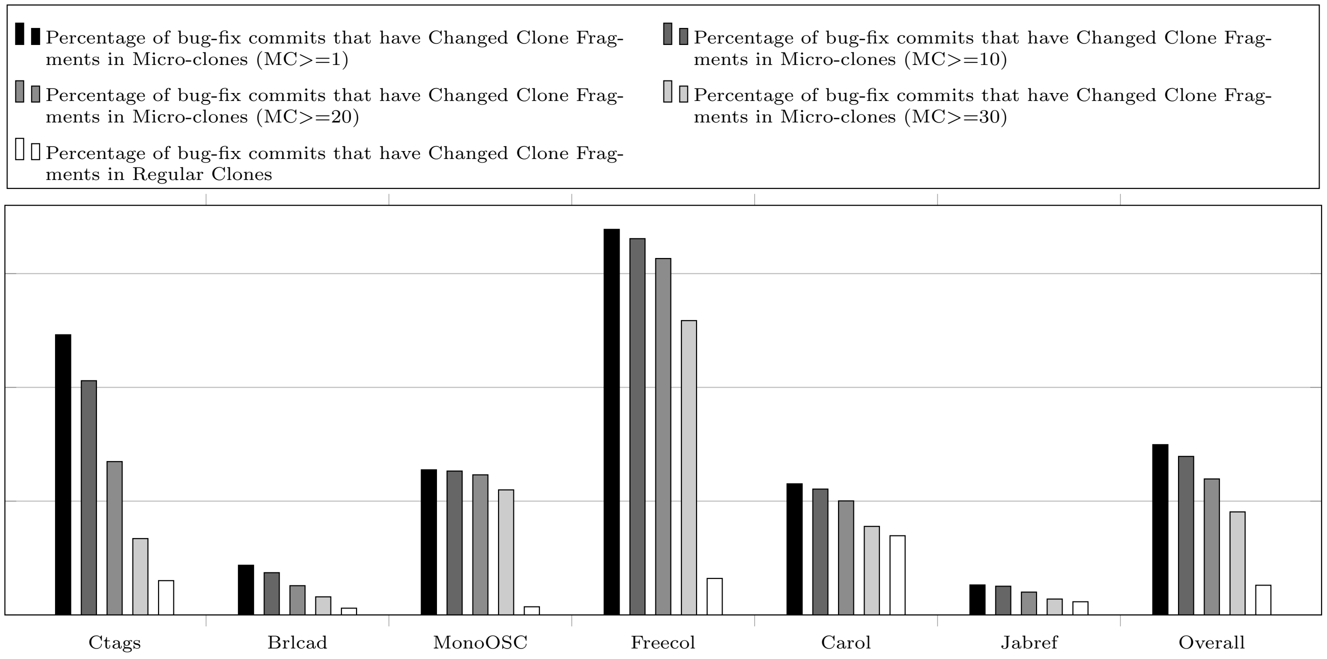


Figure 4.2: Percentage of bug-fix commits that have changed clone fragments in regular and micro-clones.

dissimilarity threshold and blind renaming of identifiers. For micro-clones using NiCad we detect block clones of a minimum size of 1 LOC and maximum size of 4 LOC with 20% dissimilarity threshold and blind renaming of identifiers as was detected by Mondal et al. [135]. For different settings of a clone detector the clone detection results can be different and thus, the findings on bugs in code clones can also be different. Hence, selection of appropriate settings (i.e., detection parameters) is important. We used the mentioned settings in our research for detecting regular clones, because Svajlenko and Roy [175] show that these settings provide us with better clone detection results in terms of both precision and recall.

4.4 Experimental Results

Our research questions are listed in Table 4.1. We represent answers to these research questions and analyze our experimental results in this section.

4.4.1 *RQ 1: Do micro-clones contain more bug-fix changes than regular code clones?*

Motivation. Finding the number of changes due to fixing a bug in a code clone is an important consideration when comparing regular and micro-clones. Intuitively, micro-clones might have a higher number of bug-fix changes than regular clones since the number of micro-clones is higher in every system. If code clones of a particular type (micro or regular) contain more bug-fix changes then we should be more careful for that kind of clone while performing software maintenance tasks.

Methodology. To answer this research question we automatically count the number of bug-fix commits for both regular and micro-clones. We found the result we expected i.e. number of bug-fix commits is higher in micro-clones than regular clones. Table 4.3 shows the experimental result for our first research question. While finding the bug-fix changes in micro-clones, we restrain the minimum number of characters per clone line. We refer it as the Minimum number of Characters per clone line or MC. We choose four different threshold values for MC (i.e. $MC \geq 1$, $MC \geq 10$, $MC \geq 20$, and $MC \geq 30$). We predicted that the number of bug-fix commits would increase with a decrease in MC value. We found the result to be as we expected. This is obvious because when we choose $MC \geq 1$, all the changes will be included. When we choose $MC \geq 30$, only clone lines containing more than 29 characters will be counted. For regular code clones we choose the default value of $MC \geq 1$. Our goal is to observe that even if we restrict the minimum number of characters per clone line for micro-clones, if there is any chance to get less number of bug-fix changes in micro-clones than regular code clones.

Figure 4.2 shows the percentages of bug-fix commits that have changed clone fragments in regular and micro-clones for six subject systems. We observe that for every subject system percentage of bug-fix commits is higher in micro-clones than regular clones. Also, if we increase the threshold of minimum characters per clone line (MC), the percentage decreases. This proves that even if we limit the minimum number of characters per clone line to equal or greater than 30 ($MC \geq 30$), still the number of bug-fix changes is higher in micro-clones than regular code clones. Overall, we can see that percentages of bug-fix commits is much lower in regular clones than micro-clones.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 1. To check the statistical significance of our experiment results we perform Mann-Whitney-Wilcoxon (MWW) test [9, 10]. We want to observe whether the percentage of commits that have changes due to fixing a bug is significantly higher in micro-clones than regular clones. MWW test is a non-parametric and does not require normal distribution of data. We consider significance level of 5% for the test. We investigate six subject systems and for each of them we calculate percentage of regular clones and micro-clones. For micro-clones we calculate four percentages of bug-fix commits considering four micro-clone sizes i.e. $MC \geq 1$, $MC \geq 10$, $MC \geq 20$, and $MC \geq 30$, where MC refers to the minimum number of characters per clone line. Thus, we compare the percentage from each set of micro-clones with that of regular clones. We found that percentages of bug-fix changes in micro-clones is significantly higher than regular clones. Table 4.4 shows that for different MC values p-values are less than 0.05 i.e. the significance level that we consider for our tests. From the data we can see that the critical value of U is 7 and for each case our value of U is less than or equal to 7. This data prevails strongly that the number of bug-fix commits is significantly higher in micro-clones than in regular code clones.

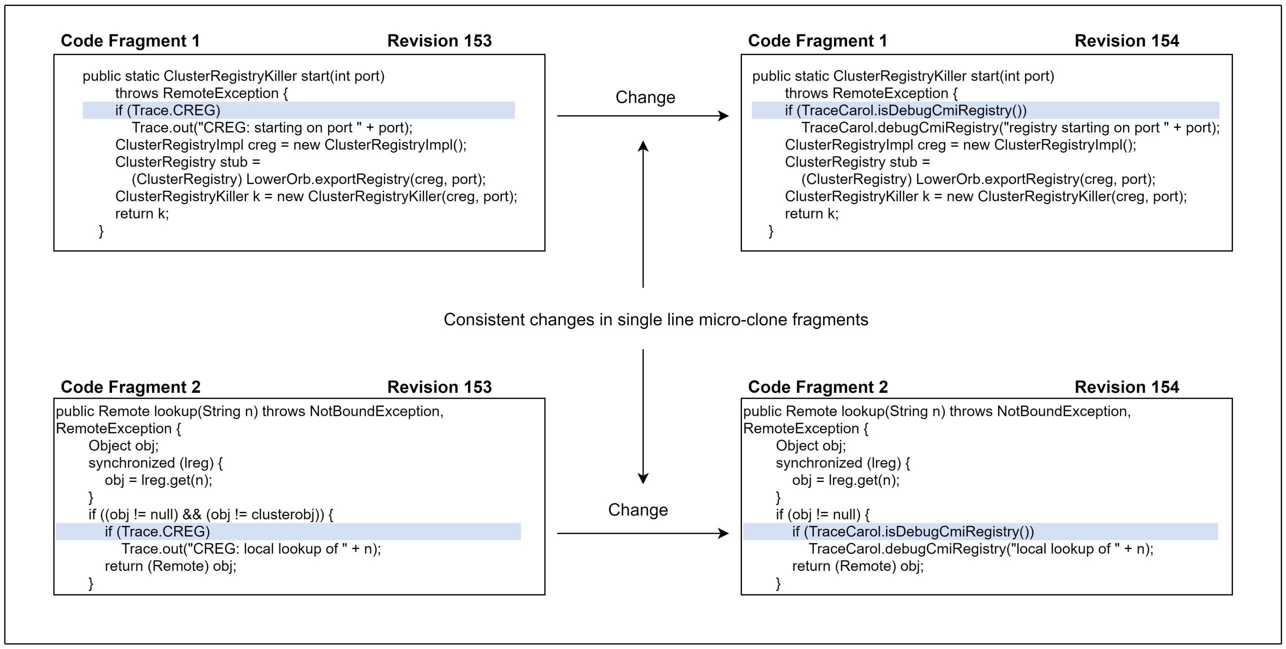
Answer to RQ 1. From our investigation and analysis we found that percentage of bug-fix commits containing changed clone fragments is significantly higher in micro-clones (from 4.52% to 7.48%) than

Table 4.3: Number of Bug-fix commits that have Changed Clone Fragments in Micro-clones and Regular Clones. ***MC*** = Minimum number of Characters per clone line.

Subject Systems	**Regular Clones**	**Micro-clones**			
		$MC \geq 1$	$MC \geq 10$	$MC \geq 20$	$MC \geq 30$
Ctags	53	433	362	237	118
Brlcad	29	214	182	126	78
MonoOSC	26	463	459	447	399
Freecol	358	3783	3692	3497	2888
Carol	450	745	715	648	503
Jabref	112	255	244	194	135

Table 4.4: Mann-Whitney-Wilcoxon Test Result for RQ1

Micro-Clones	**p-value**	**U value**
MC>= 1	0.01539	4
MC>= 10	0.01539	4
MC>= 20	0.03288	6
MC>= 30	0.04648	7
Considering level of significance is 5%. For 5% one-tailed level, Critical value of U is 7		



```
public static ClusterRegistryKiller start(int port)
      throws RemoteException {
      if (Trace.CREG)
         Trace.out("CREG: starting on port " + port);
      ClusterRegistryImpl creg = new ClusterRegistryImpl();
      ClusterRegistry stub =
         (ClusterRegistry) LowerOrb.exportRegistry(creg, port);
      ClusterRegistryKiller k = new ClusterRegistryKiller(creg, port);
      return k;
   }
```

```
public static ClusterRegistryKiller start(int port)
      throws RemoteException {
      if (TraceCarol.isDebugCmiRegistry())
         TraceCarol.debugCmiRegistry("registry starting on port " + port);
      ClusterRegistryImpl creg = new ClusterRegistryImpl();
      ClusterRegistry stub =
         (ClusterRegistry) LowerOrb.exportRegistry(creg, port);
      ClusterRegistryKiller k = new ClusterRegistryKiller(creg, port);
      return k;
   }
```

```
public Remote lookup(String n) throws NotBoundException,
RemoteException {
      Object obj;
      synchronized (lreg) {
         obj = lreg.get(n);
      }
      if ((obj != null) && (obj != clusterobj)) {
         if (Trace.CREG)
            Trace.out("CREG: local lookup of " + n);
         return (Remote) obj;
      }
```

```
public Remote lookup(String n) throws NotBoundException,
RemoteException {
      Object obj;
      synchronized (lreg) {
         obj = lreg.get(n);
      }
      if (obj != null) {
         if (TraceCarol.isDebugCmiRegistry())
            TraceCarol.debugCmiRegistry("local lookup of " + n);
         return (Remote) obj;
      }
```

Figure 4.3: This is an example of consistent changes in micro-clones. Code Fragment 1 and Code Fragment 2 contain a pair of single line micro-clone from our subject system Carol. Consistent changes in single line micro-clone are shown in between revision 153 and revision 154. We highlight the single line micro-clone in both code fragments and for both revisions. Here, similar changes occur at the commit operation on revision 153. We also observe that the surrounding code of one micro-clone fragment is not similar to that of the other micro-clone fragment.

regular clones (1.30%) in software systems.

From the above results of our first research question we can state that micro-clones experience changes in a significantly higher number of bug-fix commits compared to regular clones. This reveals that we should emphasize micro-clones during code clone management and software maintenance.

4.4.2 *RQ 2: Are the bug-fix changes consistent in micro and regular code clones?*

Motivation. From our first research question, we have found that the bug-fix commits occur more in micro-clones than regular clones in systems. However, we do not know whether these bug-fix commits are committing consistent changes or inconsistent changes in clone fragments. Consistent bug-fix changes means more than one clone fragment from the same group experienced the same bug-fix. If the same bug-fix change needs to be propagated to different clone fragments in the same group, then it indicates that the bug was replicated in different clone fragments. Replication of bugs is a severe negative impact of code cloning [72]. In RQ 2, we investigate whether the intensity of consistent bug-fix changes is higher in micro-clones or in regular code clones.

Methodology. In our first research question we find all the changes that occurred due to fixing a bug i.e. number of bug-fix commits in both regular clones and micro-clones. To answer our second research question, we find consistent bug-fix changes in regular and micro-clones. When similar changes occur in two or more clone fragments from the same clone class, we consider those changes as consistent changes. For instance, suppose CF_1 and CF_2 are two clone fragments in revision R. After the commit operation on revision R i.e. in revision $R+1$, the clone fragments CF_1 and CF_2 change to CF_1' and CF_2' respectively. If the changes between clone fragments CF_1 and CF_1' are similar to the changes between clone fragments CF_2 and CF_2', then we can consider the changes to be consistent. Figure 4.3 shows an example of consistent changes in micro-clones. Here, we can see that a single line micro-clone has been changed consistently. This example has been mined from our subject system Carol written in Java. In the commit operation on revision 153, a similar change occurred in revision 154 for both code fragments.

Table 4.5 shows the data of consistent changes of clones found in regular and micro-clones for six subject systems. Here, we can see that the total number of consistent bug-fix changes is higher in micro-clones than regular code clones for every subject system.

Figure 4.4 shows the percentage of consistently changed bug-fix commits in regular and micro-clones for six subject systems along with overall values. Here, we observe that for three systems, Ctags, MonoOSC, Freecol, the percentage of consistent changes is higher in micro-clones than regular code clones. On the other hand, for rest of the systems, i.e. Brlcad, Carol, Jabref, the percentage of consistent changes is higher in regular code clones than micro-clones. Overall, the percentage of consistent changes is approximately equal in both regular and mirco-clones. Though the result shows that there is no significant difference in the percentage of consistent changes, we believe that the actual number of consistent bug-fix changes is more important. All the bugs need to be fixed for ensuring consistency of the software system. We found that the total number of consistent bug-fix changes in regular clones is 728. On the other hand, the total number of consistent changes due to bug-fix commits in micro-clones is 4,118. For the purpose of clone refactoring or clone tracking, we have to deal with the actual number of clones. Thus, we need to emphasize on micro-clones which have higher number of clones as well as consistent changes of bug-fix commits. To evaluate the significance of the difference between regular and micro-clones, we do the MWW statistical test.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 2. To investigate statistical significance we perform Mann-Whitney-Wilcoxon test [9, 10] on the results. We found that for two-tailed, 5% significant level and critical value of U is 5, the difference between two percentages of consistent bug-fix changes in regular and micro-clones is not statistically significant. The MWW test shows that the p-value is 0.93624 which is greater than 0.05 and the value of U found 17 which is greater than the critical value of U i.e. 5. Though the result shows that there is no significant difference in the percentage of consistent changes, we should emphasize on the number of consistent bug-fix changes as we stated above. In terms of consistency, we have to observe the number of clones rather than percentages. We found that the average number of consistently changed

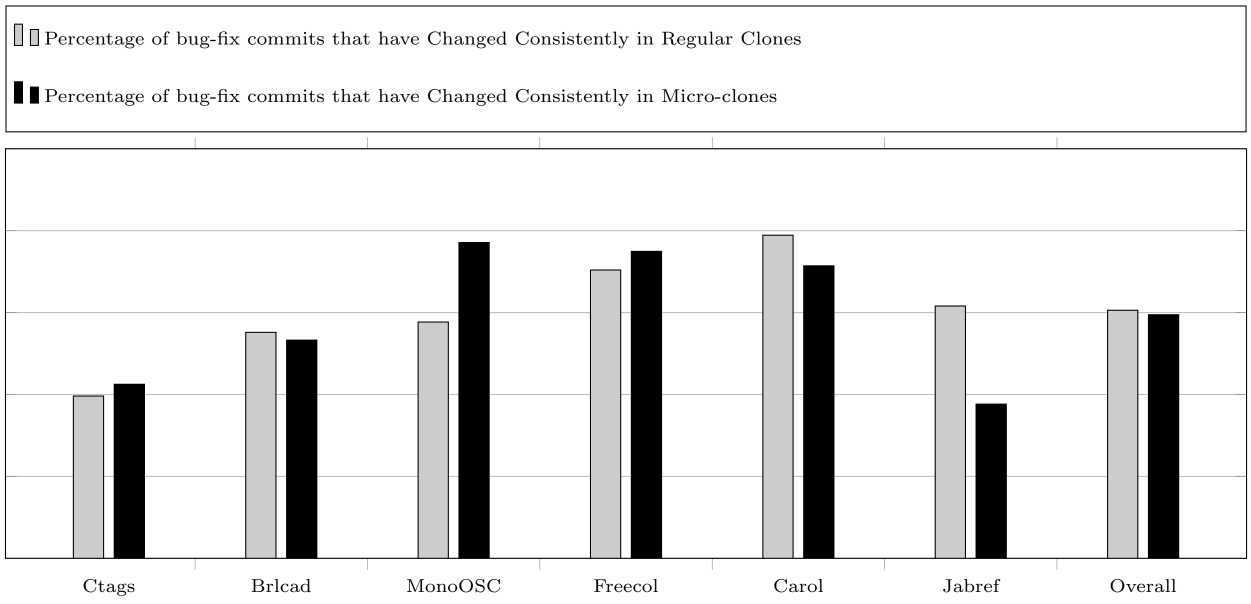


Figure 4.4: Percentage of bug-fix commits that have consistent changes of clones in regular and micro-clones.

bug-fix commits is 121.33 for regular clones and for micro-clones it is 686.33. Hence, the number of consistent changes in micro-clones is higher than that of regular code clones.

Answer to RQ 2. After investigating the bug-fix changes in clones we found that number of consistent bug-fix changes is significantly higher in micro-clones than in regular clones. The difference between two average numbers is 565.

From the above observation we can state that number of consistent bug-fix changes are more in micro-clones than regular clones. This finding is important for refactoring and tracking of code clones.

4.4.3 *RQ 3: What percentage of files get affected for fixing bugs in micro and regular clones?*

Motivation. It is important to answer this research question because the impact of bug-fix commits on micro and regular clones is revealed through this answer. Intuitively, a higher percentage of affected files indicates a higher number of changes in the system. More attention is needed when more changes occur. Answer to this research question implies which type of code clones (regular or micro clone) are affecting the system more. Knowing the information we can emphasize on that particular type of code clone while maintaining and managing code clones.

Methodology. To answer our third research question first we find out total number of files those have been changed after a bug-fix commit during software evolution. Then we find the total number of files which get affected due to fixing a bug in regular clones and micro-clones respectively. Thus, we calculate the

Table 4.5: Number of Consistent Changes in Regular and Micro-Clones

Subject Systems	Regular Clones	Consistent Regular Clones	Micro Clones	Consistent Micro Clones
Ctags	53	21	433	184
Brlcad	29	16	214	114
MonoOSC	26	15	463	357
Freecol	358	252	3783	2835
Carol	450	355	745	532
Jabref	112	69	255	96

percentage of files that get affected due to bug-fix commits. Table 4.6 shows the result of our investigation on RQ3. Here, we can see that number of affected files due to bug-fix commits is higher in micro-clones than regular code clones. Figure 4.5 illustrates the percentage of files that get affected due to bug-fix commits for six subject systems. In this figure we can see that for each subject system, percentage of affected files is higher in micro-clones than regular clones. The difference in percentage between micro and regular clones is the highest in Freecol subject system. Overall, percentage of affected files due to bug-fix commit in regular clone is 8.02% and for micro-clone the percentage is 39.15%. To understand the statistical significance of the difference between regular and micro-clones, we perform following MWW test for this research question.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 3. We perform MWW test [9, 10] to observe the statistical significance level of our experimental result. For a two tailed and significance level 0.05, we get the critical value of U is 5. MWW test results show that our value of U is 1 which is less than 5. Also, we found the p-value is 0.0083 which is much less than significance level 0.05. Thus, we find that percentage of files that get affected by bug-fix commits is significantly higher in micro-clones than regular code clones.

Answer to RQ 3. According to our investigation we found that percentage of files that get affected due to fixing a bug in the software systems is significantly higher in micro-clones than regular code clones.

During software maintenance it is important to manage clones intelligently. For this purpose finding which type of code clones are important is a crucial issue. From our above experiment we can state that micro-clones are more important than regular clones while managing code clones in software systems.

Table 4.6: Number of Files that Changes in Regular and Micro-Clones

Subject Systems	**Total Number of Files**	**Regular Clones**	**Micro Clones**
Ctags	116	9	41
Brlcad	153	3	36
MonoOSC	132	5	32
Freecol	310	52	269
Carol	367	49	186
Jabref	377	17	54

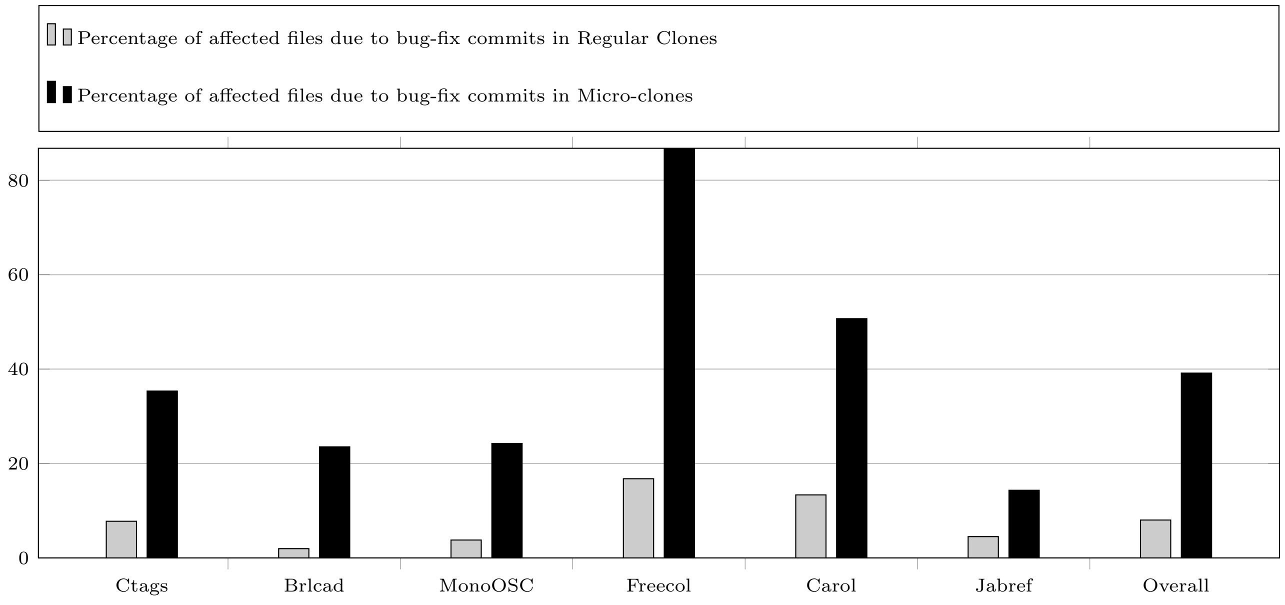


Figure 4.5: Percentage of files that get affected by bug-fix commits in regular and micro-clones.

Table 4.7: Mann-Whitney-Wilcoxon Test Result for RQ2, RQ3 and RQ4

Research Question No.	**p-value**	**U value**	**Critical Value of U**
RQ2	0.93624	17	5
RQ3	0.0083	1	5
RQ4	0.04846	11	11
Considering level of significance is 5%. For RQ3 and RQ4, U value <= Critical value of U			

4.4.4 *RQ 4: Do micro-clones contain more severe bugs compared to regular code clones?*

Motivation. From our previous research questions we tried to find out percentage of changes, consistent changes and affected files due to fixing a bug during software evolution. Still we need to understand which type of clone is a severe threat in our systems while software maintenance. Severe bugs can be more harmful than non-severe bugs. Fixing a severe bug is an emergency task compared to fixing a non-severe bug. For this purpose we observe and compare the severity of bugs in both regular and micro-clones.

Methodology. To distinguish between severe and non-severe bugs in regular and micro-clones we automatically perform a heuristic search in bug-fix commit messages of regular and micro clone code. Lamkanfi et al. [98] proposed a list of keywords which identify severe and non-severe bugs from textual information. We search these keywords in bug fixing commit messages to mine the severe bug-fixing revisions. There are different levels or categories of severity in a bug report. However, we consider only two levels of severity i.e. true or false for the simplicity of our experiment. Table 4.8 shows the result of our heuristic search. Here, we can see that number of severe bugs is low in regular clones compared with micro-clones. In fact, two subject systems, Brlcad and MonoOSC contain no severe bugs in their system for regular code clones. Highest number of severe bug-fixing commit found in Freecol where the number of micro-clones is also very high. Figure 4.6 depicts the percentage of severe bugs in six subject systems for both regular and micro-clones. For each of the subject systems percentage of severe bugs is high in micro-clones than regular clones except for Freecol. Overall, micro-clones contain 26.82% of severe bugs and regular clones contain 9.84% of severe bugs.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 4. To understand whether the difference between two percentages is significant or not we perform MWW test [9, 10]. For significance level of 5% and one-tailed MWW test, we find the critical value of U is 11. After performing MWW test on our data set of severe bugs for regular and micro-clones, it shows that U value is 11 which is equal to the critical value of U i.e. 11.

Table 4.8: Number of Severe Bugs in Regular and Micro-Clones

Subject Systems	**Regular Clones**	**Severe Bugs in Regular Clones**	**Micro Clones**	**Severe Bugs in Micro Clones**
Ctags	53	12	433	113
Brlcad	29	0	214	9
MonoOSC	26	0	463	394
Freecol	358	65	3783	607
Carol	450	46	745	91
Jabref	112	9	255	44

Also, the p-value of our data set is 0.04846 which is less than 0.05 i.e. significance level. This proves that the percentage of severe bug-fix commits is significantly higher in micro-clones than regular clones. Table 4.7 shows the MWW test results for our second, third and fourth research questions.

Answer to RQ 4. After observing the severity of the bug-fix commits in both regular and micro-clones, we found that percentage of severe bugs is significantly higher in micro-clones than regular clones.

Since severe bugs need to be fixed immediately, it is a vital issue for fixing a severe bug during software maintenance. From the above investigation, we can state that micro-clones contain more severe bugs than regular clones for which we have to deal micro-clones carefully than regular clones while software maintenance.

4.5 Manual Analysis

We manually investigate the types of bug-fix changes that occurred in micro-clones. Knowing this information is helpful for programmers so that he or she can be more careful while copying the code fragments containing those statements. The result of our manual investigation will be helpful for preventing bug-proneness in code bases in advance. Since, manually investigating all the bug-fix changes is not feasible, we inspect first 50 distinct changes from each of our subject systems. Thus we observe $50 \times 6 = 300$ distinct changes in total. For each change observation the average time is 10 minutes. Hence, for six subject systems and first fifty distinct changes from each of them, we spent approximately $(50 \times 6 \times 10) = 3000$ minutes = 50 hours. Moreover, there were some cases which implies one bug-fix change falls into more than one change types. In this case, we choose most appropriate one. We categorize the bug-fix changes and then count them for

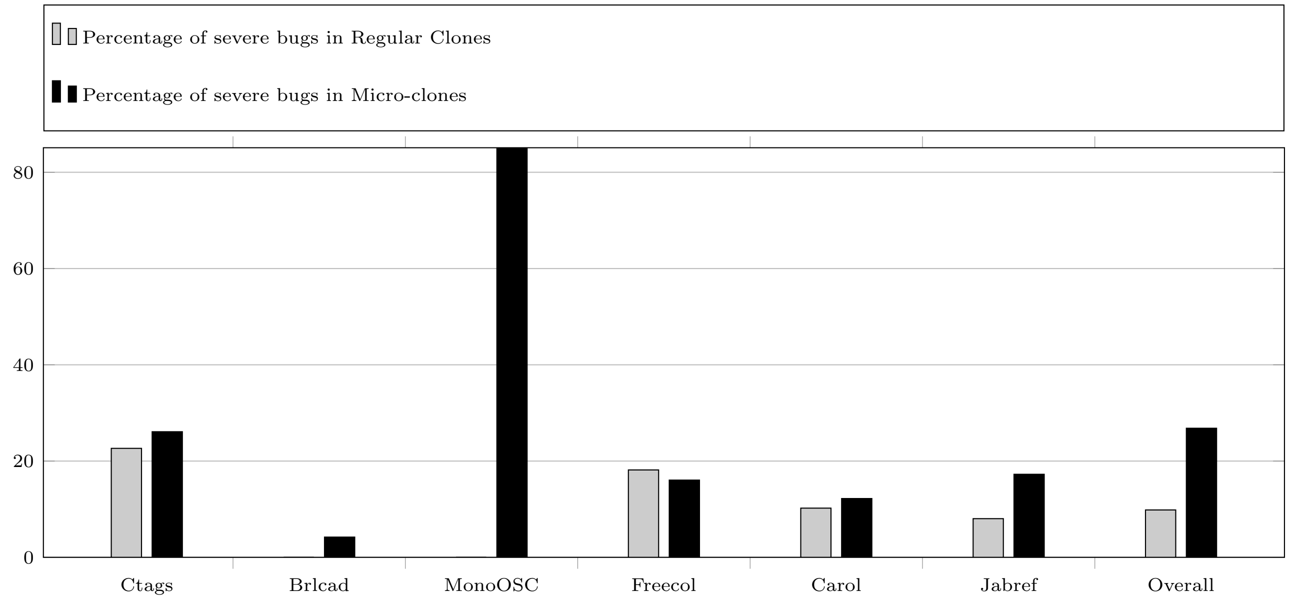


Figure 4.6: Percentage of severe bugs in bug-fix commits in regular and micro-clones.

each category. In total we define 37 categories of bugs. To visualize the changes in bug-fixing micro-clones instantly, we use an online difference checker in code [3].

Table 4.9 shows the list of most frequent change types in bug-fix changes in micro-clones. Here, we can see that "Deletion of Statement" is the highest in number i.e. 70 in total and for subject system MonoOSC has the highest number of this type of change i.e. 23. Second and third frequent changes are "Addition of Statement" and "Modification of Statement" i.e. 38 and 34 respectively. Again, MonoOSC contains the highest number of "Addition of Statement" types i.e. 21. The other frequent changes are "Modification of Function Parameter", "Modification of if-condition", "Deletion of Function Call" and "Modification of Function Call". We found that most of the change types in MonoOSC is "Addition of Statement" or "Deletion of Statement".

Finding the type of bugs or code construction for micro-clones is important because more frequent type of bug should be considered more carefully during software development. Since from our manual analysis we found that most change type is "Deletion of Statement", so when a developer deletes any statement she should be more cautious for not creating any bugs. The other frequent change types during bug-fixes in micro-clones are "Addition of Statement", "Modification of Statement", and "Modification of Function Parameter".

4.6 Related Work

A micro-clone is a recent concern of the code clone research area. The term *micro-clones* was first introduced by Beller et al. [40]. They have identified and statistically proved that majority of software bugs in micro-clones occur in the last line or statement of micro-clones. Tonder et al. [191] agreed with them and proposed detection and removal of micro-clones at a large scale. Both Beller's and Tonder's paper used PVS-Studio

Table 4.9: Most Frequent Change-Types During Fixing Bugs in Micro Clones

	Change Types	**Ctags**	**Brlcad**	**MonoOSC**	**Freecol**	**Carol**	**Jabref**	**Total**
1	Deletion of Statement	15	7	23	6	10	9	70
2	Addition of Statement	3	1	21	7	5	1	38
3	Modification of Statement	5	6	1	7	5	10	34
4	Modification of Function Parameter	8	8	2	3	1	2	24
5	Modification of if-condition	4	6	1	1	3	2	17
6	Deletion of Function Call	3	0	0	4	2	6	15
7	Modification of Function Call	3	5	0	1	3	0	12

[14] static analysis tool to detect faulty micro-clones. Also, Tonder et al. [191] used Boa [51] software mining infrastructure which contains parsed ASTs for all Java files in 380,125 Java repositories on GitHub and a domain-specific language (DSL). They [191] found that 95% of their pull requests from active GitHub repositories merged quickly and 76% of their accepted patches are removing (REM category) micro-clones. Previous studies ignore micro-clones stating that code clones of less than 6 LOC are not promising [41]. In contrast, Mondal et al. [135] investigated the importance of micro-clones during software evolution. They showed that micro-clones have a very high tendency of getting updated consistently. Recently, Mondal et al. [125] investigated near-miss micro-clones and found that consistent updates occur in near-miss micro-clones more often than exact micro-clones and regular code clones.

None of the studies discussed above investigated bugs in micro-clones and regular code clones simultaneously. Mondal et al. [131] investigated bug-proneness of code clones. While the primary target of that study was to compare the bug-proneness of three clone-types (Type 1, Type 2, and Type 3), our target is to compare the bug-proneness of micro-clones and regular code clones. Mondal et al. [131] did not investigate the bug-proneness of micro-clone code in their study. Higo et al. [64] proposed a method to distinguish problematic code clones from non-problematic code clones. They stated that not all clones are problematic for the systems. Thus, it is important to find and fix the problematic code clones. As a result of their study, they found 22 problematic code clones.

A new aspect has been discussed in a recent study [134] showing that bug-proneness is related with how recently the clone has been changed on a subject system. The more recent the changes happened the greater the possibility of bugs occurring. In another study, Mondal et al. [138] investigated bug propagation in code cloning and found that 33% of bug-fixing code clones contain propagated bugs. They have suggested to prioritize these clones for refactoring and tracking. It is noticeable in their research that near-miss code clones contain more propagated bugs than identical code clones. A context aware bug in code clone has been studied [124] recently and found that nearly 50% of clone related bug-fixings are context-bug-fixing. However, Mondal et al. [134, 138, 124] did not investigate bug-proneness of micro-clones.

On the other hand, from a different perspective Rahman and Roy [148] show the relation between stability and bug-proneness of code clones. They investigated five open source diverse subject systems written in Java. They found statistically significant relation between stability and bug-proneness of code clones. Also, buggy clones have the tendency to change more often than non-buggy clones. Moreover, they found the bug-proneness of Type 2 and Type 3 clones is strongly related to their stability compared to Type 1 clones. They also investigated from a fine-grained change perspective and found Type 2 and Type 3 clones are mostly influenced by changes of low to medium significance. However, they did not investigate bug-proneness of micro-clones in their study. We investigate bug-proneness of micro-clones in our study.

Rakibul and Zibran [79] perform a comparative investigation in between buggy and non-buggy clone code. They have studied on three open source software systems written in Java containing 2,077 revisions in total. Using SourceMeter [15] they have observed 29 source code quality metrics to characterize the buggy clone code. Their approach to finding bugs in code clones is similar to other studies like [134, 147]. They have found that buggy clones have significantly higher complexity and lower maintainability than non-buggy clone code. Also, size of the method of buggy clone is higher than non-buggy clone method. While Rakibul and Zibran investigated regular code clones, we investigate the bug-proneness of micro-clones in our study.

We see that a number of studies have been conducted on the bug-proneness of regular code clones. However, bug-proneness of micro-clones have been ignored. Focusing on this we perform an in-depth investigation on bug's impacts in micro code clones and regular code clones in our research. Our experimental results are promising and provide useful implications for better understanding of the bug-proneness of micro-clone and regular clone code.

4.7 Limitations

We used the NiCad clone detector [49] for detecting both micro and regular clones. While all clone detection tools suffer from the *confounding configuration choice problem* [197] and might give different results for different settings of the tools, the setting that we used for NiCad for this experiment are considered standard [153] and with these settings NiCad can detect clones with high precision and recall [156, 155, 175]. Thus, we believe that our findings on the bug-proneness of micro code clones and regular code clones are of significant importance.

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique proposed by Mocus and Votta [123] and also used by Barbour et al. [36]. The technique proposed by Mocus and Votta [123] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [36] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

The number of total subject systems is not enough in our research to be able to generalize our findings regarding the comparative bug-proneness of micro and regular clones. However, our candidate systems were of diverse variety in terms of application domains, sizes and revisions. Thus, we believe that our findings are important from the perspectives of managing code clones.

4.8 Summary

In this chapter, we investigate and compare the bug-proneness and their characteristics in between regular and micro-clones. From our investigation on six diverse subject systems, we found that micro-clones need to be handled more carefully than regular clones during software maintenance. We have found that changes to clone fragments due to fixing a bug occur more in micro-clones than regular clones. Moreover, total number of consistent changes due to bug-fix commits is higher in micro-clones than regular clones. Also, percentage of files that get affected due to bug-fix changes is higher in micro-clones than regular clones. Additionally, we found that percentage of severe bugs is higher in micro-clones than regular code clones. We believe that these findings are important for clone management specifically for managing micro-clones. Considering the findings of this study, in our future research we would like to investigate replicated bugs in micro-clones.

5 Bug Replication in Code Clones

From our previous two studies, we have have found that clone code are more responsible to produce bugs. This motivates us to explore more on code clones. Hence, in this chapter we studied bug-replication of clone code. Intuitively, if a buggy code fragment get copied and pasted several times in the same code base then it is likely to replicate that bug of the original copy and thus occur bug-replication through code cloning. In Section 5.4 replication of bug in clone code has been described thoroughly.

Existing studies show that clones are directly related to bugs and inconsistencies in the code-base. Code cloning (making code clones) is suspected to be responsible for replicating bugs in the code fragments. However, there is no study on the possibilities of bug-replication through cloning process. Such a study can help us discover ways of minimizing bug-replication. Focusing on this we conduct an empirical study on the intensities of bug-replication in the code clones of the major clone-types: Type 1, Type 2, and Type 3. According to our investigation on thousands of revisions of six diverse subject systems written in two different programming languages, C and Java, a considerable proportion (i.e., up to 10%) of the code clones can contain replicated bugs. Both Type 2 and Type 3 clones have higher tendencies of having replicated bugs compared to Type 1 clones. Thus, Type 2 and Type 3 clones are more important from clone management perspectives. The extent of bug-replication in the buggy clone classes is generally very high (i.e., 100% in most of the cases). We also find that overall 55% of all the bugs experienced by the code clones can be replicated bugs. Our study shows that replication of bugs through cloning is a common phenomenon. Clone fragments having *method-calls* and *if-conditions* should be considered for refactoring with high priorities, because such clone fragments have high possibilities of containing replicated bugs. We believe that our findings are important for better maintenance of software systems, in particular, systems with code clones.

The rest of the chapter is organized as follows. We introduce this chapter in Section 5.1. Section 5.2 contains the terminology, Section 5.3 discusses the experimental steps, Section 5.4 describes the process of identifying the replicated bugs, Section 5.5 answers the research questions by presenting and analyzing the experimental results, Section 5.6 discusses the related work, Section 5.7 mentions the possible threats to validity, and Section 5.8 concludes the chapter by mentioning possible future work.

5.1 Introduction

If two or more code fragments in a software system's code-base are exactly or nearly similar to one another we call them code clones [151, 157]. A group of similar code fragments forms a clone class. Code clones are mainly created because of the frequent copy/paste activities of the programmers during software development and maintenance. Whatever may be the reasons behind cloning, code clones are of great importance from the perspectives of software maintenance and evolution [157].

It is suspected that cloning is responsible for replicating bugs [157]. If a particular code fragment contains a bug and a programmer copies that code fragment to several other places in the code-base without the knowledge of the existing bug, the bug in the original fragment gets replicated. Fixing of such replicated bugs may require increased maintenance effort and cost for software systems. However, although cloning is suspected to be responsible for replicating bugs, there is no study on the possibilities of bug-replication through cloning. Such a study can provide us helpful insights for minimizing bug-replication as well as for prioritizing code clones for refactoring or tracking. Focusing on this, we conduct an in-depth empirical study regarding bug-replication in the code clones of the major clone-types: Type 1, Type 2, Type 3.

We conduct our empirical study on thousands of revisions of six diverse subject systems written in two different programming languages (Java and C). We detect code clones from each of the revisions of a subject system using the NiCad [49] clone detector, analyze the evolution history of these code clones, and investigate whether and to what extent they contain replicated bugs. We answer four important research questions (Table 5.1) regarding the intensity and cause of bug-replication through our investigation. According to our investigation involving rigorous manual analysis we can state that:

(1) A considerable percentage of the code clones can be related to bug-replication. According to our observations up to 10% of the code clones in a software system can contain replicated bugs.

(2) Both Type 2 and Type 3 clones have a higher possibility of containing replicated bugs compared to Type 1 clones. Thus, Type 2 and Type 3 clones should be given higher priorities for management.

(3) A considerable proportion (around 55%) of the bugs occurred in code clones can be replicated bugs.

(4) Most of the replicated bugs are related to the *method-calls* and *if-conditions* residing in the clone fragments. Thus, clone fragments containing *method-calls* and/or *if-conditions* should be considered for refactoring or tracking with high priorities.

Our findings imply that bug-replication tendencies of code clones should be taken in proper consideration when making clone management decisions. The findings from our study are important for better management of code clones as well as for better maintenance of software systems.

5.2 Background

5.2.1 Similarity Preserving Co-change (SPCO)

Let us consider two code fragments, CF1 and CF2, which are clones of each other in revision R of a subject system. A commit operation was applied on revision R and both of these two fragments were changed (i.e., the clone fragments co-changed) in such a way that they were again considered as clones of each other in the next revision R+1 (i.e., created because of the commit). In other words, the clone fragments preserved

Table 5.1: Research Questions

SL	Research Question
RQ 1	What percentage of the clone fragments in different clone-types takes part in bug-replication?
RQ 2	What is the extent of bug-replication in the buggy clone classes of different types of clones?
RQ 3	What percentage of the bugs that were experienced by the code clones of different clone-types are replicated bugs?
RQ 4	Which types of statements are highly related to bug-replication?

their similarity even after experiencing changes in the commit operation. Thus, we call this co-change of clone fragments (i.e., change of more than one clone fragment together) a Similarity Preserving Co-change (SPCO).

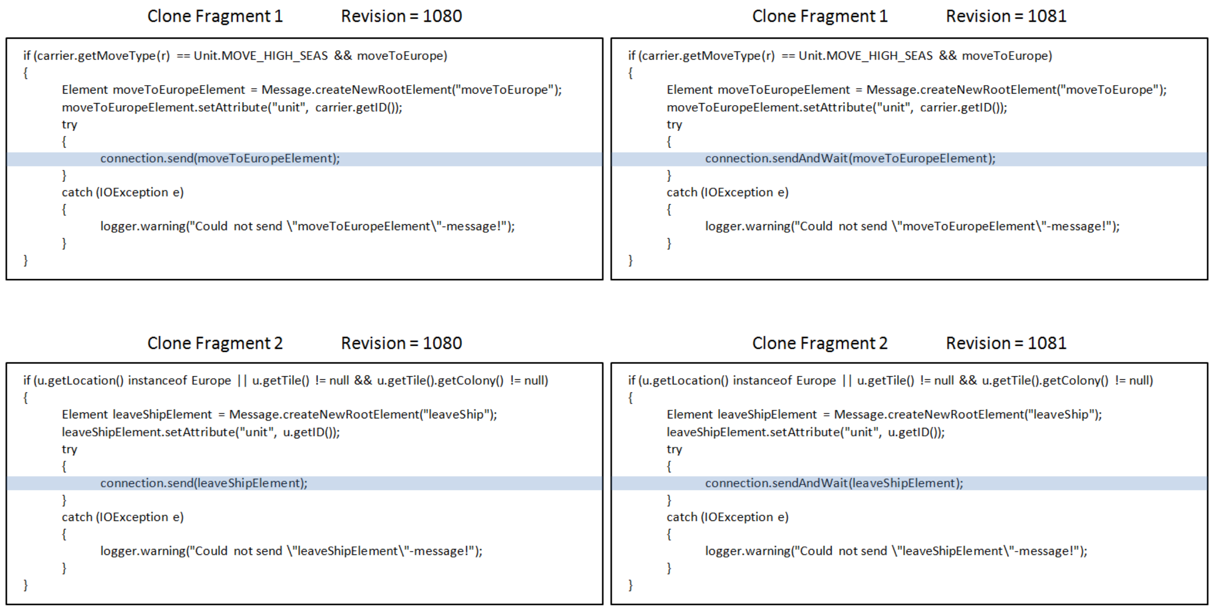
In a previous study [130] we showed that in a similarity preserving co-change (SPCO) more than one clone fragment from the same clone class are changed together consistently (i.e., the clone fragments are changed in the same way). Fig. 5.1 shows an example of SPCO of two Type 3 clone fragments from our subject system Freecol. The figure caption includes the details regarding the example. The figure demonstrates that the two clone fragments were changed together consistently in a particular commit operation for fixing a replicated-bug.

5.2.2 Late Propagation in Code Clones

Late propagation is defined as the occurrence of one or more inconsistent changes of a clone pair followed by a re-synchronizing change. The re-synchronization of the code clones indicates that the inconsistent changes were unintentional [35].

5.2.3 Late Propagation in Literature

Late propagation is another source of bug introduction in software systems. A number of studies have been conducted on late propagation in clones and its relationships with bugs. Barbour et al. [35] investigated eight different patterns of late propagation considering Type 1 and Type 2 clones of three subject systems and identified those patterns that are likely to introduce bugs and inconsistencies to the code-base. Mondal et al. [133] worked on Type 3 code clones along with Type 1 and Type 2 code clone to investigate late propagation. They have found that late propagation is high and more intense in Type 3 code clones than other two types of clone and also block clones cause more late propagation than method clones. They have suggested refactoring of clones to mitigate the inconsistencies that have aroused due to the late propagation in the systems. Later, they have extended their study in a journal paper [136] by extending the number of


Clone Fragment 1 Revision = 1080

```
if (carrier.getMoveType(r) == Unit.MOVE_HIGH_SEAS && moveToEurope)
{
        Element moveToEuropeElement = Message.createNewRootElement("moveToEurope");
        moveToEuropeElement.setAttribute("unit", carrier.getID());
        try
        {
                connection.send(moveToEuropeElement);
        }
        catch (IOException e)
        {
                logger.warning("Could not send \"moveToEuropeElement\"-message!");
        }
}
```

Clone Fragment 1 Revision = 1081

```
if (carrier.getMoveType(r) == Unit.MOVE_HIGH_SEAS && moveToEurope)
{
        Element moveToEuropeElement = Message.createNewRootElement("moveToEurope");
        moveToEuropeElement.setAttribute("unit", carrier.getID());
        try
        {
                connection.sendAndWait(moveToEuropeElement);
        }
        catch (IOException e)
        {
                logger.warning("Could not send \"moveToEuropeElement\"-message!");
        }
}
```

Clone Fragment 2 Revision = 1080

```
if (u.getLocation() instanceof Europe || u.getTile() != null && u.getTile().getColony() != null)
{
        Element leaveShipElement = Message.createNewRootElement("leaveShip");
        leaveShipElement.setAttribute("unit", u.getID());
        try
        {
                connection.send(leaveShipElement);
        }
        catch (IOException e)
        {
                logger.warning("Could not send \"leaveShipElement\"-message!");
        }
}
```

Clone Fragment 2 Revision = 1081

```
if (u.getLocation() instanceof Europe || u.getTile() != null && u.getTile().getColony() != null)
{
        Element leaveShipElement = Message.createNewRootElement("leaveShip");
        leaveShipElement.setAttribute("unit", u.getID());
        try
        {
                connection.sendAndWait(leaveShipElement);
        }
        catch (IOException e)
        {
                logger.warning("Could not send \"leaveShipElement\"-message!");
        }
}
```

Figure 5.1: Similarity Preserving Co-change (SPCO) example. This figure demonstrates an example of SPCO for two Type 3 clone fragments (Clone Fragment 1, and Clone Fragment 2) in the commit operation applied on revision 1080. We take this example from our subject system Freecol. The snapshots of each clone fragment in the two revisions 1080 and 1081 are shown in the figure, and the changes are highlighted. We can easily understand that in case of each clone fragment, a method named 'send' was replaced by a method named 'sendAndWait'. We see that the clone fragments were changed in the same way. The comment from the programmer regarding this change says that *Fixed a potential synchronization bug. "endAndWait" should be used instead of "send" in order to ensure that the server has completed handling the request before we make any modifications to the model. This is a necessary because the server and the AI share the same model.* From the changes to the clone fragments and programmer comments we realize that this SPCO is an example of fixing a replicated bug.

Table 5.2: Subject Systems

Systems	Language	Domains	LLR	Revisions
Ctags	C	Code Def. Generator	33,270	774
Camellia	C	Image Processing Library	89,063	170
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Management	45,515	1545
LLR = LOC in the Last Revision				

subject systems. They have found that the increased amount of late propagation in Type 3 clones are buggy late propagation. Additionally, they have suggested in their journal paper that refactoring and tracking of Similarity Preserving Change Pattern (SPCP) clones (details of SPCP will be found in [130]) will help to reduce the frequency of late propagation in clones. Recently in another journal paper, Mondal et al. [132] have discovered that there is no relation between bugs and late propagation. Late propagation was detected by analyzing clone genealogies through software evolution history. According to their research, only 1.4% of bug-fixing in clone code is related to late propagation as well as only 10.76% cases of late propagation in clones are related with bug.

5.3 Experimental Steps

We perform our investigation on six subject systems (Table 5.2) downloaded from Sourceforge [16]. Fig. 5.2 shows the work procedure of our experimental steps.

As demonstrated in Fig. 5.2, we perform the following experimental steps for detecting replicated bugs: **(1)** Extraction of all revisions (as mentioned in Table 5.2) of each of the subject systems from the online SVN repository; **(2)** Method detection and extraction from each of the revisions using CTAGS [2]; **(3)** Detection and extraction of code clones from each revision by applying the NiCad [49] clone detector; **(4)** Detection of changes between every two consecutive revisions using *diff*; **(5)** Locating these changes to the already detected methods as well as clones of the corresponding revisions; **(6)** Locating the code clones detected from each revision to the methods of that revision; **(7)** Detection of method genealogies considering all revisions using the technique proposed by Lozano and Wermelinger [24]; **(8)** Detection of clone genealogies by identifying the propagation of each clone fragment through a method genealogy; **(9)** Detection of SPCOs by analyzing clone change patterns; **(10)** Detection of bug-fix commit operations; and **(11)** Detection of

Table 5.3: NiCad Settings for three types of clones

Clone Types	**Identifier Renaming**	**Dissimilarity Threshold**
Type 1	none	0%
Type 2	blindrename	0%
Type 3	blindrename	20%

replicated-bugs in code clones. For completing the first nine steps we use the tool SPCP-Miner [137]. For the details of these steps we refer the interested readers to our earlier work [116]. We will describe the detection of bug-fix commits later in this section. In Section 5.4 we will describe how we detect replicated bugs.

We use NiCad [49] for detecting clones because it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [156, 155]. Using NiCad we detect block clones including both exact (Type 1) and near-miss (Type 2, Type 3) clones of a minimum size of 10 LOC with 20% dissimilarity threshold (for Type 3 clones) and blind renaming of identifiers. NiCad settings for detecting three clone-types (Type 1, Type 2, and Type 3) are mentioned in Table 5.3. These settings are explained in detail in our earlier work [116]. For different settings of a clone detector the clone detection results can be different and thus, the findings regarding bug-replication in code clones can also be different. Thus, selection of reasonable settings (i.e., detection parameters) is important. We used the mentioned settings in our research, because in a recent study [175] Svajlenko and Roy show that these settings provide us with better clone detection results in terms of both precision and recall. Moreover, code clones with a minimum size of 10 LOC are more appropriate from maintenance perspectives [152, 157, 37]. Before using the NiCad outputs of Type 2 and Type 3 cases, we processed them in the following way.

(1) Every Type 2 clone class that exactly matched any Type 1 clone class was excluded from Type 2 outputs.

(2) Every Type 3 clone class that exactly matched any Type 1 or Type 2 class was excluded from Type 3 outputs.

We processed NiCad clone detection results in the mentioned ways because we wanted to investigate bug-replication in three types of clones separately.

Clone Genealogies of Different Clone-Types. SPCP-Miner [137] detects clone genealogies considering each clone-type (Type 1, Type 2, and Type 3) separately. Considering a particular clone-type it first detects all the clone fragments of that particular type from each of the revisions of the candidate system. Then, it performs origin analysis of these detected clone fragments and builds the genealogies. Thus, all the instances in a particular clone genealogy are of a particular clone-type. An instance is a snap-shot of a clone

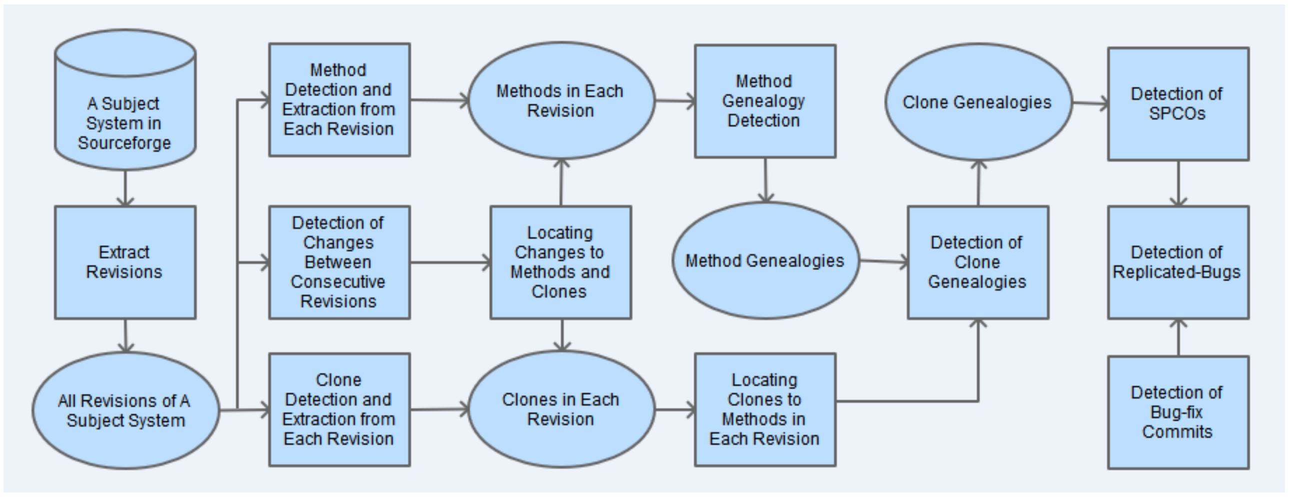

A Subject System in Sourceforge
Extract Revisions
All Revisions of A Subject System
Method Detection and Extraction from Each Revision
Detection of Changes Between Consecutive Revisions
Clone Detection and Extraction from Each Revision
Methods in Each Revision
Locating Changes to Methods and Clones
Clones in Each Revision
Method Genealogy Detection
Method Genealogies
Locating Clones to Methods in Each Revision
Clone Genealogies
Detection of Clone Genealogies
Detection of SPCOs
Detection of Replicated-Bugs
Detection of Bug-fix Commits

Figure 5.2: The execution flow diagram of the experimental steps in detecting replicated bugs in code clones. The rectangles demonstrate the steps.

fragment in a particular revision. A detailed elaboration of the genealogy detection approach is presented in our previous study [130]. As we obtain three separate sets of clone genealogies for three different clone-types, we can easily determine and compare the bug-replication tendencies of these clone-types.

Tackling Clone-Mutations. Xie et al.[200] found that mutations of the clone fragments (i.e., a particular clone fragment may change its type) might occur during evolution. If a particular clone fragment is considered of a different clone-type during different periods of evolution, SPCP-Miner extracts a separate clone-genealogy for this fragment for each of these periods. Thus, even with the occurrences of clone-mutations, we can clearly distinguish which bugs were experienced by which clone-types.

5.4 Replicated Bugs in Code Clones

In Sub-section 5.2.1 we defined SPCO (Similarity Preserving Co-change) of two or more clone fragments from the same clone class. In a previous study [130] we found that in an SPCO the participating clone fragments (i.e., from the same clone class) are likely to be changed consistently (i.e., each of the participating clone fragments is likely to be changed in the same way). We detect replicated bugs in code clones on the basis of this finding.

5.4.1 Identifying Replicated Bugs in Code Clones

If two or more clone fragments of a clone class experience an SPCO in a bug-fix commit operation (i.e., if two or more clone fragments are changed together consistently, that means in the same way, for fixing a bug), then we understand that those clone fragments contained the same bug, and for fixing that bug each of the fragments was changed in the same way. Thus, this case is an example of fixing a replicated bug in code clones.

5.4.2 Steps in Identifying Replicated Bugs in Code Clones

By analyzing the clone evolution history of a software system we identify the cases of fixing replicated bugs in code clones. Our identification consists of the following two steps.

Step 1: In this step we detect all the bug-fix commit operations of a subject system. The procedure of detecting bug-fix commits was discussed in Section 3.2.3.

Step 2: Considering each of the bug-fix commits detected in **Step 1** we determine whether clone fragments of any clone-type experienced an SPCO in this commit. Let us consider a bug-fix commit BFC which was applied on revision R of a subject system. If we see that two or more clone fragments from a clone class in revision R experienced an SPCO in the commit BFC, then this is a case of fixing a replicated bug (i.e., fixing the same bug in each of the clone fragments that experienced an SPCO), because in an SPCO the participating clone fragments are likely to be changed together in the same way (i.e., consistently). We detect SPCOs of clone fragments of different clone-types using our tool SPCP-Miner [137].

We identify all the cases of fixing replicated bugs in each of the three types (i.e., Type 1, Type 2, and Type 3) of code clones by analyzing the clone evolution history of each of our candidate systems listed in Table 5.2.

5.4.3 Manual Analysis of the SPCOs in the Bug-fix Commits

After detecting the SPCOs in the bug-fix commits, we manually investigate the changes that occurred to the participating clone fragments in each SPCO to check whether the clone fragments were actually changed in the same way (i.e., whether the clone fragments were changed consistently). Table 5.4 shows the number of SPCOs that occurred in the bug-fix commits considering each clone-type of each of the subject systems. From our manual investigation on each of these 186 SPCOs (in total) we can state that *in each SPCO, two or more clone fragments from a particular clone class were changed together consistently (i.e., in the same way).* Fig. 5.1 shows an SPCO of two Type 3 clone fragments in the bug-fix commit operation applied on revision 1080 of our subject system Freecol. In RQ 4 (i.e., the fourth research question) we investigate the change-types experienced by the clone fragments in the SPCOs that occurred in the bug-fix commits.

5.5 Experimental Results and Analysis

In this section we answer the research questions listed in Table 5.1 by presenting and analyzing our experimental results.

Table 5.4: Number of SPCOs experienced by different types of code clones during bug-fix commit operations

Clone Types	Ctags	Camellia	jEdit	Freecol	Carol	Jabref
Type 1	1	1	5	6	10	1
Type 2	0	0	5	5	14	5
Type 3	0	4	35	37	40	17

5.5.1 *RQ 1: What percentage of the clone fragments in different clone-types takes part in bug-replication?*

Motivation. Answering this question is important since a clone-type with a higher tendency of replicating bugs should be given a higher priority compared to the other clone-types when making clone management decisions. We conduct our study considering the major clone-types: Type 1, Type 2, and Type 3. We should note that any code fragment in the code-base can contain a bug. However, cloning of this fragment is responsible for replicating/propagating that bug in several other places. Without studying bug-replication in code clones we cannot understand the real impact of cloning in propagating bugs.

Methodology. For answering RQ 1 we identify bug-replication in three types (Type 1, Type 2, Type 3) of code clones by mining the clone evolution history of each of our subject systems. Section 5.4 elaborates on the procedure we follow for detecting replicated bugs in code clones.

We automatically detect the bug-fix commits and then identify the similarity preserving co-changes (SPCOs) of clones in these commits. Table III shows the number of SPCOs considering each clone-type of each of our subject systems. If an SPCO of two or more clone fragments occurs in a bug-fix commit, then it is an indication of fixing of a replicated bug. Table 5.5 shows the following four measures concerning bug-replication in each clone-type.

Measure CG: This is the total number of clone genealogies created during the whole period of evolution (the columns in Table 5.5 with the heading **CG**).

Measure CGBF: This is the total number of clone genealogies related to bug-fix (the columns with the heading **CGBF** in Table 5.5).

Measure CGBR: This is the total number of clone genealogies related to bug-replication (the columns with the heading **CGBR** in Table 5.5).

Measure PCGBR: This is the percentage of clone genealogies related to bug-replication with respect to all clone genealogies (The columns with the heading **PCGBR** in Table 5.5). We calculate this percentage

Table 5.5: Number of clone genealogies related to bug-replication

	Type 1				Type 2				Type 3			
Subject Systems	**CG**	**CGBF**	**CGBR**	**PCGBR**	**CG**	**CGBF**	**CGBR**	**PCGBR**	**CG**	**CGBF**	**CGBR**	**PCGBR**
Ctags	52	4	2	3.85%	88	4	0	0.00%	155	14	0	0.00%
Camellia	300	2	2	0.67%	48	0	0	0.00%	177	11	5	2.82%
jEdit	7398	73	6	0.08%	399	20	6	1.50%	2688	184	45	1.67%
Freecol	239	14	8	3.35%	162	12	6	3.70%	752	107	55	7.31%
Carol	415	31	12	2.89%	211	32	16	7.58%	682	134	68	9.97%
Jabref	483	8	2	0.41%	228	14	4	1.75%	1363	31	21	1.54%

CG = Number of Clone Genealogies

CGBF = Number of Clone Genealogies Related to Bug-Fix

CGBR = Number of Clone Genealogies Related to Bug-Replication

PCGBR = Percentage of Clone Genealogies Related to Bug-Replication with respect to all clone genealogies

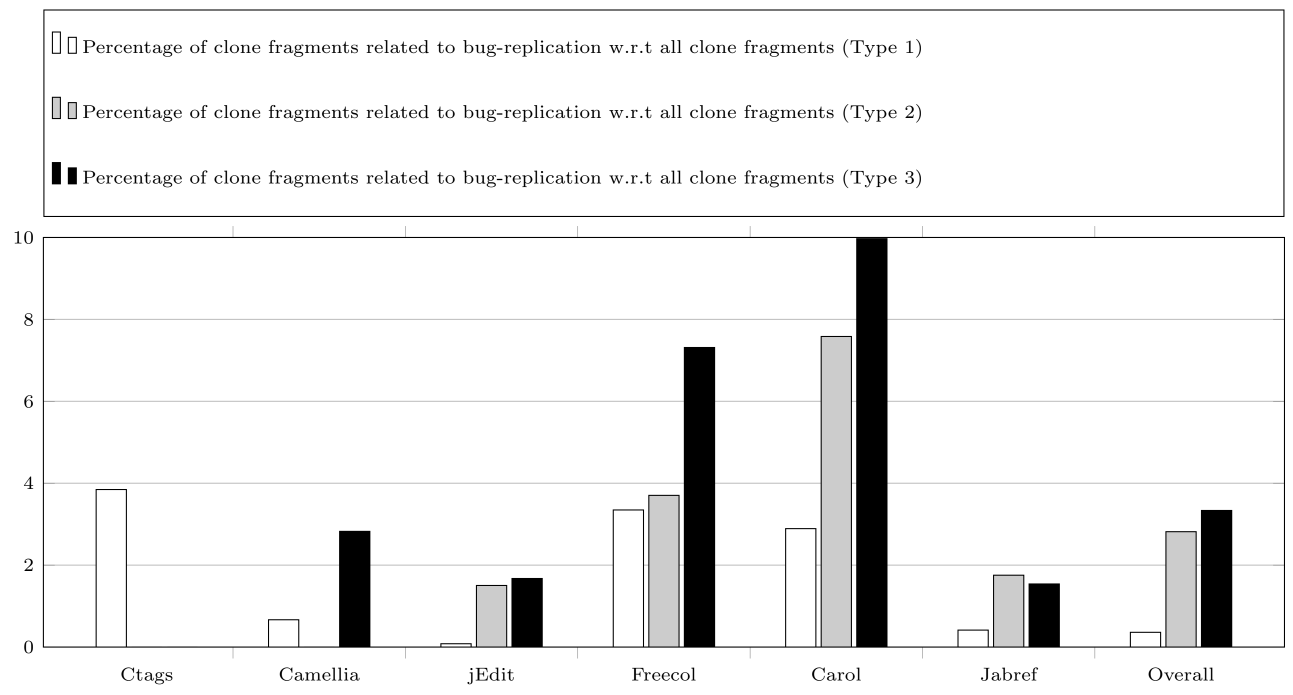


Figure 5.3: Percentage of clone fragments related to bug-replication with respect to all clone fragments.

for a particular clone-type of a particular subject system using the following equation.

$$PCGBR = \frac{100 \times CGBR}{CG} \tag{5.1}$$

We also calculate the overall percentage (i.e., considering all subject systems) of clone fragments related to bug-replication considering each clone-type. This overall percentage was calculated using the following equation.

$$OPCGBR_{T_i} = \frac{100 \times \sum_{all\ systems} CGBR_{T_i}}{\sum_{all\ systems} CG_{T_i}} \tag{5.2}$$

Here, $OPCGBR_{T_i}$ is the overall percentage of clone fragments related to bug-replication with respect to all clone fragments of clone-type T_i (i = 1, 2, or 3).

Fig. 5.3 shows the percentage **PCGBR** for each clone-type of each subject system. It also shows the overall percentage **OPCGBR** for each clone-type. Looking at the percentages (PCGBRs) we understand that although the percentage of clone fragments containing replicated bugs is low for some subject systems (such as: jEdit, Jabref), this percentage can sometimes be considerable (for example, the percentages regarding our subject systems: Freecol, and Carol). We also see that for most of the subject systems (i.e., for four systems out of six) the percentage regarding Type 3 case is the highest. From Table 5.5 we see that for the Type 2 and Type 3 cases of Ctags and Type 2 case of Camellia we did not get any clone fragments related to bug-replication.

We also wanted to investigate what percentage of code clones that are related to bugs contain replicated bugs. We calculate this percentage using the following equation.

$$PCGBRBF = \frac{100 \times CGBR}{CGBF} \tag{5.3}$$

Here, **PCGBRBF** is the percentage of clone fragments containing replicated bugs with respect to all buggy clone fragments of a particular clone-type of a particular subject system. We also calculate the overall value (i.e., considering all system) of this percentage for each clone-type using an equation which is similar to Eq. 5.2. We show these percentages (**PCGBRBF**, and the overall one) in Fig. 5.4.

From Fig. 5.4 we see that for three clone-types: Type 1, Type 2, and Type 3 overall respectively 24%, 39%, and 40% of the buggy clone fragments contain replicated bugs. From this graph we again see that Type 3 clones have the highest possibility of containing replicated bugs. The percentage regarding Type 2 clones is also very near to Type 3 clones.

Answer to RQ 1. For most of our subject systems a small percentage of all clone fragments can contain replicated bugs. However, this percentage can sometimes be considerable. According to our subject systems, the percentages for the three clone-types, Types 1, 2 and 3 can be up to 3.85%, 7.58%, and 9.97% respectively. We also see that around 24%, 39%, and 40% of the buggy clone fragments in Type 1, Type 2, and Type 3 case can contain replicated bugs. According to our observation, the percentages of code clones containing replicated bugs in Type 2 and Type 3 case are higher than in Type 1 case.

From our investigation and analysis in RQ 1 we realize that bugs in code fragments can often get replicated through cloning process. These replicated bugs can cause higher instability as well as increased maintenance efforts and costs for the software systems. Also, as Type 2 and Type 3 clones have higher possibilities of containing replicated bugs compared to Type 1 clones, we should possibly consider refactoring or tracking of Type 2 and Type 3 clones with high priorities.

5.5.2 *RQ 2: What is the extent of bug-replication in the buggy clone classes of different types of clones?*

Motivation. From our answer to the previous research question (RQ 1) we can understand what percentage of clone fragments from which clone-type contain replicated bugs. However, we still do not know the extent to which the clone fragments in a clone class of a particular clone-type can contain replicated bugs. Intuitively, a clone-type with a higher extent of bug-replication should be considered more harmful compared to the other clone-types. We answer RQ 2 in the following way.

Methodology. We first detect the bug-fix commits following the procedure described in Section 3.2.3. Considering each of these commits we determine whether a clone class was affected in that commit (i.e.,

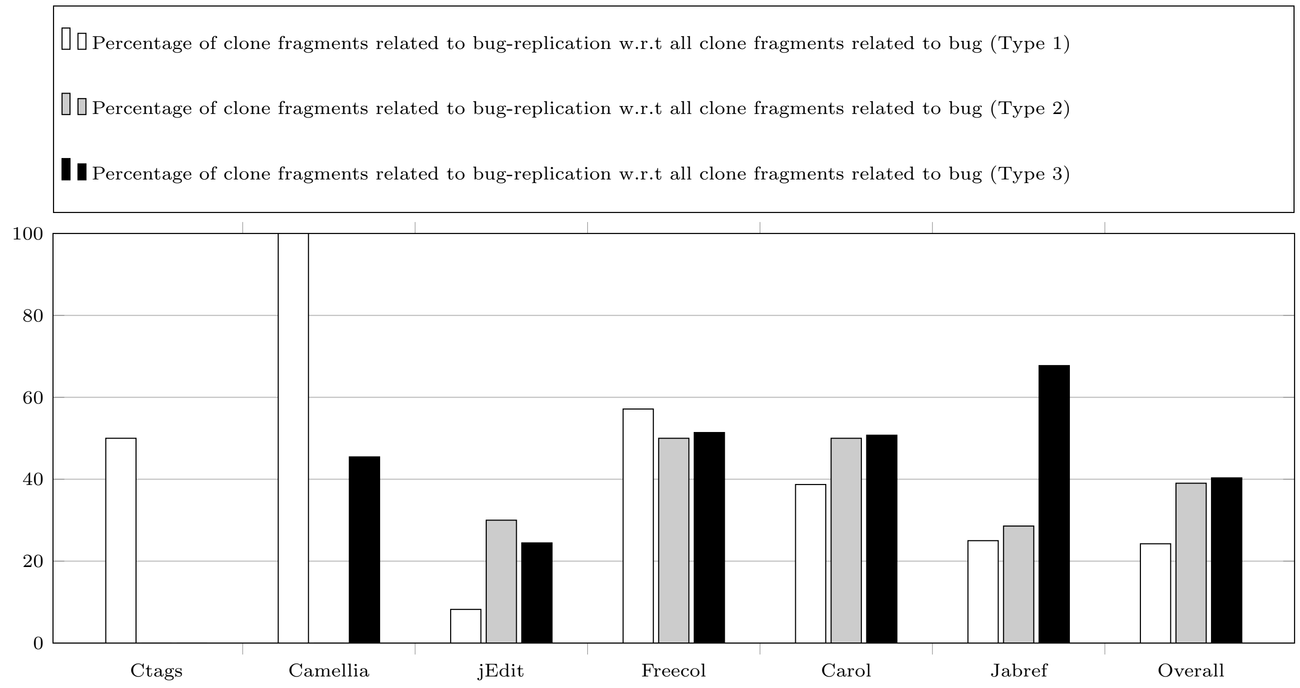


Figure 5.4: Percentage of clone fragments related to bug-replication with respect to all clone fragments related to bug-fix.

whether one or more clone fragments from the clone class were changed in that commit for fixing a bug). Let us consider a bug-fix commit BFC which was applied on a particular revision R. We determine all the clone classes in this revision, and then identify which of these clone classes were affected by the commit BFC. Let us consider a clone class CC which was affected by BFC. We determine whether two or more clone fragments from CC experienced a similarity preserving co-change (SPCO) in BFC. If this is true, then we select this clone class for our investigation in RQ 2. We call such a clone class an *Eligible Clone Class* (ECC). According to our previous discussions we can easily understand that an ECC contains a replicated bug. An affected clone class which is not ECC did not experience a similarity preserving co-change, and thus, this class does not contain replicated bugs.

Considering each of the eligible clone classes (ECCs) we determine the following two measures:

CF: The total number of clone fragments in the eligible clone class (i.e., in the ECC).

CFRB: The number of clone fragments that experienced similarity preserving co-change (SPCO). In other words, this is the number of clone fragments that contained replicated bugs.

From the above two measures of a particular ECC (eligible clone class) we can determine the extent of bug-replication in that ECC in the following way.

$$EBR = \frac{100 \times CFRB}{CF} \tag{5.4}$$

Here, EBR is the extent of bug-replication for a particular ECC. Considering all the ECCs of a particular clone-type from all the bug-fix commits of a particular subject system we determine the above two measures, and then calculate the overall extent of bug-replication using the following equation.

$$OEBR = \frac{100 \times \sum_{all\ ECCs} CFBR}{\sum_{all\ ECCs} CF} \tag{5.5}$$

Here, OEBR is the overall extent of bug-replication in the code clones of a particular clone-type of a subject system. We show the measure OEBR for the three clone-types of each of our subject systems in Fig. 5.5. The figure shows that the extent of bug-replication in the buggy clone classes of Type 1 case is 100% for five out of six subject systems (except jEdit). Also, for three subject systems (jEdit, Freecol, and Jabref) the extent of bug-replication in the buggy clone classes of Type 2 case is 100%. Bug-replication extent is also very high in the Type 3 buggy clone classes of three subject systems: Freecol, Carol, and Jabref. We also show the overall scenario (i.e., considering all subject systems) of the extent of bug-replication in the buggy clone classes of three clone-types. We see that while the buggy clone classes of Type 2 case show the highest extent of bug replication, Type 3 case shows the lowest extent. However, according to our candidate systems, the extent of bug replication in the buggy clone classes is very high in case of each of the clone-types.

Answer to RQ 2. The extent of bug replication in the buggy clone classes of each of Type 1 and Type 2 cases is higher than the extent in Type 3 case. However, the overall extent of bug-replication in each of these three clone-types is very high (i.e., more than 70%). From such a finding we believe that there should be automatic support for finding clone classes with replicated bugs. When refactoring clone classes of any clone-type we should primarily focus on the clone classes that contain replicated bugs.

Our answer to RQ 2 implies that in case of each clone-type, most of the clone fragments in a buggy clone class contain the same bug. For many cases the extent of replication is 100%. This is expected. If a particular code fragment contains a bug, then all other copies of that fragment will also contain the same bug. However, in case of each of the subject systems we find that there are some buggy clone classes where the extension of bug-replication is not 100%. Only some of the clone fragments in such a class (i.e., with partial replication of bug) were consistently modified for fixing bug leaving the other clone fragments in the class as they are. We were interested to investigate why some of the clone fragments in a partially buggy clone class were not considered buggy. We manually investigated such clone classes from each of the subject systems and had the following findings.

For most of the cases, one or more clone fragments were created by copy/pasting an original code fragment making a clone class consisting of the original one as well as the newly created ones. However, some of the newly created clone fragments were not properly changed to meet the contextual requirements. Eventually, a bug-fix commit only modified those clone fragments (i.e., the clone fragments that were not previously modified properly) leaving the other clone fragments in the class as they are. As an example of such a case we can mention the commit operation applied on revision 318 of our subject system Jabref. The commit message as was indicated by the developer says, “Fixed cut/copy/paste focus bug”. Through further investigation

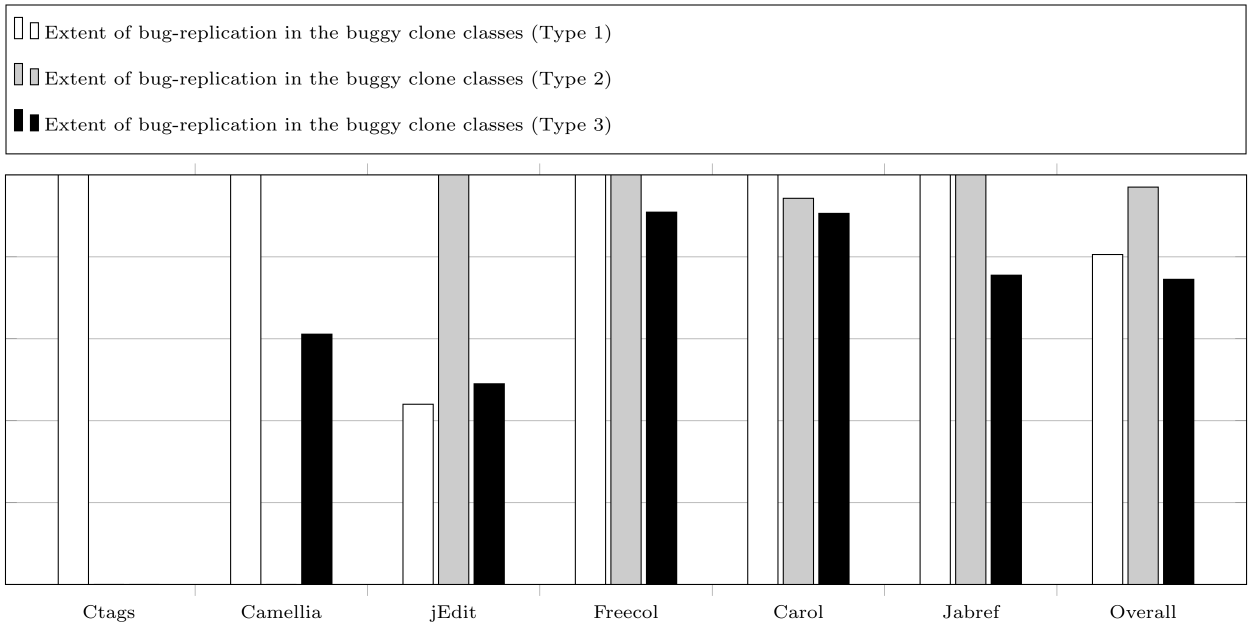


Figure 5.5: Extent of bug-replication in the buggy clone classes.

we found that a clone class in revision 318 contained two clone fragments. The commit operation on this revision changed one of these two fragments by commenting an *if-statement* in that fragment. We infer that the *if-statement* that was commented was not appropriate for the context where the copied fragment was pasted.

We finally state that although some of the eligible clone classes (ECCs) do not exhibit a complete replication (i.e., a bug-replication extent of 100%) of the bug, the overall extent of bug-replication is very high for each clone-type. Thus, bug-replication tendencies of code clones should be given importance when taking clone management decisions.

5.5.3 *RQ 3: What percentage of the bugs that were experienced by the code clones of different clone-types are replicated bugs?*

Motivation. Even after answering the previous two research questions it is still unknown whether most of the bugs experienced by the code clones are replicated bugs or not. If most of the bugs in code clones are replicated bugs then we understand that bug-replication is a common phenomenon during cloning, and in that case the programmers should be suggested to be aware that the code fragments they are going to copy are bug-free. We answer RQ 3 by investigating three clone-types of each of the subject systems.

Methodology. We detect the bug-fix commits of a candidate subject system. We sequentially examine each of these commits and update the following two counters considering each clone-type.

NBC (The number of bugs experienced by the code clones): Let us consider a bug-fix commit BFC which was applied on the revision R of a subject system. If one or more of the clone classes residing in this revision were affected by the commit BFC, then we increase the counter NBC by one.

NBRC (The number of bugs that were replicated in the code clones): Let us consider that a bug-fix commit was applied on revision R of a candidate system and one or more clone classes in this revision were affected by the commit. If any of these affected clone classes experienced a similarity preserving co-change in this commit, then we increase the counter NBRC by one. We should again note that according our former discussions, experiencing a similarity preserving co-change (SPCO) in a bug-fix commit operation is an indication of the existence of replicated bugs in the code clones.

The percentage of replicated bugs with respect to all bugs experienced by the code clones of a particular clone-type of a particular subject system is determined using Eq. 5.6.

$$\%\ of\ replicated\ bugs = \frac{100 \times NBRC}{NBC} \tag{5.6}$$

In Table 5.6 we show the two measures: NBC and NBRC for each clone-type of each subject system. We also show the percentage of replicated bugs for three clone-types of each system in Fig. 5.6. We see that this percentage is considerable for most of the cases. From Table 5.6 and Fig. 5.6 we realize that all the eight bugs experienced by the Type 2 clones of Carol were replicated bugs. Fig. 5.6 also shows the overall percentage of replicated bugs in the three clone-types considering all subject systems. This overall percentage was calculated using an equation similar to Eq. 5.2. For Type 1, Type 2, and Type 3 clones the overall percentages are respectively 30.65%, 55.26%, and 46.3%. While the overall percentage of replicated bugs is the highest in Type 2 clones, this percentage is the lowest in Type 1 clones.

Answering RQ 3. A considerable percentage of the bugs experienced by the code clones of each clone-type can be replicated bugs. This percentage can sometimes be very high (i.e., up to 100%). The overall percentage of replicated bugs in both Type 2 and Type 3 clones are higher compared to Type 1 clones.

Replication of bugs can cause increased maintenance effort and cost for software systems. Thus, it is important to investigate which code statements in the clone fragments primarily contain replicated bugs. We perform such an investigation in RQ 4.

5.5.4 *RQ 4: Which code statements are highly related to bug-replication?*

Motivation. In RQ 4 we investigate which code statements in the clone fragments have high possibilities of containing replicated bugs. Programmers might need to be careful while copying code fragments containing those statements. The clone classes containing such statements (i.e., that contain replicated bugs) could be given higher priorities for refactoring and tracking. The existing studies did not investigate the bug-replication possibilities of clone fragments. Such an investigation can provide us helpful insights regarding minimizing bug-replication. We perform our investigation for answering RQ 4 in the following way.

Table 5.6: Number of Bugs that were Replicated in Clones

Subject	Type 1		Type 2		Type 3	
Systems	**NBC**	**NBRC**	**NBC**	**NBRC**	**NBC**	**NBRC**
Ctags	3	1	4	0	11	0
Camellia	1	1	0	0	5	3
jEdit	37	5	10	5	42	15
Freecol	7	6	10	4	46	23
Carol	8	5	8	8	22	14
Jabref	6	1	6	4	23	14

NBC = Number of Bugs experienced by Clones

= The number of bug-fix commits experienced by the code clones of a particular clone type

NBRC = Number of Bugs that were Replicated in Clones

= The number of bug-fix commits having SPCOs experienced by the code clones of a particular clone-type

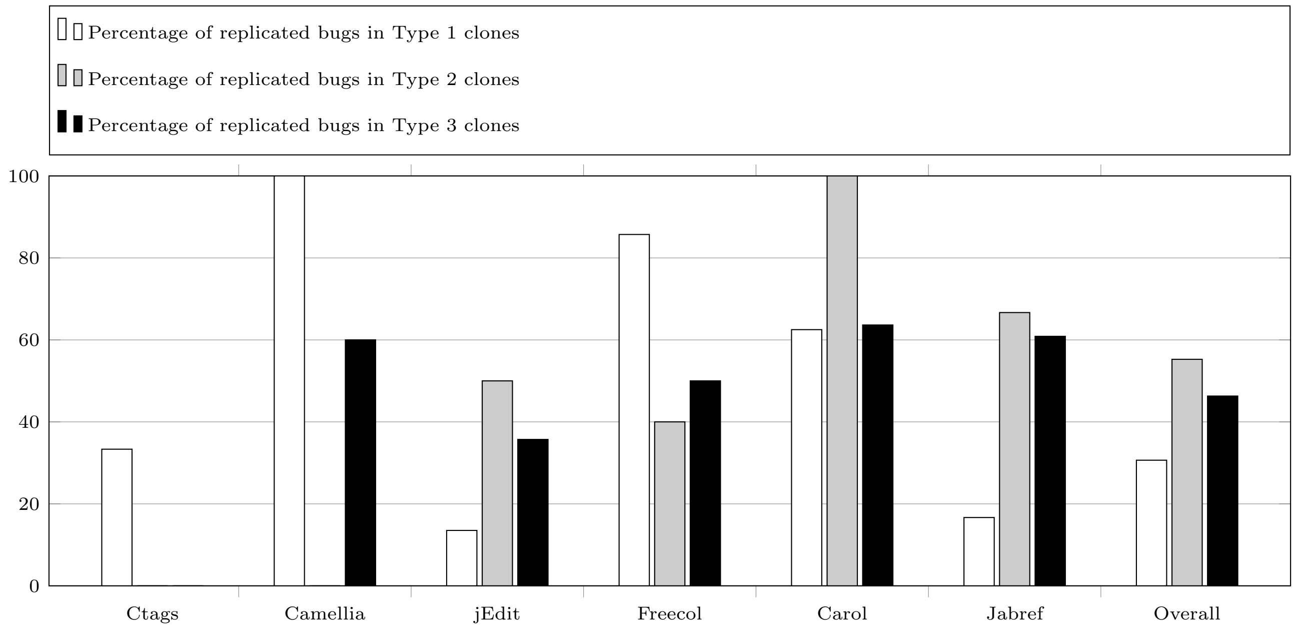


Figure 5.6: Percentage of replicated bugs in different types of code clones.

Table 5.7: Most frequent change-types during fixing replicated bugs in code clones

SL	Change Types	Ctags	Camellia	jEdit	Freecol	Carol	Jabref	Total
1	Addition of if-else blocks	4	2	17	12	17	7	59
2	Modification of if Condition			21	14	9	2	46
3	Deletion of if-else blocks			9	5	4	1	19
4	Modification of Parameters in the Called Method	2	2	23	8	26	7	68
5	Addition of Method Call			17	9	7	10	43
6	Replacement of Old Method Call by New Method Call			12	7	14		33
7	Deletion of Method Call			5	2	1	2	10

Methodology. As we did before we detect the bug-fix commits and then identify cases where the bugs replicated in the clone fragments were fixed. Fixing of replicated bugs were identified by detecting the similarity preserving co-changes (SPCOs) of clone fragments in the bug-fix commits. We should again note that if two or more clone fragments from a particular clone class undergo an SPCO in a particular commit operation, then it is very much likely that those clone fragments were changed consistently (i.e., each of the clone fragments was changed in the same way) in that commit.

We manually check each of the bug-fix commits where the clone fragments experienced SPCOs. We identify the changes in the SPCOs that occurred in a bug-fix commit, and then categorize these changes. In Table 5.6 we have already reported the number of bug-fix commits having SPCOs for each clone-type of each subject system. The column named 'NBRC'in this table reports this number. Table 5.4 shows the number of SPCO cases for each clone-type of a subject system. There can be more than one SPCO in a bug-fix commit. In other words, more than one clone classes can experience similarity preserving co-change in a bug-fix commit. The frequent change categories that we observed during our manual analysis of the SPCO cases are shown in Table 5.7. A particular clone fragment can experience changes of more than one of these categories during fixing replicated bugs.

From Table 5.7 we realize that the most frequent change experienced by the clone fragments during fixing replicated bugs is 'Modification of Parameters in the Called Method'. We suggest programmers to be more careful when copying code fragments containing *method-calls*. Before copying the programmers should ensure that the code fragment she is going to copy is bug-free. We should also prioritize refactoring of clone fragments that contain *method-calls*. Intuitively, a change in the name or in the parameters of a particular method will affect all the code fragments that call that particular method. Refactoring of clone fragments containing the same method calls can considerably minimize future software maintenance effort and cost.

We suggest that clone fragments containing *if-conditions* should also be given a high priority for refactoring, because we observe a high frequency of the occurrence of modifying *if-conditions*.

The other frequent change-types according to Table 5.7 are: addition of if-else blocks, addition of method calls, and replacement of an old method call by a new method call. We suggest that clone fragments with the possibilities of experiencing such types of changes should also be considered for management. If such clone fragments are not suitable for refactoring, then they should be tracked. We also observed a number of infrequent change-types during our investigation. These change-types include: replacement of C preprocessor by method call, addition, deletion or modification of loops, and modification of try-catch blocks.

Answer to RQ 4. According to our manual investigation it seems that clone fragments having *method calls* and *if-conditions* have a high possibility of containing replicated bugs. We suggest that such clone fragments should be considered for maintenance (i.e., refactoring or tracking) with high priorities.

We should note that as Type 1 clones are exactly the same code fragments, refactoring is possibly the best option for maintaining such clones. Tracking is the most suitable maintenance technique for the other two types (Type 2, and Type 3) of code clones.

5.6 Related Work

In a previous study Mondal et al. [131] investigated and compared the bug-proneness of three types of code clones. However, that study did not focus on the bug-replication tendencies of code clones. In Section 3.5, we see that a number of studies have been conducted on the bug-proneness of code clones. However, none of these studies focus on the bug replication tendencies of different clone-types. We believe that without studying bug-replication in code clones we cannot realize their actual impact on software maintenance and evolution. Focusing on this we perform an in-depth investigation on bug-replication in code clones in this research. Our experimental results are promising and provide useful implications for better maintenance of software systems through minimizing bug-replications.

5.7 Limitations

We used the NiCad clone detector [49] for detecting clones. While all clone detections tools suffer from the *confounding configuration choice problem* [197] and might give different results for different settings of the tools, the setting that we used for NiCad for this experiment are considered standard [153] and with these settings NiCad can detect clones with high precision and recall [156, 155, 175]. Thus, we believe that our findings on the bug-proneness of code clones are of significant importance.

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique followed by Barbour et al.[36]. Such a technique proposed by Mocus and Votta [123] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al.[36] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

In our experiment we did not study enough subject systems to be able to generalize our findings regarding the comparative bug-proneness of different types of clones. However, our candidate systems were of diverse variety in terms of application domains, sizes and revisions. Thus, we believe that our findings are important from the perspectives of clone management and can help us in better ranking of code clones for refactoring and tracking.

5.8 Summary

In this chapter, we present our empirical study on bug-replication in the major three types of code clones: Type 1, Type 2, and Type 3. Although a number of existing studies have investigated bug-proneness of code clones, none of these studies focus on analyzing the bug-replication tendencies of different clone-types. Without studying bug-replication tendencies of code clones we cannot understand the real impact of cloning on software maintenance and evolution. Focusing on this we conduct an in-depth empirical study on the three types of code clones residing in thousands of revisions of six diverse subject systems to investigate whether and to what extent bug-replication occurs in different clone-types.

According to our investigation, bug replication through code cloning is a common phenomenon during software maintenance and evolution. Around 55% of the bugs occurred in code clones can be replicated bugs. Also, up to 10% of the code clones can contain replicated bugs. Tendencies of bug-replication is higher in Type 2 and Type 3 clones than in Type 1 clones. It implies that code clones of Type 2 and Type 3 should be considered for management (such as refactoring or tracking) with high priorities. From our manual analysis we observe that *method-calls* and *if-conditions* residing in the clone fragments exhibit high tendencies of being related to bug-replication. From this we suggest that clone fragments containing *method-calls* and/or *if-conditions* should be prioritized for refactoring and tracking to minimize bug-replication. We finally believe that the outcomes of our bug-replication study are important for better management of code clones as well as for better maintenance of software systems. In future we would like to investigate ranking of code clones for management considering their bug-replication tendencies.

6 Bug Replication in Micro Clones

From our previous study in Chapter 5, we have found that bug-replication through code cloning is a common phenomenon during software evolution. Also, from the results of our second study in Chapter 4, we found that micro-clones are more bug-prone than regular code clones. Thus, we were interested to know that how replicated bugs behave in micro-clones and then compare the result with regular code clones. In this chapter, we discover bug-replication in both regular and micro-clones and compare the results.

Clone fragments with a minimum size of 5 LOC were usually considered in previous studies. In recent studies, clone fragments which are less than 5 LOC are referred as micro-clones. It has been established by the literature that code clones are closely related with software bugs as well as bug replication. None of the previous studies have been conducted on bug-replication of micro-clones. In this paper we investigate and compare bug-replication in between regular and micro-clones. For the purpose of our investigation, we analyze the evolutionary history of our subject systems and identify occurrences of similarity preserving co-changes (SPCOs) in both regular and micro-clones where they experienced bug-fixes. From our experiment on thousands of revisions of six diverse subject systems written in three different programming languages, C, C# and Java we find that the percentage of clone fragments that take part in bug-replication is often higher in micro-clones than in regular code clones. The percentage of bugs that get replicated in micro-clones is almost the same as the percentage in regular clones. Finally, both regular and micro-clones have similar tendencies of replicating severe bugs according to our experiment. Thus, micro-clones in a code-base should not be ignored. We should rather consider these equally important as of the regular clones when making clone management decisions.

The rest of the chapter is organized as follows. Section 6.1 introduces this chapter, Section 6.2 discusses the experimental steps, Section 6.3 answers our research questions by presenting and analyzing the experimental results. Section 6.4 discusses the related work and compares our study with the existing ones, Section 6.5 discusses possible threats to validity, and Section 6.6 concludes the chapter and discusses possible future work.

6.1 Introduction

Recurrent activities of copy-pasting code fragments is very common in everyday life of software development cycle. The act of copying a piece of code and then pasting it without any modification (exactly similar) or with modifications (nearly similar) is known as code cloning [157, 151]. A group of similar code fragments constructs a clone class. Code clones are mainly created because of the frequent copy/paste activities of programmers during software development and maintenance. Beside copy/paste activities there can be various other reasons behind creating clone code [152]. Whatever may be the reasons behind cloning, code clones are significantly important from the perspectives of software maintenance and evolution [157].

Recently, Islam et al. [73] performed a comparative study on the bug-proneness between regular and micro-clones. They found that micro-clones are more bug-prone than regular clones. Moreover, the amount of severe bugs in micro-clones is comparatively higher than in regular clones. However, they did not investigate bug replication of micro-clones. This motivates us to investigate the bug replication in both micro and regular code clones. In order to explore the effects of bug replication in micro-clones and regular code clones we perform a comparative study. To the best of our knowledge, our research is the first comparative study on bug replication in between micro-clones and regular code clones.

We consider thousands of revisions of six diverse open source software systems written in three languages, Java, C# and C. For detecting code clones from each of the revisions of a subject system we use the NiCad clone detector [49]. We analyze the evolution history of both micro and regular code clones, and investigate whether they contain replicated bugs and to what extent.

In our experiment, the minimum and maximum size of a micro-clone fragment can be 1 LOC and 4 LOC respectively. We consider those micro-clones that are not parts of regular clones. We detect regular code clones of at least 5 LOC because Wang et al. [197] reported this size as the best minimum threshold for detecting regular clones.

To investigate bug replication in regular and micro-clones, we observe bug-fix commits reported by the developers from thousands of commits in our candidate open source projects. We observe the intensity of bug-replication in both micro-clones and regular code clones and answer four important research questions listed in Table 6.1. The major findings from our research are as follows.

- The percentage of clone fragments that experienced bug-replication in micro-clones is often higher than that of regular code clones.
- The percentage of bugs that get replicated in micro-clones is almost the same as of the percentage of bugs that get replicated in regular code clones.
- The replicated bugs in both regular and micro-clones have similar tendencies of being severe.

From these findings we see that micro-clones are equally important as regular code clones. Thus, we should also consider micro-clones when making clone management decisions.

6.2 Experimental Steps

We conduct our research on six subject systems (two C, one C# and three Java systems). We consider these six subject systems since these systems have variations in application domains, sizes, and revisions. These

Table 6.1: Research Questions

SL	Research Question
RQ 1	What percentage of the clone fragments in Regular and Micro-clones takes part in bug-replication?
RQ 2	What is the extent of bug-replication in the buggy clone classes of Regular code clones and Micro-clones?
RQ 3	What percentage of bugs experienced by regular and micro-clones are replicated bugs?
RQ 4	Are the replicated bugs in micro-clones more likely to be severe than the replicated bugs in regular clones?

Table 6.2: Subject Systems

Subject Systems	Language	Domains	LLR	Revisions
Ctags	C	Code Def. Generator	33,270	774
Brlcad	C	Solid Modeling CAD	39,309	735
MonoOSC	C#	Formats and Protocols	18,991	355
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Management	45,515	1545
LLR = LOC in the Last Revision				

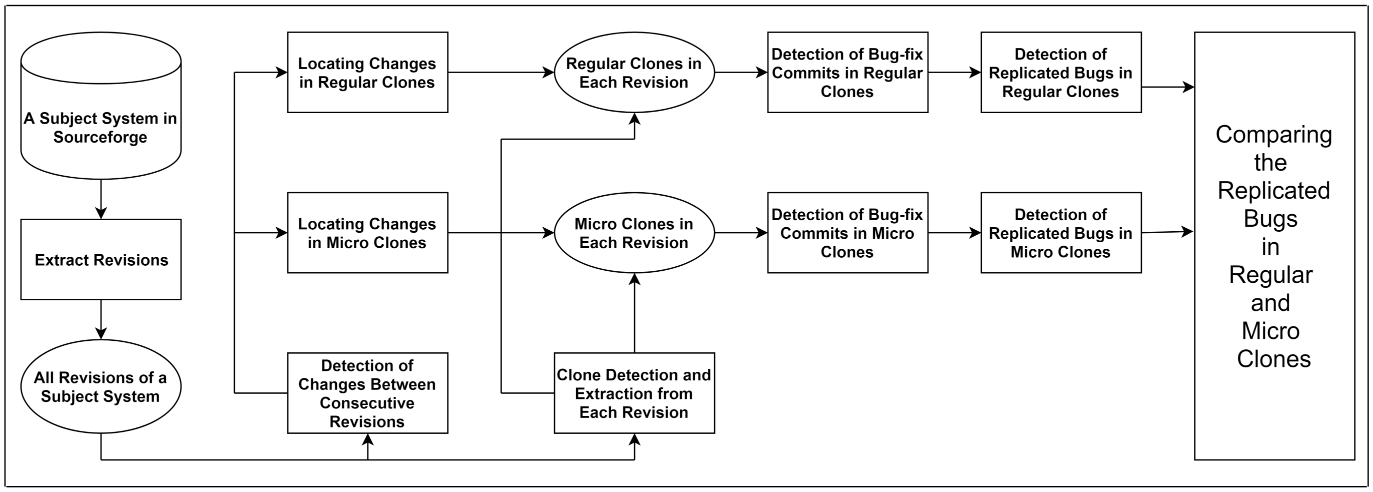

A Subject System in Sourceforge
Extract Revisions
All Revisions of a Subject System
Locating Changes in Regular Clones
Locating Changes in Micro Clones
Detection of Changes Between Consecutive Revisions
Regular Clones in Each Revision
Micro Clones in Each Revision
Clone Detection and Extraction from Each Revision
Detection of Bug-fix Commits in Regular Clones
Detection of Bug-fix Commits in Micro Clones
Detection of Replicated Bugs in Regular Clones
Detection of Replicated Bugs in Micro Clones
Comparing the Replicated Bugs in Regular and Micro Clones

Figure 6.1: The execution flow diagram of the experimental steps in detecting replicated bug-fix commits in Regular and Micro clones. The rectangles demonstrate the steps.

Table 6.3: NiCad Settings for Detecting Regular and Micro-clones

Clones	**Clone Granularity**	**Minimum Line**	**Maximum Line**	**Identifier Renaming**	**Dissimilarity Threshold**
Regular Clones	Block	5	500	Blind Rename	30%
Micro Clones	Block	1	4	Blind Rename	30%

subject systems are listed in Table 6.2 which were downloaded from the SourceForge online SVN repository [16]. In this table, the total number of revisions of each subject system is given along with the lines of code (LOC) in the last revision. Figure 6.1 shows the simple flow diagram of our work procedure for this study.

We perform the following steps for detecting fixed bugs: **(1)** Extraction of all revisions (as stated in Table 6.2) of each of the subject systems from the online SVN repository; **(2)** Detection and extraction of code clones from each revision by applying NiCad [49] clone detector; **(3)** Detection of changes between every two consecutive revisions using diff; **(4)** Locating these changes to the already detected clones of the corresponding revisions; and **(5)** Detection of bug-fix commit operations. For completing the first four steps we use the tool SPCP-Miner [137]. We perform steps 1 to 5 for both regular and micro-clones. We will describe the detection of bug-fix commits later in this section. In Section 6.3 we will describe how we detect bug-fix changes in regular code clones and micro-clones.

We use NiCad [49] for detecting clones since it can detect code clones with high precision and recall [156, 155]. Using NiCad, we detect block clones (both exact and near-miss) of at least 5 lines with 30% dissimilarity threshold and blind renaming of identifiers. For micro-clones using NiCad we detect block clones of a minimum size of 1 LOC and maximum size of 4 LOC with 30% dissimilarity threshold and blind renaming of identifiers as was detected by Mondal et al. [135]. NiCad settings for detecting both micro and regular code clones are shown in Table 6.3.

For different settings of a clone detector the clone detection results can be different and thus, the findings on bugs in code clones can also be different. Hence, selection of appropriate settings (i.e., detection parameters) is important. We used the mentioned settings in our research for detecting regular clones, because Svajlenko and Roy [175] show that these settings provide us with better clone detection results in terms of both precision and recall.

6.3 Experimental Results and Analysis

Our research questions are listed in Table 6.1. In this section, we analyze our experiment results for six subject systems and answer the research questions from our analysis.

6.3.1 *RQ 1: What percentage of the clone fragments in Regular and Micro-clones takes part in bug-replication?*

Motivation. Measuring the percentage of clone fragments which take part in bug replication is an important way to contrast between micro and regular code clones. If a particular category of code clones (regular clones or micro-clones) contains more replicated bugs, we should be more careful about that category while making clone management decisions. The software bug that replicates through clone code have a ripple effect in the whole system. Thus answering this research question has an important impact on clone research since code clones that gravitate to replicate bugs should be emphasized during software maintenance.

Methodology. To answer this research question we identify bug-replication in code clones by mining the clone evolution history of each of our subject systems. Detail procedure of detecting replicated bugs in regular and micro-clones is elaborated in Section 5.4. We automatically detect the bug-fix commits and then identify the similarity preserving co-changes (SPCOs) of clones in these bug-fix commits. If an SPCO of two or more clone fragments occurs in a bug-fix commit, then it is an indication of fixing of a replicated bug.

We use the following two measures to calculate the percentage of clone fragments containing replicated bugs.

CFRB: The number of clone fragments that experienced similarity preserving co-change (SPCO) i.e. clone fragments which contain replicated bugs.

BFCR: The number of distinct bug-fix commits where regular and/or micro-clones experienced similarity preserving co-changes (SPCOs).

To determine the percentage of clone fragments that takes part in bug-replication we use Equation 6.1.

$$PCFRB = \frac{100 \times CFRB}{BFCR} \tag{6.1}$$

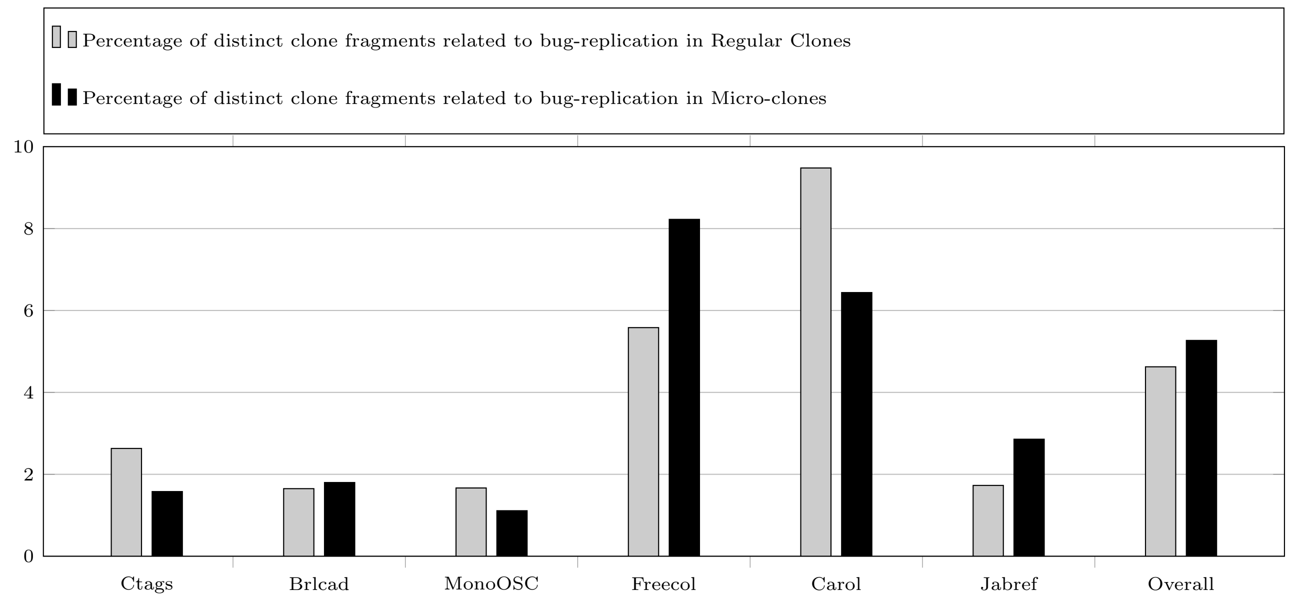


Figure 6.2: Percentage of clone fragments that related to bug-replication in regular and micro-clones.

Here, PCFRB is the percentage of clone fragments containing replicated bugs per revision. We compute the overall percentage of the clone fragments that experienced bug-replication for both regular and micro-clones using the following equation.

$$OPCFRB = \frac{100 \times \sum_{all\ systems} CFRB}{\sum_{all\ systems} BFCR} \tag{6.2}$$

Table 6.4 shows the number of distinct clone code fragments which took part in bug-replication in both regular and micro-clone code for six subject systems. From this table we observe that the number of distinct clone fragments is higher in micro-clones for three subject systems i.e. for Brlcad, Freecol and Jabref. For other three subject systems i.e. for Ctags, MonoOSC and Carol number of replicated clone fragments is higher in regular code clones than that of micro-clones. Figure 6.2 shows the percentage of distinct clone fragments which are related to bug-replication in regular and micro-clones. In overall, percentage of clone fragments that experienced bug-replication is slightly higher in micro-clones than the regular code clones.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 1. We perform Mann-Whitney-Wilcoxon (MWW) test [9, 10] to check the statistical significance of the result. We are interested to know that if the percentage of the clone fragments that takes part in bug-replication in micro-clones is significantly higher than in regular clones. MWW test is non-parametric and does not require normal distribution of data. The significance level of our MWW test is 5% i.e. if the p-value is less than 0.05 and the U-value is less than or equal to critical U-value then the difference will be significant. We found that the p-value is 0.93624 which is much greater than the significance level 0.05. Also, the U-value is 17 and the critical value of U at $p<0.05$ is 5. So the U-value is greater than the critical U-value. This MWW test result prevails that the difference between the two percentages of clone fragments containing replicated bugs in regular and micro-clones is insignificant.

Table 6.4: Number of Clone Fragments that Experienced Bug-replication in Regular and Micro-Clones

Subject Systems	**Regular Clones (CFRB)**	**Micro Clones (CFRB)**	**Revisions (BFCR)**
Ctags	50	30	19
Brlcad	33	36	20
MonoOSC	15	10	9
Freecol	547	806	98
Carol	455	309	48
Jabref	121	200	70

Answer to RQ 1. The percentage of the clone fragments in micro-clones that takes part in bug-replication (5.27%) is slightly higher than the percentage of the clone fragments in regular clones that takes part in bug-replication (4.63%).

From our above experimental results we can state that the difference between two overall percentages (regular and micro-clones) is not prominent. This implies that micro-clones are as important as regular code clones. Thus, we can not disregard micro-clones during software maintenance. Developers should equally emphasize on both micro-clones and regular code clones while performing clone management.

6.3.2 *RQ 2: What is the extent of bug-replication in the buggy clone classes of Regular code clones and Micro-clones?*

Motivation. After knowing the percentage of clone fragments containing replicated bugs in both regular and micro-clones, we still need to understand the extent of bug-replication in the clone classes which contains bugs for regular and micro-clones. Instinctively, the code clone that contains higher extent of bug-replication expected to be more harmful for the system. To answer this research question we perform the following procedures.

Methodology. Following the procedure of Section 3.2.3, we first detect the bug-fix commits. For each of the bug-fix commits we detect whether a clone class was affected in that commit. This means whether one or more clone fragments from the clone class were changed for fixing a bug in that commit. Suppose a bug-fix commit was applied on the revision R of a subject system. Let us consider a clone class CC which was affected by BFC. If two or more clone fragments from CC are experienced a similarity preserving co-change

(SPCO) in BFC then we select this clone class as an *Eligible Clone Class* (ECC). In other words, an ECC contains replicated bug. We discard all the clone classes which did not experienced similarity preserving co-change (SPCO) despite of being affected by BFC.

For each eligible clone class (ECC) we calculate following two measures:

CF: The total number of clone fragments in the eligible clone class i.e. in the ECC.

CFRB: The number of clone fragments that experienced similarity preserving co-change (SPCO) i.e. clone fragments which contain replicated bugs (same as illustrated in Section 6.3.1).

From these two measures of an eligible clone class (ECC), we calculate the extent of bug-replication in that ECC using the following equation.

$$EBR = \frac{100 \times CFRB}{CF} \tag{6.3}$$

Here, EBR stands for the extent of bug-replication for a particular ECC. We get the percentage of the extent of bug-replication (PEBR) by considering all the ECCs of regular or micro-clones from all the replicated bugs with respect to all bugs of a subject system. We use Equation 6.4 for finding the value of PEBR.

$$PEBR = \frac{100 \times \sum_{all} CFRB}{\sum_{all} CF} \tag{6.4}$$

The overall percentage of the extent of bug-replication for all subject systems is calculated using following equation.

$$OPEBR = \frac{100 \times \sum_{all\ systems} CFRB}{\sum_{all\ systems} CF} \tag{6.5}$$

In Equation 6.5, OPEBR refers to the overall percentage of the extent of bug-replication with respect to all bugs. The OPEBR is calculated for both regular and micro-clones.

Table 6.5 shows the extent of bug-replication experienced by both regular and micro-clones. Figure 6.3 depicts the extent of replicated bugs in the buggy clone classes for regular and micro-clones. From Table 6.5 and Figure 6.3 we observe that the extent of bug-replication is a bit higher in regular code clones than that of micro-clones for each of the subject systems. The highest extent (97.06%) of replicated bugs found in Brlcad for regular code clones which is nearly 100%. Second highest extent (88.24%) of the bug-replication found in MonoOSC for regular clone code. Other than these two cases, for rest of the cases, extent of bug-replication for regular clones varies from 5% to 17% and for micro-clones it varies from 0.25% to 2%.

In this research question we did not perform the MWW test since we believe that the actual number of extension of replicated bugs is more important than the percentage of extension. This is because we found that the number of clone fragments is very high in micro-clones compared with regular clones. For instance, from Table 6.5 we see that in Freecol, total number of clone fragments in eligible clone classes is 207765 in micro-clones whereas in regular clones it is only 10507 which is almost 20 times. However, the number of clone fragments experienced bug-replication is only 806 and 547 in micro clones and regular clones respectively. For

Table 6.5: Extent of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones

Subject	**Regular Clones**		**Micro Clones**	
Systems	**CFRB**	**CF**	**CFRB**	**CF**
Ctags	50	795	30	1745
Brlcad	33	34	36	6189
MonoOSC	15	17	10	517
Freecol	547	10507	806	207765
Carol	455	2784	309	29907
Jabref	121	742	200	79019
Total	1221	14879	1391	325142

the purpose of clone refactoring or clone tracking, we have to deal with the actual number of clone fragments (all the bugs need to be fixed) rather than percentages. Thus, though the extent of bug-replication in the buggy clone classes is higher in regular clones than micro-clones, we should not neglect micro-clones because of its characteristic of having higher number of clone fragments compared to regular clones.

Answer to RQ 2. The extent of bug replication in the buggy clone classes of regular code clones is higher than the extent in micro-clones. The overall extent of replicated bugs in regular clones is 8.21% and in micro-clones the overall extent of bug-replication is 0.43%.

Clone classes containing replicated bugs are considered more harmful. While refactoring clone classes of code clones, developers should focus on clone classes that have replicated bugs. In this research question, our findings show that both regular code clones and micro-clones have replicated bugs in their clone classes. Hence, both regular and micro-clones should be equally considered carefully while clone management (such as refactoring).

6.3.3 *RQ 3: What percentage of bugs experienced by regular and micro-clones are replicated bugs?*

Motivation. We are interested to identify whether most of the bugs in clone code are replicated bugs or not. Finding the answer of this research question will help us to understand that if bug-replication is a recurrent event in software development. Intuitively, if almost all bugs are found to be replicated bugs then developers should be more concerned about copy/pasting source code without containing any bugs.

Methodology. To answer this research question, we calculate the total number of bugs and also the

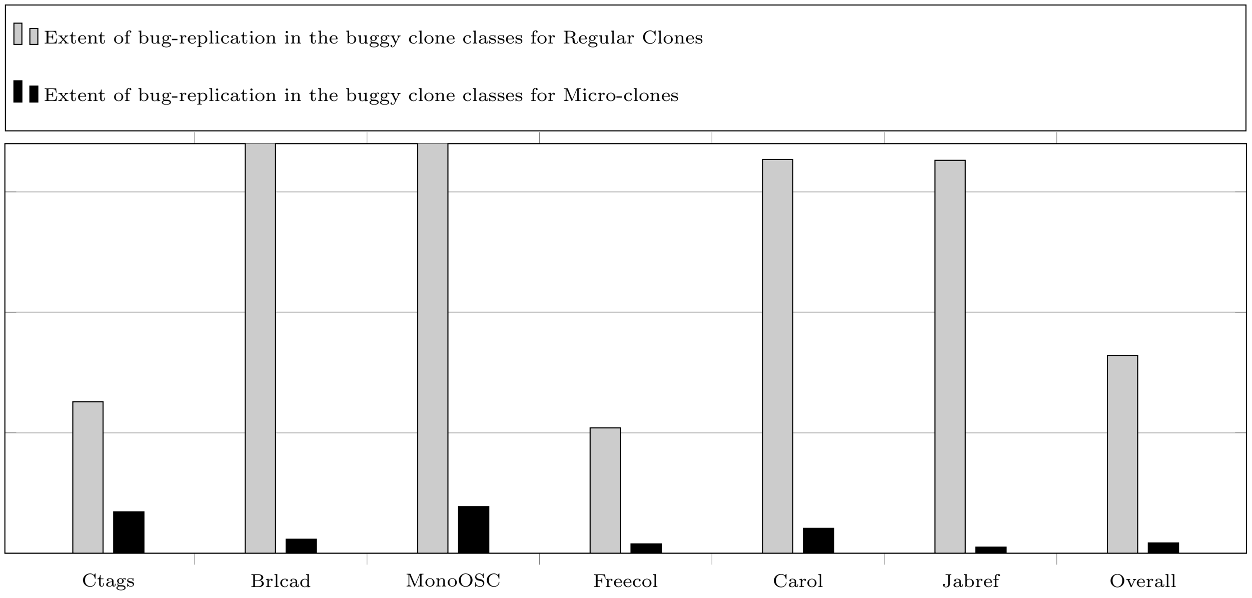


Figure 6.3: Extent of bug-replication in the buggy clone classes for regular and micro-clones.

total number of bugs that got replicated respectively for each of the subject systems. We calculate these numbers for both regular and micro-clones. We use following two counters to demonstrate our methods.

NBC (The number of bugs experience by the code clones): Suppose a bug-fix commit is denoted by BFC and it was applied on the revision R of a candidate system. We increase the counter NBC by one if one or more clone classes residing in this revision R were affected by the commit BFC.

NBRC (The number of bugs that were replicated in the code clones): Suppose a bug-fix commit was applied on the revision R of a subject system and one or more clone classes in this revision were affected by this commit. We increase the counter NBRC by one if any of these affected clone classes experienced a similarity preserving co-change (SPCO) in this commit. According to our previous discussion in Section 5.4 experiencing a similarity preserving co-change (SPCO) in a bug-fix commit operation is an evidence of the existence of replicated bugs in the code clones.

The percentage of replicated bugs with respect to all bugs found in code clones for both regular and micro-clones of a subject system is calculated by Equation 6.6.

$$PNBRC = \frac{100 \times NBRC}{NBC} \tag{6.6}$$

We calculate overall percentage of replicated bugs with respect to all bugs for six subject systems using following equation.

$$OPNBRC = \frac{100 \times \sum_{all\ systems} NBRC}{\sum_{all\ systems} NBC} \tag{6.7}$$

Table 6.6 shows the number of replicated bugs with respect to all bugs for both regular and micro-clones for each of the subject systems. Here, we observe that for three subject system i.e. Ctags, Brlcad and Carol

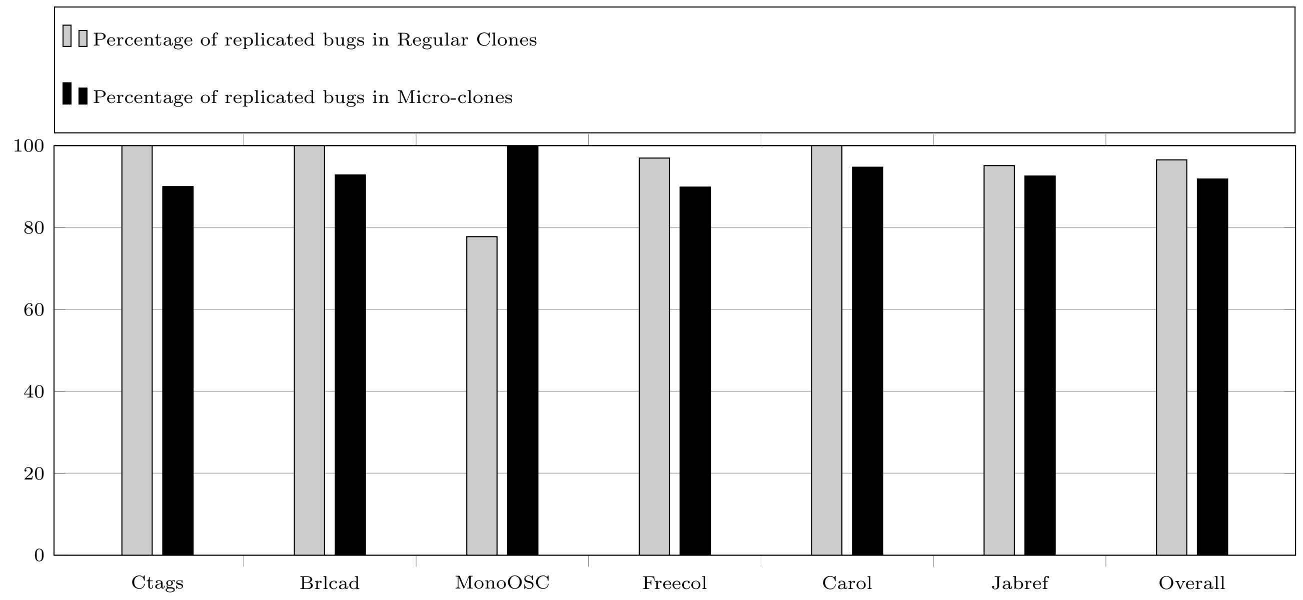


Figure 6.4: Percentage of replicated bugs in regular and micro-clones.

in regular code clones all of the bugs are replicated bugs. In case of micro-clones this scenario is true for MonoOSC. For the rest of the subject systems, in both regular and micro-clones majority number of bugs are found to be replicated bugs. Figure 6.4 depicts the percentage of replicated bugs in regular and micro-clones. In this figure we see that for all of the subject systems, percentage of replicated bugs is slightly higher in regular clones than in micro-clones except for the MonoOSC. The overall percentage of replicated bugs in regular code clones is 96.53% and in micro-clones it is 91.87%. To better understand the difference of these two percentages we perform following MWW test.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 3. We want to know whether the difference between two percentages of replicated bugs with respect to all bugs in regular and micro-clones is significant or not. To investigate statistical significance we perform Mann-Whitney-Wilcoxon (MWW) test [9, 10] on the results. We found that for two-tailed, 5% significance level the p-value is 0.20054 and critical U-value is 5. Here, p-value is greater than the significance level 0.05. Also, we found that the U-value is 9.5 which is greater than the critical U-value 5. Hence, the MWW test result reveals that the difference of percentages is not significant.

Answer to RQ 3. A substantial number of bugs in both regular code clones and micro-clones are found to be replicated bugs. The overall percentage of replicated bugs is higher by 4.66% in regular clones than micro-clones.

Since the difference of the two percentages (regular and micro-clones) is insignificant, we can say that both micro-clones and regular clones contain a high percentage (over 90%) of replicated bugs in code bases. Hence, both micro-clones and regular code clones carry equivalent importance in software maintenance.

Table 6.6: Number of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones

Subject	**Regular Clones**		**Micro Clones**	
Systems	**NBRC**	**NBC**	**NBRC**	**NBC**
Ctags	15	15	9	10
Brlcad	11	11	13	14
MonoOSC	7	9	4	4
Freecol	64	66	80	89
Carol	31	31	36	38
Jabref	39	41	50	64

6.3.4 *RQ 4: Are the replicated bugs in micro-clones more likely to be severe than the replicated bugs in regular clones?*

Motivation. Finding the severity of replicated bugs is an important criterion to compare between regular and micro code clones. Answering this research question will let us know which code clone requires more attention than the other during fixing a bug. Since severe bugs need to be fixed immediately, if severe replicated bugs occur more in a certain type (regular or micro-clones) then developers can be more careful about that type of clone in advance during software maintenance.

Methodology. To find the severe replicated bugs we automatically perform a heuristic search in bug-fix commit messages of regular and micro-clones for the six subject systems. According to Lamkanfi's [98] proposal, a list of keywords can identify severe and non-severe bugs from textual information. We use these keywords to identify severe bugs in replicated bug-fixing commit messages of our candidate systems.

NSBRC (The number of severe bugs that were replicated in the code clones): We denote the number of severe replicated bugs of a subject system by NSBRC. We increase the value of NSBRC by one if a replicated bug found in code clones is severe.

We calculate the percentage of severe replicated bugs with respect to all replicated bugs in regular code clones and micro-clones using Equation 6.8. Here, NBRC is the number of bugs that were replicated in code clones as defined in previous Section 7.3.3.

$$PNSBRC = \frac{100 \times NSBRC}{NBRC} \tag{6.8}$$

We also calculate overall percentage of severe replicated bugs with respect to all replicated bugs in regular and micro-clones using the following way.

$$OPNSBRC = \frac{100 \times \sum_{all\ systems} NSBRC}{\sum_{all\ systems} NBRC} \tag{6.9}$$

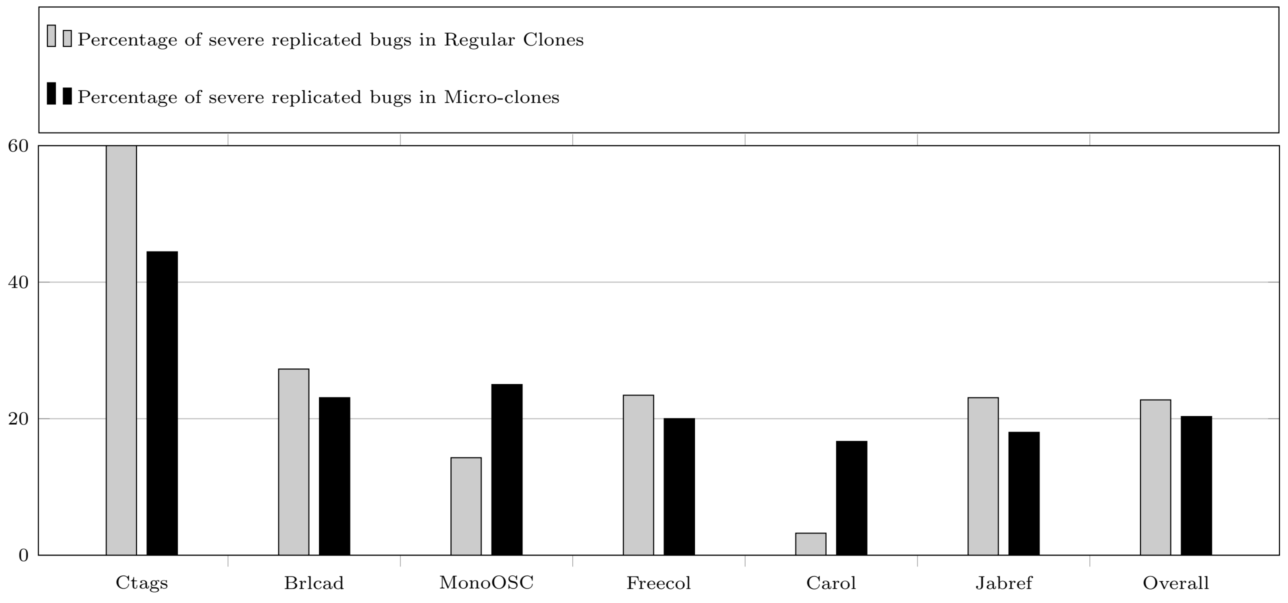


Figure 6.5: Percentage of severe replicated bugs in regular and micro-clones.

Here, OPNSBRC refers to overall percentage of severe bugs which are related to bug-replication with respect to all bugs related to bug-replication in regular and micro-clones.

Table 6.7 shows the number of severe bugs that experienced bug-replication in code clones for both regular and micro-clones in six subject systems. Figure 6.5 shows the percentage of severe replicated bugs with respect to all replicated bugs for regular and micro-clones. For subject system MonoOSC and Carol percentage of severe replicated bugs is higher (i.e. 25% and 16.67% respectively) in micro-clones than that (i.e. 14.29% and 3.23% respectively) of regular code clones. For the rest of the subject systems percentage of severe replicated bugs is higher in regular code clones than the micro-clones. In overall, percentage of severe replicated bugs is slightly higher in regular code clones than micro-clones.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 4. To understand that if the difference between the percentages of severe replicated bugs in regular and micro-clones is significant or not, we conduct MWW test [9, 10] on RQ4's result. For significance level of 5% and two-tailed MWW test we find the critical value of U is 5. Our MWW test result shows that the p-value is 1 which is far greater than the significance level 0.05 and U-value is 17.5 which is greater than critical U-value 5. This proves that the percentage of severe replicated bugs in regular and micro-clones is nearly equal. Table 6.8 shows all the result of MWW tests for three research questions. Here, we can see that for all RQs p-values are greater than significance level 0.05 and U-values are greater than critical U-value.

Answer to RQ 4. Replicated bugs in regular code clones are more severe than the replicated bugs in micro-clones. The overall percentage of severe replicated bugs for regular clones is 22.75% and for micro-clones it is 20.31%.

Table 6.7: Number of Severe Replicated Bugs that Experienced by code clones in Regular and Micro-Clones

Subject	**Regular Clones**		**Micro Clones**	
Systems	**NBRC**	**NSBRC**	**NBRC**	**NSBRC**
Ctags	15	9	9	4
Brlcad	11	3	13	3
MonoOSC	7	1	4	1
Freecol	64	15	80	16
Carol	31	1	36	6
Jabref	39	9	50	9

Table 6.8: Mann-Whitney-Wilcoxon Test Result for RQ1, RQ3 and RQ4

Research Question No.	**p-value**	**U-value**	**Critical Value of U**
RQ1	0.93624	17	5
RQ3	0.20054	9.5	5
RQ4	1	17.5	5
Considering level of significance is 5%. For all RQs, U-value > Critical value of U			

Since the difference between the percentage of severe replicated bugs in regular and micro-clones is minor i.e. 2.44%, we can say that both regular and micro-clones should be taken care of during clone management from the bug severity perspective.

For all research questions, we perform manual analysis to check our implementation is correct. For all subject systems, we investigated at least first 50 clone fragments and/or revisions manually to confirm our analysis.

6.4 Related Work

From Section 4.6, we see that [40, 191, 135] introduced research on micro-clones. However, none of these three studies on micro-clones [40, 191, 135] showed any comparison between micro-clones and non-micro clones or regular clones. Recently, Islam et al. [73] performed a comparative study on bug-proneness in between regular and micro-clones and showed that micro-clones are more bug-prone than regular clones. However, they did not compare the bug-replication in micro-clones and regular clones. In a previous study Islam et al. [72] investigated bug-replication on three types of regular clones (i.e. Type 1, Type 2 and Type 3) and found that bug-replication is a common phenomenon in code cloning. They did not consider micro-clones in their study. Whereas in this study, we compare bug-replication between regular clones and micro clones.

6.5 Limitations

We used the NiCad clone detector [49] for detecting both micro and regular clones. While all clone detection tools suffer from the *confounding configuration choice problem* [197] and might give different results for different settings of the tools, the setting that we used for NiCad for this experiment are considered standard [153] and with these settings NiCad can detect clones with high precision and recall [156, 155, 175]. Thus, we believe that our findings on the bug-replication of micro code clones and regular code clones are of significant importance.

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique proposed by Mocus and Votta [123] and also used by Barbour et al. [36]. The technique proposed by Mocus and Votta [123] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [36] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

The number of total subject systems is not enough in our research to be able to generalize our findings regarding the comparative bug-replication of micro and regular clones. However, our candidate systems were of diverse variety in terms of application domains, sizes and revisions. Thus, we believe that our findings are important from the perspectives of managing code clones.

6.6 Summary

In this paper, we investigate and compare the aspects of bug-replication in between regular code clones and micro-clones. After investigating on six diverse subject systems, we found that micro-clones are as equally important as regular code clones. We have found that the percentage of clone fragments which are related with bug-replication is some times higher in micro-clones than that of regular code clones. Moreover, the percentage of replicated bugs in micro-clones is almost the same as the percentage in regular clones. Additionally, both regular code clones and micro code clones have the similar tendencies of replicating severe bugs. We believe that these findings are important from the clone management perspective. Both micro-clones and regular clones should be considered for software evaluation and maintenance. Since only a limited number of studies have been performed on micro-clones, more research is needed to elaborate micro-clone more profoundly. In future we would like to explore micro-clones for more number of systems of various programming languages so that we can perform a language centric empirical study on bug-proneness of micro-clones.

7 Empirical Study on Bug Replication in Regular and Micro Clones

Duplicating source code is a common phenomenon during software development by programmers. Previous research did not consider clone fragments, which are less than 5 LOCs. In recent literature, these clones are defined as micro-clones. Previous studies on clone code showed that there is a relationship between code clones and software bugs as well as replicated bugs. No other study has been performed on replicated bugs in micro-clones. In this study, we explore bug replication in both regular and micro-clones using three different clone detection tools (NiCad, ConQat, and iClones). In order to achieve our goal, we analyze bug-fixing similarity preserving co-changes (SPCOs) in regular and micro-clones through the evolutionary archive of our subject systems. After investigating thousands of revisions in five diverse candidate systems written in C and Java, we find that the percentage of replicated clone fragments is often higher in micro-clones than in regular clones. The percentage of bug-replication that occurs in micro-clones is nearly the same as the percentage as regular clones. Our study indicates that regular code clones and micro-clones both replicate severe bugs likewise. Finally, the percentage of line coverage is higher in micro-clones than in regular clones. Thus, we should not disregard micro-clones in software systems. Micro-clones should be considered equally important as of regular code clones while making clone management resolutions.

7.1 Introduction

This study is a significant extension of previous chapter i.e. Chapter 6, our earlier study [75] on bug replication in regular and micro-clones. In our previous study, we used NiCad clone detection tool to find clones. In this study, we use two more state-of-the-art clone detection tools i.e. ConQat [87] and iClones [59], for detecting both regular and micro-clones. We wanted to investigate whether our previous study is biased by the clone detection tool. We also answer one more research question in this extensive study and compare the results among three clone detection tools. Unfortunately, ConQat and iClones do not support C# programming language. Hence, we discard the C# language for our extended study. Thus, we discard one subject system (MonoOSC) from our previous study [75], written in the C# language. Also, to make the experiment feasible, we work on fewer revisions than our previous study [75] for Java subject systems only. Thus, all the reported data in tables and figures are revised in this new study. To explore the effects of bug replication in micro-clones and regular code clones, we perform a comparative study. To the best of our knowledge, our research is the first study that compares bug-replication between micro-clones and regular code clones using three clone detectors.

We consider thousands of revisions of five diverse open-source software systems written in two languages, Java and C. We use the NiCad [49], ConQat [87], and iClones [59] clone detector for clone detection from each of the revisions of a subject system. We analyze the evolution history of micro-clones and regular clones and investigate whether they contain replicated bugs and to what extent.

In our experiment, the minimum size of a code fragment is 1 LOC for micro-clones, and the maximum size of a micro-clone code fragment is 4 LOC. We consider those micro-clones that are not part of regular clones. We detect regular code clones that have a minimum of 5 LOCs since Wang et al. [197] reported that this is the best minimum threshold for detecting regular clones.

To explore bug replication, we discover bug-fix commits reported by the developers from thousands of commits in our open source subject systems in cases of both regular and micro-clones. We observe bug-replication intensity in both micro-clones and regular code clones and answer five important research questions listed in Table 7.1. We answered first four research questions in our previous study [75] using NiCad clone detection tool only. In this extended study we have added the fifth research question and answer five research questions for all three clone detection tools with revised data set. The significant findings from our research are as follows.

- The percentage of clone fragments that experienced bug-replication in micro-clones is often higher than regular code clones.
- The percentage of bugs that get replicated is almost the same for micro-clones and regular code clones.
- The severe replicated bugs are found in both regular and micro-clones almost equally.
- The percentage of line coverage of replicated clone fragments is higher in micro-clones than in regular code clones.

These findings prove that micro-clones are equally important as regular code clones. Thus, we should also consider micro-clones while making clone management resolution.

The organization of this chapter is as follows. Section 7.2 discusses the experimental steps; Section 7.3 presents and analyzes the experimental results and answers our research questions. Section 7.4 discusses the related work and compares our study with the existing ones, Section 7.5 discusses possible threats to validity, and Section 7.6 concludes the paper and discusses possible future work.

7.2 Experiment Steps

We conduct our research on five subject systems (two C and three Java systems). We consider these five subject systems since these systems have variations in application domains, sizes, and revisions. These subject

Table 7.1: Research Questions

SL	Research Question
RQ 1	What percentage of the clone fragments in Regular and Micro-clones takes part in bug-replication?
RQ 2	What is the extent of bug-replication in the buggy clone classes of Regular code clones and Micro-clones?
RQ 3	What percentage of bugs experienced by regular and micro-clones are replicated bugs?
RQ 4	Are the replicated bugs in micro-clones more likely to be severe than the replicated bugs in regular clones?
RQ 5	What is the percentage of line coverage of replicated clone fragments in both regular and micro-clones?

Table 7.2: Subject Systems

Systems	Lang.	Domains	LLR	Revisions
Ctags	C	Code Def. Generator	33,270	774
Brlcad	C	Solid Modeling CAD	39,309	735
Freecol	Java	Game	30,297	1025
Carol	Java	Game	3,886	50
Jabref	Java	Reference Management	12,311	100
LLR = LOC in the Last Revision				

Table 7.3: Parameters of Clone Detection Tools

Tools	**Clones**	**Parameters**
NiCad	Regular	clone size: 5-500, block clones, blind renaming, 30% dissimilarity
	Micro	clone size: 1-4, block clones, blind renaming, 30% dissimilarity
ConQat	Regular	min. clone length: 5, block clones, gap ratio: 0.3
	Micro	min. clone length: 1, block clones, gap ratio: 0.3
iClones	Regular	min. clone: 50, min. block: 30, all transformation
	Micro	min. clone: 10, min. block: 30, all transformation

systems are listed in Table 7.2, which were downloaded from the SourceForge online SVN repository [16]. For each subject system, the total number of revisions and the total lines of code (LOC) in the last revision is given in Table 7.2.

7.2.1 Preliminary Steps

We accomplish the following steps for fixed bugs detection:

(1) For each subject system, we extract all revisions (as stated in Table 7.2) from the online SVN repository;

(2) We detect and extract code clones from each revision by applying NiCad [49], ConQat [87], and iClones [59] clone detectors;

(3) Using UNIX *diff* command, we detect the changes between every two consecutive revisions;

(4) We locate these changes to the already detected clones of the corresponding revisions;

(5) We detect bug-fix commit operations.

We use the tool SPCP-Miner [137] for completing the first four steps. We conduct steps 1 to 5 for both regular and micro-clones. The detection of bug-fix commits is described later in this section. In Section 7.3, we will describe how we detect bug-fix changes in regular code clones and micro-clones.

We use NiCad [49] for detecting clones in our previous study [75] since it can detect code clones with high precision and recall [156, 155]. Using NiCad for regular clones, we detect block clones (both exact and near-miss) of a minimum of 5 lines and a maximum of 500 lines. We consider 70% similarity threshold with the blind renaming of identifiers. For micro-clones using NiCad, we use the same similarity threshold and blind renaming of identifiers of a minimum 1 line and maximum 4 lines, as was detected by Mondal et al. [135]. We select the other two clone detection tools i.e. ConQat [87] and iClones [59], since the literature [175, 141] on evaluating different clone detection tools supports that these two tools outperformed than other clone detection tools. Also, we select those tools which are compatible with NiCad. Both ConQat and iClones can find clones in two programming languages i.e. Java and C. They both detect Type 1, Type 2, and Type 3 clones like NiCad. Other parameter settings for detecting both micro and regular code clones for all three clone detection tools i.e. NiCad, ConQat, and iClones, are shown in Table 7.3.

The clone detection results can be different for different settings of a clone detector. Thus, the results on bugs in code clones can also be different. Hence, it is important to select the appropriate settings (i.e., detection parameters). In our research, we used the mentioned settings to detect regular clones because, in terms of precision and recall, Svajlenko and Roy [175] show that these settings provide us with better clone detection results.

7.3 Analysis of Experimental Results

In Table 7.1, our research questions are listed. Here in this section, we analyze our experiment results for five subject systems and answer the research questions from our analysis.

7.3.1 First research question (RQ 1): What percentage of the clone fragments in Regular and Micro-clones takes part in bug-replication?

The methodology of finding the answer of this research question is same as shown in Section 6.3.1. However, in this study we perform it for three clone detection tools i.e. NiCad, ConQat, and iClones. For NiCad clone detection tool we reproduce the experiment since we consider revised number of revisions.

Table 7.4 shows the number of distinct clone code fragments that took part in bug-replication in both regular and micro-clones code for five subject systems using NiCad clone detection tools. Table 7.5 and Table 7.6 show the same value for ConQat and iClones clone detection tools respectively. From Table 7.4, we observe that the number of distinct clone fragments is higher in micro-clones for four subject systems i.e. for Brlcad, Freecol, Carol, and Jabref. For one subject system (i.e., for Ctags) number of replicated clone fragments is higher in regular code clones than that of micro-clones. For ConQat in Table 7.5, the number of distinct clone fragments is higher in micro-clones for all of the subject systems. On the contrary, for iClones, Table 7.6 shows that for all subject systems, the number of distinct clone fragments is higher in regular code clones than in micro-clones. Figure 7.1 shows the percentage of discrete clone fragments that are related to bug-replication in regular and micro-clones using the NiCad clone detection tool. Figure 7.2 and 7.3 show the same value for ConQat and iClones clone detection tools respectively. Overall, the percentage of clone fragments that experienced bug-replication is higher in micro-clones than the regular code clones for both NiCad and ConQat clone detection tools. On the other hand, iClones shows the opposite scenario i.e. the percentage of clone fragments that experienced bug-replication is higher in regular code clones than the micro-clones.

Investigating Manually for iClones Results. From Table 7.4, 7.5, 7.6, and Figure 7.1, 7.2, 7.3, we

Table 7.4: Number of Clone Fragments that Experienced Bug-replication in Regular and Micro-Clones for NiCad clone detection tool

Subject Systems	**Regular Clones (CFRB)**	**Micro Clones (CFRB)**	**Revisions (BFCR)**
Ctags	50	30	19
Brlcad	33	36	20
Freecol	8	16	5
Carol	0	11	2
Jabref	0	2	1

observe that NiCad and ConQat are showing similar results, whereas iClones shows opposite results compared to NiCad and ConQat clone detection tools. We are interested to learn about the reason behind this result. We manually investigate all the results for at least one bug-fixing revision (e.g. Brlcad, revision-12) for all three clone detection tools. We found that the total number of micro-clones is very low in iClones, and thus the number of replicated clone fragments in micro-clones for iClones is almost zero. In Table 7.6, we can see that Freecol has only two replicated clone fragments in micro-clones, and for the rest of the subject systems, there is none. The detection of a low number of micro-clones using iClones reflects in all results. This is the reason for getting different results in iClones compared to NiCad and ConQat clone detection tools. This is expected since these tools are mainly designed for regular code clones ignoring the micro-clones. Despite this constraint, NiCad and ConQat caught a good number of micro-clones, whereas iClones failed. Thus, we expect this trend will continue for the rest of the research questions in this paper.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 1. To inspect the statistical significance of the result, we perform the Mann-Whitney-Wilcoxon (MWW) test [9, 10] for all three clone detection tools i.e. NiCad, ConQat, and iClones. We are interested to know that if the percentage of the clone fragments that participate in bug-replication in micro-clones is substantially higher than in regular clones. It is a non-parametric test and does not require the normal distribution of data. The level of significance of our MWW test is 5%. This implies that if the p-value is less than 0.05 and the U-value is less than or equal to the critical U-value, then the difference will be significant. We found that for NiCad, the p-value is 0.1443, which is greater than the significance level 0.05. Also, the U-value is 5 and the critical value of U at $p<0.05$ is 2. Here, the U-value is greater than the critical U-value. For both ConQat and iClones, the p-value is 0.0601, the U-value is 3, and at $p<0.05$, the critical value of U is 2. This MWW test result conquers that the difference between the two percentages of clone fragments that contain replicated bugs in regular and micro-clones is insignificant for all three clone detection tools.

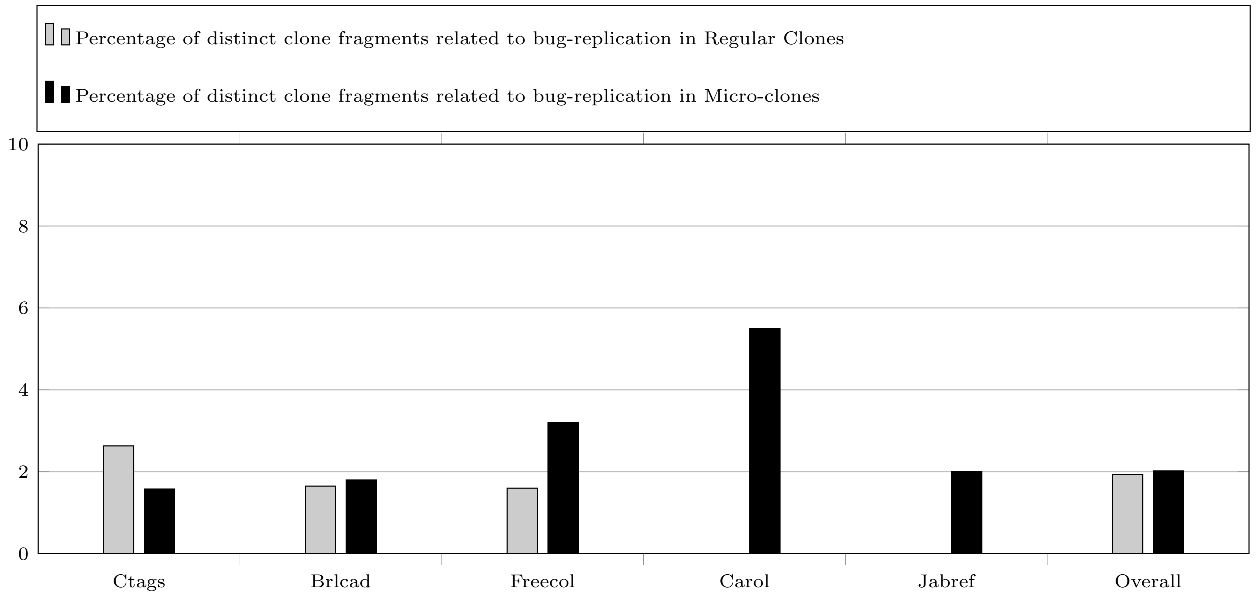


Figure 7.1: Percentage of clone fragments that related to bug-replication in regular and micro-clones for NiCad clone detection tool.

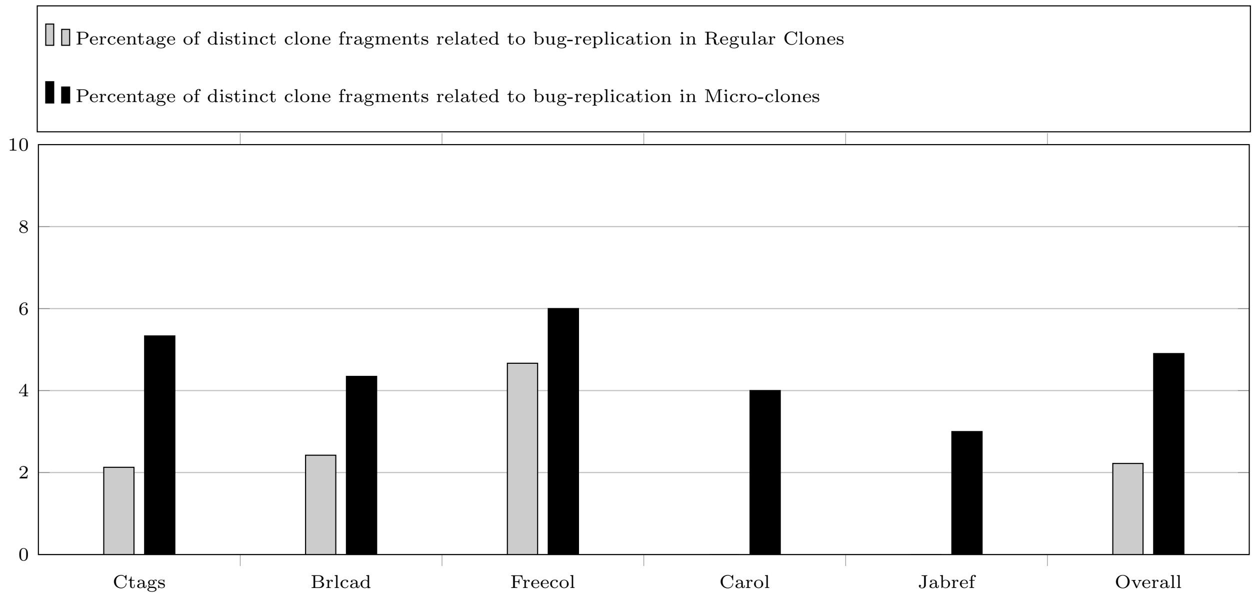


Figure 7.2: Percentage of clone fragments that related to bug-replication in regular and micro-clones for ConQat clone detection tool.

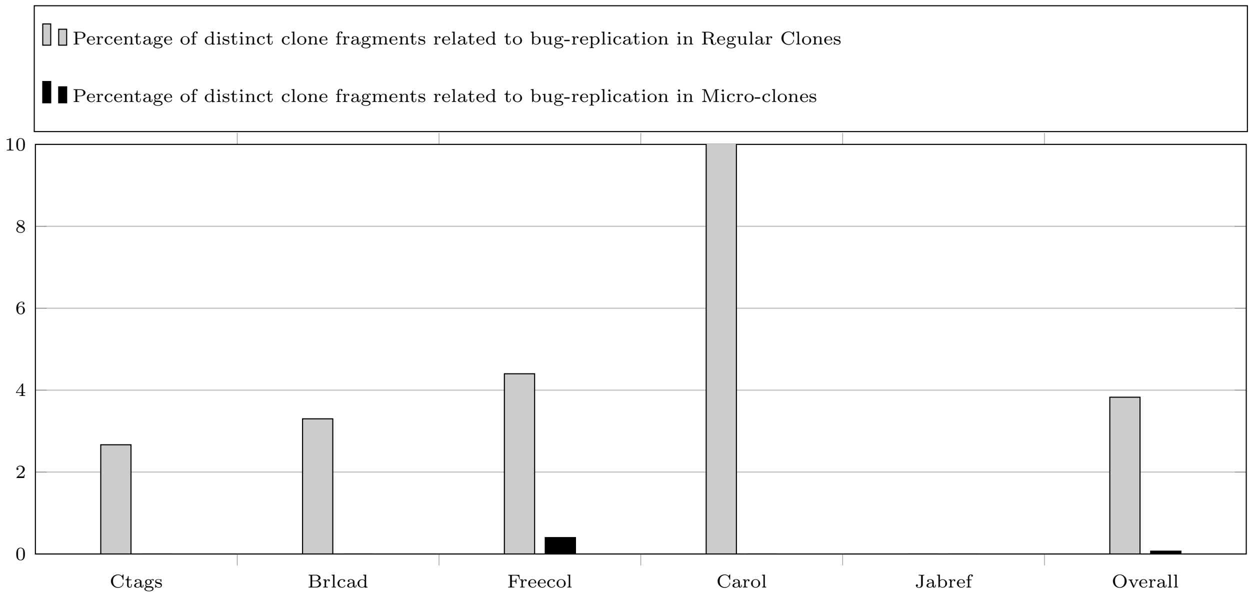


Figure 7.3: Percentage of clone fragments that related to bug-replication in regular and micro-clones for iClones clone detection tool.

Table 7.5: Number of Clone Fragments that Experienced Bug-replication in Regular and Micro-Clones for ConQat clone detection tool

Subject Systems	**Regular Clones (CFRB)**	**Micro Clones (CFRB)**	**Revisions (BFCR)**
Ctags	83	208	39
Brlcad	63	113	26
Freecol	14	18	3
Carol	0	8	2
Jabref	0	6	2

Table 7.6: Number of Clone Fragments that Experienced Bug-replication in Regular and Micro-Clones for iClones clone detection tool

Subject Systems	**Regular Clones (CFRB)**	**Micro Clones (CFRB)**	**Revisions (BFCR)**
Ctags	32	0	12
Brlcad	33	0	10
Freecol	22	2	5
Carol	24	0	2
Jabref	0	0	0

Answer to RQ 1. The percentage of the clone fragments in micro-clones that take part in bug-replication is higher than that of regular code clones for NiCad and ConQat clone detection tools. However, for iClones, the percentage is higher in regular code clones than micro-clones.

We can state that the difference between the two overall percentages (regular and micro-clones) is not prominent from our above experimental results. This indicates that micro-clones are equally important as regular code clones. Thus, we can not ignore micro-clones during software maintenance tasks. While performing clone management, developers should equally emphasize on both micro-clones and regular code clones.

7.3.2 Second research question (RQ 2): What is the extent of bug-replication in the buggy clone classes of Regular code clones and Micro-clones?

We have follow the same procedure as described in Section 6.3.2. However, this time we apply three clone detection tools (NiCad, ConQat, iClones) along with the revised data set.

Table 7.7 shows the extent of bug-replication experienced by both regular and micro-clones for the NiCad clone detection tool. Figure 7.4 depicts the extent of replicated bugs in the buggy clone classes for regular and micro-clones using NiCad. From Table 7.7 and Figure 7.4 for the NiCad clone detection tool, we perceive that the extent of replicated bugs is a bit higher in regular code clones than in micro-clones for each of the subject systems. The highest extent (100%) of replicated bugs found in Freecol for regular code clones. Second highest extent (97.06%) of the bug-replication found in Brlcad for regular clone code. Other than these two cases, for the rest of the cases, the extent of bug-replication for regular clones varies from 0% to 7%, and for micro-clones, it varies from 0.22% to 8%. From Table 7.8 and Figure 7.5 for the ConQat clone detection tool, we see a similar result compared with NiCad. The highest extent (100%) of replicated bugs

Table 7.7: Extent of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for NiCad clone detection tool

Subject	**Regular Clones**		**Micro Clones**	
Systems	**CFRB**	**CF**	**CFRB**	**CF**
Ctags	50	795	30	1745
Brlcad	33	34	36	6189
Freecol	8	8	16	7425
Carol	0	0	11	145
Jabref	0	0	2	322
Total	91	837	95	15826

found in Freecol for regular code clones. On the other hand, for micro-clones, the highest extent (88.89%) of replicated bugs found in Carol. For iClones from Table 7.9 and Figure 7.6, we observe that the extent of replicated bugs is greater in micro-clones than regular code clones. The reason behind this was explained previously (Section 7.3.1). Again, the number of micro-clones detected by iClones is very low; thus, the number of replicated micro-clones is zero for all subject systems except Freecol.

For our second research question, we did not perform the MWW test because we believe that the total number of bug-replication extensions is more important than the percentage. We found that the total number of clone fragments is very high in micro-clones compared with regular clones. For example, from Table 7.7 we see that in Freecol, the total number of clone fragments in eligible clone classes is 7425 in micro-clones, whereas in regular clones, it is only 8 which is more than 900 times. However, the number of clone fragments encountered bug-replication is only 16 and 8 in micro clones and regular clones respectively. Table 7.8 shows a similar scenario for ConQat. We have to deal with the total number of clone fragments (all the bugs need to be fixed) rather than percentages for the purpose of clone refactoring or clone tracking. Therefore, though the extent of bug-replication in the buggy clone classes is higher in regular clones than micro-clones for both NiCad and ConQat, we should not neglect micro-clones because of its characteristic of having a higher number of replicated clone fragments compared to regular clones.

Answer to RQ 2. The extent of replicated bugs in the clone classes of regular code clones is higher than the extent in micro-clones. The overall extent of bug-replication in regular clones is 10.87%, and in micro-clones the overall extent of bug-replication is 0.60% for NiCad. For ConQat the overall extent of replicated bugs in regular clones is 95.81%, and in micro-clones the overall extent of bug-replication is 2.52%.

Table 7.8: Extent of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for ConQat clone detection tool

Subject	**Regular Clones**		**Micro Clones**	
Systems	**CFRB**	**CF**	**CFRB**	**CF**
Ctags	83	86	208	10411
Brlcad	63	67	113	2901
Freecol	14	14	18	548
Carol	0	0	8	9
Jabref	0	0	6	147
Total	160	167	353	14016

Table 7.9: Extent of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for iClones clone detection tool

Subject	**Regular Clones**		**Micro Clones**	
Systems	**CFRB**	**CF**	**CFRB**	**CF**
Ctags	32	32	0	0
Brlcad	33	36	0	0
Freecol	22	398	2	2
Carol	24	24	0	0
Jabref	0	0	0	0
Total	111	490	2	2

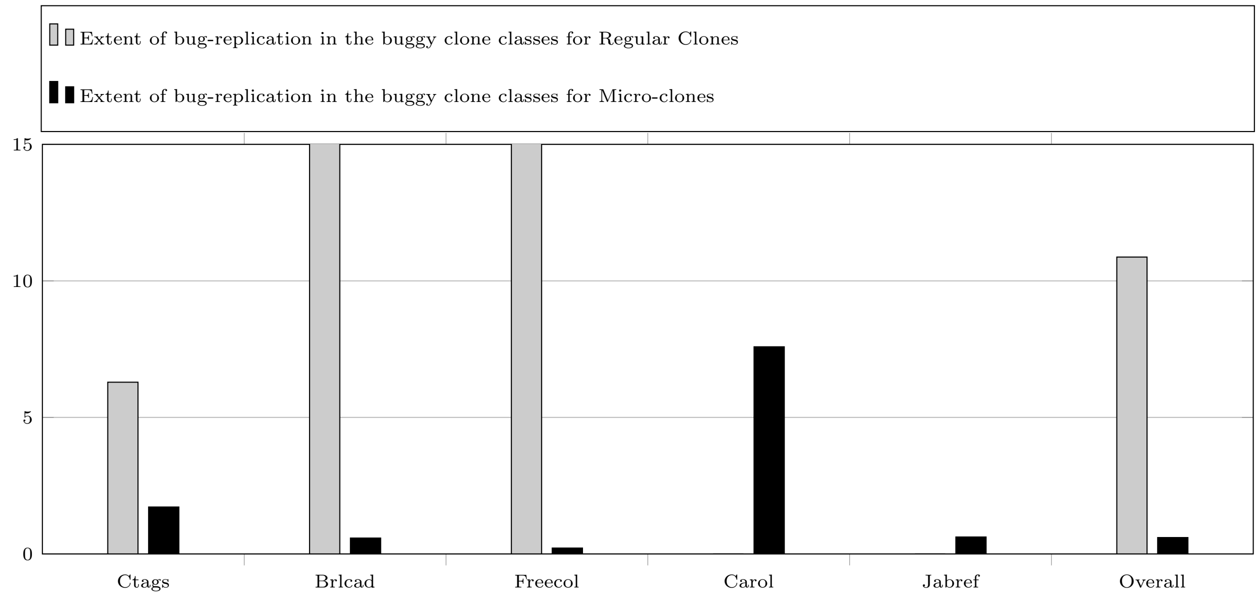


Figure 7.4: Extent of bug-replication in the buggy clone classes for regular and micro-clones for NiCad clone detection tool.

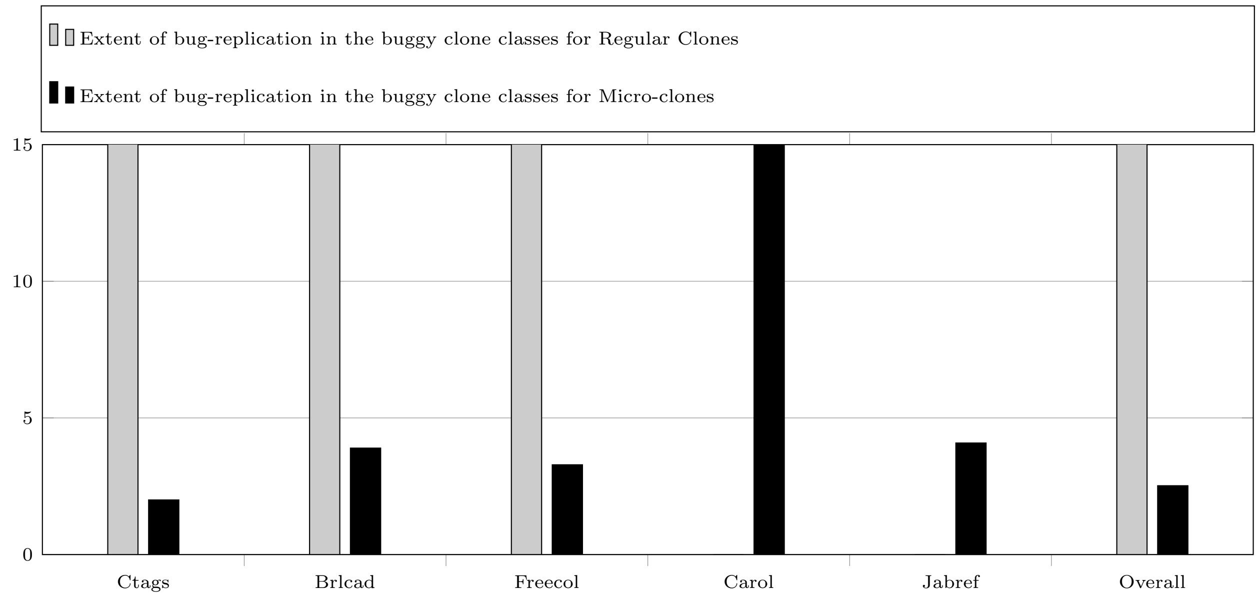


Figure 7.5: Extent of bug-replication in the buggy clone classes for regular and micro-clones for ConQat clone detection tool.

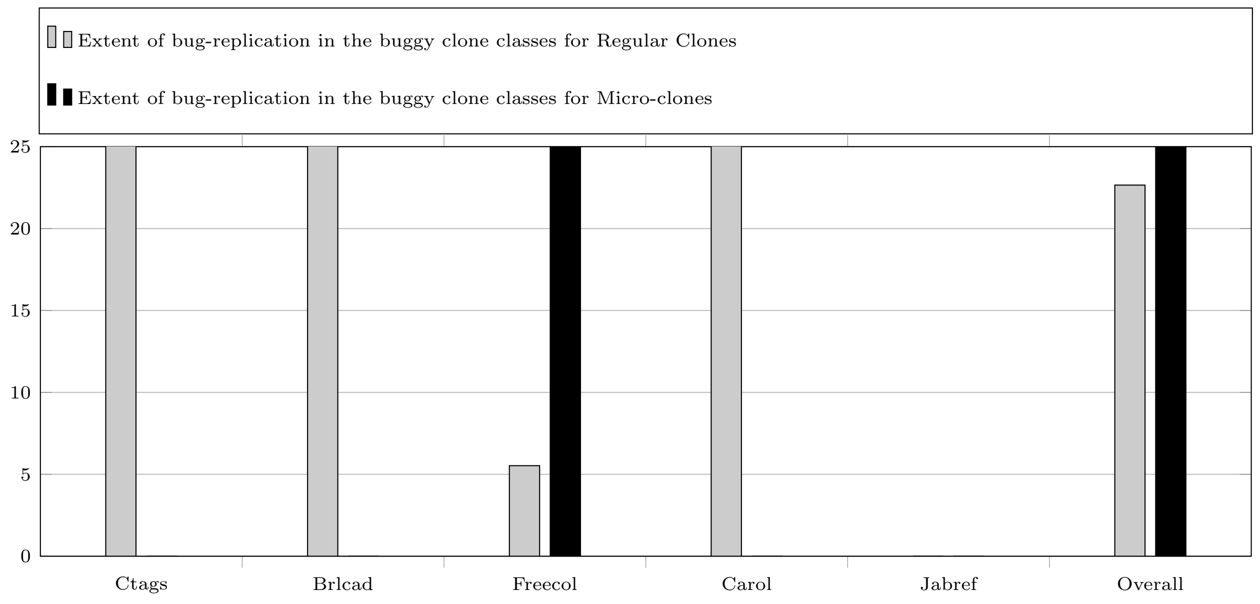


Figure 7.6: Extent of bug-replication in the buggy clone classes for regular and micro-clones for iClones clone detection tool.

Clone classes that have replicated bugs are considered more harmful. Developers should focus on clone classes that contain replicated bugs when refactoring clone classes of code clones. In this research question, we found that both regular clones and micro-clones have replicated bugs in their clone classes. Thus, both regular and micro-clones should be equally considered carefully while clone management (such as refactoring).

7.3.3 Third research question (RQ 3): What percentage of bugs experienced by regular and micro-clones are replicated bugs?

Following the same methodology described in Section 6.3.3, we consider three clone detection tools and perform the experiment with revised data set to answer our third research question.

Table 7.10 shows the number of replicated bugs with respect to all bugs for regular and micro-clones for each of the subject systems for NiCad clone detection tool. Table 7.11 and 7.12 show the same value for ConQat and iClones. Here in Table 7.10, we observe that all bugs are replicated for all five subject systems in regular code clones. For micro-clones this scenario is true for Freecol, Carol, and Jabref. For the rest of the candidate systems, in micro-clones, most bugs are found to be replicated bugs. In Table 7.11 for ConQat, we see that for all five subject systems in regular code clones, all bugs are replicated bugs. On the other hand, for micro-clones this ranges from 50% to 100%. On the contrary, in Table 7.12 for iClones, shows that in micro-clones for all subject systems, all the bugs are replicated bugs. For regular code clones, all the bugs are replicated bugs except for the Freecol (83.33%). Figure 7.7, 7.8, 7.9 depict the percentage of replicated bugs in regular and micro-clones for NiCad, ConQat, and iClones respectively. In Figure 7.7 for NiCad, we see that the percentage of replicated bugs is slightly higher or equal in regular clones than in micro-clones

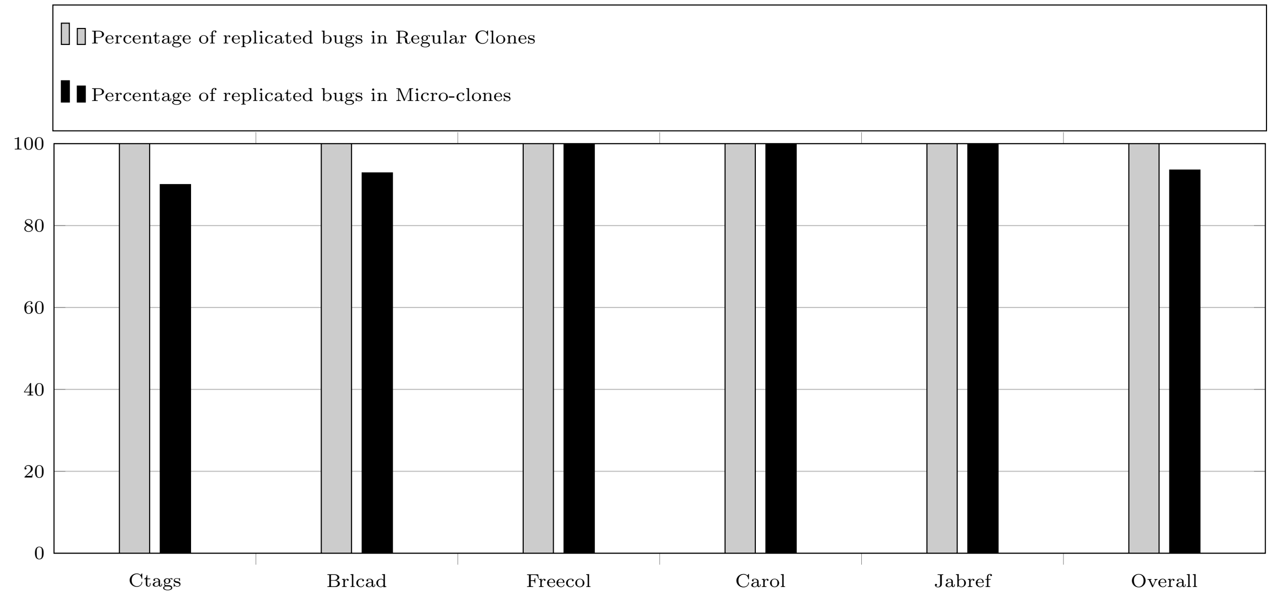


Figure 7.7: Percentage of replicated bugs in regular and micro-clones for NiCad clone detection tool.

for all of the subject systems. The overall percentage of bug-replication in regular code clones is 100%, and in micro-clones it is 93.55% for NiCad. In figure 7.8 for ConQat, we see that percentage of replicated bugs is much higher (or equal in the case of Carol) for all subject systems in regular clones than in micro-clones. In the case of iClones in figure 7.9, we observe that for all subject systems, the percentage of replicated bugs are equal in both regular and micro-clones except Freecol. We perform the following MWW test to understand the difference between these two percentages better.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 3. We want to confirm whether the two percentages of replicated bugs with respect to all bugs in regular and micro-clones differ significantly using the MWW test. To investigate statistical significance, we perform the MWW test [9, 10] on the results. We found that for NiCad, for a two-tailed, 5% significance level the p-value is 0.34722 and the critical U-value is 2. Here we see, the p-value is greater than the significance level 0.05. Also, we found that the U-value is 7.5, which is greater than the critical U-value 2. For ConQat, p-value is 0.0477, U-value is 2.5, and critical U-value is 2. For iClones, p-value is 0.67448, U-value is 10, and critical U-value is 2. Hence, the MWW test results reveal that the difference in percentages for NiCad, ConQat and iClones is not significant.

Answer to RQ 3. A significant number of bugs found in regular code clones and micro-clones as replicated bugs. The overall percentage of bug-replication is higher in regular clones than micro-clones for NiCad and ConQat. For iClones, the overall percentage of replicated bugs in slightly higher in micro-clones than regular clones.

Since the difference between the two percentages (regular and micro-clones) is insignificant, we can say that both micro-clones and regular clones contain a high percentage of replicated bugs in code bases. Therefore,

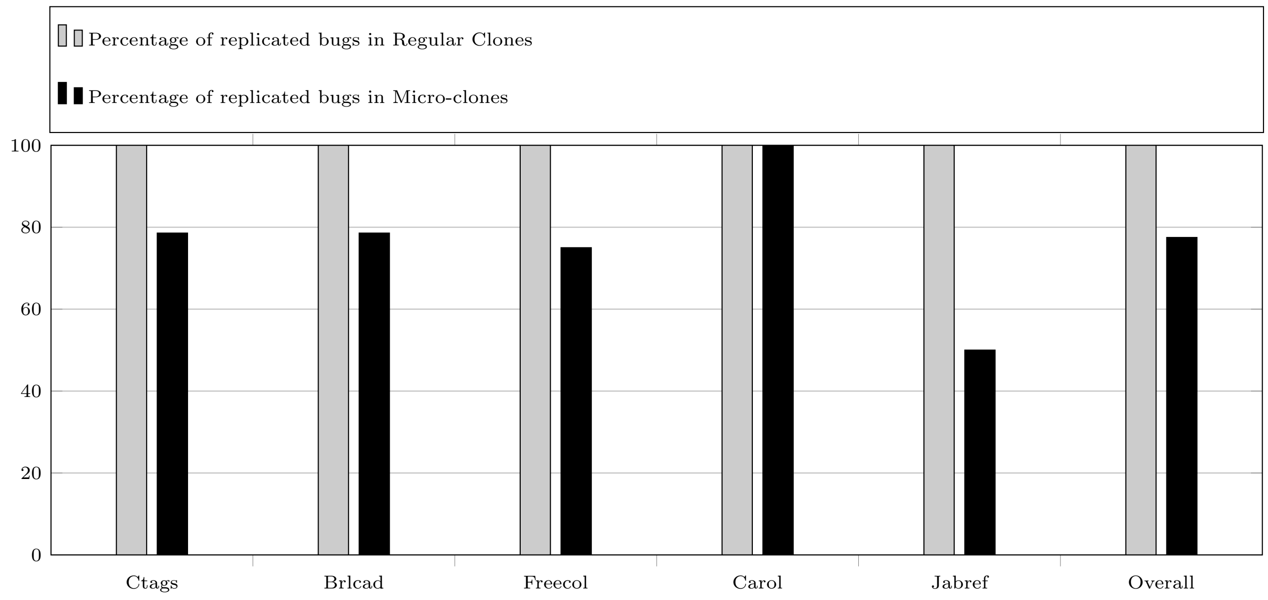


Figure 7.8: Percentage of replicated bugs in regular and micro-clones for ConQat clone detection tool.

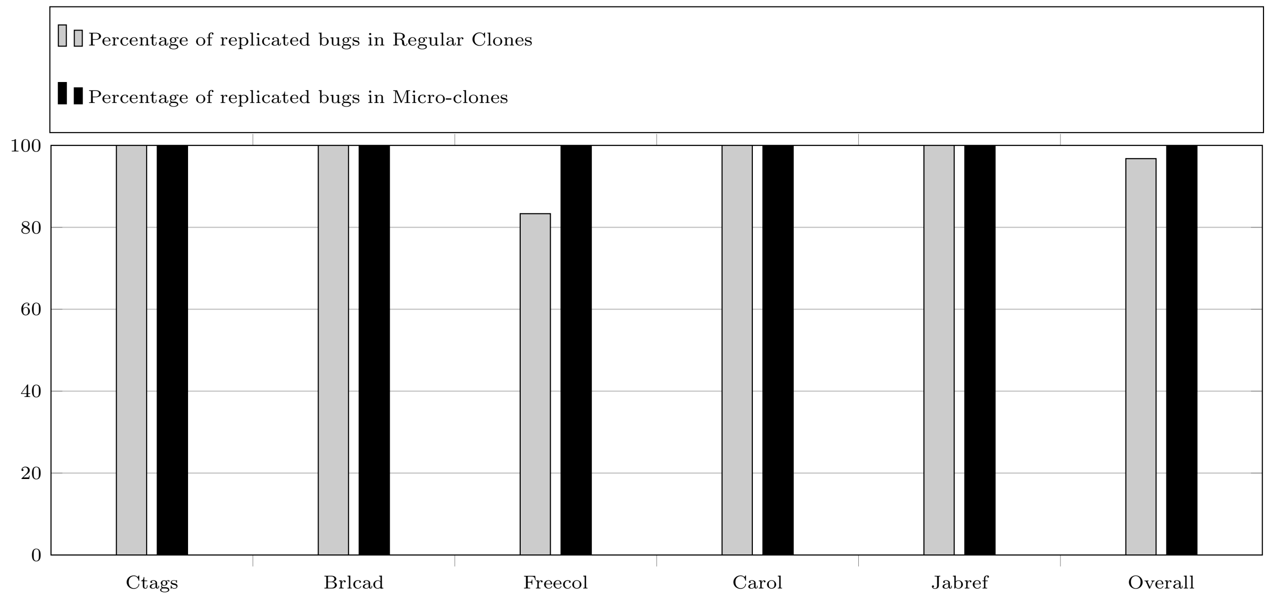


Figure 7.9: Percentage of replicated bugs in regular and micro-clones for iClones clone detection tool.

Table 7.10: Number of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for NiCad clone detection tool

Subject	Regular Clones		Micro Clones	
Systems	NBRC	NBC	NBRC	NBC
Ctags	15	15	9	10
Brlcad	11	11	13	14
Freecol	4	4	4	4
Carol	1	1	2	2
Jabref	1	1	1	1

Table 7.11: Number of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for ConQat clone detection tool

Subject	Regular Clones		Micro Clones	
Systems	NBRC	NBC	NBRC	NBC
Ctags	16	16	33	42
Brlcad	12	12	22	28
Freecol	3	3	3	4
Carol	1	1	2	2
Jabref	1	1	2	4

Table 7.12: Number of Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for iClones clone detection tool

Subject	Regular Clones		Micro Clones	
Systems	NBRC	NBC	NBRC	NBC
Ctags	12	12	1	1
Brlcad	10	10	1	1
Freecol	5	6	1	1
Carol	2	2	1	1
Jabref	1	1	1	1

both micro-clones and regular code clones carry equal importance in software maintenance.

7.3.4 Fourth research question (RQ 4): Are the replicated bugs in micro-clones more likely to be severe than the replicated bugs in regular clones?

The methodology of how we answer our fourth research question has been described in previous Chapter 6 in Section 6.3.4. However, we use three clone detection tools instead of one in this study along with revised data set.

Table 7.13 shows the number of severe bugs that experienced bug-replication in code clones for both regular and micro-clones in five subject systems for NiCad. Table 7.14 and Table 7.15 shows the same value for ConQat and iClones clone detection tools respectively. Figure 7.10 shows the percentage of severe replicated bugs with respect to all replicated bugs for regular and micro-clones for NiCad. For the subject system Freecol and Jabref, the percentage of severe replicated bugs are equal (i.e. 50% and 100% respectively) in micro-clones and regular code clones. For the rest of the candidate systems percentage of severe replicated bugs is higher in regular code clones than in micro-clones. Figure 7.11 shows the percentage of severe replicated bugs with respect to all replicated bugs for regular and micro-clones for ConQat. Here, we see that Freecol has an equal percentage (33.33%) for both regular and micro-clones. For the rest of the candidate systems percentage of severe replicated bugs is higher in regular clones than in micro-clones. Overall for both NiCad and ConQat, the percentage of severe replicated bugs is higher in regular code clones than micro-clones. On the other hand, Figure 7.12 shows the percentage of severe replicated bugs for the iClones clone detection tool. Here, we observe that for Jabref, the percentage is equal (100%) for both regular and micro-clones. For the rest of the subject systems and in overall, the percentage of severe replicated bugs with respect to all replicated bugs is higher in micro-clones than regular code clones.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 4. We conduct the MWW test [9, 10] on RQ4's result to understand whether the variance between the two percentages (regular and micro-clones) of severe replicated bugs is significant. For the NiCad significance level of 5% and two-tailed MWW test, we find the critical value of U is 2. The MWW test result shows that the p-value is 0.4009 and U-value is 8 which is greater than the critical U-value 2. For ConQat, p-value is 0.5287, U-value is 9, and critical U-value is 2. Here, U-value 9 is greater than critical U-value 2 and thus, the difference is insignificant. On the other hand for iClones, the p-value is 0.0477, U-value is 2.5, and the critical U-value 2; thus, the difference is not significant. The test result shows that the percentage of severe replicated bugs in regular and micro-clones is nearly equal for all three clone detection tools.

Answer to RQ 4. The percentage of replicated bugs that are severe in regular code clones is generally higher than the corresponding percentage in micro-clones. For NiCad and ConQat, the overall percentage of severe replicated bugs for regular clones is 50% and 36.36%; for micro-clones it is 37.93%

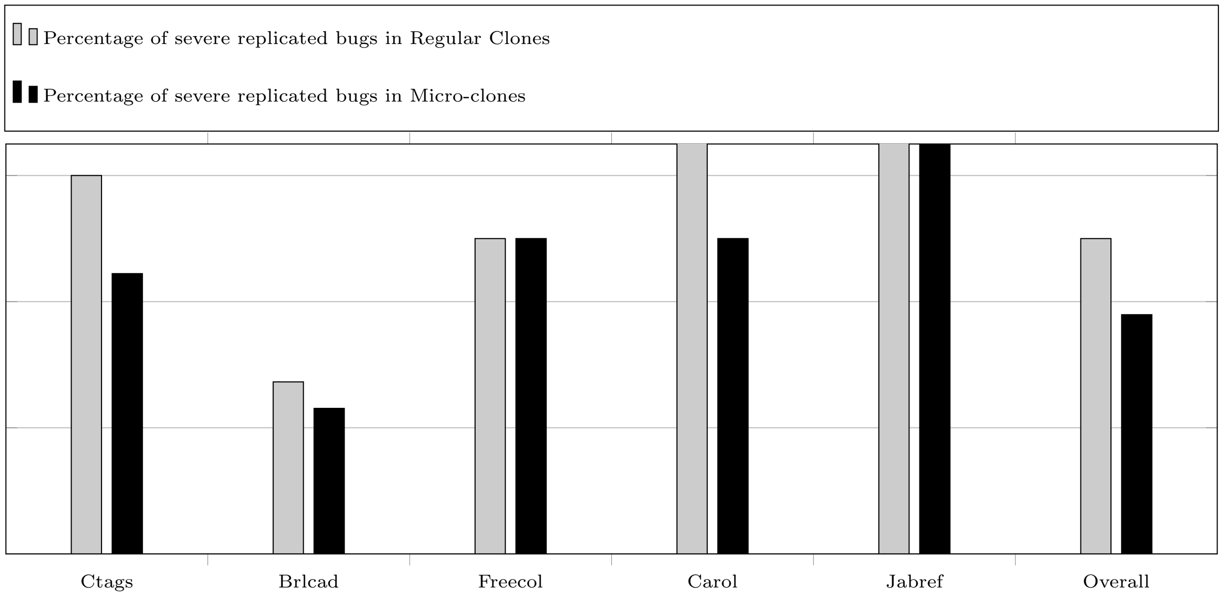


Figure 7.10: Percentage of severe replicated bugs in regular and micro-clones for NiCad clone detection tool.

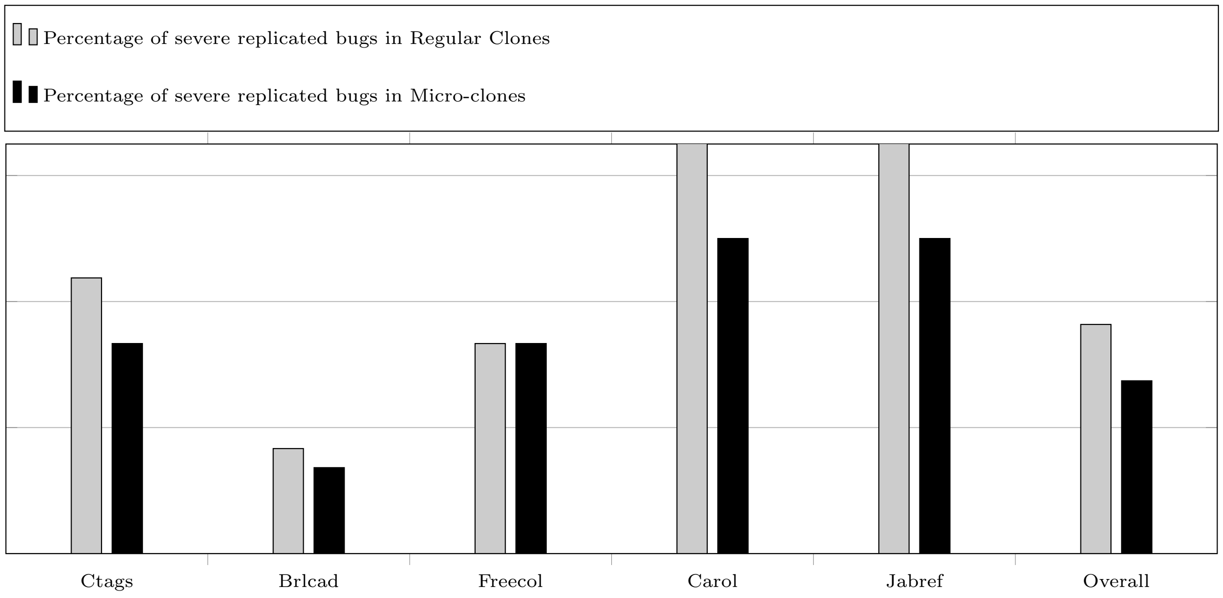


Figure 7.11: Percentage of severe replicated bugs in regular and micro-clones for ConQat clone detection tool.

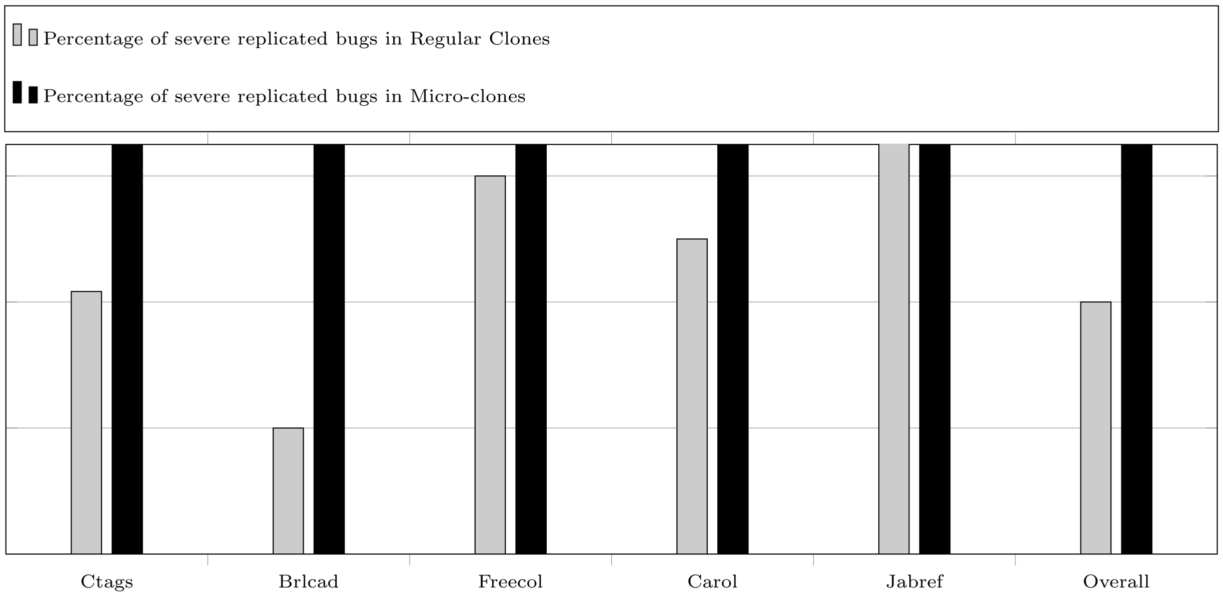


Figure 7.12: Percentage of severe replicated bugs in regular and micro-clones for iClones clone detection tool.

Table 7.13: Number of Severe Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for NiCad clone detection tool

Subject	**Regular Clones**		**Micro Clones**	
Systems	**NBRC**	**NSBRC**	**NBRC**	**NSBRC**
Ctags	15	9	9	4
Brlcad	11	3	13	3
Freecol	4	2	4	2
Carol	1	1	2	1
Jabref	1	1	1	1

Table 7.14: Number of Severe Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for ConQat clone detection tool

Subject	**Regular Clones**		**Micro Clones**	
Systems	**NBRC**	**NSBRC**	**NBRC**	**NSBRC**
Ctags	16	7	33	11
Brlcad	12	2	22	3
Freecol	3	1	3	1
Carol	1	1	2	1
Jabref	1	1	2	1

Table 7.15: Number of Severe Replicated Bugs that Experienced by code clones in Regular and Micro-Clones for iClones clone detection tool

Subject	**Regular Clones**		**Micro Clones**	
Systems	**NBRC**	**NSBRC**	**NBRC**	**NSBRC**
Ctags	12	5	1	1
Brlcad	10	2	1	1
Freecol	5	3	1	1
Carol	2	1	1	1
Jabref	1	1	1	1

and 27.42% respectively.

The difference between the percentage of severe replicated bugs in regular and micro-clones is not significant for all three clone detection tools. Therefore, we can state that both regular and micro-clones should be taken care of during clone management from the bug severity perspective.

7.3.5 Fifth research question (RQ 5): What is the percentage of line coverage of replicated clone fragments in both regular and micro-clones?

In this extended study from previous chapter, we add our fifth research question. The procedure of our experiment to answer this research question is as follows.

Motivation. While comparing with three clone detection tools, we found that the number of clones detected by each tool varies a lot, especially for micro-clones. Since these tools i.e. NiCad, ConQat, and iClones are designed for regular clones ignoring the micro-clones, this result is not surprising. We think that this would be unfair if we compare these three tools disregarding the number of lines in micro-clones detected by each tool. To be fair, we calculate the line coverage i.e. percentage of lines of code of replicated clone fragments for each clone detection tool compared to non-replicated ones. Without answering this research question, we can not understand the actual scenario of replicated bugs in regular and micro-clones.

Methodology. To implement this research question, we first calculate the total number of lines of code (LOCs) for all buggy clone fragments for all revisions. We then calculate the total number of lines of code (LOCs) for all replicated buggy clone fragments for all revisions of a subject system. We calculate these lines of code for both regular and micro-clones. We analyze the following two measures for answering this research question.

LOCCF (Total lines of code of all clone fragments): Previously, we measured the total number of clone fragments in the eligible clone classes in section 7.3.2. Here, we calculate the total number of clone fragments for all clone classes which take part in bug-fixing commit operations. Then we calculate the total number of lines of code for all of these buggy clone fragments.

LOCCFRB (Total lines of code of all replicated clone fragments): We calculated the total number of clone fragments that encountered similarity preserving co-change (SPCO), i.e. clone fragments which contain replicated bugs (same as illustrated in Section 7.3.1). Then we calculate the total number of lines of code for all of these replicated clone fragments.

From these two measures, we calculate the percentage of line coverage (PLC) for both regular and micro-clones. We use the following equation to find the value of PLC for all revisions of a subject system.

$$PLC = \frac{100 \times \sum_{all} LOCCFRB}{\sum_{all} LOCCF} \tag{7.1}$$

To find the overall percentage of line coverage for all subject systems, we use the following equation for both regular and micro-clones.

$$OPLC = \frac{100 \times \sum_{all\ systems} LOCCFRB}{\sum_{all\ systems} LOCCF} \tag{7.2}$$

In Equation 7.2, OPLC stands for the overall percentage of line coverage of replicated clone fragments with respect to all buggy clone fragments. We calculate OPLC for both regular and micro-clones.

Table 7.16, 7.17, 7.18 show the total number of lines of code for replicated and non-replicated clone fragments for both regular and micro-clones for NiCad, ConQat, and iClones clone detection tools respectively. From RQ 2 for NiCad and ConQat, we know that there are no replicated clone fragments for Carol and Jabref. Thus, in Table 7.16 and 7.17 we see that the total lines of code for replicated clone fragments are zero for these two subject systems. From Table 7.16 for NiCad, we see that for all subject systems the total number of lines of code for replicated clone fragments is higher in micro-clones than regular code clones except for Ctags and Brlcad. For ConQat from Table 7.17, the total number of lines of code of replicated clone fragments is higher in micro-clones than in regular clones for all subject systems. On the other hand from RQ 2 for iClones, we know that there are only two replicated clone fragments found in Freecol for micro-clones. Also, there are no replicated clone fragments in Jabref for regular clones. Thus from Table 7.18, we see that in these cases there are no lines of code for replicated clone fragments either. For all subject systems in iClones, the total number of lines of code of clone fragments that contain replicated bugs is higher in regular code clones than in micro-clones. Figure 7.13, 7.14, and 7.15 show the percentage of line coverage in both regular and micro-clones for NiCad, ConQat, and iClones respectively. From figure 7.13 for NiCad, we observe that the percentage of line coverage of replicated clone fragments is higher in micro-clones than regular clones except for Ctags. Also, from figure 7.14 for ConQat, we see that the percentage of line coverage of replicated clone fragments is higher in micro-clones than in regular clones for all five subject systems. On the contrary in figure 7.15 for iClones, we observe that for all subject systems the percentage of line coverage of replicated clone fragments is higher in regular code clones than in micro-clones except for Jabref where both percentages are zero. To better understand the differences of these percentages in between regular and micro-clones, we perform the MWW test for each clone detection tool's result.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 5. To understand the significance of the difference between percentages of line coverage of regular and micro-clones, we conduct the MWW test [9, 10]. We found that for 5% significance level for NiCad the p-value is 0.29834, U-value is 7, and the critical U-value is 2. For ConQat, p-value is 0.09492, U-value is 4, and critical U-value is 2. On the other hand for iClones, the p-value is 0.0601, U-value is 3, and the critical U-value is 2. For all the three code clone detection tools, U-value is greater than the critical U-value. Hence, none of the differences in percentages of replicated clone fragments between regular clones and micro-clones is significant.

Since the difference of the percentages in between regular and micro-clones do not differ significantly, we can say that both micro-clones and regular clones are equally important for software maintenance tasks.

Table 7.16: Total Number of Lines of Code for NiCad clone detection tool

Subject	**Regular Clones**		**Micro Clones**	
Systems	**CFRB***	**CF***	**CFRB***	**CF***
Ctags	119135	2289959	21316	889023
Brlcad	139366	4300934	103666	2622517
Freecol	80352	2108656	164026	2152926
Carol	0	34591	10679	42084
Jabref	0	327064	4698	208756
*LOC = Total Lines of Code				

Table 7.17: Total Number of Lines of Code for ConQat clone detection tool

Subject	**Regular Clones**		**Micro Clones**	
Systems	**CFRB***	**CF***	**CFRB***	**CF***
Ctags	437894	5177512	2925129	17036371
Brlcad	1363688	12733290	2317473	19098646
Freecol	202006	3030064	563836	6576597
Carol	0	17116	9148	47747
Jabref	0	480701	46320	705773
*LOC = Total Lines of Code				

Table 7.19 shows all the results of MWW tests for NiCad, ConQat, and iClones for four research questions. Here, we see that for all RQs U-values are greater than corresponding critical U-values.

Answer to RQ 5. The percentage of line coverage of replicated clone fragments is higher in micro-clones than regular code clones for both NiCad (by 1.41%) and ConQat (by 4.14%) clone detection tools.

We perform manual analysis to inspect our implementation is correct for all research questions. We investigated at least the first 50 clone fragments and/or revisions manually to confirm our analysis for all subject systems.

Table 7.18: Total Number of Lines of Code for iClones clone detection tool

Subject	**Regular Clones**		**Micro Clones**	
Systems	**CFRB***	**CF***	**CFRB***	**CF***
Ctags	62554	1944358	0	293188
Brlcad	244275	6140060	0	611781
Freecol	256310	2485730	2188	229463
Carol	37878	254505	0	6596
Jabref	0	386711	0	17615
*LOC = Total Lines of Code				

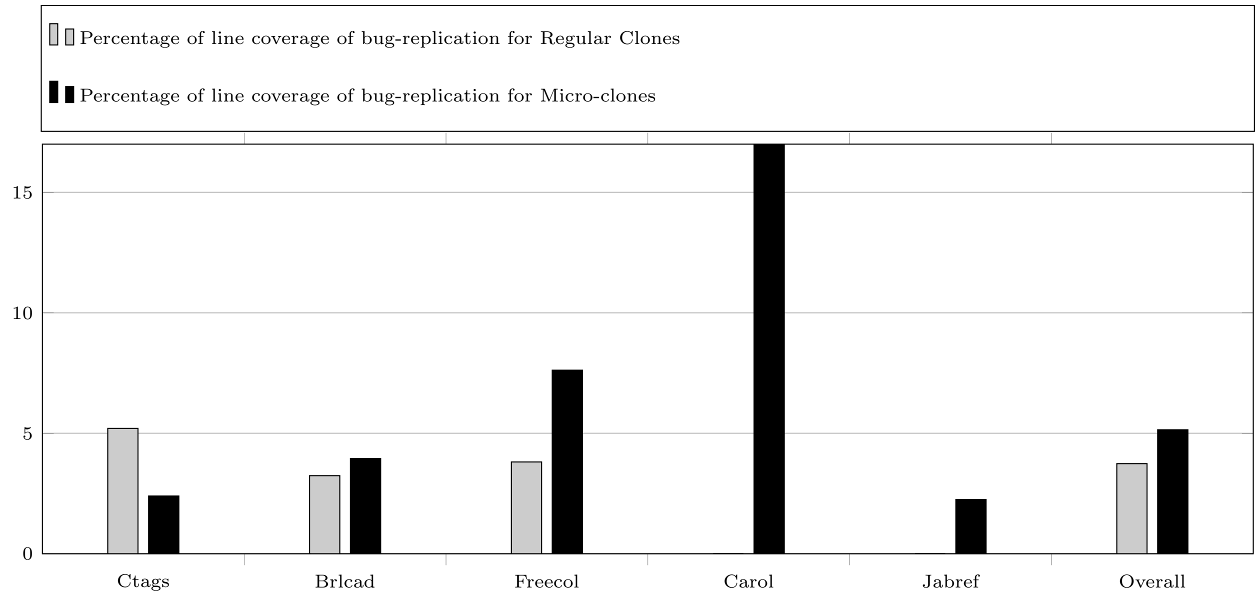


Figure 7.13: Percentage of line coverage of replicated bugs compared to non-replicated bugs in code clones for regular and micro-clones for NiCad clone detection tool.

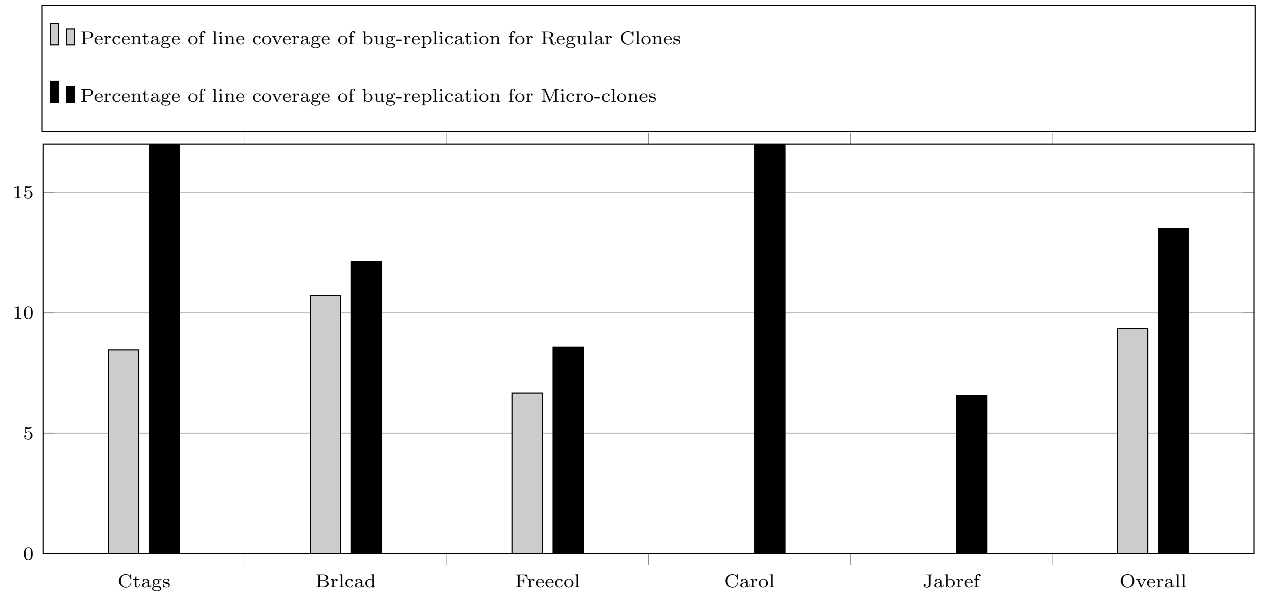


Figure 7.14: Percentage of line coverage of replicated bugs compared to non-replicated bugs in code clones for regular and micro-clones for ConQat clone detection tool.

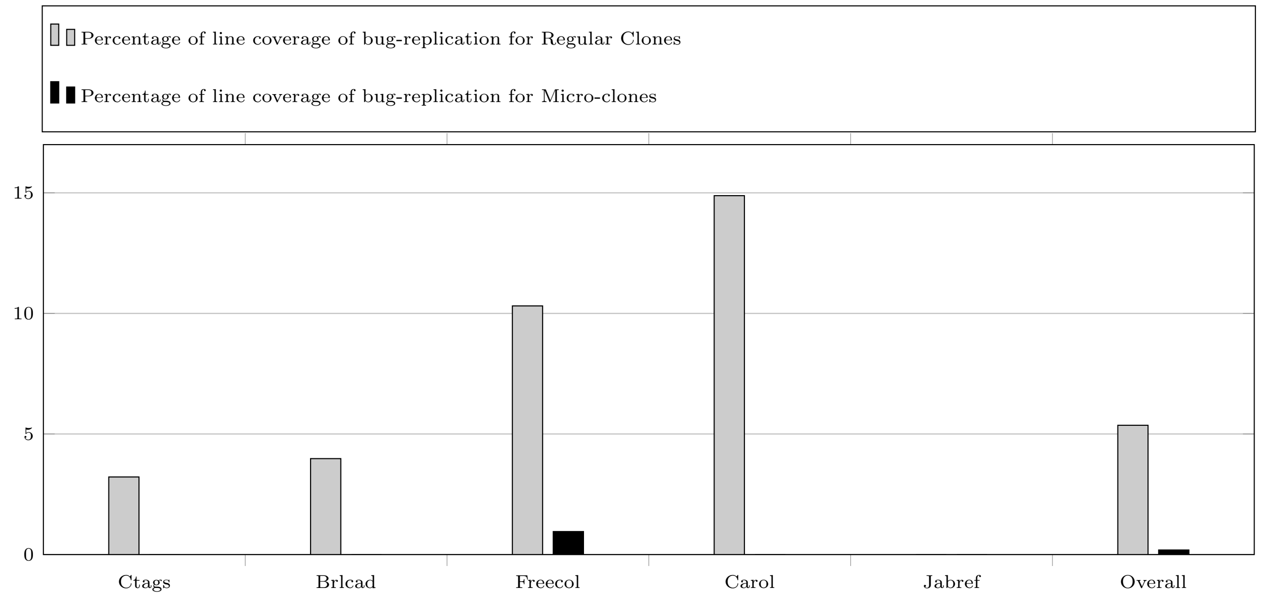


Figure 7.15: Percentage of line coverage of replicated bugs compared to non-replicated bugs in code clones for regular and micro-clones for iClones clone detection tool.

Table 7.19: Mann-Whitney-Wilcoxon Test Result for RQ1, RQ3, RQ4 and RQ5 for NiCad, ConQat, and iClones clone detection tools

Tools	**Research Question No.**	**p-value**	**U-value**	**Critical Value of U**
NiCad	RQ1	0.1443	5	2
	RQ3	0.34722	7.5	2
	RQ4	0.4009	8	2
	RQ5	0.29834	7	2
ConQat	RQ1	0.0601	3	2
	RQ3	0.0477	2.5	2
	RQ4	0.5287	9	2
	RQ5	0.09492	4	2
iClones	RQ1	0.0601	3	2
	RQ3	0.67448	10	2
	RQ4	0.0477	2.5	2
	RQ5	0.0601	3	2
Considering level of significance is 5%. For all RQs, U-value > Critical value of U				

7.4 Related Work

Our research in this paper is a significant extension of our previous study [75]. In our previous study [75] on bug-replication in regular and micro-clones we used NiCad clone detection tool. However, we wanted to know if the findings of our previous study [75] are affected by the result of clone detection tool NiCad. To this extent we use two more clone detection tools i.e. ConQat and iClones in this extended study. We also answer one more additional research question (RQ 5) in this journal paper. Our fifth research question finds the percentage of line coverage of replicated clone fragments compared to non-replicated one. This is necessary because otherwise we can not compare the result in between three clone detection tools. We found that different clone detection tool finds different number of micro-clones. So to be fair in measurements, we conduct the fifth research question. We did not perform these additional tasks in our previous study [75]. From our extended study we found that NiCad and ConQat have the similar results whereas iClones shows the opposite. We conduct manual investigation on this regard and found that iClones detects extremely low number of micro-clones which is opposite compared to NiCad and ConQat. However, since the statistical significance test shows that the differences between regular and micro-clones are insignificant for all three clone detection tools, we can say that micro-clones are equally important as of the regular code clones from clone management perspective.

7.5 Threats to Validity

We used the NiCad [49], ConQat [87], and iClones [59] clone detector for detecting both micro and regular clones. According to Wang et al. [197], all clone detection tools suffer from the *confounding configuration choice problem* and might give different results for different tools' settings. Despite this constraint, the settings used for three clone detection tools for our experiment are considered standard [153, 175, 141]. NiCad, ConQat, and iClones can detect clones with high precision and recall [156, 155, 175] with these settings. Therefore, we believe that our findings on the bug-replication of micro-clones and regular clones have significant importance.

Our research requires the detection of bug-fix commits. Our detection technique is similar to the technique proposed by Mocus and Votta [123] and also used by Barbour et al. [36]. The technique proposed by Mocus and Votta [123] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [36] showed that the probability of selecting a non-bug-fix commit as a bug-fix commit is very low. According to their inspection, the technique is 87% accuracy in detecting bug-fix commits.

To generalize our findings concerning the comparative bug-replication of micro and regular clones, the total number of subject systems is not adequate in our research. However, our subject systems were of diverse variety in terms of application domains, sizes and revisions. Therefore, we believe that our findings are important from the perspectives of managing code clones.

7.6 Conclusion

This chapter investigates and compares the aspects of bug-replication between regular code clones and micro-clones. After investigating five diverse subject systems using three state-of-the-art clone detection tools, we found that micro-clones are equally important as regular clones. We have observed that the percentage of clone fragments that are related to bug-replication is higher in micro-clones in some cases than that of regular code clones for both NiCad and ConQat. Moreover, the percentage of bug-replication in micro-clones is almost the same as the percentage in regular clones for NiCad and ConQat. Additionally, both regular code clones and micro-clones have similar tendencies of replicating severe bugs in NiCad and ConQat. Furthermore, the percentage of line coverage of replicated clone fragments is higher in micro-clones than in regular clones for NiCad and ConQat clone detection tools. We believe that from the clone management perspective, these findings are important. Both regular clones and micro-clones should be considered for software evaluation and maintenance tasks. Since only a limited number of research has been performed on micro-clones, more research is needed to elaborate micro-clones more profoundly. In the future, we would like to explore micro-clones for more systems of various programming languages to perform a language centric empirical study on the bug-proneness of micro-clones. Also, since the current clone detection tools are not designed for detecting micro-clones, a clone detection tool explicitly designed for micro-clones is required for further study on micro-clones.

8 Conclusion

Reusing a code fragment through copy/pasting, also known as code cloning, is a common practice during software development and maintenance. Code clones are exactly or nearly similar code fragments in the code-base of a software system. Code cloning has both positive and negative impacts in the software systems. Some positive impacts are faster development, cost reduction, program comprehension etc. Negative impacts are hidden bug propagation, unintentional inconsistent changes, high instability etc. We need to examine the trade off between the positive and negative impacts of clone code so that we can get both the benefits of code cloning and remove the harmful clones which produce bugs. Through our four studies, we have discovered that not all clones are bad. However, we had to filter out the bad clones and then we suggested some improvement based on different scenarios that how we can reduce these negative impacts of code clones. In this chapter, we summarized our findings of all studies i.e. Chapter 3-7. This chapter is organized as follows. Section 8.1 summarizes this thesis and describes our major contributions of this research, Section 8.2 elaborates some threats to validity, and then Section 8.3 concludes this chapter by directing some future research.

8.1 Summary

Throughout four major studies we intend to investigate how we can help the developer to improve the software systems by maintaining bug free code clones. To reach this goal, we first compared bug-proneness between clone and non-clone code and found that clone code are more bug-prone than non-clone code. Thus, we motivated to explore more on clone code and performed our second study. We compared bug-proneness between micro-clones and regular clones. From our second study, we found that micro-clones are equally important like the regular clones. Since our first study reveals that clone code are more bug-prone than non-clone code, we were interested to know how these bugs replicate in code clones through software evolution. Our third study discovers that bug-replication occurs more in Type 3 code clones compared to Type 1 and Type 2 code clones. This study motivates us to explore bug-replication in micro-clones. We found that bug replicates in micro-clones as equally as in regular code clones and thus it is important to emphasize on micro-clones which were ignored in the past research.

We have stated our research motivation along with the research problem and a brief description of each of our studies in the first chapter i.e. Chapter 1. A background knowledge has been provided in Chapter 2. We have contrasted clone and non-clone code by analysing bug-proneness in Chapter 3. In Chapter

4, we described our comparative research on bug-proneness of micro-clones and regular code clones. We analyzed bug-replication between different types of code clones in Chapter 5. We compared bug-replication between micro-clones and regular code clones in Chapter 6. We performed an empirical study to compare bug-replication between regular and micro-clones using three clone detection tools in Chapter 7.

In this section, I have summarized my major contributions to the research during my PhD. In total I have published one short paper, one journal paper, four full conference papers and one journal paper is under review. These are described shortly below.

- **Comparing software bugs in between clone and non-clone code.** We have identified and compared software bugs in clone and non-clone code. After detailed investigation we have found that clone code is more bug-prone than non-clone code. We have published our work in SEKE 2017 [76]. Later, we have enhanced our investigation in this direction and published it in a well reputed journal IJSEKE 2017 [74].

- **Comparing software bugs in between micro and regular code clones.** We worked on micro-clones and identified software bugs in both micro and regular code clones. We found that micro-clones are more bug-prone than regular code clones. We have published our work in a leading international conference SANER 2019 [73].

- **Finding and analyzing replicated bugs in different types of clones.** We identified and observed bug-replication in code clones. We investigated replicated bugs in three types of clones i.e. Type 1, Type 2, and Type 3 and compared among them. We found that replication of bugs through cloning is a common phenomenon. We have published our work in a prominent international software engineering conference SANER 2016 [72].

- **Comparing bug-replication in between regular and micro-clones.** We identified replicated bugs in both regular and micro-clones and compared between them. We found that micro-clones have almost the same contribution as the regular code clones in bug-replication. We have published our work in a preeminent international conference ICPC 2019 [75]. We extend this study by using two more clone detection tools and adding one more research question. We also used revised data set for all three clone detection tools. The significant extension of this research has been submitted to journal JSS 2020 [77].

Besides these major publications, I have another short paper on big data analytic [172]. This paper [172] contributes to build a clone benchmark, BigCloneBench [177] which is used to evaluate clone detection tools for large data sets.

8.2 Threats to Validity

In Section 2.6, we discussed some background research on software bugs. We shortly described *performance bugs* and *concurrency bugs*. We did not consider such type of bugs in our research. We described a few state-of-the-art software bug detection tools. We did not use any of the bug detection tools in our research. Using such tools might vary the result of our research. However, we used bug-fixing commit messages which were reported by the developers in real time. Thus, our approach does not depend on the report of any bug detection tool.

In Section 2.7, we also discussed about different software testing tools and techniques. In our research, we detected bug-fix commits. We did not use any software testing tools to detect the bugs in our research. Using software testing tools for bug detection in code clone might vary the result of our research. However for detecting bug-fix commit messages, we followed the similar technique that proposed by Mockus and Votta [123]. Barbour et al. [36] used the same procedure and stated that this method has 87% of accuracy.

8.3 Future Research Direction

In our research we have analyzed rigorously on finding which type of code or which type of clones are more likely to introduce software bugs in the system. To extend our research in future we would like to propose mechanisms for fixing these bugs without deteriorating the software quality. Fixing a bug especially clone bug is crucial for every system since it influences the entire code base. Also, doing such manually is a huge task and almost impossible. So, if it is possible to make the process automatic then it would leverage human labor and will be error free. Some plausible future research are discussed in the following sub-sections.

8.3.1 Buggy Micro-clones

Research on micro-clones are still another naive avenue in clone research. Exploring micro-clones could lead us to new solution of buggy code clones. Finding bugs in micro-clones and perform an analytical study is required to understand the characteristics of buggy micro-clones. Very few research has been conducted to this direction.

8.3.2 Context aware bugs in clones

Programmer sometimes copy and paste a code fragment just to make new changes to the copied code fragment. There are two possible cases in this circumstance. Either they copy a non-buggy code fragment or copy a buggy code fragment and then paste it. For both cases it is necessary to make changes (non-buggy code fragment) or fix bugs (buggy code fragment) to adapt the context of the code base. Failing to change appropriately might lead to introducing bug (non-buggy code fragment) or preserving the bug (buggy code fragment). We can define these bugs as context aware bugs. In our research we did not investigate these context aware bugs and thus it requires exploration.

8.3.3 Increasing the number of subject systems

Most of our studies have been conducted on six or seven subject systems. The number of subject systems may not be sufficient despite of the fact that our subject systems are of diverse variety in terms of size (LOC), application domains, and revision history lengths (number of revisions). Investigating large number of subject systems or in large scale might show some new insights on buggy code clones. The only plausible limitation might be that for micro-clones it would be a huge number of clones and thus complicated and time consuming experiment would have to be executed.

8.3.4 Considering Type 4 code clones

In any of our studies, we did not consider Type 4 clone code or semantic code clones. This is because there is very limited resource for technical support to investigate Type 4 clones. The availability of Type 4 clone detection tool would make it easier to conduct more future research on this direction.

8.3.5 Automatic tool support for recommendation

In all of our studies, we have recommended programmers about which type of code they should emphasize during clone management (e.g. clone tracking) and maintenance tasks (e.g. refactoring). This would be easy if there is an automatic tool support like an IDE plugin while they write code. An automatic tool can reduce future bugs by predicting high risk bugs in code clones.

8.3.6 Clone detection tool for micro-clones

While investigating bug-replication between regular and micro-clones using three clone detection tools, we realized that current clone detection tools have limited support for micro-clones. This is obvious since these clone detection tools are designed for detecting regular code clones. Thus, a specialized clone detection tool that can detect micro-clones is required. In future we can design such clone detection tool for micro-clones.

8.3.7 Bug tracking

Bug tracking is a very renowned area of study. However, tracking bugs in case of micro-clones and comparing with regular clones would be challenging as a future research. No other studies have explored this avenue and thus, we have scope for new research.

8.3.8 ML in buggy clone research

Using machine learning algorithm in this research area would be interesting. Very few research have been conducted to explore software bugs in clone code using ML. I believe that ML can solve complex problems like handling semantic or Type 4 clones. If a semantic clone data set can be trained as human decisions, it would be useful for test data set. A substantial improvement could be possible if more research is conducted in this direction.

References

[1] Bugzilla. URL: `https://www.bugzilla.org`.

[2] CTAGS. URL: `http://ctags.sourceforge.net/`.

[3] Difference checker. URL: `https://www.diffchecker.com/`.

[4] Eclipse bug report. URL: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=76138`.

[5] FindBugs. URL: `http://findbugs.sourceforge.net/`.

[6] Google. Android Monkey. URL: `https://developer.android.com/studio/test/monkey`.

[7] JIRA. URL: `https://www.atlassian.com/software/jira`.

[8] JLint. URL: `http://jlint.sourceforge.net/`.

[9] Mann-Whitney U Test. URL: `https://en.wikipedia.org/wiki/Mann-Whitney_U_test`.

[10] Mann-Whitney U Test. URL: `http://www.socscistatistics.com/tests/mannwhitney/Default2.aspx`.

[11] Mann-Whitney-Wilcoxon Test Online. URL: `http://scistatcalc.blogspot.ca/2013/10/mann-whitney-u-test-calculator.html`.

[12] Mantis. URL: `https://www.mantisbt.org`.

[13] PMD. URL: `https://pmd.github.io/`.

[14] PVS-Studio Analyzer. URL: `https://www.viva64.com/en/pvs-studio/`.

[15] SourceMeter: Static source code analysis solution for Java, C/C++, C#, Python and RPG. URL: `https://www.sourcemeter.com`.

[16] SVN repository. URL: `http://sourceforge.net/`.

[17] 1012a-1998 - IEEE standard for software verification and validation - content map to IEEE 12207.1. 1998.

[18] 1012-2012 - IEEE standard for system and software verification and validation. 2012.

[19] Boeing eyes lion air crash software upgrade in 6 to 8 weeks. https://goo.gl/Xy1qFa, 2018.

[20] The 737Max and why software engineers might want to pay attention, 2019. URL: `https://bit.ly/2CmeTqB`.

[21] Boeing 737 jets grounded globally as officials investigate technical issues behind fatal crash, 2019. URL: `https://goo.gl/ieBgYN`.

[22] Heres the terrifying reason Boeings 737 MAX 8 is grounded across the globe, 2019. URL: `https://goo.gl/GwXv6H`.

[23] ISO/IEC 29119. Software testing standard. 2008.

[24] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proceedings International Conference on Software Maintenance (ICSM)*, pages 227–236, 2008.

[25] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software (JSS)*, 82(11):1780-1792, 2009.

[26] M. N. Al-Ameen, M. M. Hasan, and A. Hamid. Making findbugs more powerful. In *Proceedings 2nd International Conference on Software Engineering and Service Science*, pages 705–708, Beijing, China, July, 2011.

[27] F. Al-omari, C. K. Roy, and T. Chen. SemanticCloneBench: A semantic code clone benchmark using crowd-source knowledge. In *Proceedings 14th International Workshop on Software Clones (IWSC)*, pages 57–63, 2020.

[28] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson. Models are code too: Near-miss clone detection for Simulink models. In *Proceedings 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 295–304, 2012.

[29] J. E. M. Araújo, S. Souza, and M. T. Valente. Study on the relevance of the warnings reported by Java bug-finding tools. In *Proceedings The Institution of Engineering and Technology (IET) Software*, pages 5(4):366–374, 2011.

[30] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *Proceedings Australian Software Engineering Conference (ASWEC)*, pages 68–75, 2001.

[31] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *Proceedings 9th International Symposium on Software Testing and Analysis (ISSTA)*, pages 49–60, 2010.

[32] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 265–274, 2010.

[33] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *Proceedings 11th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 81–90, 2007.

[34] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 1–8, San Diego, California, USA, June, 2007.

[35] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proceedings International Conference on Software Maintenance (ICSM)*, pages 273–282, 2011.

[36] L. Barbour, F. Khomh, and Y. Zou. An empirical study of faults in late propagation clone genealogies. *Journal of Software: Evolution and Process (JSEP)*, 25(11):1139-1165, 2013.

[37] I. D. Baxter, M. Conradt, J. R. Cordy, and R. Koschke. Software clone management towards industrial application (Dagstuhl seminar 12071). *Dagstuhl Reports*, 2(2):21–57, 2012.

[38] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *Proceedings 26th International Conference on Software Engineering (ICSE)*, pages 625–634, 2004.

[39] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman. On the dichotomy of debugging behavior among programmers. In *Proceedings of The 40th International Conference on Software Engineering (ICSE Companion)*, pages 572–583, Gothenburg, Sweden, 2018.

[40] M. Beller, A. Zaidman, and A. Karpov. The last line effect. In *Proceedings of the 23rd International Conference on Program Comprehension (ICPC)*, pages 240–243, Florence, Italy, 2015.

[41] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[42] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller. Poster: How developers debug software – the DBGBENCH dataset. In *Proceedings of The 39th International Conference on Software Engineering (ICSE Companion)*, pages 244–246, Buenos Aires, Argentina, May, 2017.

[43] P. A. Brooks and A. M. Memon. Automated GUI testing guided by usage profile. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 333–342, Atlanta, Georgia, USA, November, 2007.

[44] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings 2nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 36–43, 2002.

[45] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, and N. A. Kraft. Measuring the efficacy of code clone information in a bug localization task: An empirical study. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 20–29, 2011.

[46] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings 27th International Conference on Software Engineering (ICSE)*, pages 342–351, 2005.

[47] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. P. Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings 22nd International Conference on Software Engineering (ICSE)*, pages 439–448, 2004.

[48] J. R. Cordy, T. R. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *Proceedings 14th Annual International Conference on Computer Science and Software Engineering (CASCON)*, pages 1–12, 2004.

[49] J. R. Cordy and C. K. Roy. The nicad clone detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension ICPC Tool Demo*, pages 219–220, Washington, DC, USA, 2011.

[50] G. C. B. Costa, R. Braga, J. M. N. David, and F. Campos. A scientific software product line for the bioinformatics domain. *Journal of Biomedical Informatics - Elsevier (JBI)*, 56: 239-264, 2015.

[51] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 422–431, San Francisco, CA, USA, May, 2013.

[52] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering (ESE)*, 13(6): 601-643, 2008.

[53] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings ACM SIGPLAN, conference on Programming language design and implementation (PLDI)*, pages 234–245, 2002.

[54] TMMi Foundation. Test maturity model integration (TMMi) reference model. Version 2.0, 2009.

[55] J. Gao, H. Liu, Y. Li, C. Liu, Z. Yang, Q. Li, Z. Guan, and Z. Chen. Towards automated testing of blockchain-based decentralized applications. In *Proceedings of the 27rd International Conference on Program Comprehension (ICPC)*, Montreal, Canada, 2019.

[56] V. Garousi and M. V. Mäntylä. When and what to automate in software testing? a multi-vocal literature review. *Information and Software Technology (IST)*, 76: 92-117, 2016.

[57] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering (TSE)*, 45(1):34–67, 2019.

[58] N. Göde and J. Harder. Clone stability. In *Proceedings Software Maintenance and Reengineering (CSMR), 15th European Conference on*, pages 65–74, 2011.

[59] N. Göde and R. Koschke. Incremental clone detection. In *Proceedings Software Maintenance and Reengineering (CSMR), 13th European Conference on*, pages 219–228, March 2009.

[60] N. Göde and Rainer Koschke. Frequency and risks of changes to clones. In *Proceedings 33rd International Conference on Software Engineering (ICSE)*, pages 311–320, 2011.

[61] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing. In *Proceedings 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12, 2015.

[62] Y. Higo and S. Kusumoto. Enhancing quality of code clone detection with program dependency graph. In *Proceedings 16th Working Conference on Reverse Engineering (WCRE)*, pages 315–316, 2009.

[63] Y. Higo and S. Kusumoto. Code clone detection on specialized PDGs with heuristics. In *Proceedings 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 75–84, 2011.

[64] Y. Higo, K. Sawa, and S. Kusumoto. Problematic code clones identification using multiple detection results. In *Proceedings 16th Asia-Pacific Software Engineering Conference (APSEC)*, pages 365–372, 2009.

[65] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto. Incremental code clone detection: A PDG-based approach. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 3–12, 2011.

[66] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: An empirical study on open source software. In *Proceedings Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 73–82, 2010.

[67] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. An empirical study on the impact of duplicate code. *Advances in Software Engineering*, 2012: 1-22, 2012.

[68] D. Hovemeyer and W. Pugh. Finding bugs is easy. *Proceedings ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[69] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proceedings 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 9–14, San Diego, California, USA, June, 2007.

[70] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: Incremental, distributed, scalable. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–9, 2010.

[71] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee. Experience of finding inconsistently-changed bugs in code clones of mobile software. In *Proceedings 6th International Workshop on Software Clones (IWSC)*, pages 94–95, 2012.

[72] J. F. Islam, M. Mondal, and C. K. Roy. Bug replication in code clones: An empirical study. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 68–78, 2016.

[73] J. F. Islam, M. Mondal, and C. K. Roy. A comparative study of software bugs in micro-clones and regular code clones. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 73–83, Hangzhou, China, 2019.

[74] J. F. Islam, M. Mondal, C. K. Roy, and K. A. Schneider. Comparing software bugs in clone and non-clone code: An empirical study. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 27(9&10):1507-1527, 2017.

[75] J. F. Islam, M. Mondal, C. K. Roy, and K. A. Schneider. Comparing bug replication in regular and micro code clones. In *Proceedings of the 27rd International Conference on Program Comprehension (ICPC)*, pages 81–92, Montreal, Canada, 2019.

[76] J. F. Islam, M. Mondal, C. K. Roy, and K. A. Schneider. A comparative study of software bugs in clone and non-clone code. In *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 436–443, USA, 2017.

[77] J. F. Islam and C. K. Roy. An empirical study on bug-replication in regular and micro-clones. *Journal of Systems and Software (JSS)*, (submitted), 2020.

[78] M. R. Islam and M. F. Zibran. A comparative study on vulnerabilities in categories of clones and non-cloned code. In *Proceedings 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 8–14, 2016.

[79] M. R. Islam and M. F. Zibran. On the characteristics of buggy code clones: A code quality perspective. In *Proceedings of the 12th International Workshop on Software Clones (IWSC)*, pages 23–29, Campobasso, Italy, 2018.

[80] M. R. Islam, M. F. Zibran, and A. Nagpal. Security vulnerabilities in categories of clones and non-cloned code: An empirical study. In *Proceedings ACM/IEEE International Symposium on Empirical Software Engineering and Measurement Security (ESEM)*, pages 20–29, 2017.

[81] International Software Testing Qualifications Board (ISTQB). Standard glossary of terms used in software testing. version 2.0, December 2007.

[82] J. Krinke. Is cloned code more stable than non-cloned code? In *Proceedings 8th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 57–66, 2008.

[83] L. Jiang, G. Misherghi, and S. Glondu Z. Su. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings 29th International Conference on Software Engineering (ICSE)*, pages 96–105, 2007.

[84] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE)*, pages 55–64, 2007.

[85] S. Jindal and A. Saha. Efficiency of bug finding tools for refactoring prediction. In *Proceedings 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*, pages 1–6, 2015.

[86] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 273–282, 2005.

[87] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Clonedetective - a workbench for clone detection research. In *Proceedings 31st International Conference on Software Engineering (ICSE)*, pages 603–606, 2009.

[88] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings 31st International Conference on Software Engineering (ICSE)*, pages 485–495, 2009.

[89] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[90] U. Kanewala and J. M. Bieman. Testing scientific software: A systematic literature review. *Information and Software Technology (IST)*, 56(10): 1219-1232, 2014.

[91] C. Kapser and M. W. Godfrey. “Cloning considered harmful” considered harmful: patterns of cloning in software. *Empirical Software Engineering (ESE)*, 13(6): 645-692, 2008.

[92] J. Kasurinen. Elaborating software test processes and strategies. In *Proceedings 3rd International Conference on Software Testing, Verification and Validation (ICST)*, pages 355–358, 2010.

[93] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and David Lo. Understanding the test automation culture of app developers. In *Proceedings 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[94] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings 14th Working Conference on Reverse Engineering (WCRE)*, pages 170–178, 2007.

[95] J. Krinke. Is cloned code older than non-cloned code? In *Proceedings 5th International Workshop on Software Clones (IWSC)*, pages 28–33, 2011.

[96] D. E. Krutz and W. Le. A code clone oracle. In *Proceedings 11TH Working Conference on Mining Software Repositories (MSR)*, pages 388–391, 2014.

[97] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *Proceedings 7th Working Conference on Mining Software Repositories (MSR)*, pages 1–10, 2010.

[98] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *Proceedings 7th Working Conference on Mining Software Repositories (MSR)*, pages 1–10, 2010.

[99] T. Lavoie and E. Merlo. Automated type-3 clone oracle using Levenshtein metric. In *Proceedings 5th International Workshop on Software Clones (IWSC)*, pages 34–40, 2011.

[100] T. D. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC-FSE)*, pages 579–590, Bergamo, Italy, 2015.

[101] T. D. B. Le, S. Wang, and D. Lo. Multi-abstraction concern localization. In *Proceedings IEEE International Conference on Software Maintenance (ICSM)*, pages 364–367, 2013.

[102] J. Li and M. D. Ernst. CBCD: Cloned buggy code detector. In *Proceedings 34th International Conference on Software Engineering (ICSE)*, pages 310–320, 2012.

[103] X. Li, S. Zhu, M. dAmorim, and A. Orso. Enlightened debugging. In *Proceedings of The 40th International Conference on Software Engineering (ICSE)*, pages 82–92, Gothenburg, Sweden, 2018.

[104] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings 6th conference on Symposium on Operating Systems Design & Implementation (OSDI)*, pages 289–302, 2004.

[105] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering (TSE)*, 32(3):176–192, March 2006.

[106] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 15–26, June 2005.

[107] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 286–295, 2005.

[108] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *Proceedings 29th International Conference on Software Engineering (ICSE)*, pages 106–115, 2007.

[109] A. Lozano and M. Wermelinger. Tracking clones' imprint. In *Proceedings 4th International Workshop on Software Clones (IWSC)*, pages 65–72, 2010.

[110] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. In *Proceedings 4th International Workshop on Mining Software Repositories (MSR)*, pages 18–21, 2007.

[111] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 103–116, October 2007.

[112] L. Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *Proceedings 26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[113] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172219, 2014.

[114] L. Lucia, D. Lo, and X. Xia. Fusion fault localizers. In *Proceedings 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 127–138, 2014.

[115] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information & Software Technology*, 52(9):972-990, 2010.

[116] M. Mondal and C. K. Roy and K. A. Schneider. Connectivity of co-changed method groups: A case study on open source systems. In *Proceedings 22nd Annual International Conference on Computer Science and Software Engineering (CASCON)*, pages 205–219, 2012.

[117] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings 15th International Symposium on Software Testing and Analysis (ISSTA)*, pages 94–105, 2016.

[118] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *Proceedings 5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 81–90, 2012.

[119] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Automatic testing of GUI-based applications. *Software Testing, Verification and Reliability (STVR)*, 24(5): 341-366, 2014.

[120] L. Mariani, M. Pezzè, and D. Zuddas. Augusto: Exploiting popular functionalities for the generation of semantic GUI tests with oracles. In *Proceedings 40th International Conference on Software Engineering (ICSE)*, pages 280–290, 2018.

[121] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability (STVR)*, 14: 105-156, 2004.

[122] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering (TSE)*, 27(2):144–155, 2001.

[123] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings International Conference on Software Maintenance (ICSM)*, pages 120–130, 2000.

[124] M. Mondal, Banani Roy, C. K. Roy, and K. A. Schneider. Investigating context adaptation bugs in code clones. In *Proceedings of the 35th International Conference on Software Maintenance and Evolution (ICSME)*, pages 157–168, 2019.

[125] M. Mondal, Banani Roy, C. K. Roy, and K. A. Schneider. Investigating near-miss micro-clones in evolving software. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC)*, pages 208–218, July 2020.

[126] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings 27th Annual ACM Symposium on Applied Computing (SAC)*, pages 1227–1234, 2012.

[127] M. Mondal, C. K. Roy, and K. A. Schneider. Dispersion of changes in cloned and non-cloned code. In *Proceedings 2012 6th International Workshop on Software Clones (IWSC)*, pages 29–35, 2012.

[128] M. Mondal, C. K. Roy, and K. A. Schneider. An empirical study on clone stability. In *ACM SIGAPP Applied Computing Review (ACR)*, pages 12(3): 20–36, 2012.

[129] M. Mondal, C. K. Roy, and K. A. Schneider. An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study. In *Science of Computer Programming (SCP)*, pages 95:445–468, 2013.

[130] M. Mondal, C. K. Roy, and K. A. Schneider. Automatic ranking of clones for refactoring through mining association rules. In *Proceedings CSMR-Working Conference on Reverse Engineering (WCRE)*, pages 114–123, 2014.

[131] M. Mondal, C. K. Roy, and K. A. Schneider. A comparative study on the bug-proneness of different types of code clones. In *Proceedings 31st International Conference on Software Maintenance and Evolution(ICSME)*, pages 91–100, 2015.

[132] M. Mondal, C. K. Roy, and K. A. Schneider. Bug-proneness and late propagation tendency of code clones: A comparative study on different clone types. *Journal of Systems and Software (JSS)*, 144:41-59, 2018.

[133] M. Mondal, C. K. Roy, and K. A. Schneider. Late propagation in near-miss clones: An empirical study. In *Proceedings 8th International Workshop on Software Clones (IWSC)*, page 15 pp, Antwerp, Belgium, February 2014.

[134] M. Mondal, C. K. Roy, and K. A. Schneider. Identifying code clones having high possibilities of containing bugs. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC)*, pages 99–109, Buenos Aires-Argentina, May 2017.

[135] M. Mondal, C. K. Roy, and K. A. Schneider. Micro-clones in evolving software. In *Proceedings of the 25th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 50–60, Campobasso, Italy, March, 2018.

[136] M. Mondal, C. K. Roy, and K. A. Schneider. A comparative study on the intensity and harmfulness of late propagation in near-miss code clones. *Software Quality Journal (SQJ)*, 24(4):883-915, December 2016.

[137] M. Mondal, C. K. Roy, and K. A. Schneider. SPCP-Miner: A tool for mining code clones that are important for refactoring or tracking. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 484–488, Montreal, Canada, March, 2015.

[138] M. Mondal, C. K. Roy, and K. A. Schneider. Bug propagation through code cloning: An empirical study. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME)*, pages 227–237, Shanghai, China, September 2017.

[139] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering (ESE)*, 23(5): 2901-2947, 2018.

[140] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Gapped code clone detection with lightweight source code analysis. In *Proceedings 21st International Conference on Program Comprehension (ICPC)*, pages 93–102, 2013.

[141] M. Nadim, M. Mondal, and C. K. Roy. Evaluating performance of clone detection tools in detecting cloned cochange candidates. In *Proceedings 14th International Workshop on Software Clones (IWSC)*, pages 15–21, 2020.

[142] A. Nanthaamornphong and J. C. Carver. Test-driven development in scientific software: a survey. *Software Quality Journal (SQJ)*, 25(2): 343-372, 2017.

[143] B. N. Nguyen, B. Robbins, I. Banerjee, and A. M. Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering (ASE)*, 21(1): 65-105, 2014.

[144] M. Pezzè, P. Rondena, and D. Zuddas. Automatic GUI testing of desktop applications: an empirical assessment of the state of the art. In *Proceedings Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 54–62, Amsterdam, Netherlands, July, 2018.

[145] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings 14th IEEE International Conference on Program Comprehension (ICPC)*, pages 137–146, 2006.

[146] D. Poshyvanyk, Y. Guhneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering (TSE)*, 33(6):420–432, 2007.

[147] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *Proceedings 7th Working Conference on Mining Software Repositories (MSR)*, pages 72–81, 2010.

[148] M. S. Rahman and C. K. Roy. On the relationships between stability and bug-proneness of code clones: An empirical study. In *Proceedings 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 131–140, 2017.

[149] S. Rao and A. C. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings 8th International Working Conference on Mining Software Repositories, (MSR)*, pages 43–52, 2011.

[150] B. Ray, V. Hellendoorn, and S. Godhane. On the "naturalness" of buggy code. In *Proceedings 38th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 428–439, 2016.

[151] C. K. Roy. Detection and analysis of near-miss software clones. In *Proceedings International Conference on Software Maintenance (ICSM)*, pages 447–450, 2009.

[152] C. K. Roy and J. R. Cordy. A survey on software clone detection research. In *Technical Report 2007-541*, page 115, School of Computing, Queen's University, Canada, 2007.

[153] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, pages 172–181, 2008.

[154] C. K. Roy and J. R. Cordy. Towards a mutation-based automatic framework for evaluating code clone detection tools. In *Proceedings Canadian Conference on Computer Science & Software Engineering (C3S2E)*, pages 137–140, 2008.

[155] C. K. Roy and J. R. Cordy. A mutation / injection-based automatic framework for evaluating code clone detection tools. In *Proceedings International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 157–166, April 2009.

[156] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. In *Science of Computer Programming*, pages 74 (2009): 470–495, 2009.

[157] C. K. Roy, M. F. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 18–33, 2014.

[158] N. Ruter, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *Proceedings 15th International Symposium on Software Reliability Engineering (ISSRE)*, pages 245–256, 2004.

[159] F. Van Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proceedings 19th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 336339, 2004.

[160] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry. Are these bugs really "normal"? In *Proceedings 12th Working Conference on Mining Software Repositories (MSR)*, pages 258–268, 2015.

[161] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *Proceedings 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355, 2013.

[162] Z. Sahaf, V. Garousi, D. Pfahl, R. Irving, and Y. Amannejad. When to automate software testing? decision support based on system dynamics: An industrial case study. In *Proceedings International Conference on Software and System Process (ICSSP)*, pages 149–158, 2014.

[163] V. Saini, H. Sajnani, and C. V. Lopes. Comparing quality metrics for cloned and non cloned Java methods: A large scale empirical study. In *Proceedings 32th International Conference on Software Maintenance and Evolution(ICSME)*, pages 256–266, 2016.

[164] H. Sajnani, V. Saini, and C. V. Lopes. A comparative study of bug patterns in Java cloned and non-cloned code. In *Proceedings 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 21–30, 2014.

[165] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of The 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, Austin, TX, USA, May 2016.

[166] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou. Studying the impact of clones on software defects. In *Proceedings 17th Working Conference on Reverse Engineering (WCRE)*, pages 13–21, 2010.

[167] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition? DefUse: Definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings 25th Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 160–174, 2010.

[168] B. Sisman and A. C. Kak. Incorporating version histories in information retrieval based bug localization. In *Proceedings 9th IEEE Working Conference of Mining Software Repositories (MSR)*, pages 50–59, 2012.

[169] K. Solanki and S. Kumari. Comparative study of software clone detection techniques. In *Proceedings The 2016 Management and Innovation Technology International Conference (MITiCON-2016)*, pages 152–156, 2016.

[170] S. Srivastva and S. Dhir. Debugging approaches on various software processing levels. In *Proceedings International conference of Electronics, Communication and Aerospace Technology (ICECA)*, pages 302–306, India, 2017.

[171] D. Steidl and N. Göde. Feature-based detection of bugs in clones. In *Proceedings 7th International Workshop on Software Clones (IWSC)*, pages 76–82, 2013.

[172] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 476–480, 2014.

[173] J. Svajlenko, I. Keivanloo, and C. K. Roy. Big data clone detection using the classical detectors: An exploratory study. *Journal of Software Evolution and Process*, 27(6):430–464, 2015.

[174] J. Svajlenko, I. Keivanloo, and C. K. Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *Proceedings of the ICSE 7th International Workshop on Software Clones (IWSC)*, pages 16–22, San Francisco, CA, May 2013.

[175] J. Svajlenko and C. K. Roy. Evaluating modern clone detection tools. In *Proceedings 30th International Conference on Software Maintenance and Evolution(ICSME)*, pages 321–330, 2014.

[176] J. Svajlenko and C. K. Roy. The mutation and injection framework: Evaluating clone detection tools with mutation analysis. *IEEE Transactions on Software Engineering (TSE)*, 28 pages (in press), 2020.

[177] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *Proceedings 31st International Conference on Software Maintenance and Evolution(ICSME)*, pages 131–140, Bremen, Germany, October 2015.

[178] J. Svajlenko and C. K. Roy. CloneWorks: a fast and flexible large-scale near-miss clone detection tool. In *Proceedings of The 39th International Conference on Software Engineering (ICSE Companion)*, pages 177–179, Buenos Aires, Argentina, May, 2017.

[179] J. Svajlenko and C. K. Roy. Fast and flexible large-scale clone detection with CloneWorks. In *Proceedings of The 39th International Conference on Software Engineering (ICSE Companion)*, pages 27–30, Buenos Aires, Argentina, May, 2017.

[180] J. Svajlenko and C. K. Roy. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In *Proceedings of the Tools Demos Track of the 32nd International Conference on Software Maintenance and Evolution(ICSME)*, pages 596–600, Raleigh, North Carolina, USA, October 2016.

[181] J. Svajlenko, C. K. Roy, and F. Al-Omari. The Mutation and Injection Framework, 2018. URL: `https://github.com/jeffsvajlenko/MutationInjectionFramework`.

[182] J. Svajlenko, C. K. Roy, and J. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *Proceedings of Short/Tool Papers Track of the ICSE 7th International Workshop on Software Clones (IWSC)*, pages 8–9, San Francisco, CA, May 2013.

[183] A. Machiry R. Tahiliani and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 21st ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 224–234, 2013.

[184] H. Tang, S. Tan, and X. Cheng. A survey on sentiment detection of reviews. *Expert Systems with Applications: An International Journal*, 36(7):10760-10773, September 2009.

[185] G. Tassey. The economic impacts of inadequate infrastructure for software testing. In *RTI Final Report*, page 309, National Institute of Standards and Technology, 2002.

[186] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering (ESE)*, 15(1): 1-34, 2009.

[187] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra. Automating test automation. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 881–891, 2012.

[188] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *Proceedings 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59, 2012.

[189] S. Tsakiltsidis, A. Miranskyy, and E. Mazzawi. On automatic detection of performance bugs. In *Proceedings Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*, pages 132–139, 2016.

[190] S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, pages 13–22, 2011.

[191] R. van Tonder and C. Le Goues. Defending against the attack of the micro-clones. In *Proceedings of the 24th International Conference on Program Comprehension (ICPC)*, pages 1–4, Austin, TX, USA, 2016.

[192] T. Vislavski, G. Rakic, N. Cardozo, and Z. Budimac. LICCA: A tool for cross-language clone detection. In *Proceedings of the 25th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 512–516, Campobasso, Italy, March, 2018.

[193] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy. CCAligner: a token based large-gap clone detector. In *Proceedings 40th International Conference on Software Engineering (ICSE)*, pages 1066–1077, 2018.

[194] S. Wang and D. Lo. History, similar report, and structure: putting them together for improved bug localization. In *Proceedings 22nd International Conference on Program Comprehension (ICPC)*, pages 53–63, 2014.

[195] S. Wang and D. Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software Evolution and Process*, 28:921-942, October 2016.

[196] S. Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *Proceedings 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 171–180, 2014.

[197] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proceedings 21st ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 455–465, 2013.

[198] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can I clone this piece of code here? In *Proceedings 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 170–179, 2012.

[199] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proceedings IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 181–190, 2014.

[200] S. Xie, F. Khomh, and Y. Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *Proceedings 10TH Working Conference on Mining Software Repositories (MSR)*, pages 149–158, 2013.

[201] X. Ye, R. C. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 689–699, 2014.

[202] N. Yoshida, T. Hattori, and K. Inoue. Finding similar defects using synonymous identifier retrieval. In *Proceedings 4th International Workshop on Software Clones (IWSC)*, pages 49–56, 2010.

[203] Y. Yuki, Y. Higo, K. Hotta, and S. Kusumoto. Generating clone references with less human subjectivity. In *Proceedings of the 24th International Conference on Program Comprehension (ICPC)*, pages 1–4, Austin, TX, USA, 2016.

[204] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10, 2002.

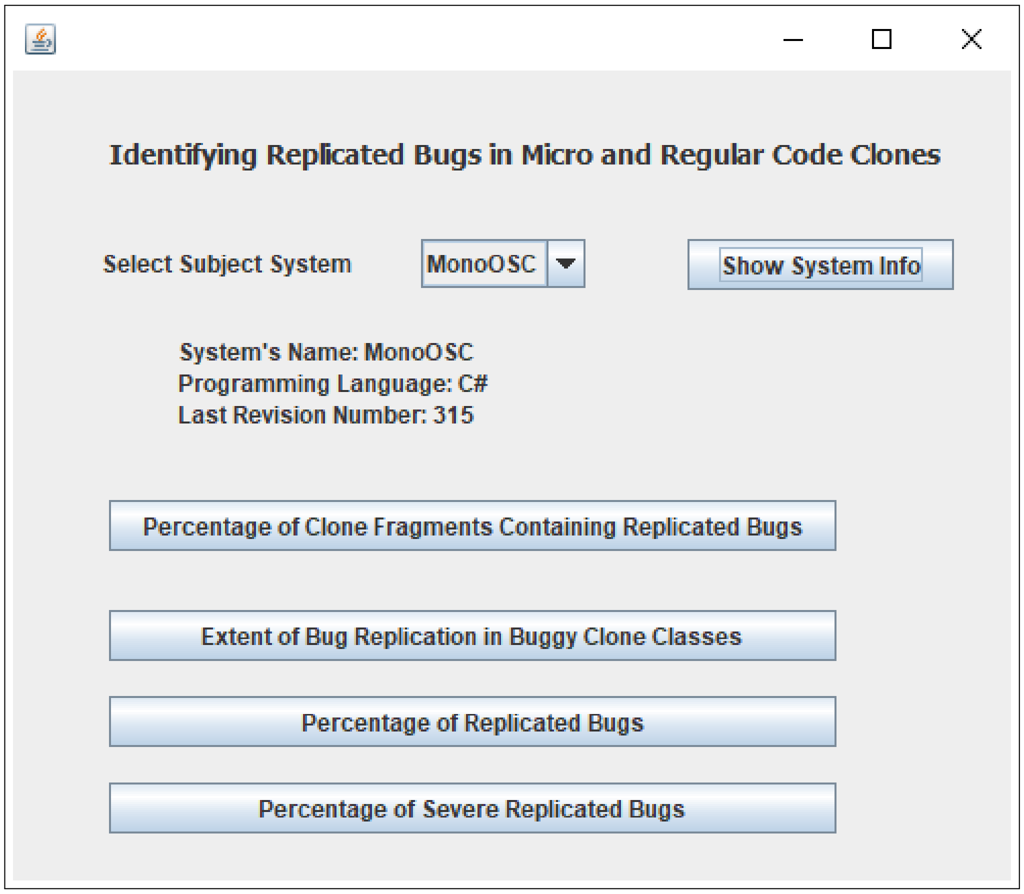
[205] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transaction on Software Engineering*, 28(2):183–200, 2002.

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

Appendix A

Detection of Replicated Bugs in Regular and Micro-Clones

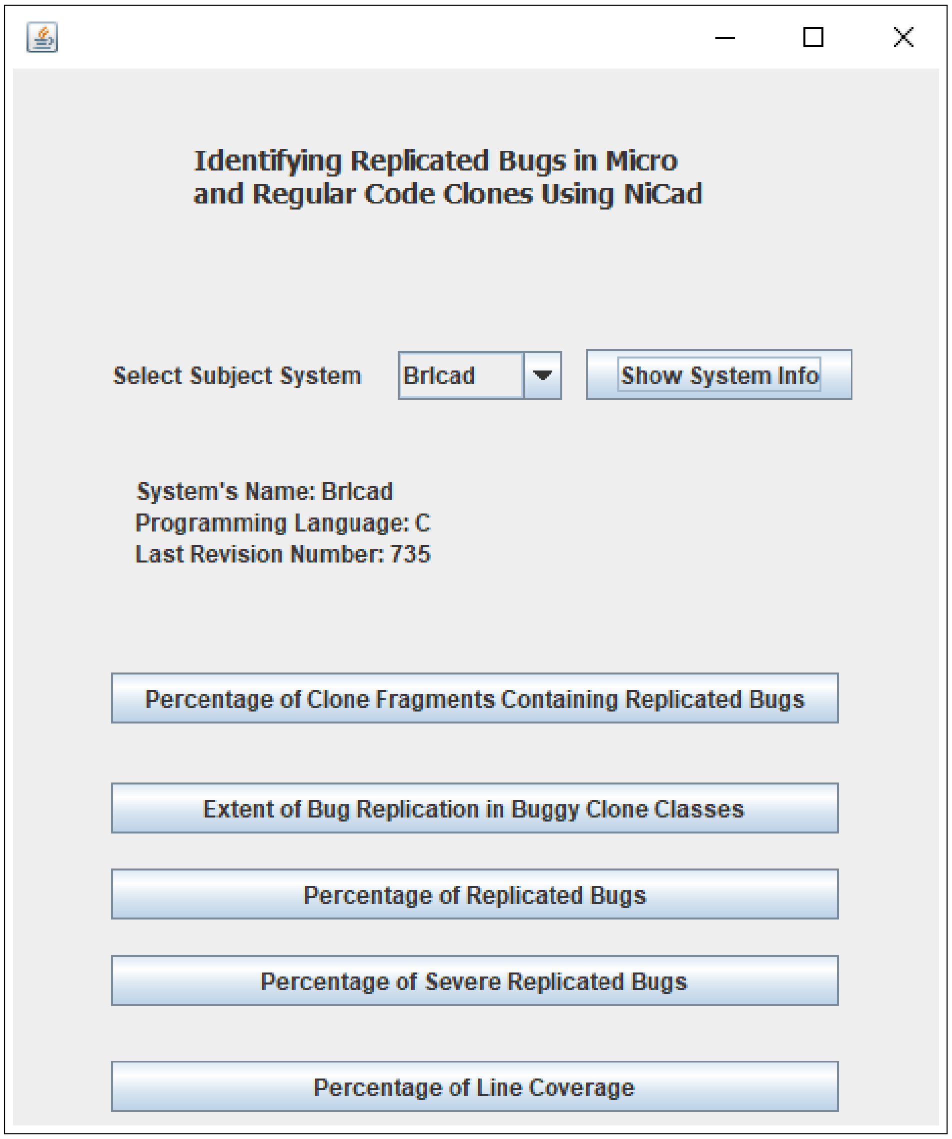
As described in Chapter 6 to detect replicated bugs in regular and micro-clones using NiCad clone detection tool, we implemented a GUI user interface. We used NetBeans IDE version 8.2 for front end development of NiCad experiment. Figure A.1 shows the GUI interface for detecting replicated bugs in regular and micro-clones. As an extension in Chapter 7, we detected replicated bugs in regular and micro-clones using three different clone detection tools. We used IntelliJ IDEA version 2020.1.4 for front end development in experiment with ConQat and iClones. We used WampServer and MySQL database for back end development for our fourth study (both previous and extended). Our machine configuration was Intel(R) Core(TM) i7 8th Generation, CPU 3.20 GHz, 16 GB DDR4 RAM, 64-bit OS. Here, Figure A.2 shows the GUI interface for detecting replicated bugs using NiCad clone detection tool. Any subject system can be selected from the drop down menu as shown in the Figure A.2. Information about the selected subject system such as programming language and last revision number can be shown after clicking the *Show System Info* button. Our five research questions can be asked individually using five different buttons. Each of these five buttons can give the corresponding result for selected subject system. Our first research question represents the *Percentage of Clone Fragments Containing Replicated Bugs* button. Second research question presents the *Extent of Bug Replication in Buggy Clone Classes* button. Third research question represents *Percentage of Replicated Bugs* button. Fourth research question represents the *Percentage of Severe Replicated Bugs* button. Finally, answer of our fifth research question can be found by clicking *Percentage of Line Coverage* button. Similarly, we implemented another GUI interface for detecting replicated bugs in code clones using ConQat clone detection tool. Figure A.3 shows the GUI interface for ConQat. Lastly, we implemented the GUI interface for iClones clone detection tool as well. Figure A.4 shows the GUI interface for detecting replicated bugs using iClones. In Figure A.5, we showed the result of our third research question for ConQat when candidate system Carol is selected.


Identifying Replicated Bugs in Micro and Regular Code Clones
Select Subject System
MonoOSC
Show System Info

System's Name: MonoOSC
Programming Language: C#
Last Revision Number: 315

Percentage of Clone Fragments Containing Replicated Bugs
Extent of Bug Replication in Buggy Clone Classes
Percentage of Replicated Bugs
Percentage of Severe Replicated Bugs

Figure A.1: A GUI interface used for detection of replicated bugs in code clones

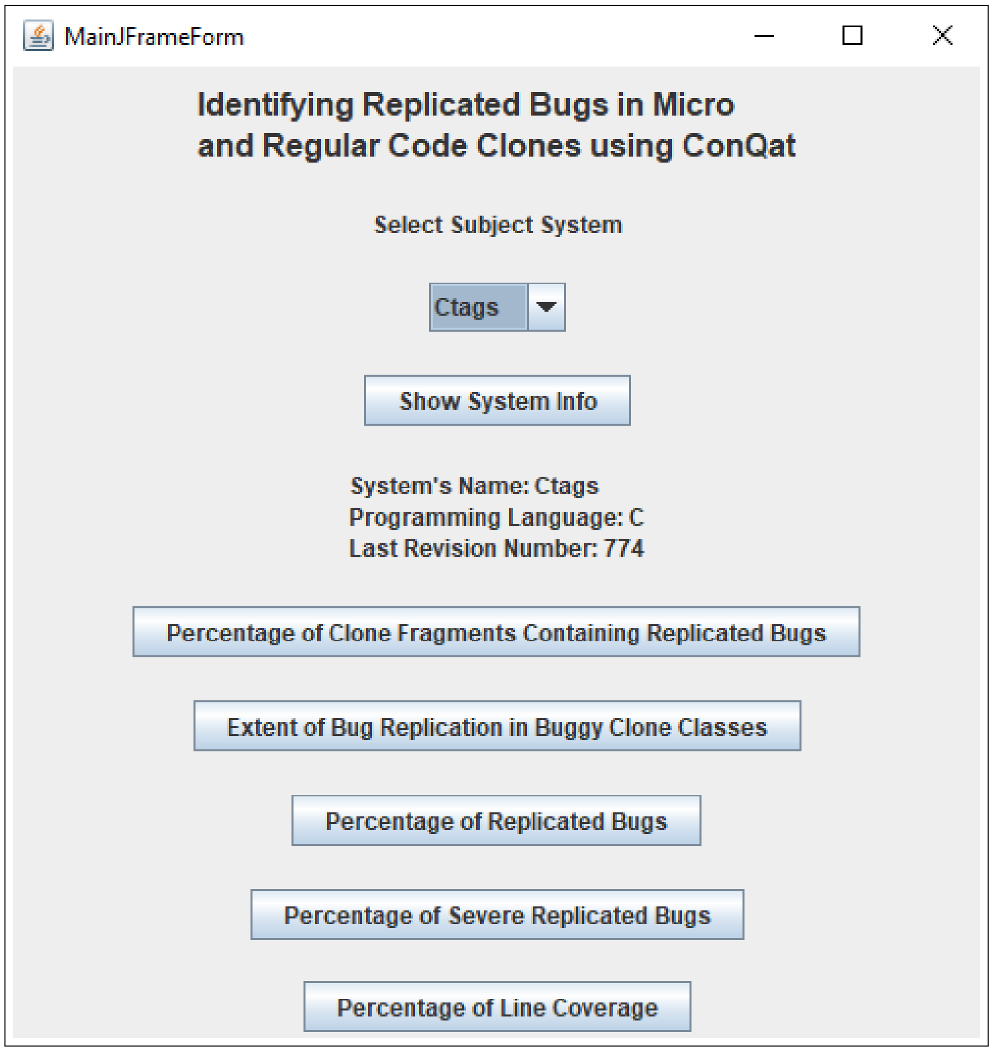

Identifying Replicated Bugs in Micro and Regular Code Clones Using NiCad

Select Subject System
Brlcad
Show System Info

System's Name: Brlcad
Programming Language: C
Last Revision Number: 735

Percentage of Clone Fragments Containing Replicated Bugs
Extent of Bug Replication in Buggy Clone Classes
Percentage of Replicated Bugs
Percentage of Severe Replicated Bugs
Percentage of Line Coverage

Figure A.2: A GUI interface used for detection of replicated bugs using NiCad clone detection tool


MainJFrameForm

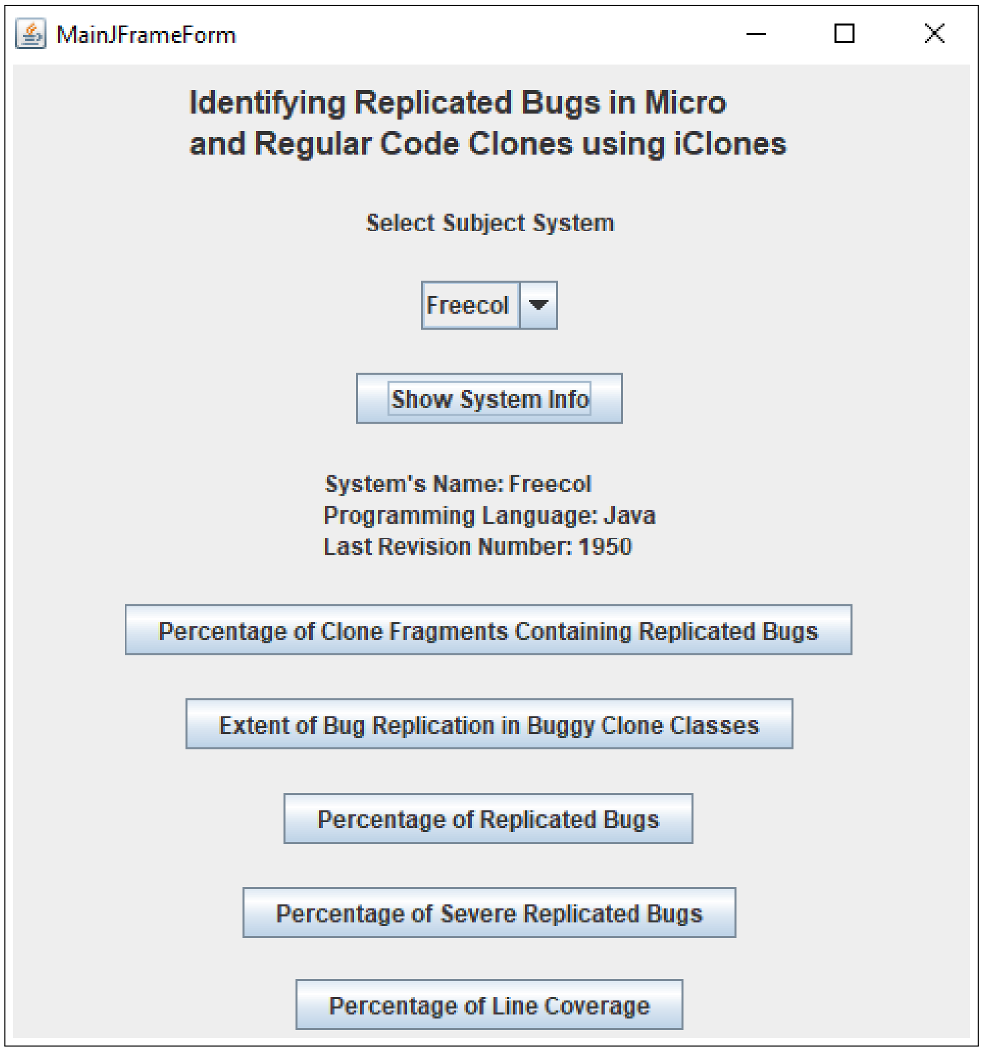
Identifying Replicated Bugs in Micro
and Regular Code Clones using ConQat

Select Subject System

System's Name: Ctags
Programming Language: C
Last Revision Number: 774

Percentage of Clone Fragments Containing Replicated Bugs
Extent of Bug Replication in Buggy Clone Classes
Percentage of Replicated Bugs
Percentage of Severe Replicated Bugs
Percentage of Line Coverage

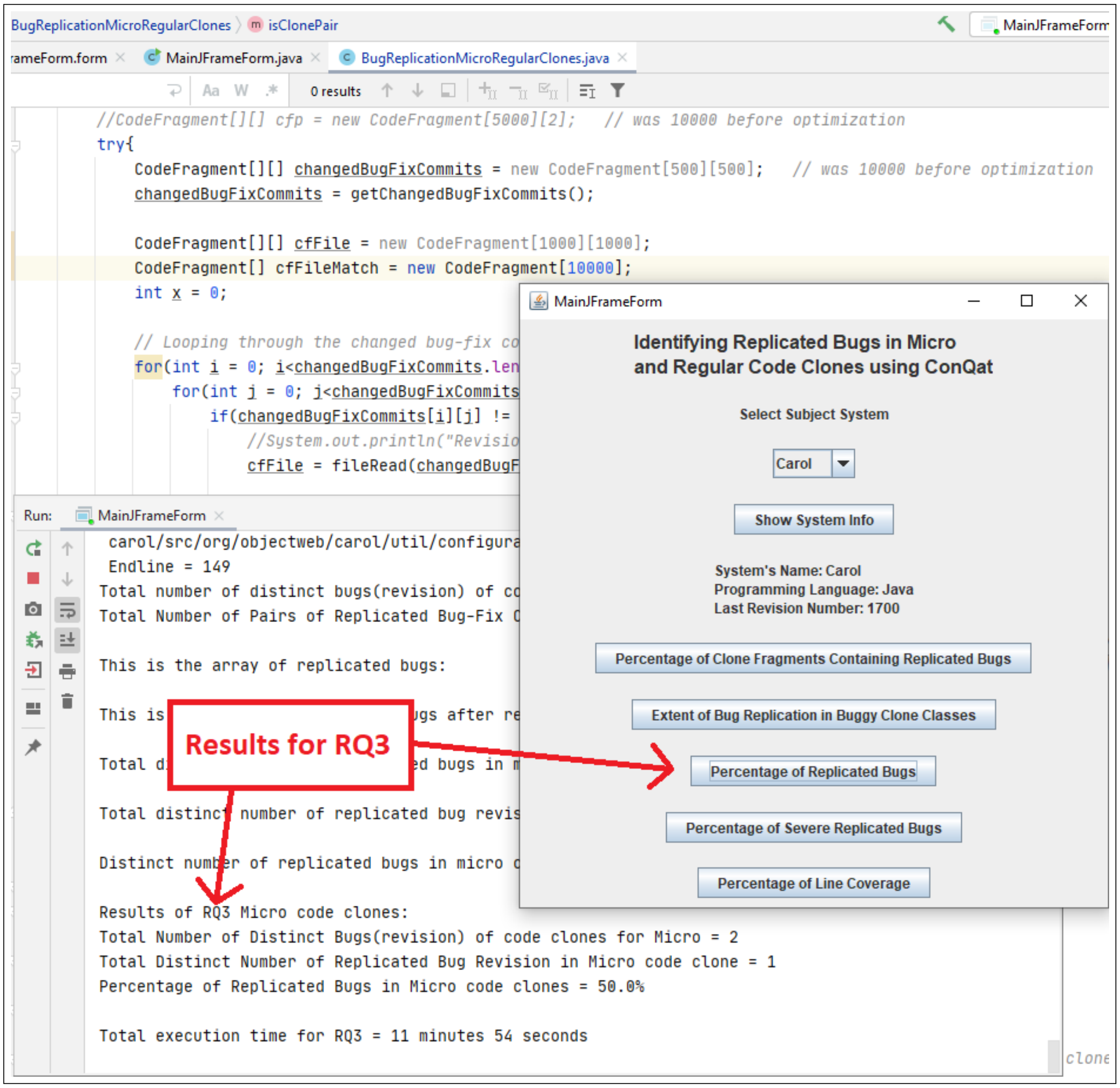
Figure A.3: A GUI interface used for detection of replicated bugs using ConQat clone detection tool


MainJFrameForm
Identifying Replicated Bugs in Micro
and Regular Code Clones using iClones
Select Subject System
Freecol
Show System Info

System's Name: Freecol
Programming Language: Java
Last Revision Number: 1950

Percentage of Clone Fragments Containing Replicated Bugs
Extent of Bug Replication in Buggy Clone Classes
Percentage of Replicated Bugs
Percentage of Severe Replicated Bugs
Percentage of Line Coverage

Figure A.4: A GUI interface used for detection of replicated bugs using iClones clone detection tool


BugReplicationMicroRegularClones
isClonePair
MainJFrameForm
rameForm.form
MainJFrameForm.java
BugReplicationMicroRegularClones.java
0 results
Run:
MainJFrameForm
Results for RQ3
Show System Info
Percentage of Clone Fragments Containing Replicated Bugs
Extent of Bug Replication in Buggy Clone Classes
Percentage of Replicated Bugs
Percentage of Severe Replicated Bugs
Percentage of Line Coverage

```
    //CodeFragment[][] cfp = new CodeFragment[5000][2];   // was 10000 before optimization
    try{
        CodeFragment[][] changedBugFixCommits = new CodeFragment[500][500];   // was 10000 before optimization
        changedBugFixCommits = getChangedBugFixCommits();

        CodeFragment[][] cfFile = new CodeFragment[1000][1000];
        CodeFragment[] cfFileMatch = new CodeFragment[10000];
        int x = 0;

        // Looping through the changed bug-fix co
        for(int i = 0; i<changedBugFixCommits.len
            for(int j = 0; j<changedBugFixCommits
                if(changedBugFixCommits[i][j] !=
                    //System.out.println("Revisio
                    cfFile = fileRead(changedBugF
```

```
 carol/src/org/objectweb/carol/util/configura
 Endline = 149
Total number of distinct bugs(revision) of co
Total Number of Pairs of Replicated Bug-Fix C

This is the array of replicated bugs:

This is                              ugs after re

Total d                              ed bugs in m

Total distinct number of replicated bug revis

Distinct number of replicated bugs in micro c

Results of RQ3 Micro code clones:
Total Number of Distinct Bugs(revision) of code clones for Micro = 2
Total Distinct Number of Replicated Bug Revision in Micro code clone = 1
Percentage of Replicated Bugs in Micro code clones = 50.0%

Total execution time for RQ3 = 11 minutes 54 seconds
```

MainJFrameForm

Identifying Replicated Bugs in Micro and Regular Code Clones using ConQat

Select Subject System

Carol

System's Name: Carol
Programming Language: Java
Last Revision Number: 1700

Figure A.5: Result showing percentage of replicated bugs (RQ3) for Carol subject system using ConQat clone detection tool