Change Impact Analysis of 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
Md Saidur Rahman

Permission to Use

Abstract

Copying a code fragment and reusing it with or without modifications is known to be a frequent activity in software development. This results in exact or closely similar copies of code fragments, known as code clones, to exist in the software systems. Developers leverage the code reuse opportunity by code cloning for increased productivity. However, different studies on code clones report important concerns regarding the impacts of clones on software maintenance. One of the key concerns is to maintain consistent evolution of the clone fragments as inconsistent changes to clones may introduce bugs. Challenges to the consistent evolution of clones involve the identification of all related clone fragments for change propagation when a cloned fragment is changed. The task of identifying the *ripple effects* (*i.e.,* all the related components to change) is known as Change Impact Analysis (CIA). In this thesis, we evaluate the impacts of clones on software systems from new perspectives and then we propose an evolutionary coupling based technique for change impact analysis of clones. First, we empirically evaluate the comparative stability of cloned and non-cloned code using fine-grained syntactic change types. Second, we assess the impacts of clones from the perspective of coupling at the domain level. Third, we carry out a comprehensive analysis of the comparative stability of cloned and non-cloned code within a uniform framework. We compare stability metrics with the results from the original experimental settings with respect to the clone detection tools and the subject systems. Fourth, we investigate the relationships between stability and bug-proneness of clones to assess whether and how stability contribute to the bug-proneness of different types of clones. Next, in the fifth study, we analyzed the impacts of co-change coupling on the bug-proneness of different types of clones. After a comprehensive evaluation of the impacts of clones on software systems, we propose an evolutionary coupling based CIA approach to support the consistent evolution of clones. In the sixth study, we propose a solution to minimize the effects of atypical commits (extra large commits) on the accuracy of the detection of evolutionary coupling. We propose a clustering-based technique to split atypical commits into pseudo-commits of related entities. This considerably reduces the number of incorrect couplings introduced by the atypical commits. Finally, in the seventh study, we propose an evolutionary coupling based change impact analysis approach for clones. In addition to handling the atypical commits, we use the history of fine-grained syntactic changes extracted from the software repositories to detect typed evolutionary coupling of clones. Conventional approaches consider only the frequency of co-change of the entities to detect evolutionary coupling. We consider both change frequencies and the fine-grained change types in the detection of evolutionary coupling. Findings from our studies give important insights regarding the impacts of clones and our proposed typed evolutionary coupling based CIA approach has the potential to support the consistent evolution of clones for better clone management.

Acknowledgements

I would like to express my sincere gratitude to my respected supervisor Dr. Chanchal K. Roy for his continuous guidance, timely advice and extraordinary support in every aspect throughout this thesis. Without his inspiration and extended co-operation, this thesis work would have been simply impossible.

I am thankful to Dr. Kevin A. Schneider, Dr. Gord McCalla, Dr. Ralph Deters and Dr. Shahedul Khan for being in my PhD committee, evaluating my thesis and supporting me with valuable advice, thoughtful suggestions, and comments. I would like to thank all the co-authors

I would like to thank all the co-authors of my publications related to this thesis for their collaborations. I am really in debt to all my fellow lab members in the Software Research Lab (SRLab) for being on my side with cordial support whenever I needed. Especially, I would like to thank Manishankar Mondal, Farouq Al Omari, Mohammad Masudur Rahman, Ripon Saha, Muhammad Asaduzzaman, Sharif Uddin, Khalid Billah, Shamima Yeasmin, Amit Kumar Mondal, and Judith Islam.

I am grateful to the Department of Computer Science of the University of Saskatchewan for the necessary technical support and supporting me financially with scholarships and teaching fellowships during the thesis. I would like to thank all the staff members in Computer Science for their help. Especially, I would like to thank Gwen Lancaster, Heather Webb, Findlay Sophie, and Merlin Hansen.

I am thankful to the anonymous reviewers for their valuable comments on the papers published from this thesis.

My daughter Samiha Farheen and my son Muntasir Rahman are the two most precious gifts of my life, my source of motivation to dream big and work hard. They along with their great mother, my wife Fahmida Nawab sacrificed a lot for me to complete this thesis. I am in debt for their patience and sacrifice.

I express my humble gratitude to my beloved mother Rizia Khatun and my wonderful father Md Akbar Ali Sordar for their limitless sacrifice, love, care, and prayers in every moment of my life. I am thankful to my sisters Shamsunnaher, Zabeda, and Anjumanara and my brother Anisur Rahman for their love and inspirations.

Above all, I am grateful to the Almighty for all his blessings on me to complete this thesis.

I would like to dedicate my thesis to my mother **Rizia Khatun** and my father **Md Akbar Ali Sordar** for their endless love, care, inspiration, and sacrifice throughout my life; to my beloved wife **Fahmida Nawab** for her invaluable support and care during this thesis and beyond.

Contents

List of Tables

List of Figures

List of Abbreviations

CIA	Change Impact Analysis
EC	Evolutionary Coupling
TEC	Typed Evolutionary Coupling
MSR	Mining Software Repositories
AST	Abstract Syntax Tree
LOC	Lines of Code
DBSCAN	Density-Based Spatial Clustering of Applications with Noise

Chapter 1

Introduction

1.1 Motivation

Code reuse by copy-paste is a common practice in software development, and as result software systems often have sections of code that are identical or similar, called software clones or code clones. Clones comprise a significant proportion of the code base in the software systems(between 7% and 23% [151] or sometimes even 50% [140]). Regardless of the intentional and unintentional reasons behind code cloning, the impact of clones on software maintenance has been a great concern [148].

Although it is believed that code cloning speeds up software development and facilitates the reuse of mature and tested code, clones are often accused of introducing maintenance challenges by making consistent changes more difficult leading to the introduction of bugs [69], propagation of existing bugs and thus resulting in increasing maintenance efforts [97]. Having both positive and negative impacts on software maintenance, code cloning has been under significant research focus [18, 49, 56, 57, 83, 84, 86, 21, 22, 34, 69, 51, 95, 94, 97, 100, 122, 162, 185, 178, 60] towards the evaluation of the impacts of clones on software maintenance and evolution. Some of the existing studies conclude in favour of clones suggesting that clones are not harmful [18, 49, 56, 57, 83, 84, 86] , rather, cloning can be beneficial to software development [74]. A good number of studies on the other hand conclude with empirical evidences that clones have negative impacts on software maintenance and evolution [21, 22, 34, 69, 51, 95, 94, 97, 100, 122, 162, 185].

Given the negative impacts of clones, researchers agree that clones need to be managed [148]. Effective clone management aims to minimize the negative impacts of clones while leveraging the benefits of code cloning. One important approach to minimize the negative impacts of clones is to remove clones by refactoring [181, 87]. However, not all of the clones are suitable for refactoring [75, 175]. Clones those are not suitable for refactoring are candidates for tracking [39, 38]. One key task in clone management is to ensure the consistent evolution of clones so that they do not introduce inconsistencies or bugs.

As there are thousands of cloned fragments in the software systems, it is very important and crucial task to carry out Change Impact Analysis (CIA) for clones *i.e.*, identifying all co-change candidates when a cloned fragment is changed. Moreover, a cloned fragment may have evolutionary dependencies with clones of the same *clone class*(set of code fragments that are clone to each other), clones from different clone class or even with non-clone code fragments [114]. Thus, traditional static analysis (on source code) and dynamic analysis (on execution trace) may not be able to capture all aspects of dependencies between clones for change impact analysis. For example, two clone fragments may require to co-evolve to ensure consistency without having any syntactic or execution dependencies. As a result, evolutionary coupling (based on the extent of co-change in evolution history) have been found effective in identifying important candidate clones for refactoring and tracking [113, 101, 124, 116]. However, none of the existing studies use the important fine-grained syntactic typed-change history of clones in either assessing the impacts of clones or in identifying co-change candidates for clones.

In the proposed thesis, we first assess the impacts of clones from different new perspectives and then we propose an evolutionary coupling based improved (minimizing the effects of atypical commits *i.e.,* commits affecting an excessively large set of entities) change impact analysis technique for code clones. For our research, we first analyze the comparative stability of cloned and non-cloned code using fine-grained syntactic change information. We then investigate the impacts of clones at domain-level by comparing the correspondence between the existence of domain based coupling [15, 14] and code clones among the software components. Next, we evaluate different existing metrics on the impacts of clones within a uniform framework to look for firm conclusions given the contradictory conclusions on the impacts of clones by the existing studies [56, 84, 86, 97, 100, 49]. We then study the relationships between stability and bug-proneness of different types of clones by analyzing the evolution of clones from the perspective of fine-grained change types. After that, we investigate the impacts of co-change coupling on the bug-proneness of clones. Then we focus on developing a technique to support the consistent evolution of clones. We propose a clustering-based splitting technique for atypical commits (extra-large commits) to minimize the effects of atypical commits on the detection of evolutionary coupling. Finally, we propose an evolutionary coupling based improved technique for change impact analysis of code clones using fine-grained change types.

In conclusion, the proposed research first comprehensively assesses the impacts of clones from different new perspectives and then proposes an evolutionary coupling based improved technique for change impact analysis of code clones to support the consistent evolution of clones, an important and prime task in clone management.

1.2 Problem Definition and Scope

1.2.1 Background and Research Problem

As software systems contain a significant proportion of cloned code [151, 140], there has been a significant research focus on the impacts of clones on software systems. Consequently, there are many studies from different perspectives investigating the impacts of clones. Some well-investigated perspectives of assessing the impacts of clones are stability [84, 86, 122, 123], bug-proneness [60, 61, 116, 120], and the effects of late propagation [21, 22, 125]. However, the studies concluded in contradictory outcomes regarding the impacts of clones on software systems. The contradictory outcomes are likely due to the differences in the experimental settings, clone detection tools, subject systems studied and even the metrics measured. Thus, several studies [122, 123] focused on the assessment of the impacts of clones using a common framework to avoid possible biases due to differences in experimental settings. Mondal *et al.* [122, 123] investigated the impacts of clones within a uniform framework using different clone detection tools, subjects systems of different programming languages to assess the comparative stability of cloned and non-cloned code. Although these studies give important insights regarding the stability of clones, the debate on the impacts of clones is yet to be resolved. These studies did not consider the original experimental settings (clone detection tools and subject systems). Mondal *et al.* in a recent study [121] investigated eight existing stability metrics within a uniform framework to study the comparative stability of cloned and non-cloned code. The study applies clone detection tools NiCad [144] and CCFinderX [1] on 12 subject systems of three (Java, C, C#) programming languages. This study also evaluates the differences in the metrics by systematic replication of the original studies using original settings (subject systems and clone detection tools). However, the study did not evaluate the metrics by applying different clone detection tools on the subject systems used in the original studies. Thus, it is important to investigate the comparative stability of cloned and non-cloned code by applying different clone detection tools on the subject systems used in original studies.

Again, existing stability analysis approaches do not consider the fine-grained change types in their proposed stability metrics. However, different types of syntactic changes have different impacts on the functional modification of the software systems. Types of changes also define the extent of required change propagation due to ripple effects of changes. However, none of the existing stability analysis considers fine-grained change types.

1.2.2 Research Goals and Scope

In this thesis, we address the problems outlined above from two distinct aspects: ***assessing the impacts of clones on software systems*** from new perspectives and to develop ***evolutionary coupling based change impact analysis*** approach for clones that incorporates fine-grained typed change information from software repositories for more accurate detection of evolutionary coupling of clones.

1.2.3 Research Contributions

To address the two primary research goals defined above, we carry out our research studies in the following two phases:

- **Phase 1**: Comprehensive assessment of the impacts of clones on software systems from different perspectives.
- **Phase 2**: Developing an evolutionary coupling based improved approach for change impact analysis of code clones.

In Phase 1, we assess the problem by investigating the impacts of clones from different new perspectives. First, we measure comparative *stability* of cloned code by considering fine-grained syntactic change types and their significance. As different syntactic change types have different impacts on the components related to a changed component, this study investigates the comparative stability of cloned and non-cloned code considering syntactic change types and their significance. Second, in an exploratory study, we evaluate the impacts of code clones at the domain level. This study explores the relationships between domain-level coupling among the user interface components (UIC) and code clones. Studies show that domain level coupling corresponds to dependencies at code level such as conceptual and evolutionary coupling [14, 48]. Our study shows that there exists a strong correspondence between code clones and domain-based coupling. Thus, the existence of clones corresponds to the existence of couplings at domain-level as well as code level. This shows that code clones corresponds to coupling and thus have negative impacts on software systems. Third, we evaluate the comparative stability of cloned and non-cloned code within a uniform framework employing different clone detection tools and subject systems used in original studies to investigate the impacts of clones avoiding possible biases. We also evaluate the metrics for the impacts of clones in their original experimental settings (tools, parameters, and subject systems) to comprehensively assess the impact of clones. Fourth, we investigate the relationships between stability and the bug-proneness of clones. Fifth, we investigate the impacts of co-change coupling on the bug-proneness of code clones.

In Phase 2, we aim to develop an improved change impact analysis approach for clones. To improve the existing evolutionary coupling based change impact analysis technique we focus on two key issues: (1) resolving the effects of atypical commits on the detection of evolutionary coupling, and (2) detecting evolutionary coupling of clones more accurately using fine-grained change information. To minimize the effects of atypical commits, we propose a clustering-based technique to split atypical commits for more accurate detection of evolutionary coupling. In this study, we evaluate the extent to which atypical commits affect the detection of evolutionary coupling. We then apply DBSCAN [42] clustering algorithm to split the atypical commits into pseudo commits of related items. We then detect evolutionary coupling by replacing the atypical commits by the corresponding pseudo commits obtained from the clustering. Finally, we propose an improved technique for change impact analysis of clones by incorporating fine-grained change types in detecting typed evolutionary coupling of clones.

1.3 Summary of Our Studies and Approaches to Address the Research Problem

To address the above research problems, we have conducted a number of empirical studies to answer our research questions. We conducted the following seven studies with specific research objectives:

- **Study 1**: *Assessing Clone Stability From the Perspective of Fine-grained Change Types.*
- **Study 2**: *Relationships between Domain-Based Coupling and Code Clones*

- **Study 3**: *Analyzing Comparative Stability of Cloned and Non-Cloned Code within Uniform Framework*
- **Study 4**: *Analyzing the relationships between Stability and Bug-proneness of Code Clones*
- **Study 5**: *Analyzing the impacts of co-change coupling on the Bug-proneness of Code Clones*
- **Study 6**: *Analyzing the Effects of Atypical Commits on the detection of Evolutionary Coupling*
- **Study 7**: *A Change-type Based Approach for Detecting Evolutionary Coupling of Code Clones*

Here, the first five studies (Study 1, Study 2, Study 3, Study 4 and Study 5) belong to Phase 1 of our research while the last two studies (Study 6 and Study 7) belong to Phase 2. The following sections briefly describe our studies to outline our research contributions.

1.3.1 Study 1: Assessing Clone Stability from the Perspective of Fine-grained Change Types

In this study, we empirically evaluate the comparative stability of cloned and non-cloned code using fine-grained syntactic change types. A large number of studies concerning the impacts of clones on software systems mainly focus on the frequency of changes to evaluate stability, consistency in evolution and introduction of bugs. Although it is obvious that not each type of changes has equal impact on software systems, none of the existing studies take the types of changes and their significance into account during comparative evaluation of the stability of cloned and non-cloned code. In this empirical study, we analyze the comparative stability of cloned and non-cloned code from the perspective of different change types. Changes from successive revisions are extracted and classified using ChangeDistiller which employs Abstract Syntax Tree (AST) differencing of the successive revisions of source code files. We detect exact (Type 1) and near-miss (Type 2 and Type 3) clones using the clone detection tool NiCad. Extracted and classified changes and clone information are then analyzed to compare the stability of cloned and non-cloned code from three different perspectives: types of clones, types of changes with respect to the significance of changes, and size and extent of evolution of the systems. Our study on seven open-source Java systems with diversity in their size, length of evolution and application domain shows that changes are more frequent in cloned code than in non-cloned code and Type 1 clones are comparatively more vulnerable to the stability of the systems. Therefore, cloned code is less stable than non-cloned code suggesting that cloned code is likely to pose more maintenance challenges than non-cloned code.

1.3.2 Study 2: Evaluating the Impacts of Clones at Domain Level

Knowledge of the existence of code clones is important to many software maintenance activities including bug fixing, refactoring, and impact analysis and program comprehension. While a great deal of research has been conducted for finding techniques and implementing tools to identify code clones, little research has been done to analyze the relationships between code clones and other aspects of software. In this study, we attempt to uncover the relationships between code clones and coupling among domain-level components. We report on a case study of a large-scale open source enterprise system, where we demonstrate that the probability of finding code clones among components with domain-based coupling (similarity of domain level information in user interfaces) is more than 90%. While such a probabilistic view does not replace a clone detection tool per se, it certainly has the potential to complement the existing tools by providing the probability of having code clones between software components. For example, it can both reduce the clone search space and provide a flexible and language independent way of focusing only on a specific part of the system. It can also provide a higher level of abstraction to look at the cloning relationships among software components.Again, our study shows that there exists a strong correspondence between code clones and domain-based coupling. We observe that the number of clones among the software components increases with the increase in couplings at the domain-level. Again studies [14, 48] show that domain level coupling is also related to coupling at the code level. As high coupling is not a desirable software quality, existence of clones likely to have negative impacts on software systems as it contributes to high coupling at the domain level.

1.3.3 Study 3: Comprehensive Stability Analysis of Clones within Uniform Framework

The impact of clones is of great importance from software maintenance perspectives. Stability is a well-investigated feature in assessing the impacts of clones on software maintenance. While some studies show that code clones are more stable than non-cloned code, the other studies provide empirical evidence of higher instability of code clones. The possible reasons behind these contradictory findings are that different studies investigated different aspects of stability using different clone detection tools on different subject systems using different experimental setups. We evaluate the comparative stability of cloned and non-cloned code within a uniform framework. We employ different clone detection tools and subject systems to investigate the impacts of clones avoiding possible biases. We also evaluate the stability metrics for the impacts of clones in their original experimental settings (tools, parameters, and subject systems) to comprehensively assess the impact of clones. Our investigation of 16 diverse subject systems for 5 stability metrics suggests that it is very hard to draw a general conclusion on the impacts of clones. The comparative stability scenarios vary with experimental settings, subject systems, and their evolution. We also systematically replicated the original studies with their original settings and found mostly equivalent results as of the original studies.

1.3.4 Study 4: Analyzing the relationships between Stability and Bug-proneness of Code Clones

Stability is a widely investigated perspective of assessing the impacts of clones on software systems. A number of existing studies show that clones are often less stable than non-cloned code. This suggests that clones change more frequently than non-cloned code and thus may require comparatively more maintenance efforts.

Again, frequent changes to clones may increase the likelihood of missing change propagation to the co-change candidates leading to inconsistencies or bugs. However, none of the existing studies investigate whether the stability of clones is related to the bug-proneness. In this empirical study, we analyze the relationships between stability and bug-proneness of clones. At first, we identify bug-fix commits by analyzing the commit messages from software repositories. Then we extract classified changes using ChangeDistiller. Clones those are changed in the bug-fix commits are considered as bug-prone clones. We compare the stability of buggy and non-buggy clones considering the fine-grained syntactic change types and their significance. Our experimental results based on five open-source Java systems show that (1) stability and bug-proneness of code clones are related and this relationship is statistically significant, (2) for both exact (Type 1) and near-miss (Type 2 and Type 3) clones, buggy clones tend to have higher frequency of changes than non-buggy clones, (3) the bug-proneness of Type 2 and Type 3 clones tend to be strongly related with their stability compared to Type 1 clones, and (4) the relationship between the stability and the bug-proneness of clones with respect to fine-grained change types is likely to be influenced by the changes of low to medium significance. We believe that our findings are important and potentially useful in identifying and prioritizing candidate clones for management.

1.3.5 Study 5: Analyzing the Impacts of Co-change Couplings on the Bug-proneness of Code Clones

One of the important claims against code clone is that if clones evolve inconsistently due to missing change propagation, they are likely to introduce bugs. Clones might have evolutionary couplings with clones from same clone class, clones from different clone class or even with non-clones. When a cloned fragment is coupled with higher number of other code fragments, it intuitively increases the complexity and cost of change propagation. Moreover, the higher the number of co-change candidates, the higher will be the likelihood of missing change propagation and so the likelihood of introducing inconsistencies. Although there are many different studies investigating the bug-proneness of different types of clones, none of the existing studies investigate whether the bug-proneness of clones are related to the degree of co-change coupling. In this empirical study, we analyze the relationships between the co-change coupling and the bug-proneness of clones. We identify bug-fix commits by analyzing the commit messages from software repositories. Clone fragments those are changed in the bug-fix commits are considered as bug-prone clones. We compare the co-change couplings of the buggy and non-buggy clones with respect to couplings with clones and non-clones. Our experimental results based on six open-source Java systems show that (1) co-change coupling and bug-proneness of code clones have statistically significant relationship and this relationship is likely to be dominated by the couplings of clones with non-clones (2) Type 1 and Type 3 clones exhibit significant association between the co-change couplings and the bug-proneness while the co-change coupling of Type 2 clones might not have significant relationship with the bug-proneness (3) the degree of co-change couplings of clones with non-clones might be a significant indicator of their bug-proneness.

1.3.6 Study 6: Evaluating and Minimizing the Impacts of Atypical Commits on the detection of Evolutionary Coupling

Detection of evolutionary coupling is a promising technique to discover dependencies among the evolving program entities. The conventional approaches detect evolutionary coupling by mining association rules based on changes in commit histories from software repositories. However, a considerable proportion of the commits are atypical commits in the software repositories involving changes to an excessively large number of program entities. Atypical commits are likely to result from major refactoring activities and may involve changes to more than one independent group of related entities. Here, intra-group entities are likely to be related where inter-group entities might not have any dependencies among them although co-changed in the atypical commits. However, conventional approaches for detecting evolutionary coupling usually discard the larger commits based on a selected threshold. However, most of the entities are changed infrequently and the exclusion of atypical commits may cause a significant loss of important evolution information. Considering all commits also likely to affect the accuracy of the detection of evolutionary coupling. We carry out an empirical study using clustering-based (DBSCAN) technique to split atypical transactions. The goal is to split the atypical transactions (commits) into a set of pseudo transactions of related entities based on the evolution information in the non-atypical transactions. Our experimental results based on the analysis of thousands of revisions of five Java subject systems show that transaction splitting considerably minimizes the impacts of atypical commits. The proposed approach gives association rules that yield considerable improvement in average precision of the detection of evolutionary coupling as compared to the conventional approach with a small improvement in average recall.

1.3.7 Study 7: A Change-type Based Approach for Detecting Evolutionary Coupling of Code Clones

Evolutionary coupling is detected by mining association rules based on change histories from software repositories. Conventionally, detection of evolutionary coupling considers only the change frequencies of the co-changed entities no matter whether those changes are syntactically relevant to represent dependencies. This most likely affects the accuracy of the conventional approaches. Moreover, because of considering only the change frequencies, the conventional approaches may also fail to identify evolutionary coupling among the infrequently co-changed program entities. Evolutionary coupling is capable of capturing the evolutionary dependencies not detectable by static or dynamic analysis. Thus, evolutionary coupling is likely to be more suitable to identify co-change candidates for clones. Clones may need to co-evolve consistently without having any syntactic dependencies among them. Change propagation requirement for a change to clone may include changes to clones of the same clone class, clones of different clone class or even changes to non-cloned code fragments. When a clone fragment is changed, other clones from the same clone class may need same or similar changes. For such cases of changes to be consistent, same or similar part of the clones need to be changed by same syntactic change types. However, clones from other clone class or other non-clone code fragments may need corresponding change propagation because of the existence of syntactic and semantic dependencies. The type of syntactic changes two entities evolve through may provide useful insights of their dependency relation. In this study, we propose the concept of typed evolutionary coupling that uses change history and the fine-grained syntactic change types extracted using ChangeDistiller. Here, in addition to the information regarding 'which' software entities have changed, we augment the information of 'how' they have changed syntactically. For each pair of co-changed entities, we mine association rules for all distinct pairs of change types. We then measure the aggregated coupling value for each pair of evolutionary coupled entities (clone fragments). Our experimental results based on the analysis of six diverse subject systems written in Java show that our approach demonstrates improved accuracy over the conventional approach.

1.4 Organization of the Thesis

This chapter introduces our research problem with motivations and then briefly describes how we address our defined research problem through different studies. The rest of the dissertation is organized as follows:

- Chapter 2 briefly discusses the background concepts and terminologies related to our research and necessary to follow the remaining parts of the thesis.
- Chapter 3 represents our change-typed based empirical study to analyze the comparative stability of cloned and non-cloned code.
- In Chapter 4, we explore the relationships between domain based coupling and code clones.
- Chapter 5 represents a uniform framework for a comprehensive analysis of the stability of clones.
- In Chapter 6, we represent an empirical study on the relationships between stability and bug-proneness of different types of clones.
- In Chapter 7, we study the impacts of co-change coupling on the bug-proneness of different types of clones.
- Chapter 8 evaluates the impacts of atypical commits on the detection of evolutionary coupling and proposes a clustering-based splitting technique for atypical commits to minimize the impacts.
- In Chapter 9, we present a change-type based technique for the detection of evolutionary couplings of clones.
- And finally, Chapter 10 concludes the thesis.

Parts of this thesis have been published in the peer-reviewed journal and international conferences which we list in Appendix A.

Chapter 2

Background

In this chapter, we introduce the necessary concepts and terminologies related to code clones and Change Impact Analysis that help to follow the remaining parts of this thesis.

2.1 Code Clones

2.1.1 Definition of Code Clones

As defined in the literature [150], exact or nearly similar code fragments are called clones to each other.

2.1.2 Types of Code Clones

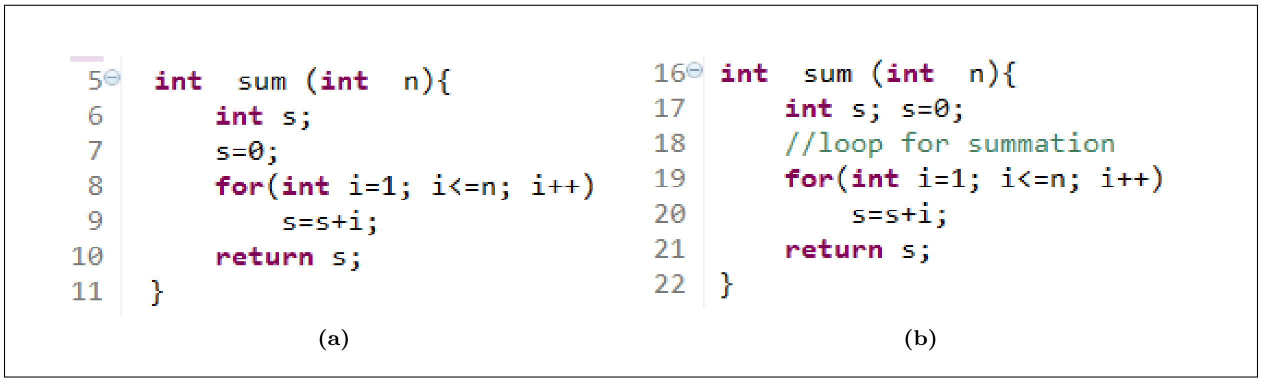
Based on syntactic and semantic similarities code clones are classified into the following four types [150, 147]:

- Type 1 Clone: Identical copies of code fragments with differences in layouts, white spaces and comments are called *Type 1* or *exact* clones. As shown in Figure 2.1 if we ignore the differences in layouts and comments, the pair of code fragments become exact copies of each other.

- Type 2 Clone: Code fragments that are syntactically identical with only differences in identifiers, literals, data types, layouts, white spaces, and comments are called *Type 2* clones. As in Figure 2.2, the identifier s in the left fragment has been renamed to tot in the code fragment on the right side. In addition, there is a comment in the code fragment on the right side. The example code fragments are *Type 2* clones of each other.

- Type 3 Clone: In addition to the differences as in *Type 2* clones, similar code fragments with variations due to addition, deletion and modification of statements are called *Type 3* clones. Example Code fragments in Figure 2.3 have differences in identifier names (s renamed to tot), comments (line 43) and the addition of line (line 44). These two code fragments are *Type 3* clones.

- Type 4 Clone: Functionally or semantically identical code fragments with syntactically different implementations are called *Type 4* or *semantic* clones. Both code fragments in Figure 2.4 are implementing the functionality of summation of numbers up to a given value n using loop and recursion respectively. These code fragments are *Type 4* or semantic clones.



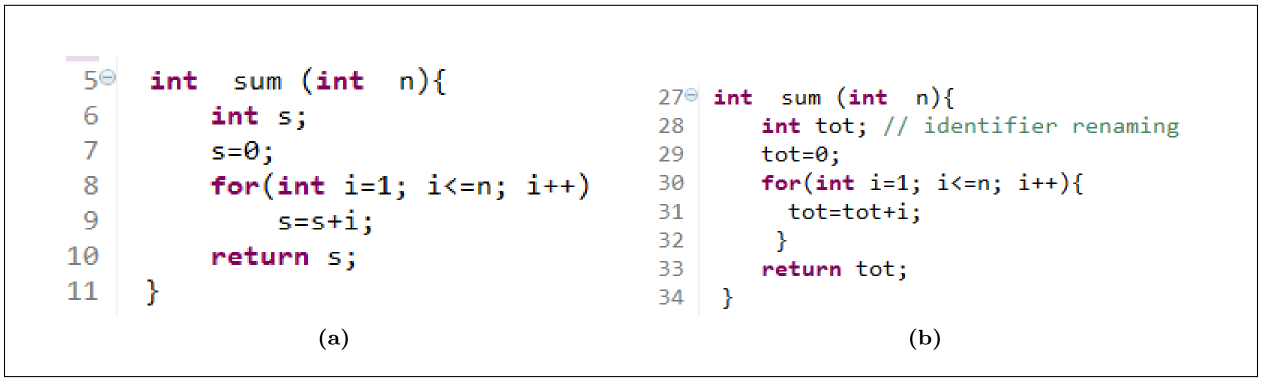
```
int  sum (int  n){
    int s;
    s=0;
    for(int i=1; i<=n; i++)
        s=s+i;
    return s;
}
```

(a)

```
int  sum (int  n){
    int s; s=0;
    //loop for summation
    for(int i=1; i<=n; i++)
        s=s+i;
    return s;
}
```

(b)

Figure 2.1: Type 1 Clones



```
int  sum (int  n){
    int s;
    s=0;
    for(int i=1; i<=n; i++)
        s=s+i;
    return s;
}
```

(a)

```
int  sum (int  n){
    int tot; // identifier renaming
    tot=0;
    for(int i=1; i<=n; i++){
      tot=tot+i;
     }
    return tot;
 }
```

(b)

Figure 2.2: Type 2 Clones

2.1.3 Representation of Clone Relationships

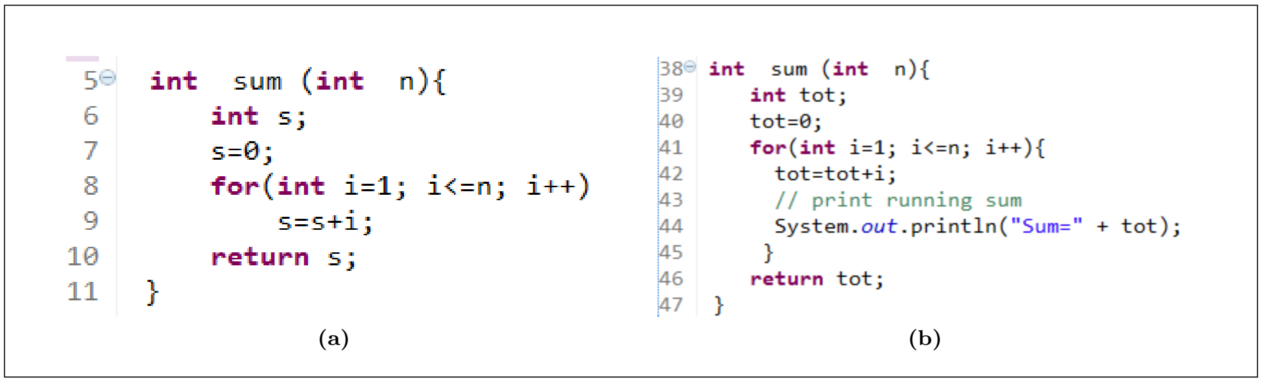
Clone relationships among the code fragments are represented as clone pairs or clone classes defined as follows:

- **Clone Pair**: A pair of code fragments that are clones to each other form a *clone pair*.
- **Clone class:** A set of two or more code fragments that are clones to each other is represented as a *clone class* or *clone family*. In a clone class, any pair of code fragments belongs a clone pair.

2.1.4 Clone Granularity

Detection and analysis of code clones are done at different levels of granularity. The choice of clone granularity depends on the context and the objectives of the analysis. Different granularity levels for code clones are as follows:

- **File Clone:** Two source code files with exact or similar code are called file clones.
- **Class Clone:** When two classes in object-oriented programming (OOP) context contain identical or closely similar code, they are considered as class clones.

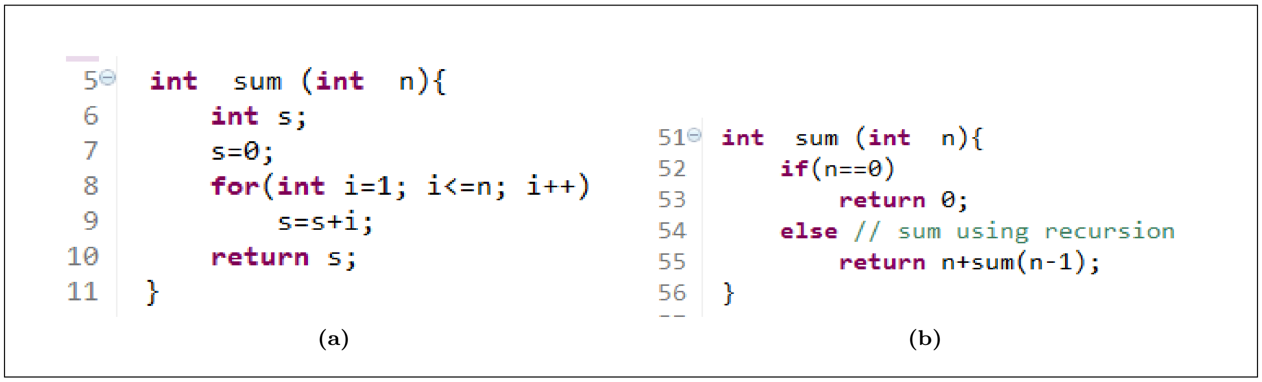

```
int  sum (int  n){
    int s;
    s=0;
    for(int i=1; i<=n; i++)
        s=s+i;
    return s;
}
```

(a)

```
int  sum (int  n){
    int tot;
    tot=0;
    for(int i=1; i<=n; i++){
      tot=tot+i;
      // print running sum
      System.out.println("Sum=" + tot);
    }
    return tot;
}
```

(b)

Figure 2.3: Type 3 Clones


```
int  sum (int  n){
    int s;
    s=0;
    for(int i=1; i<=n; i++)
        s=s+i;
    return s;
}
```

(a)

```
int  sum (int  n){
    if(n==0)
        return 0;
    else // sum using recursion
        return n+sum(n-1);
}
```

(b)

Figure 2.4: Type 4 Clones

- **Method of Function Clone:** Two methods of functions with exact or similar code are called method or function clones.
- **Block Clone:** When two arbitrary blocks of code are identical or similar they are called block clones. Blocks are defined with code boundaries marked with opening and closing brackets, indentation or so.
- **Statements Clone:** Two groups of similar statements in arbitrary locations can also be considered as clones.

2.1.5 Code Clone Genealogy

Clone genealogy [76] represents the evolution lineage of a particular clone group or clone fragment over a series of revisions. It is important to detect clone genealogy for the analysis of clone evolution. There are different studies on the detection of clone genealogies [18, 52, 83, 100, 126, 153, 154]. Clone fragments may be created at any revision of software systems. Then these clone fragments may either remain unchanged or may be modified or even be deleted. Depending on the changes, a clone fragment may or may not remain as a clone in the same clone class. It may become a cloned fragment in another clone class or may even become a non-cloned code fragment. As long as a cloned fragment appears as clone in successive revisions, it is considered as *alive*. In each successive revision where a clone fragment is alive, there is a snapshot of that cloned fragment. By clone genealogy, we refer to the set of successive snapshots of a particular clone fragment. Analysis of clone genealogies is important to observe and understand how a particular clone fragment or a clone group changed over time. Clone genealogies are extracted by mapping clones in successive revisions of the software systems.

2.1.6 Benefits of Code Cloning

A number of studies [18, 49, 56, 57, 83, 84, 86] report that code clones do not have negative impacts on software systems. Instead, code cloning may have some important benefits to software development and maintenance [74]. Some benefits of code clones are briefly described below:

Increased Productivity

One of the most common reasons of code cloning is the frequent copy-paste activity by the developers to reuse existing code. Reusing existing code by slightly modifying to meet new functional requirements saves a lot of development time and cost contributing to faster software development. Code reuse by cloning thus results in increased productivity.

Easing Program Comprehension

Code clones are exact or similar copies of code fragments. Understanding one fragments in a clone class thus helps in understanding the code and functionality of other fragments in the same clone class. As clones constitute a considerable proportion of code base, the existence of clones thus can ease program comprehension.

Avoiding Risks

Code cloning might also be a software development strategy. Developers often implement similar functionality by copying existing code and modifying it for adapting new requirements. Reuse of existing tested code reduces the risks of introducing possible errors and inconsistencies than the reimplementation from the scratch.

In addition, code clones may reduce couplings among the components by implementing required functionalities for each module by cloning instead of sharing the same code.

2.1.7 Negative Impacts of Code Clones

Despite some important benefits of code clones, a good number of studies [21, 22, 34, 69, 51, 95, 94, 97, 100, 122, 162, 185] report that code clones have some negative impacts on software development and maintenance. Some key concerns of code clones regarding software maintenance are described below:

Bug Propagation

When clones are created by copy-paste, any unidentified or hidden bug in the original fragment is propagated to all new cloned copies. When these hidden bugs are identified, the necessary fix is also needed to be propagated to all the cloned fragments. This may make software systems more vulnerable due to duplicate or replicated bugs [59] in clones and it is likely to increase efforts and cost for software maintenance.

Inconsistent Change

Code clones may introduce unintentional inconsistencies in the software systems. When one clone fragment in a clone class is changed, corresponding changes may need to be propagated to other clone fragments in the class. If clones are not changed accordingly, possibly because of the developers were unaware of the existence of those clones, software systems may experience unintentional inconsistencies leading to bugs [71]. However, if such cases are identified later and those can be updated accordingly by late propagation of changes [21, 22]. Thus it is very important to keep track of the clones to ensure consistent evolution where necessary.

Higher Change Proneness

When there are duplicate copies of code fragments in the software systems, changes to one of the fragments may trigger changes to similar other fragments to ensure consistency in the system during evolution. Several studies [51, 97] report that cloned code changes more frequently than non-cloned code. Thus, the presence of code clones as they are comparatively less stable than non-cloned code may make software systems more change-prone. This is in turn likely to increase efforts and costs for software maintenance [97, 119].

Moreover, creating duplicate copies of code for same or similar functionalities may increase the size of the code base.

2.1.8 Clone Management

As mentioned earlier, clones constitutes a significant proportion (around 7%-13% [151], or even up to 50% [140]) of code base of the software systems. Given the positive and negative impacts of clones, one common view of the researchers is that clones should be managed. Developers thus must be aware of the existence and evolution of clones to make proper decisions for managing clones. Some of the important activities in clone management are described below:

Clone Detection

To manage code clones in software systems, the very first thing to do is to identify the clones by clone detection. There are a good number of tools [147] for clone detection. Some of the tools are text-based [102, 40, 68, 144], some are token based [20, 50, 73, 95, 179], some are graph based [46, 79, 85], some are tree-based [23, 64, 82] while some tools are hybrid in nature [144]. Clone detection tools can be configured to detect clones for supported program languages, clone types, clone size, and clone granularities [147]. Once detected from the software systems, clones can be analyzed for making proper decision to manage them.

Clone Analysis And Visualization

To have a better insight of clones in order to manage them properly, clones are to be analyzed. The types of the analysis may vary depending on the concerned management objectives. For example, to identify bug-prone clones developers might need to analyze the evolution history to separate which clones are changed in bug-fix commits. Again, for proper management of clones, developers need to be aware of different aspects of clone relations among the code fragments. Tools for visualization of clones [17] and their evolution [155] can provide the developers with the support of summarizing and visualizing different aspects of clones in software systems for managing clones.

Clone Refactoring

Clone refactoring refers to the task of removing clones where possible by merging or unifying clones from a clone class into a single fragment. However, refactoring must not change the functional behavior or integrity of the systems. There are a good number of tools that support clone refactoring [181, 87]. It is desirable that clone refactoring tools provide supports for assessing which clones are suitable for refactoring, may also prioritize and rank clones for refactoring [101, 113, 117].

Clone Tracking

Due to different limitations, not all clones are suitable for refactoring [76]. However, to ensure consistent evolution of clones, there should be support available to keep track of the evolution of clones. The task of keeping track of clone relations in software systems is known as clone tracking [39, 38]. Clone trackers [39] remembers the clone relation and notify developers of changes to any of the clones in a clone class and identify potential candidate clones for change. The developers then can assess the proposed co-change candidates to make decisions for managing the clones.

Other activities regarding code clone management include the identification of clones that are harmful to software systems such as the detection of clones associated with bugs [94].

2.2 Software Change and Impact Analysis

Software systems evolve through a lot of changes during the life-cycle. As the components in software systems are interrelated and interdependent, changes in one component may trigger changes to other related components to ensure the consistency and integrity of the systems. Identification of all related components to be changed, known as change impact analysis (CIA) is an important task for software maintenance and evolution.

2.2.1 Change Impact Analysis

The task to identify the candidate entities for a proposed change is known as Change Impact Analysis (CIA) or simply Impact Analysis (IA). Bohner and Arnold [13] define Impact Analysis as "*identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change*". As changes are obvious in all phases of life-cycle of the software systems, impact analysis plays an important role in software maintenance and evolution by estimating the various costs and risks related to the proposed change before the actual change takes place and provides systematic guidance for change propagation. In this thesis, we focus on the change impacts analysis of clones only to support the consistent evolution of clones during clone management.

2.2.2 Formal Definition of Change Impact Analysis

Ajila [9] defined change impact analysis more precisely in the context of object oriented paradigm. According to his definition, for a system S represented by a set of objects $O_S = \{o_1, o_2, \cdots, o_n\}$ with the set of change types $T_S = \{t_1, t_2, \cdots, t_m\}$ to be carried out on the objects of S, impact for a given change $\{t_i, o_j\}$ is given by

$$f_{impact}\{t_i, o_j\} \rightarrow \{o_1, \cdots, o_i, \cdots, o_k, \cdots\}$$

Where the impact set $\{o_1, \cdots, o_i, \cdots, o_k, \cdots\}$ includes only the objects directly affected by the change t_i applied to object o_j. However, when required changes are propagated to the objects in the impact set $\{o_1, \cdots, o_i, \cdots, o_k, \cdots\}$, some other objects may need to be changed due to ripple effects. Thus, Ajila extended his definition to include these indirectly affected objects due to change propagation as

$$f_{impact}\{t_i, o_j\} \rightarrow \{o_1(o_{11}, o_{12}, \cdots, o_{1n}), \cdots, o_i(o_{i1}, o_{i2}, \cdots, o_{in}), o_k(o_{k1}, o_{k2}, \cdots, o_{kn}), \cdot\cdot\}$$

This definition implies that impact analysis determines the set of all objects directly or indirectly affected by a proposed set of changes. Although this definition is in the object oriented context, it can be can be easily extended by generalizing objects as any other software artifacts.

2.2.3 CIA Approaches

Depending on the types of software artifacts CIA methods are dealing with, change impact analysis methods are broadly grouped into two categories: *static impact analysis* and *dynamic impact analysis.* Static impact analysis deals with artifacts like source code and design documents for static analysis of dependencies [19, 27]. On the other hand, dynamic analysis is concerned with dynamic program behavior and it analyzes execution traces for impact analysis [128, 91, 12]. However, some *hybrid* approaches combine both static and dynamic analysis for change impact analysis [187, 55, 26].

Li et al. [93] in their survey on code-based change impact analysis, considered 30 publications with 23 different techniques. They grouped the impact analysis techniques into four broad categories listed below:

- *Traditional dependency analysis:* This group of impact analysis techniques carry out static program analysis to extract dependency relationships among the program components [19, 27, 164]. The relationships among the program components are represented as dependency graph (e.g., call graph) and change impacts are identified based on the reachability of graph.

- *Execution information collection:* Dynamic information collected during program execution is useful to describe the run-time behavior of the system. Some dynamic impact analysis techniques [128, 91, 12] use execution trace, coverage information to identify dependency relation to compute the impact set corresponding to a changed entity.

- *Software repository mining:* This category of impact analysis techniques [190, 29, 63] analyze the evolution history to identify the dependency relationships among the program components. Source code repositories contain information that describes the evolutionary dependency among the program components which cannot be extracted by traditional static analysis techniques. Mining software repository can identify co-changed entities to compute change impacts.

- *Coupling measurement:* There exist different types of coupling among the program elements such as structural coupling and conceptual coupling among the program elements. This group of techniques [28, 132] identify impacts of change by using the coupling values between a changed entity and other system entities.

2.2.4 Concepts of Evolutionary Couplings

Evolutionary Coupling

When two or more entities (e.g, files, classes, methods) in a software system co-change frequently (in many revisions) during evolution, these entities are said to have *evolutionary coupling*, also known as change coupling [47]. The existence of evolutionary coupling among the software entities is an indication of dependencies among the associated entities and changes to one such entity may introduce corresponding changes to other coupled entities in the software system. As evolutionary coupling reveals the underlying relationships among software components, it helps in Change Impact Analysis (CIA) by identifying co-change candidates when a related entity is changed during evolution [189].

Association Rule

An association rule is an expression of the form $X \Rightarrow Y$ where X is called the antecedent and Y is called the consequent of the association rule. X and Y are sets of one or more software entities. In the context of our study, an association rule is interpreted as *"if method X is changed in a revision, Y is likely to be changed in the same revision"*. The strength of this likelihood is expressed in terms of the *support* and *confidence* of the association rule defined below.

Support and Confidence

Support is defined as the number of commits (revisions) an entity or a set of entities were changed [189]. For example, let two co-changed methods $M1$ and $M2$ where set of revisions method $M1$ was changed in is $R1 = \{3, 4, 7, 15\}$. On the other hand, set of revisions method $M2$ was changed in is $R2 = \{4, 6, 7, 15, 17\}$. Here, $support(M1) = |R1| = 4$, the cardinality of $R1$. Similarly, $support(M2) = |R2| = 5$. However, $support(M1, M2) = |R1 \cap R2| = |\{4, 7, 15\}| = 3$, the number of revisions $M1$ and $M2$ have co-changed and thus $support(M1, M2) = support(M2, M1)$. For co-changed methods $M1$ and $M2$, we can have two association rules: $M1 \Rightarrow M2$ and $M2 \Rightarrow M1$. Again, support of an association rule is defined as the number of revisions (commits) where the antecedent and consequent changed together (i.e, co-changed). For an association rule $X \Rightarrow Y$,

$$support(X \Rightarrow Y) = support(X, Y)$$

In our example, $support(M1 \Rightarrow M2) = support(M1, M2) = 3$. *Confidence* of any association rule $X \Rightarrow Y$ is defined as,

$$confidence(X \Rightarrow Y) = support(X, Y)/support(X)$$

2.3 Summary

In this chapter, we have discussed the basic concepts of code clones and defined the key terminologies used in this thesis. Some of the concepts are further elaborated in detail within each chapter where necessary. Having the basic concepts presented for the readers, we present our research studies and findings in the following chapters.

Chapter 3

Change Type Based Analysis of the Stability of Cloned Code

This chapter presents an empirical study on the comparative stability of cloned and non-cloned code from the perspective of different syntactic change types. Although it is obvious that not each type of changes has equal impact on software systems, none of the existing studies take the types of changes and their significance into account during comparative evaluation of stability of cloned and non-cloned code. We consider fine-grained syntactic changes types for the analysis of the comparative stability of cloned and non-cloned code. We extract changes from successive revisions using *ChangeDistiller* which employs Abstract Syntax Tree (AST) differencing of the successive revisions of source code and assigns the corresponding level of significance to each of the classified changes. We detect exact (Type 1) and near-miss (Type 2 and Type 3) clones using the hybrid clone detection tool NiCad. Extracted and classified changes and clone information are then analyzed to compare the stability of cloned and non-cloned code from three different perspectives: types of clones, types of changes with respect to the significance of changes, and size and extent of evolution of the systems. Our study on seven open-source Java systems with diversity in their size, length of evolution and application domain shows that changes are more frequent in cloned code than in non-cloned code and Type 1 clones are comparatively more vulnerable to the stability of the systems. Therefore, cloned code is less stable than non-cloned code suggesting that cloned code is likely to pose more maintenance challenges than non-cloned code. The findings of this study have been published in SCAM 2014[1].

The rest of this chapter is organized as follows: Section 3.2 outlines the important motivation of the study. Section 3.3 briefly describes the taxonomy of changes used in this study. Section 3.4 represents the experimental settings and steps used in the study including the subject systems used, change extraction and classification procedure, clone detection and the metrics we measure. Section 3.5 represents the experimental results and analysis. Threats to the validity of the study are represented in section Section 3.6. Section 3.7 discusses the related works followed by the conclusion in Section 3.8.

[1]M. S. Rahman, Chanchal K. Roy. "A Change Type-Based Empirical Study on the Stability of Cloned Code". In Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2014, pp. 31–40.

3.1 Introduction

Code reuse by copy-paste is a common practice in software development, and as result software systems often have sections of code that are identical or similar, called software clones or code clones. Clones constitute a significant fraction of code (between 7% and 23% [151] or sometimes even 50% [140]). Regardless of the intentional and unintentional reasons behind code cloning, the impact of clones on software maintenance has been a great concern [148]. Although it is believed that code cloning speeds up software development and facilitates the reuse of mature and tested code, clones are often accused of introducing maintenance challenges by making consistent changes more difficult leading to the introduction of bugs [70], propagation of existing bugs and thus resulting in increasing maintenance efforts [97]. Having both positive and negative impacts on software maintenance, code cloning has been under significant research focus towards the evaluation of the impacts of clones on software maintenance and evolution. Some of the existing studies concluded in favour of clones suggesting that clones are not harmful [18, 56, 84, 86], rather, cloning can be beneficial to software development [74]. A good number of studies, on the other hand, concluded that clones have negative impacts on software maintenance and evolution [71, 97, 56, 100].

One of the well-studied perspectives of the evaluation of the comparative impacts of cloned and non-cloned code on software maintenance and evolution is the *stability.* Stability measures the extent to which cloned and non-cloned code regions remain unchanged. Code regions that change less frequently are more stable and thus require less maintenance effort. Krinke [84] measured the comparative stability of clone and non-clone code by measuring the volume of code change, *i.e.,* by counting the number of lines added, deleted and changed to cloned and non-cloned code. His study concludes that clone code is more stable than non-clone code. Göde and Harder [49] extended Krinke's study [84] with token-based incremental clone detection tool and experimented with different settings of clone detection parameters, clone types and change operations. Their study agrees with the findings of Krinke's study. Krinke's other study [86] measures stability in terms of the comparative age of the cloned and non-cloned code based on the average last change date. This study agrees with his previous findings that cloned code is more stable. Hotta *et al.* [56] measured the modification frequency of cloned and non-cloned code to evaluate their comparative stability. This study concludes that cloned code is less frequently modified than non-cloned code, *i.e.,* cloned code is more stable. Lozano and Wermelinger [97, 100] conducted studies on assessing the impacts of clones and they concluded that clones are harmful to software maintenance because they often increase the maintenance efforts and also they are vulnerable to the stability of the software systems. Their findings disagree with the findings of Hotta *et al.* [56], Krinke [84] and Göde and Harder [49].

Because of these contradictory findings in earlier studies Mondal *et al.* [123, 122] carried out stability analysis within a uniform evaluation framework considering the existing metrics (mostly taken from [56, 84, 86]) and their proposed metrics to measure stability. Their study concluded that a firm conclusion cannot be drawn on the stability of cloned and non-cloned code, rather, the comparative stability varies with programming languages, types of clones and overall system development strategies. However, none of the metrics used or proposed in their study consider actual syntactic change types and the different levels of the significance of changes.

As it is obvious that different types of changes have different impacts on change propagation, it is important to consider the distinct change types in evaluating the impacts of clones on maintenance. Considering the differences in the impacts of different change types, this study carries out an empirical evaluation of the impacts of clones on software maintenance. In particular, we investigate the frequency of changes in cloned and non-cloned code considering fine-grained change types and their levels of significance. In our empirical study, we consider a comprehensive taxonomy of source code change proposed by Fluri and Gall [43]. This taxonomy assigns a level of significance to each type of source code change where *significance* is defined as how strong impact a change may have on other source code entities. In this taxonomy there are four different levels of change significance: *low, medium, high,* and *crucial* where *low* is the lowest level and *crucial* is the highest level of significance respectively.

We extract fine-grained changes from all successive revisions of software systems using *ChangeDistiller* [43] which employs Abstract Syntax Tree (AST) differencing between two consecutive revisions of source code files. One important advantage of the AST-based code differencing over UNIX *diff* is that AST-based differencing considers the syntactic context of the code change while *diff* considers code merely as text. In addition, the results of *diff* might be sensitive to the formatting of the source code. AST-based code differencing, on the other hand, gives more fine grained changes of source code artifacts [43]. Extracted and classified changes from *ChangeDistiller* are mapped to the source code entities and stored for further analysis. We detect both exact (Type 1) and near-miss (Type 2 and Type 3) clones of all the revisions of the systems using the hybrid clone detection tool NiCad [144, 37]. The classified changes are then mapped to the cloned and non-cloned code. We then measure the frequency metrics which we define for assessing the comparative impacts of clones on software maintenance with respect to the levels of significance of changes to cloned and non-cloned code.

In particular, we evaluate the comparative stability of cloned and non-cloned code with respect to different levels of significance of changes by answering the following research questions:

RQ1 *To what extent different types of clones exhibit different stability scenarios?*

RQ2 *Do the changes of different levels of significance show different stability scenarios?*

RQ3 *Is the stability of cloned and non-cloned code system dependent?*

Changes of the higher level of significance are likely to have a higher change impact. Therefore, the higher the frequency of changes of a particular level of significance in the code, the higher maintenance challenges it is likely to pose during evolution.

From this empirical study considering fine-grained change types and their different levels of significance we have the following findings:

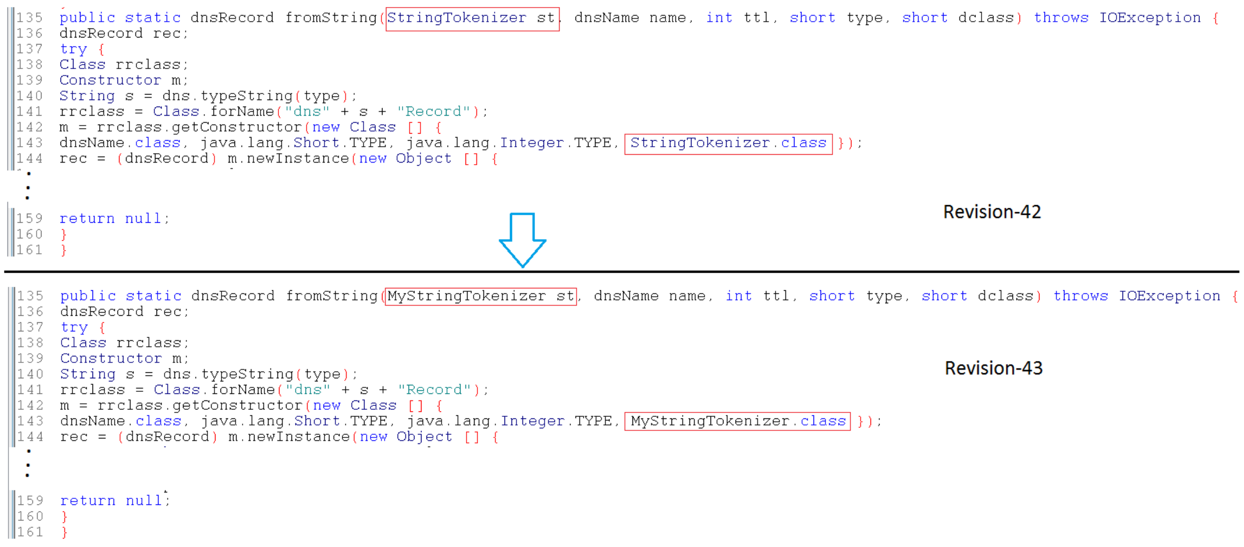
(i) Cloned code is less stable than non-cloned code and Type 1 clones are comparatively more vulnerable to the stability of the system. (ii) Stability of cloned code is mostly affected by the changes of lower (*low, medium*) levels of significance. (iii) System size and length of evolution do not have significant effects on stability. Moreover, our fine-grained analysis gives important insights into the better management of clones.

3.2 Motivation

This study is inspired by two previous studies by Krinke [84] and Hotta *et al.* [56] both of which measure the comparative stability of cloned and non-cloned code. Krinke defines *instability* of cloned and non-cloned code in terms of the number of added, deleted and changed lines with respect to the total number of lines in cloned and non-cloned code respectively. A higher value of instability indicates that the code region is less stable *i.e.,* changes more frequently. Although his method considers the volume of change in terms of lines of code modification to measure instability, it does not consider the actual syntactic change types.

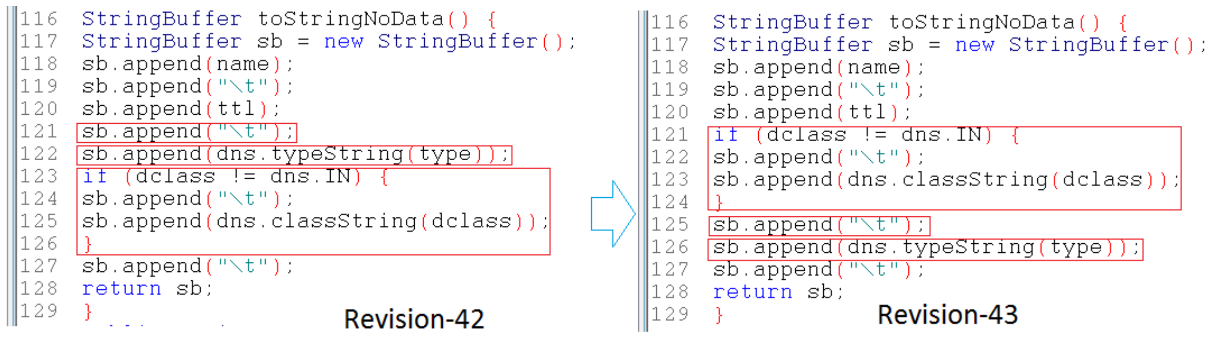
Hotta *et al.* [56], on the other hand, measure the stability of cloned and non-cloned code in terms of *modification frequency.* Their approach counts the number of regions (blocks of consecutive lines of code) modified in cloned and non-cloned code. The average modification counts per revision in cloned and non-cloned code are then used to measure the modification frequency by scaling the modification count to the ratio of cloned and non-cloned LOC (lines of code) to the total LOC. The less the modification frequency, the more stable the code region is. One of the major limitations of this approach is that it ignores the volume of change (as opposed to Krinke [84]), neither does it consider the types of actual syntactic changes. This is likely to affect the accuracy and reliability of the stability results. Thus, the measure of the stability of the code should consider the volume of change. And it is reasonable to assume that this stability measured in terms of frequency of changes represents the impact of changes on maintenance more precisely only when the actual syntactic types of changes are taken into consideration. This is because different types of syntactic changes have different impacts (likelihood of affecting other entities) from the perspective of change propagation [43].

To have a deeper understanding of the aforementioned problems let us consider the two example change scenarios as shown in Figure 3.1 and Figure 3.2 as observed in revision 43 of DNSJava. In the first case, the method `fromString()` in file `DNS/Record.java` has two changes from revision 42 to revision 43. The first change is a *parameter type change* (`StringTokenizer` to `MyStringTokenizer`) which is of *crucial* significance because all the callers of the method need the corresponding *parameter type change* to be propagated. The second change is of type *statement update* with *low* significance as the effect of the change is local to the method.


```
public static dnsRecord fromString(StringTokenizer st, dnsName name, int ttl, short type, short dclass) throws IOException {
dnsRecord rec;
try {
Class rrclass;
Constructor m;
String s = dns.typeString(type);
rrclass = Class.forName("dns" + s + "Record");
m = rrclass.getConstructor(new Class [] {
dnsName.class, java.lang.Short.TYPE, java.lang.Integer.TYPE, StringTokenizer.class });
rec = (dnsRecord) m.newInstance(new Object [] {
:
return null;
}
}
```

```
public static dnsRecord fromString(MyStringTokenizer st, dnsName name, int ttl, short type, short dclass) throws IOException {
dnsRecord rec;
try {
Class rrclass;
Constructor m;
String s = dns.typeString(type);
rrclass = Class.forName("dns" + s + "Record");
m = rrclass.getConstructor(new Class [] {
dnsName.class, java.lang.Short.TYPE, java.lang.Integer.TYPE, MyStringTokenizer.class });
rec = (dnsRecord) m.newInstance(new Object [] {
:
return null;
}
}
```

Figure 3.1: Changes in method `fromString()` in file `DNS/Record.java` in revision-43 of DNSJava


```
StringBuffer toStringNoData() {
StringBuffer sb = new StringBuffer();
sb.append(name);
sb.append("\t");
sb.append(ttl);
sb.append("\t");
sb.append(dns.typeString(type));
if (dclass != dns.IN) {
sb.append("\t");
sb.append(dns.classString(dclass));
}
sb.append("\t");
return sb;
}
```

```
StringBuffer toStringNoData() {
StringBuffer sb = new StringBuffer();
sb.append(name);
sb.append("\t");
sb.append(ttl);
if (dclass != dns.IN) {
sb.append("\t");
sb.append(dns.classString(dclass));
}
sb.append("\t");
sb.append(dns.typeString(type));
sb.append("\t");
return sb;
}
```

Figure 3.2: Changes in method `toStringNoData()` in file `DNS/Record.java` in revision-43 of DNSJava

From our manual analysis of clones, we see that this method has three Type 1 (exact) clones in files `org/xbill/DNS/dnsRecord.java`, `org/xbill/DNS/Record.java` and `DNS/dnsRecord.java`. All of the three clones have been updated consistently with the two above changes. Again, the method `fromString()` is called by the method `doAdd (dnsMessage query, StringTokenizer st)` in `update.java` and the method `doAdd()` is again called by the method `main()` in `update.java`. So, the first change introduces another *parameter type change* in `doAdd()` which is of *crucial* significance and also it requires one *statement update* in the method `doAdd()`. Therefore, the first change introduces three changes (*crucial*) to the three clones and one parameter change (*crucial*) and one statement update to the caller `doAdd()`, and one statement update change (*low*) to the `main()` method. So, the total number of changes triggered by the first change is six (3+2+1) while the second change introduces only three (1+1+1) changes of low significance to the three clones. Based on the depth of method call chain, the impact of changes can be even more in some other cases. This example makes it evident that not all changes pose equal challenges in the maintenance process.

Again, in Figure 3.2 method `toStringNoData()` in `DNS/Record.java` has three changes in revision 43. All these changes are *statement reordering* in the method body with *low* significance because the impact of this type of change is local to the changed method. However, by investigating clones and source code we see that these changes have been consistently propagated to three cloned methods of the method `toStringNoData()` in source files `org/xbill/ DNS/Record.java`, `DNS/dnsRecord.java` and `org/ xbill/DNS/dnsRecord.java`. Thus, each of the changes in `toStringNoData()` introduces one corresponding change in the three Type 1 clone fragments. Now, we see that although the method `toStringNoData()` has more changes (three) than the number of changes (two) in `fromString()` in the first example, method `fromString()` poses higher maintenance cost than `toStringNoData()` as it introduces more changes due to having the change with higher levels of significance. Thus, from the above examples we could draw the following conclusions:

(i) As different change types have different levels of significance, *i.e.,* different degrees of likelihood of affecting other entities, the significance levels of different types of code changes should be taken into account while comparing the impacts of changes in cloned and non-cloned code on software maintenance.

(ii) Because of having different levels of significance, changes of different types should be considered as separate as possible while measuring the frequency of changes in code (*i.e., stability*) in order to have more precise results of comparative evaluation of the stability of cloned and non-cloned code.

Although the existing studies give important insights regarding the impacts of clones on software maintenance, one important limitation is that none of the existing studies addressed the two important points mentioned above. Thus, the existing stability metrics [56, 84] have limitations from the perspectives of reliability and precision.

Our proposed study incorporates the actual change types and their levels of significance in measuring the frequency of changes in cloned and non-cloned code. Our study considers the type-wise relative volume of changes and their frequencies in cloned and non-cloned code for the comparative stability analysis. This enables us to comparatively evaluate the impacts of clones on maintenance in terms of stability from a more reliable and precise point of views.

3.3 Taxonomy of Software Change at Method Level

Software systems evolve through different kinds of changes. Each type of change refers to a particular syntactic context of the change. Again, each type of changes affects the software systems from different functional and structural contexts. For our study, we consider the taxonomy of change proposed by Fluri and Gall [43] which is a comprehensive taxonomy for fine-grained source code change and it defines the significance levels of changes. As our clone analysis is at method level granularity, we consider only the method level changes in the taxonomy proposed by Fluri and Gall. The taxonomy of changes used in this empirical study is presented in Table 3.1.

Changes to individual methods are referred to as method-level changes. Changes to methods are further divided into two groups, changes to method declaration and changes to the method body, based on where the changes occur in the methods. Changes to method declaration part (*method signature*) include changes in accessibility, overridability, method renaming, parameter change and changes to return type. Changes to parameters include addition, deletion, renaming, reordering and parameter type change. Changes to *method body* comprise changes to statements and structure statements (*e.g.,* loop, branching). Statements might be added, deleted, modified and reordered. Each of these fine-grained change types is assigned a level of significance based on their likelihood of affecting other code entities and the extent they modify the functionality of the system as proposed by Fluri and Gall [43].

3.4 Experimental Setup

This section outlines the experimental settings for different components and steps of our empirical study including preprocessing of subject systems, clone detection, change extraction and classification and the metrics measured.

3.4.1 Subject Systems

This study is based on seven open source systems implemented in Java with diversified size, evolution history and application domain. Table 3.2 briefly represents the features the software systems including the application domain, length of evolution history, size in lines of code (LOC) and the total number of revisions. The size of the systems represents the lines of code (LOC) in the last revision of the systems counted after removal of comments and pretty-printing. In selecting subject systems we include some systems used in previous studies (Openymsg, Squirrel) by Hotta *et al.* [56] and Krinke [84] (ArgoUML) to have a better comparative analysis of the findings.

3.4.2 Metrics Measured

To compare the stability of cloned and non-cloned code, we measure the frequency of changes to cloned and non-cloned code with respect to different levels of significance of changes and types of clones. The frequency metrics represent how frequently the cloned and non-cloned code encounter changes. To define the frequency metrics we assume that:

- $R = \{r_1, r_2, r_3, ..., r_n\}$ is the set of revisions, and
- $S = \{low, medium, high, crucial\}$ is the set of possible levels of change significance

Now, we define the number of changes in cloned and non-cloned code with significance level s in revision r as $CC_{c_s}(r)$ and $CC_{n_s}(r)$ respectively.

Table 3.1: TAXONONY OF METHOD-LEVEL CHANGES WITH SIGNIFICANCE (adapted from [43])

Change Level	Changed Part	Change Group	Change Type	Significance
Method	Declaration	Accessibility Change (MAC)	Accessibility Increase (MAI)	Medium
			Accessibility Decrease (MAD)	High
		Overridability Change (MOC)	Add Method Overridability *final* (AMO)	Crucial
			Delete Method Overridability (DMO)	Low
		Parameter Change (MPC)	Parameter Insert (PI)	Crucial
			Parameter Delete (PD)	Crucial
			Parameter Ordering (PO)	Crucial
			Parameter Renaming (PR)	Medium
			Parameter Type Change (PTC)	Crucial
		Method Identifier Change (MIC)	Method Renaming (MR)	High
		Return Type Change (RTC)	Return Type Insert (RTI)	Crucial
			Return Type Delete (RTD)	Crucial
			Return Data Type Change (RDC)	Crucial
	Body	Statement Change (SC)	Statement Insert (SI)	Medium
			Statement Delete (SD)	Medium
			Statement Update (SU)	Low
			Statement Re-ordering (SO)	Low
		Structure Statement Change (SSC)	Condition Expression Change (CEC)	Medium
			Statement Parent Change (SPC)	Medium
			Alternative- part (*else*) Insert (API)	Medium
			Alternative- part (*else*) Delete (APD)	Medium

Table 3.2: SUBJECT SYSTEMS

Systems	Type	Size (LOC)	Evolution Period	#Revision
JabRef	Bibliography Manager	153952	OCT 2003 - NOV 2011	3718
DNSJava	DNS Protocol	20831	SEP 1998 - FEB 2013	1679
OpenYMSG	Open Messenger	8821	MAR 2007 - MAR 2013	233
Ant-Contrib	Web Server	79434	JUL 2006 - MAR 2009	177
Carol	Driver Application	13213	JUN 2002 - SEP 2013	2237
ArgoUML	UML Modeling Tool	157573	JAN 1998 - JUN 2014	19915
Squirrel	SQL Client	332635	JUN 2001 - JAN 2013	6737

We then define the following two frequency metrics in terms of the average number of changes to cloned and non-cloned code with a particular level of change significance as follows:

(i) The frequency of changes to cloned code (CF_c) is measured as

$$CF_c = \frac{\sum_{r\epsilon R, s\epsilon S} CC_{c_s}(r)}{|R|} \times \frac{\sum_{r\epsilon R} LOC(r)}{\sum_{r\epsilon R} LOC_c(r)} \tag{3.1}$$

(ii) The frequency of changes to non-cloned code (CF_n) is measured as

$$CF_n = \frac{\sum_{r\epsilon R, s\epsilon S} CC_{n_s}(r)}{|R|} \times \frac{\sum_{r\epsilon R} LOC(r)}{\sum_{r\epsilon R} LOC_n(r)} \tag{3.2}$$

In equations Equation 3.1 and Equation 3.1,

- $LOC_c(r)$, $LOC_n(r)$ and $LOC(r)$ represent the cloned, non-cloned and total lines of code in revision r respectively.
- the terms $\frac{\sum_{r\epsilon R, s\epsilon S} CC_{c_s}(r)}{|R|}$ and $\frac{\sum_{r\epsilon R, s\epsilon S} CC_{n_s}(r)}{|R|}$ represent the average number of changes per revision with significance s to cloned and non-cloned code respectively.

As the frequencies of changes to cloned and non-cloned code are sensitive to $LOC_c(r)$ and $LOC_n(r)$, we normalize the change frequencies by multiplying with the ratio of $LOC(r)$ to $LOC_c(r)$ and $LOC_n(r)$ for the cloned and non-cloned code respectively in equations (1) and (2).

Our proposed metrics are similar to the metrics proposed by Hotta *et al.* [56], but different from the following two points:

(i) Metrics proposed by Hotta *et al.* [56] count changes in a range of consecutive lines of code as a single change whereas we count each fine-grained change as a single change, and

(ii) We consider the significance levels of changes and measure metrics for four levels of significance of changes separately.

3.4.3 Experimental Steps

We analyze the frequency of changes to cloned and non-cloned code by the following six steps:

Preprocessing

For the proposed analysis of the frequencies of changes in cloned and non-cloned code, we first extract all the revisions of the subject systems from their corresponding SVN repository. As our goal is to analyze the changes to the source code we remove comments from the source files. Pretty-printing of the source files is then carried out to eliminate the formatting differences using the tool *ArtisticStyle*[2].
We extract the file modification history using *SVN diff* to list added, modified and deleted files in successive revisions. This information is used to exclude the unchanged files during change analysis to speed up the process.

Method Extraction and Origin Analysis

To analyze the changes to methods throughout all the revisions, we extract method information from the successive revisions of the source code. We store the method information in a database to use for mapping changes to the corresponding methods. Again, SVN keeps track of the files that are added, deleted or modified and the history of changes to individual file content are preserved as the modification of lines. This line-level change information is not sufficient to describe the evolution of source code entities at higher granularity levels such as classes or methods. As a result, to map changes to methods throughout the development cycle, we need to map the methods acyross the revisions. Therefore, we carry out origin analysis [97] of the methods on the revisions of the systems to gather mapping information for methods and preserve in a database. This information is used to map the classified changes and cloning information back to the corresponding methods to measure the frequency of changes.

Change Extraction and Classification

The change analysis system in this study is implemented in Java based on the change extraction and classification core of *ChangeDistiller* [43]. The system imports copies of changed files from successive versions and uses JDT API of Eclipse for the extraction of methods and extracting the differences between the copies of each of the files in any two successive revisions. The details of change extraction and classification are as follows:

- *Change Extraction:* Code changes are extracted using *ChangeDistiller* classifier. *ChangeDistiller* extracts changes by taking differences between two versions of ASTs of the same file and are stored as a sequence of tree-edit operations. The generic operations contain insert, delete, move and update operations on the

[2]http://astyle.sourceforge.net/

Table 3.3: NiCad SETTINGS FOR THE STUDY

Parameters	Values
Minimum Size	5 lines
Maximum Size	500 lines
Granularity	Method Level
Threshold	0% (Type 1, Type 2), 30% (Type 3)
Identifier Renaming	blindrename (Type 2, Type 3)

nodes in the AST. The tree-edit operations encoded as edit scripts are then processed by *ChangeDistiller* to classify extracted changes to fine-grained change types. We have customized the *ChangeDistiller* classifier to suit for analyzing local repository exported from SVN. To extract source code changes, two successive versions of the same file are selected from the source repository and then are passed to the differencing engine of the change classifier. The extracted changes are then passed to the classifier for classification. The process is repeated for all changed files (identified by SVN *diff*) and for all the revisions of the subject systems.

- *Change Classification:* Changes extracted by AST-differencing of two successive revisions of source code files are classified into fined-grained changes to source code entities. Changes are classified according to the defined taxonomy and are assigned the corresponding levels of significance. For the analysis, classified changes are mapped to the corresponding source code entities based on the information extracted during the origin analysis.

Mapping Change Data

After classification, the classified changes are mapped to their corresponding source code entities (methods) with the help of extracted origin mapping information for the associated entities. We preserve the extracted, classified and mapped changes into a database to measure metrics for the frequency of changes at the method level granularity.

Clone Detection

For this study we use the hybrid clone detection tool NiCad [144]. NiCad is reported to have a higher level of precision and recall [145] and supports the detection of Type 1 (exact copies), Type 2 (syntactically exact with identifier naming differences) and, Type 3 clones (in addition to Type 2 differences lines are added, deleted or modified) clones. We run NiCad on all revisions of the subject systems to detect clones at method level granularity. The clone detection results are then processed to store the clone information in the database. Table 3.3 lists the parameter settings for NiCad used for this study.

Measurement of Metrics and Comparison

We calculate the defined metrics from the extracted changes and clone data stored in the database using equations (1) and (2). For all the subject systems, we measure the frequency metrics for all types of clones (Type 1, Type 2 and Type 3) and for all the four levels of significance of changes. Higher values of frequency metrics refer to lower stability of the corresponding subject system. The metrics are then analyzed for the comparative evaluation of the impacts of clones on maintenance in terms of the frequency of changes to cloned and non-cloned code.

3.5 Results and Analysis

This section represents the results of our empirical study by answering three research questions we defined in Section 3.1. We evaluate the stability of clones by analyzing the values of the metrics for the frequency of changes in cloned and non-cloned code. We calculated the values of the metrics for all seven subject systems, for each of the three clone types (Type 1, Type 2 and Type 3) and for each of the four levels of significance (*low, medium, high and crucial*) of changes.

Table 3.4 represents the comparative frequencies of changes in cloned and non-cloned code for Type 1 clones in the seven subject systems considered. Values for the metrics calculated for the frequency of changes in cloned and non-cloned code considering each of the levels of change significance are represented in the corresponding columns in the table. Similarly, Table 3.5 and Table 3.6 represent the comparative frequencies of changes in cloned and non-cloned code for Type 2 and Type 3 clones respectively for all seven subject systems.

Table 3.7 represents the summary of the stability scenarios of cloned and non-cloned code. This table is derived from Table 3.4, Table 3.5, and Table 3.6. The symbol '⊛' in Table 3.7 indicates that the frequency of changes in cloned code is higher than the frequency of changes in non-cloned code ($CF_c > CF_n$). This implies that cloned code is less stable as compared to non-cloned code for the corresponding subject system, type of clones and the level of significance of changes. A '⊙' symbol, on the other hand, refers to a case where the frequency of changes in cloned code is less than the frequency of changes in non-cloned code ($CF_c < CF_n$). This indicates that cloned code is more stable as compared to non-cloned code for the corresponding subject system, type of clones and the level of significance of changes. This table represents the 84 (7x3x4) decisions points regarding comparative stabilities of cloned and non-cloned code for seven subject systems, three types of clones and four different levels of the significance of changes. Based on the analysis of these decision points we answer the research questions in the following sections.

Table 3.4: COMPARATIVE FREQUENCIES OF CHANGES OF DIFFERENT LEVELS OF SIGNIFICANCE FOR TYPE 1 CLONES.

Change Significance→	Low		Medium		High		Crucial	
Systems ↓	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n
OpenYmsg	197.16	6.34	96.74	4.86	2.95	0.15	0.74	0.28
DNSJava	203.60	25.47	183.13	25.15	10.98	3.07	8.26	2.46
JabRef	36.78	12.59	18.78	10.09	0.06	0.14	0.12	0.74
Carol	10.73	14.47	14.44	11.13	0.80	0.64	1.20	0.81
Ant-Contrib	1.13	0.30	0.93	0.21	0.02	0.02	0.01	0.01
ArgoUML	130.74	25.91	81.13	16.78	0.96	0.61	1.80	1.13
Squirrel	14.30	6.94	10.78	4.24	0.30	0.26	0.54	0.49

CF_c=Frequency of Changes in Cloned Code

CF_n=Frequency of Changes in Non-cloned Code

Table 3.5: COMPARATIVE FREQUENCIES OF CHANGES OF DIFFERENT LEVELS OF SIGNIFICANCE FOR TYPE 2 CLONES.

Change Significance→	Low		Medium		High		Crucial	
Systems ↓	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n
OpenYmsg	16.64	7.30	12.74	5.28	0.26	0.17	0.26	0.28
DNSJava	44.44	43.32	41.33	41.00	5.61	3.82	2.15	3.07
JabRef	19.05	15.92	11.04	11.31	0.06	0.13	1.66	0.63
Carol	45.39	13.13	36.59	10.16	0.66	0.64	0.42	0.84
Ant-Contrib	0.37	0.71	0.47	0.55	0.00	0.02	0.00	0.01
ArgoUML	68.23	26.42	47.00	16.87	0.49	0.63	2.84	1.06
Squirrel	10.28	7.47	7.25	4.70	0.16	0.27	0.90	0.47

CF_c=Frequency of Changes in Cloned Code

CF_n=Frequency of Changes in Non-cloned Code

Table 3.6: COMPARATIVE FREQUENCIES OF CHANGES OF DIFFERENT LEVELS OF SIGNIFICANCE FOR TYPE 3 CLONES.

Change Significance→	Low		Medium		High		Crucial	
Systems ↓	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n	CF_c	CF_n
OpenYmsg	10.39	7.19	9.17	5.07	0.00	0.18	0.00	0.30
DNSJava	52.03	42.48	48.38	40.27	10.26	3.23	2.73	3.07
JabRef	36.16	14.31	21.96	10.42	0.17	0.13	1.40	0.59
Carol	33.24	11.70	29.31	8.60	0.47	0.67	0.97	0.80
Ant-Contrib	2.43	0.35	2.31	0.21	0.03	0.02	0.01	0.01
ArgoUML	79.70	20.56	53.34	12.95	0.42	0.65	1.77	1.05
Squirrel	12.08	6.44	8.07	4.00	0.21	0.28	0.62	0.47

CF_c=Frequency of Changes in Cloned Code

CF_n=Frequency of Changes in Non-cloned Code

Table 3.7: COMPARATIVE STABILITY OF CLONED AND NON-CLONED CODE

Significance→	Low			Medium			High			Crucial		
Clone Types→ Systems ↓	1	2	3	1	2	3	1	2	3	1	2	3
OpenYmsg	⊛	⊛	⊛	⊛	⊛	⊛	⊛	⊛	⊙	⊛	⊙	⊙
DNSJava	⊛	⊛	⊛	⊛	⊛	⊛	⊛	⊛	⊛	⊛	⊙	⊙
JabRef	⊛	⊛	⊛	⊛	⊙	⊛	⊙	⊙	⊛	⊙	⊛	⊛
Carol	⊙	⊛	⊛	⊛	⊛	⊛	⊛	⊛	⊙	⊛	⊙	⊛
Ant-Contrib	⊛	⊙	⊛	⊛	⊙	⊛	⊙	⊙	⊛	⊙	⊙	⊙
ArgoUML	⊛	⊛	⊛	⊛	⊛	⊛	⊛	⊙	⊙	⊛	⊛	⊛
Squirrel	⊛	⊛	⊛	⊛	⊛	⊛	⊛	⊙	⊙	⊛	⊛	⊛

⊛=cases where frequency of changes in cloned code is higher than that in non-cloned code (CF_c>CF_n)

⊙=cases where frequency of changes in cloned code is less than that in non-cloned code (CF_c<CF_n)

3.5.1 Answer to RQ1: Analysis from the perspective of clone types

In this analysis, we evaluate the comparative stability scenarios of the subject systems with respect to three different types of clones and we try to answer the first research question (RQ1). From Table 3.7, we have 28 (7 systems x 4 levels of change significance) decision points for each of the three types of clones in columns labeled with corresponding clone types (Type 1, Type 2 and Type 3).

Based on the summary in Table 3.7, we represent the comparative stabilities of cloned and non-cloned code from the perspective of the types of clones in Figure 3.3. Now, for Type 1 clones there are 82.14% (23/28) of cases (marked with '⊛' in Table 3.7) where the frequency of changes in cloned code is higher than the frequency changes in non-cloned code ($CF_c > CF_n$). For the remaining 17.86% (5/28) cases (marked with '⊙' in Table 3.7) the frequency of changes in cloned code is less than that of non-cloned code ($CF_c < CF_n$).

Again, for Type 2 clones there are 60.71% (17/28) cases where the frequency of changes in cloned code is higher than the frequency changes in non-cloned code. For Type 3 clones, on the other hand, 75% (21/28) cases show that the frequency of changes in cloned code is higher than that of non-cloned code. Here, we observe that for all clone types (Type 1, Type 2 and Type 3) the frequency of changes in cloned code is higher than the frequency of changes in non-cloned code suggesting that cloned code is modified more frequently than non-cloned code. This is an indication that cloned code is less stable as compared to non-cloned code.

To evaluate how statistically significant the comparative frequencies of cloned and non-cloned code are with respect to different types of clones, we carry out Mann-Whitney Wilcoxon (MWW) test [5]. We consider the corresponding frequencies of changes for cloned and non-cloned code for a particular clone type and for all the systems. Here, our null hypothesis is that the frequencies of changes to cloned and non-cloned code are not significantly different. Mann-Whitney Wilcoxon test (U test) is a nonparametric test, it does not require data to be normally distributed and can be applied on datasets with small sample size. Here, our samples are continuous and independent. We observe that for Type 1 clones the p value for *low* and *medium* significance is 0.03102 (<0.05, two-tailed test) which implies that the difference between the two samples is marginally significant. However, for the *high* and *crucial* levels of significance of changes for Type 1 clones and for all levels of significance for Type 2 and Type 3 clones p values are >0.05 indicating that values for frequencies for cloned and non-cloned code are not significantly different. However, we observe that the frequency of changes to cloned code is higher than that of non-cloned code in higher percentages of cases (Figure 3.3).

Summary- According to our findings from the perspective of the types of clones, the *stability* of non-cloned code is higher than that of cloned code meaning that cloned code poses higher maintenance challenges than non-cloned code during maintenance and evolution. Also, Type 1 clones are the most vulnerable to the stability of the software systems.

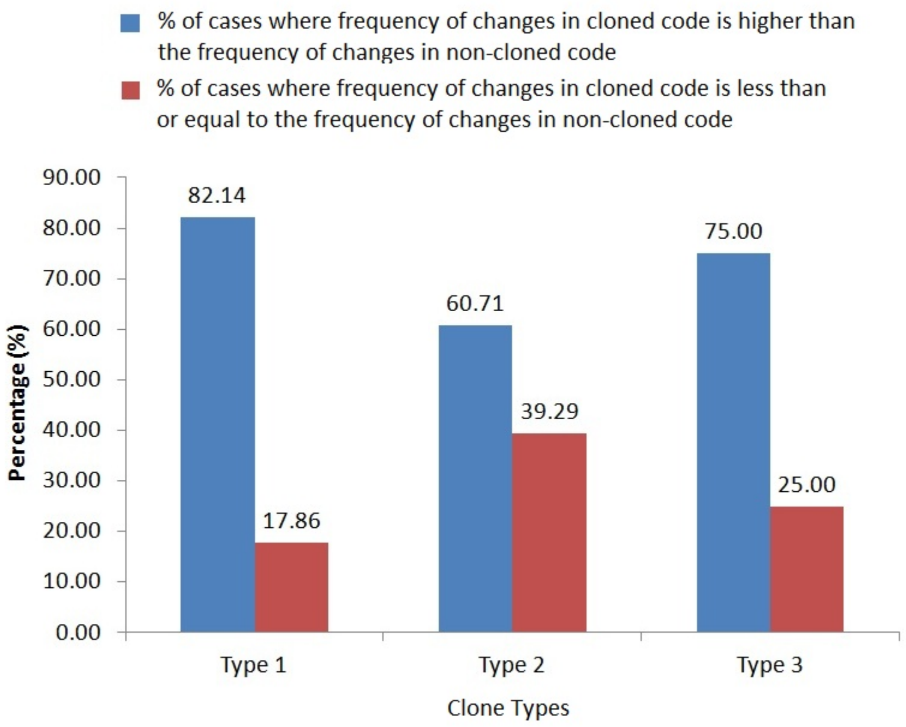


Figure 3.3: Comparison of change frequency of cloned and non-cloned code based on clone types

3.5.2 Answer to RQ2: Analysis from the perspective of change significance

Here, we analyze the comparative frequencies of changes to cloned and non-cloned from the perspective of the four different levels (*low, medium, high, crucial*) of the significance of changes. For each of the four levels of significance, we have 21 decision points (7 systems x 3 clone types) in Table 3.7 in the column groups labeled with the corresponding levels of significance of changes to answer the second research question (RQ2). The comparative stability scenarios with respect to the levels of change significance are presented in Figure 3.4.

As shown in Figure 3.4, 90.48% (19/21) of cases cloned code has higher frequency of changes than the frequency of changes in non-cloned code in both *low* and *medium* levels of significance. For more significant (*high, crucial*) changes, the percentages of cases where the frequencies of changes are higher in cloned code than in non-cloned code are comparatively closer. However, for both *high* and *crucial* levels of significance, the frequencies of changes in cloned code are higher than that of non-cloned code. In Figure 3.4, we observe that cloned code is less stable than non-cloned code and this is mostly dominated by the changes of *low* to *medium* levels of significance.

To evaluate whether there are statistically significant differences among the values for the metrics we obtained with respect to different levels of significance of changes, we carry out Mann-Whitney Wilcoxon (MWW) test [5]. We consider the corresponding frequencies of changes for cloned and non-cloned code for a particular level of significance of changes and for all the systems and all types of clones. We observe that

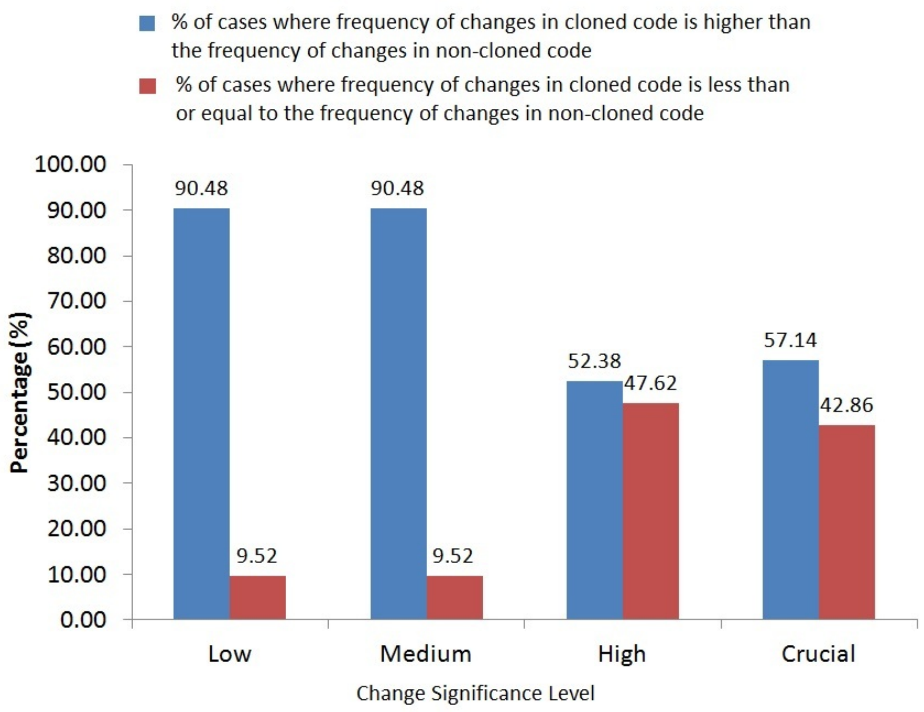


Figure 3.4: Comparison of change frequency of cloned and non-cloned code based on different levels of significance of changes

for changes with *low* and *medium* significance the p values are 0.0207 and 0.0209 respectively and both of the values are <0.05 (two-tailed test) which imply "the difference between the two samples is marginally significant". However, for the high and crucial significance levels p values are >0.05 indicating that values for frequencies for cloned and non-cloned code are not significantly different. So, we see that higher frequency of changes in cloned code as compared to non-cloned code is mostly influenced by changes of lower significance levels (*low, medium*) whereas for changes of higher significance levels (*high, crucial*) the differences among the frequencies are not statistically significant. However, from the perspective of change significance, we observe the frequency of changes to cloned code is higher than that of non-cloned code in higher percentages of cases.

Summary- Our findings from the analysis regarding the perspective of the different levels of significance of changes suggest that cloned code is less stable than non-cloned code and this stability is influenced mostly by the changes of *low* to *medium* significance.

3.5.3 Answer to RQ3: Analysis from the perspective of systems

We carry out analysis from the perspective of individual systems to answer the third research question (RQ3). We consider the 12 (3 clone types x 4 levels of significance) decision points for each of the subject systems from the corresponding rows in Table 3.7. The system-centric comparative stability scenarios for cloned and non-cloned code are presented in Figure 3.5.

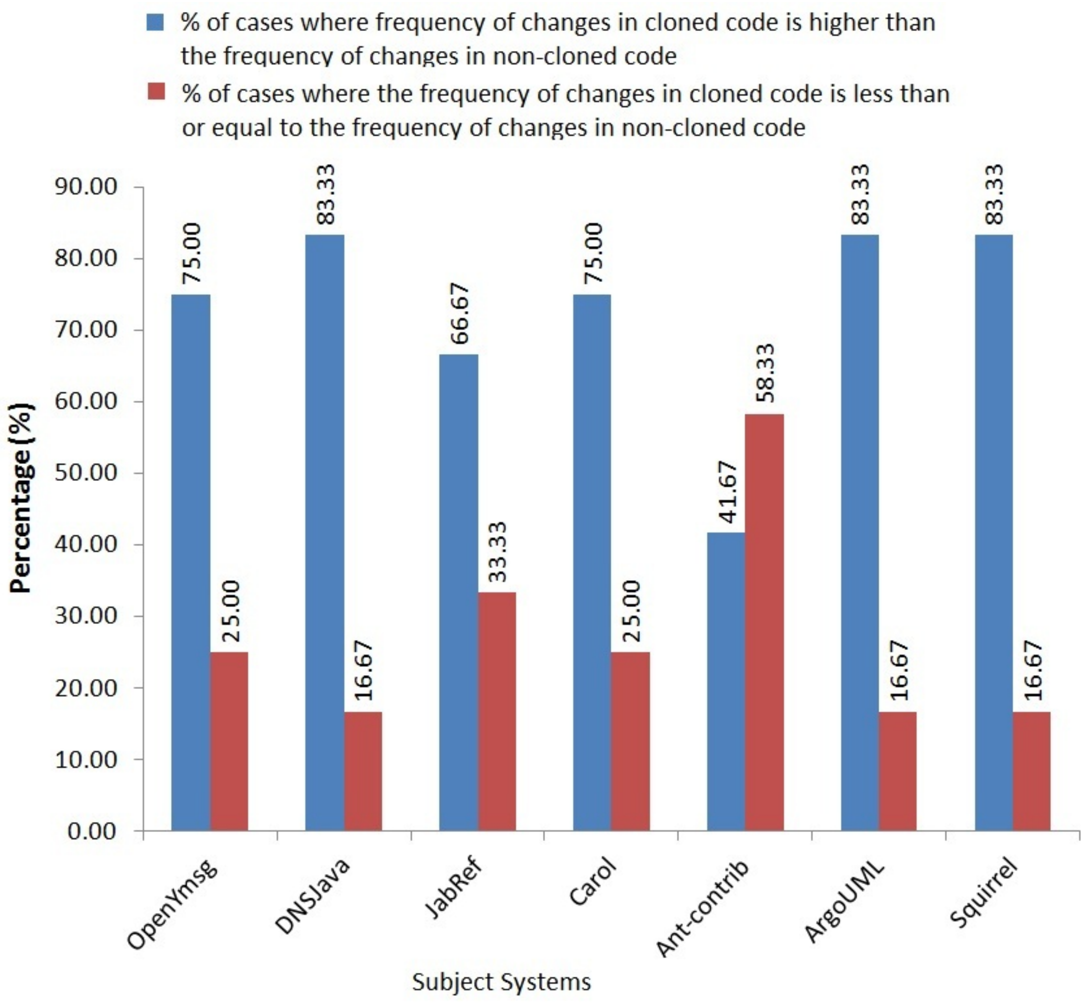


Figure 3.5: Comparison of change frequency of cloned and non-cloned code from the perspective of the systems

As shown in Figure 3.5, our study shows that for six out of seven subject systems the frequency of changes in cloned code is higher than that of non-cloned code considering all types of clones and all the four levels of the significance of changes. Also, we observe that for larger systems with a comparatively large number of revisions and with a longer period of evolution (Jabref, Argouml and Sqiurrel in Table 3.2) tend to have higher percentages of decision points supporting the higher frequency of changes in cloned code as compared to non-cloned code. This suggests that clones in larger systems might be more change prone and comparatively more vulnerable to the stability of the system.

One important point is that for the system Openymsg, our findings agree with the findings from Hotta *et al.* [56] but our findings disagree with their findings for the system Squirrel. This disagreement might be due to the differences in how we count changes. Hotta *et al.* count changes in consecutive lines of code as a single change whereas we count individual fine-grained changes to program entities. Considering changes at finer granularity level is highly likely to increase the change count and thus the increase in the values of the change frequency. For comparatively smaller systems with a smaller number of revisions like Openymsg, this difference might not be significant. However, for larger systems with a higher number of revisions like Squirrel, changes at finer and coarser granularity may make significant differences in the metrics values calculated which might have affected in having differences in the results. In addition, the relative proportion of cloned and non-cloned code in the software systems might also have an influence on the measurement of stability metrics. Moreover, we consider individual levels of significance of changes and clone types which might have effects on decisions regarding the comparative stability of cloned and non-cloned code.

Summary- Our system-centric analysis suggests that cloned code is less stable in general and software systems with a comparatively larger code base and a higher number of revisions tend to have higher probabilities of having a higher frequency of changes in cloned code as compared to non-cloned code.

3.5.4 Significance of the findings

This study re-investigates the comparative stability of cloned and non-cloned code. Stability is a widely focused measure to evaluate the impacts of clones on software maintenance. But our study quantifies stability from a new perspective considering the fine-grained types of changes and their levels of significance. From the study, we conclude that although the magnitude of stability may vary with different types of clones and the changes of different significance levels, cloned code generally tend to have less stability. In addition, Type 1 clones are more vulnerable to the stability of the system and thus, need more attention during the evolution.

We also observe that the stability of cloned code is influenced mostly by the changes of lower (*low, medium*) levels of significance. However, for changes with higher (*high, crucial*) levels of significance the comparative frequency of changes in cloned code is still higher than that of non-cloned code. Although our fine-grained analysis disagrees with the stability decisions of some of the related existing studies [56, 84, 49], our methodology gives us the confidence that our stability decisions are more precise and reliable as those take into account the actual impacts of changes. Again, the way we analyze the stability gives us an insight into the detail evolution of how individual clones change throughout their period of evolution. The information regarding the frequency of changes with distinct levels of significance might be valuable in ranking and prioritizing clones by the developers during clone management. The evolution information might also be used as constraints for an automatic scheduler for clone refactoring [188].

3.6 Threats to Validity

The change analyzer in this study is based on the change extraction and classification engine of the *ChangeDistiller*. Although *ChangeDistiller* is reported to have good performance, the validity of the outcomes of the study is dependent upon the accuracy of the core classifier used.

Another source of potential threats is the clone detection tool used. We used NiCad, a recently introduced hybrid clone detection tool which detects both exact (Type 1) and near-miss (Type 2, and Type 3) clones with high precision and recall. Again, different settings for clone detection tools might result in different stability scenarios because of the variations in clone detection results. This is termed as *confounding configuration choice problem* [180]. However, the NiCad settings we used are considered standard [151, 154] and are close enough to the optimal configuration settings identified in recent study [180] for NiCad to detect clones in Java systems. This is likely to mitigate the potential adverse effects of the configuration settings on our findings.

We selected subject systems with diversified size, number of revisions, length of evolution and application domain to avoid potential biasing. However, due to the limitation of existing change classifier our study is limited to Java systems only. Although Java is considered to be a widely used language with a comprehensive set of language features for software development, the inclusion of subject systems of other languages might help in more generalization of the findings.

3.7 Related Work

There is a large body of research work investigating the impacts of clones on software maintenance and evolution. The primary focus of these studies are evaluating the impacts of clones in terms of the degree of consistency in comparative evolution, measuring stability to study how frequently a code region is modified and the likelihood of introducing bugs during evolution.

Krinke [84] proposed a measure for the comparative stability of cloned and non-cloned code. His approach counts the modification of code in terms of the number of lines added, deleted and modified in cloned and non-cloned code. The more the number of changes to a code region, the less stable the code is. His study concluded that clone code is more stable than non-cloned code. Krinke's other study [86] measures the average age of the cloned and non-cloned code. This study concludes that clone code is older meaning more stable than non-cloned code. Göde and Harder [49] extended Krinke's [84] approach with token-based incremental clone detection tool and experimented with different settings of clone detection parameters, clone types and their changes. This study agrees with the findings of Krinke [84] that cloned code is more stable than non-cloned code but this does not hold in case of deletion.

Hotta *et al.* [56], on the other hand, measure the stability of cloned and non-cloned code in terms of *modification frequency*. Their study concludes that cloned code has lower modification frequency and thus more stable than non-cloned code. Their approach counts the number of blocks (consecutive lines) modified in cloned and non-cloned code. The average modification count per revision in cloned and non-cloned code are then used to measure the modification frequency. The less the modification frequency, the more stable the code region is. One of the key limitations of this approach is that it ignores the volume of change and also the types of actual syntactic changes. This is likely to affect the accuracy and reliability of the stability of the code measured. Although this study considers fine-grained changes because of the change analysis at token level, it does not differentiate between the types of syntactic changes and their levels of significance.

Mondal *et al.* [123, 122] carried out a comprehensive stability analysis within a uniform evaluation framework considering existing (taken from [56, 84, 86]) and their proposed metrics to measure stability. Their study concluded that there is no firm conclusion on the stability of cloned and non-cloned code, rather, the comparative stability varies with programming languages, clone types and overall system development strategies. However, none of the metrics used or proposed in this study consider actual syntactic change types.

Lozano *et al.* [99] investigated whether clones are harmful or not. Their study concludes that clones tend to be more frequently modified than non-cloned code. To assess the impacts of clones on software maintenance Lozano and Wermelinger [97] also investigated the impacts of clones on software maintenance. Their study shows that although the *likelihood* (the ratio of the number of change to an entity to the total number of changes in the system) in cloned code is not very different than in non-cloned code, in some cases the *impact* (percentage of the system affected by a change) of cloned methods is greater than that of non-cloned methods. Their study also suggests that the presence of clones may decrease the changeability of the code entities containing clones.

Our proposed study to measure the comparative stability of cloned and non-cloned code is kind of related to the study proposed by Hotta *et al.* [56]. However, our study is different from their study because we count each of the fine-grained changes whereas Hotta *et al.* consider a range of consecutive lines of modified code as a single change. Another difference is that we consider the different significance levels of changes and we measure the frequency of changes based on the levels of significance of the changes. Our study differs from Krinke's study [84] because we consider every revision of the subject systems whereas Krinke's study considers revisions on weekly intervals. Again, Krinke's study [84] differentiates between the change types as the addition deletion and update to lines of code but does not consider the actual syntactic change types as in our study. Although there are some similarities, our study is different from all others in the sense that we study the stability in the fine-grained levels considering the different change types and their levels of significance. Our objective is to focus on how cloned and non-cloned code are evolving through different changes and the impact of clones on software maintenance in terms of the frequency of changes of different levels of significance, which none of the above existing studies consider.

3.8 Summary

This empirical study evaluates the comparative stability of cloned and non-cloned code from the perspective of different syntactic change types and the levels of significance of changes. To the best of our knowledge, this is the first ever approach to the comparative evaluation of the stability of cloned and non-cloned code that considers actual syntactic change types and their corresponding levels of significance.

We use the hybrid clone detection tool NiCad to detect exact and near-miss clones. By analyzing the changes to the cloned and non-cloned code of seven software systems, we observe that the frequency of changes is higher in cloned code than that in non-cloned code in most cases. Again, from the perspective of the clone types our study shows that in most of the cases the frequency of changes in cloned code is higher than that in non-cloned code and Type 1 clones are more change prone affecting the stability of the software systems. Another important point is that the comparatively higher instability of clones is mostly influenced by the changes of lower (*low, medium*) significance. For changes of higher significance (*high, crucial*) the stability of cloned and non-cloned code are closer unlike the stabilities for changes of lower significance. However, for all levels of significance of changes the stability of non-cloned code is higher than the cloned code. Thus, cloned code is likely to pose higher challenges during the evolution and maintenance of the software systems.

Moreover, in this study, our analysis at the fine-grained change types considering their levels of significance facilitates to have a deeper insight into how the clones evolve with changes of different levels of significance. The history regarding the evolution of the clones representing how individual clones are changed over time might be an important source of information to rank the clones based on the types and volume of changes they evolve through. This might help in better managing clones by tracking and/or refactoring clones to minimize the negative impacts of clones. In the next chapter (Chapter 4) we explore the relationships between domain based coupling and code clones to analyze the impacts of clones at the domain level.

Chapter 4
Relationships between Domain-Based Coupling and Code Clones

In Chapter 3 we studied the comparative stability of cloned and non-cloned code for different fine-grained change types. However, there remain other important relationships to consider to fully understand the impacts of clones at the domain level. Existing studies mostly focus on the analysis of source code and their evolution to investigate the impacts of clones on software systems. While a great deal of research has been conducted to develop techniques and tools to identify code clones, little research has been done to analyze the relationships between code clones and other aspects of software beyond source code such as domain-level relationships among the software components. In this chapter, we present a study that explores the relationships between code clones and the coupling among domain-level components. We report on a case study based on a large-scale open source enterprise system, where we demonstrate that there is a high probability of finding code clones among components with domain-based coupling. The findings of this study have been published in ICSE 2013 (NIER-Track)[1].

The rest of this chapter is organized as follows: Section 4.1 outlines the motivation and the context of the study. Section 4.2 describes the system under analysis. Section 4.3 describes domain-based coupling analysis. Section 4.5 discusses code clones analysis. Section 4.6 presents the experimental results. Section 4.7 discusses the potential threats to the validity of our results, Section 4.8 describes the related works, and finally Section 4.9 concludes this chapter with a discussion on the findings and future areas of investigation.

4.1 Introduction

Clones represent important information regarding syntactic and semantic similarities among the program entities, thus studying the clones might assist in program comprehension [67]. Moreover, by refactoring code clones, one can potentially improve the quality and maintainability of the source code, and reduce the complexity of the system [44]. Given the importance, many tools and techniques for detecting clones have been proposed [147]. Some of these are text-based [41, 66, 104, 144], some are token-based [20, 73, 95],

[1]M. S. Rahman, Amir Aryani, Chanchal K. Roy, Febrizio Perin. "On the relationships between Domain-Based Coupling and Code Clones: An Exploratory Study". In proceedings of the International Conference on Software Engineering (ICSE NIER Track), 2013, pp. 1265-1268.

some are tree-based [23, 64], some are metrics-based [81, 105] , some are graph-based [46, 80, 85], and also based on functional similarity analysis [88]. Despite a good number of tools for clone detection and analysis, cross-language and source code independent clone analysis are still open research challenges. Although complete solutions to these open problems are yet to come, artifacts and information beyond the source code are potentially useful to formulate complementary solutions for identifying dependencies and change propagation between software components [16].

In contrast to a great deal of research about code clones, there is little information available about the relationships between clones and other aspects of software systems such as the domain level artifacts. Recently, relationships among software components at the domain level, termed as *domain-based coupling* [14, 48], have been shown useful in tracing dependencies at the source code and database levels. In this study, we focus on the domain-based coupling and its relationships with code clones. We hypothesize that coupling at the domain level can approximate clones in the source code.

We evaluate our hypothesis with a case study on a large scale enterprise system called ADempiere. We derive domain-based coupling from the domain information available at the user interface level. Code clones, on the other hand, are detected using the clone detection tool. The relationships between domain-based coupling and code clones are then determined based on the extent of co-existence of both relationships between all pairs of software components. Our results from the case study show that the coupling between domain level components can approximate clones in the associated code with high precision without even accessing the source code. In particular, we focus on the following research questions in conducting the exploratory study:

RQ1 *What is the probability of finding code clones where there is a domain-based coupling?*

RQ2 *How efficiently can domain-based coupling assist software maintainers in discovering code clones?*

RQ3 *Is there any correlation between the number of clones and the value of domain-based coupling?*

While such a probabilistic approach of predicting the existence of clones does not replace the traditional clone detection tools, it certainly has the potential to complement the existing state of the art tools by providing the probability of having clones among software components. Moreover, the approach is independent of program source code. Consequently, it can be used for the analysis of hybrid systems developed with multiple programming languages and legacy systems with missing or obsolete source code. In addition, based on the domain-based coupling, it provides a flexible way of focusing only on the relevant parts of the systems and thus reduces the clone search space. Once the relevant parts are identified, one can either use the existing tools for actual clone detection or opt for manual analysis where feasible. This essentially helps to predict goal-driven clones without being overwhelmed with hundreds of other irrelevant clones.

4.2 Case Study: ADempiere

For this study, we use ADempiere[2], a large-scale Enterprise Resource Planning (ERP) system. The qualities that persuaded us to choose ADempiere as our case study are:

Integration into multiple business domains: We need a system that manifests the integration between software and domain requirements. ADempiere includes multiple subsystems across various domains such as finance, human resource, customer relationship management (CRM), material management and sales, *etc.* Such a system provides a good example of how a system can be modeled based on domain concepts and relationships, and how it evolves by changes in domain level requirements.

Matured and evolving system: We need a system that typified large-scale enterprise systems. ADempiere represents cutting-edge open-source software technology. It is designed based on model-view-control (MVC) architecture and manifests a clear separation between the different application tiers. This system has multiple distinct front-ends from which the user can choose including a Java GUI, web interfaces and mobile devices. At the data tier, it uses relational database management systems (*e.g.,* PostgreSQL and Oracle) for data storage as well as for storing business logic. The ADempiere project traces its evolution back more than a decade. Created in 2006 as a fork of the Compiere open-source ERP, itself founded in 1999, ADempiere has been among the most popular open source enterprise software systems. It is a multi-language system that aggregates more than 6 million lines of code. The core part is written contains more than 3,531 JAVA classes with more than half a million lines of code (version 3.6 LTS). The system is used by a large number of companies around the world.

For these reasons, we deem ADempiere to be relevant and representative for enterprise systems and for the state of the art in open-source software at the moment of writing this article, and appropriate for our analysis.

4.3 Domain-Based Coupling

In this section, we describe the methodology for domain-based coupling analysis, and how it is implemented for ADempiere, the system under analysis.

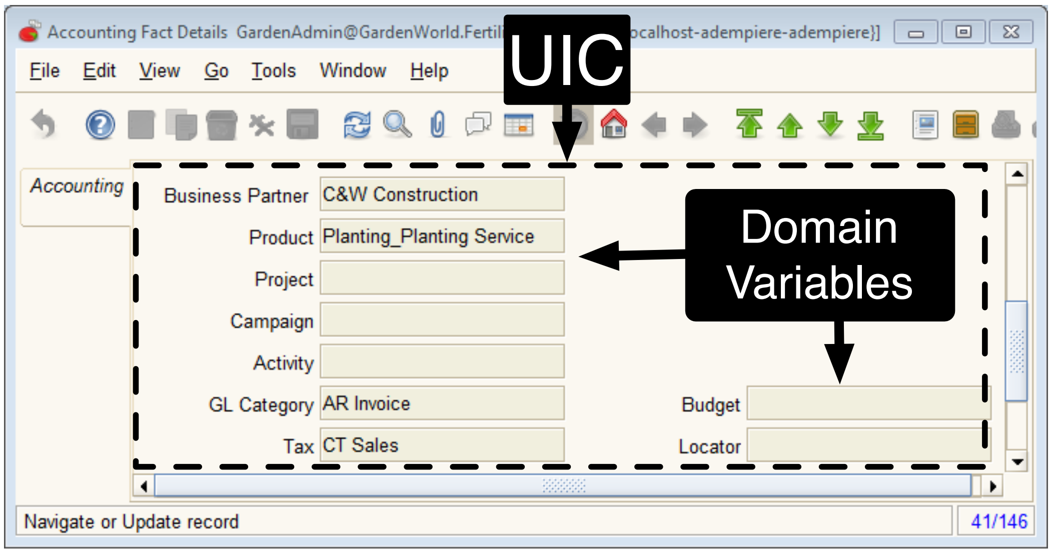
4.3.1 Notation and Definitions

Our domain analysis model is derived from the models proposed by Aryani *et al.* [11, 14, 15, 16]. The three key elements at the domain level are modelled as follows:

- A *domain variable* is a variable unit of data that has a clear identity at the domain level. *Domain variables* are modelled by a finite set V, called *variable symbols.*

- A *domain function* represents a domain-level behaviour of the system associated with at least one domain variable as an input or output. *Domain functions* are modelled by a finite set F, called *function*

[2]http://www.adempiere.com


Accounting Fact Details GardenAdmin@GardenWorld.Fertil... ocalhost-adempiere-adempiere}]
File
Edit
View
Go
Tools
Window
Help
UIC
Domain Variables
Accounting
Navigate or Update record
41/146

Business Partner: C&W Construction
Product: Planting_Planting Service
Project:
Campaign:
Activity:
GL Category: AR Invoice
Tax: CT Sales
Budget:
Locator:

Figure 4.1: ADempiere user interface component *Accounting*

symbols, and the binary relation $USE \subseteq F \times V$ represents the relation between functions and variables as the input-output of the functions.

- A *user interface component (UIC)* is a system component which directly interacts with users and contains one or more domain functions. *UICs* are modelled by a finite set C, called the *component symbols*, and $HAS \subseteq C \times F$ represents the relation between components and functions.

For example, Figure 4.1 shows the *Accounting* tab in ADEMPIERE. It is a UIC that provides the domain function of *Query Accounting Transactions*, and data fields visible on this tab are domain variables.

Definition 1. *The conceptual connection relation $CNC \subseteq C \times C$ is defined by*

$$CNC = HAS.USE.USE^{-1}HAS^{-1}$$

The domain-based coupling is derived based on the following measurements:

Definition 2. *The number of common variables among two UICs is modelled by the function $\vartheta : C \times C \to \mathbb{N}$ where*

$$\vartheta(c, c') = |c.HAS.USE \cap c'.HAS.USE|$$

Note that the definition of common domain variables is symmetric, i.e., $\vartheta(c, c') = \vartheta(c', c)$.

Definition 3. *The domain-based coupling graph of a system is a symmetric weighted graph, $G = (C, CNC \backslash ID, \omega)$ where ID indicates the identity relation and ω indicates the coupling weight function $\omega : C \times C \to [0..1]$ by*

$$\omega(c, c') = \frac{\vartheta(c, c')}{|c.HAS.USE \cup c'.HAS.USE|}$$

Here, by *domain-based coupling* we refer to ω. The next example demonstrates how to measure ϑ and ω for two example UICs.

4.3.2 Example

In ADempiere, *Daily Balances* (c_1) and *Accounting* (c_2) are two UICs which we use in this example. Each one of these UICs has only one domain functions, as follows:

$c_1.HAS = \{$ QueryDailyAccountBalances $\}$.

$c_2.HAS = \{$ QueryAccountingTransactions$\}$.

Daily Balances (c_1) contains 22 domain variables:

$c_1.HAS.USE = \{$ Account, Budget, BusinessPartner, PostingType, Product, Project, Quantity, SalesRegion,... $\}$.

Accounting (c_2) has all *Daily Balances'* domain variables and 11 others:

$c_2.HAS.USE = c_1.HAS.USE \cup \{$ Currency, GLCategory, Locator, Period, SourceCredit, UOM, Tax, TransactionDate,... $\}$.

and in total 33 variables used by either of these UICs; thus:

$$\vartheta(c_1, c_2) = 22$$

$$\omega(c_1, c_2) = 22/33 = 0.67$$

The next section describes how to derive the *HAS* and *USE* relationships for ADempiere, and create the domain-based coupling graph.

4.4 Experimental Setup

Our overall process of the exploratory study is shown in Figure 4.2. Here, the domain information is extracted from the User Interface Components (UIC) of the software systems. In case of ADempiere this data is available in the dictionary database which we used to map UIC to source code files. We measure domain-based coupling based on the extent of how UICs have common domain variables. We detect clones Using NiCad. Then we analyze the probability of UICs having domain-based coupling to have clones and vice versa. The following sections briefly discuss the experimental steps.

4.4.1 Domain Analysis in ADempiere

The user interfaces in ADempiere are structured in the following hierarchy: a data field is associated with a tab, and each tab belongs to a window (Figure 4.1). A tab is the minimum software component which interacts with the users, and it contains at least one domain function. Hence, we consider tab as a UIC in this study.

In ADempiere, data fields are mostly domain variables. However, there are some exceptions such as *Active flag* and *Search Key.* We reviewed data fields and excluded fields which are not related to the domain functions. The remains are considered domain variables.

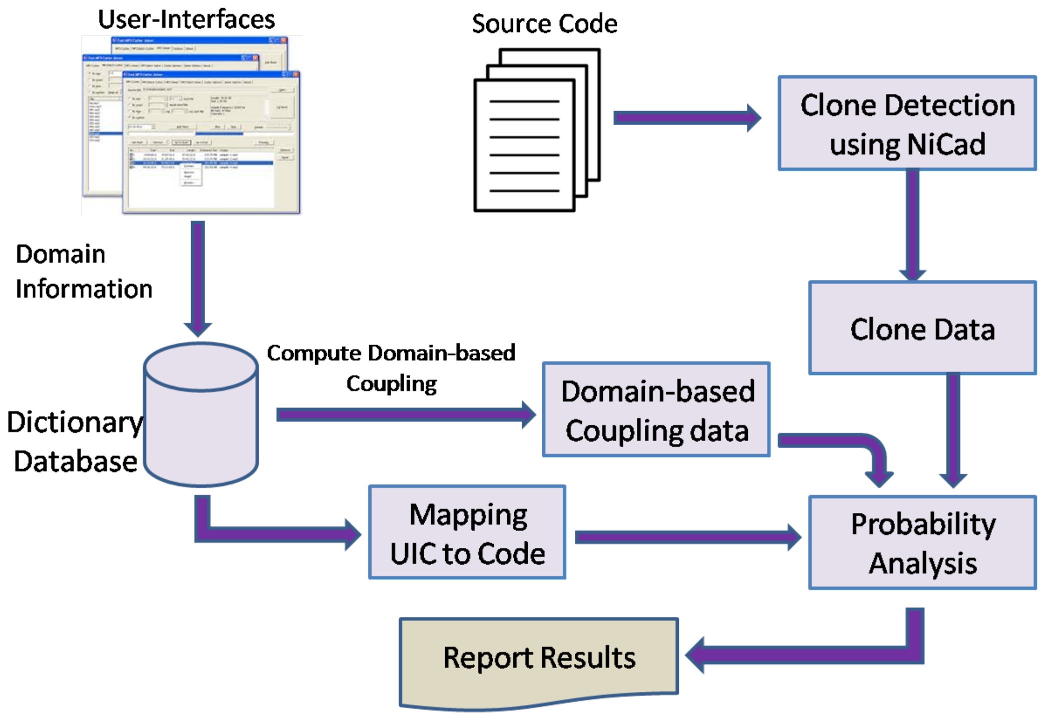


Figure 4.2: Overall Domain-based Coupling Analysis Process

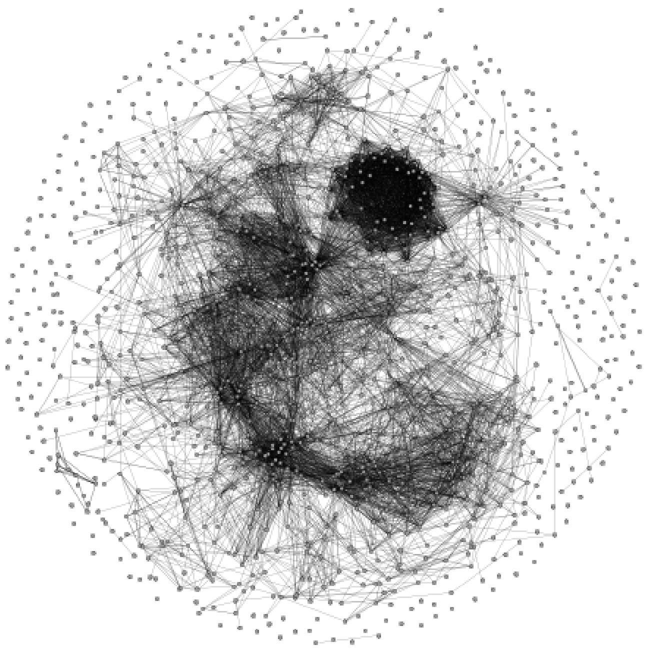
The *HAS.USE* relationships can be derived by manual analysis of the working software or analysis of help documents. In case of ADEMPIERE, the help description and the relations among these elements are documented and stored in the database. The result of our domain analysis includes 889 UICs and 2,359 domain variables, leading to 49,854 pairwise *CNC* relationships.

We used this information to create the domain-based coupling graph (Figure 4.3). Fruchterman and Reingold's [45] force-based graph layout (known as spring layout) is used to visualize the graph. This layout clusters the UICs based on the edges' weight and helps us to identify the highly coupled UICs. It turns out that it is practically useful to apply a threshold $\omega \geq t$ to the coupling weight function. Filtering of edges by their weight improves the readability of the graph, and reduce the computation cost of the layout algorithm. Later in Section 4.6.4, we use this graph to compare domain-based coupling and code clones.

4.5 Code Clone Analysis

For our study on ADEMPIERE we used the NiCad [144] tool for clone detection as NiCad is reported to have high precision and recall [145, 151]. We detected clones with a minimum size of 5 lines of code (LOC) for function granularity. At function granularity, each function in the source code is considered as a unit of code for comparison and detection of clones.

To determine the number of the *clone class* (set of all fragments that are clones of each other) and clone fragments associated with each pair of UICs, all the clone classes are checked. We count only those clone



Legend: Nodes represent UICs and edges represent their *CNC* relationships with coupling weight $\omega > 0.1$.

Figure 4.3: Domain-based Coupling Graph of ADempiere

classes which have at least one clone fragment from each of the source files behind the selected UIC pair. For each of these clone classes, all clone fragments originated from the source files behind the selected pair of UICs are counted. Definition 4 formally defines this procedure.

Definition 4. *Let $c, c' \in C$ be any pair of user interface components with their associated source code files S and S'. The number of cloned code fragments $\xi : C \times C \to \mathbb{N}$ associated with c, c', is defined by*
$\xi(c, c') = \sum_i \mid \{f : (f \in F_S \vee f \in F_{S'}) \wedge f \in CC_i \wedge (CC_i \cap F_S) \neq \phi \wedge (CC_i \cap F_{S'}) \neq \phi\} \mid.$
where F_S and $F_{S'}$ are the sets of all code fragments in S and S' respectively for a defined granularity, and CC is the set of clone classes in the system.

We observed that out of 889 UICs in ADempiere there are 813 UICs with their associated source code, and the other 76 UICs are created at runtime based on predefined business rules. We consider exact clones for these 813 UICs with source code for the evaluation of our study.

4.6 Experimental Results

In this section, we provide empirical evidence on the relationships between code clones and domain-based coupling and demonstrate how these relationships can be used to assist software maintainers by answering the defined research questions.

4.6.1 RQ1: Probability of Finding Clones

For our analysis, we use Fruchterman and Reingold's [45] force-based graph layout (known as spring layout) to visualize the domain-based coupling graph (Figure 4.5a). Now, let us consider $E \subseteq C \times C$ to be the set of domain-based coupling relations including all combinations of pairs between UICs. Our query is $Q = \{(c, c')|c, c' \in C, \omega(c, c') > 0\}$, the set of the couplings connecting UICs by coupling weight ω, and $A = \{(c, c')|c, c' \in C, \xi(c, c') > 0\}$ is the expected answer, the set of UIC pairs with one or more clones in their code behind.

There are 813 UICs in our analysis with source code (Section 4.5), leading to $|E|$ =330,078 disjoint UIC pairs out of which $|A|$ =68,563 UIC pairs have clones in their code behind it. Again, the domain-based coupling relation contains $|Q|$ =44,163 couplings, representing UIC pairs with $\omega > 0$, from which $|Q \cap A|$ =39,850 have exact clones in their code behind. The probabilities of finding code clones between UIC pairs with and without domain-based coupling represented by P_Q and $P_{\bar{Q}}$ respectively and measured as

$$P_Q = \frac{|Q \cap A|}{|Q|} = 90.23\% \qquad P_{\bar{Q}} = \frac{|A \cap E \backslash Q|}{|E \backslash Q|} = 10.04\%$$

The higher value of probability P_Q, also can be considered as *precision*, implies that domain-based coupling can be used to provide a precise prediction of the places in the source code that have cloned code (or vice versa). In addition, the high probability of co-existence of domain-based coupling and code clones suggests that domain-based coupling can be used to single out a subset of code base having code clones with a selected component for a goal-driven clone analysis.

Summary: *Based on our analysis on* ADempiere*, we observe that the probability of finding code clones where there is domain-based coupling is more than 90%.*

4.6.2 RQ2: Discovering Code Clones

Clone detection in the source code is often computationally expensive and it may generate an exhaustive list of clones. To carry out maintenance, software maintainers need precise and concise suggestions about the locations of code clones, as inconsistent updates to clone fragments may introduce bugs in the system. Instead of reviewing an exhaustive list of all clones in the system, it is more useful to have a short list of higher abstraction level components that most likely contain code clones. The maintainers can then confidently look at those selected components for their intended task. Using domain-based coupling we can provide such a short list based on the topmost coupled UICs, and which might in turn help in prioritizing the components to work on.

For a given UIC, $c \in C$, we derive the short list of $Q_{c,n} \subseteq C$ which contains the top n UICs with highest domain-based coupling to c. To evaluate how useful such a short list is, we measure the *likelihood* of finding one or more clone fragments in the code behind of UICs in the shortlist. More formally, if $A_{c,k} \subseteq C$ represents the set of UICs with at least k number of clone fragments to c, then the *likelihood* measure is defined as

$$L_{n,k} = \frac{|\{c|c \in C, Q_{c,n} \cap A_{c,k} \neq \emptyset\}|}{|\{c|c \in C, Q_{c,n} \neq \emptyset\}|}$$

Table 4.1 shows the results for the likelihood of finding code clones in the shortlist of UICs that include top 3, 5 and 10 highly coupled UICs. The likelihood of finding 10 code clones or more in the top 3 UICs is 3.94%. The likelihood increases by reducing the threshold for code clones, up to 88% for k =1. In addition, by increasing the size of the short list one can find more code clones. This is more notable for the higher thresholds. For the threshold of k =10, increasing the short list size from three to ten, almost doubles the chance of finding code clones, while for the threshold k =1, increasing the size of the list improves the likelihood from 88.81% to 91.64%. This is a trade-off between accuracy and cost since it is less expensive and more convenient for software maintainer to review a list of three UICs compared to a list of ten.

Table 4.1: LIKELIHOOD (%) OF FINDING CLONES IN TOP n RESULTS

$k \rightarrow$	1	3	5	10
Top 3	88.81	38.75	17.47	3.94
Top 5	90.53	44.77	21.40	5.78
Top 10	91.64	54.40	24.97	7.75

Legend: k is the threshold for the minimum number of clones.

Summary: *Based on our analysis on* ADempiere*, we observe that the likelihood of finding code clones in the efficient shortlist of top three results is more than 88%.*

4.6.3 RQ3: Correlation

To explore the correlation between domain-based coupling and code clones, we examined two measurements: (1) correlation coefficient between coupling weight function and the number of code clones, and (2) their average trend.

Correlation coefficient

We hypothesized that the value of the weight function increases with the number of code clone fragments. To evaluate this hypothesis, we examine the relationships between coupling weight function and the number of cloned fragments by measuring the Spearman's rank correlation. Here, Spearman's rank correlation $\rho(\omega, \xi) \rightarrow [-1, +1]$ measures the monotonic relationships between coupling weight function ω and the number of clone fragments ξ.

The value of Spearman's rank correlation (ρ) is 0.63 that is an indication that the value of coupling weight function tend to increase where there are more clone fragments. Given that the p-value is very small (2.2e-16), the observed correlations are statistically significant.

Average Trend

We grouped all the UICs pairs by the number of clone and measured the average coupling weight function for these groups. Each group of G_n consist of pairs $\langle c, c' \rangle$ where $n - \Delta < \xi(c, c') \leq n$. We measured the average coupling weight function for group G_n as

$$\bar{\omega}_n = \frac{\sum_i \omega(c, c') : \langle c, c' \rangle \in G_n}{|G_n|}$$

As presented in Figure 4.4 the average value of coupling weight function increases with the increase in the number of code clones.

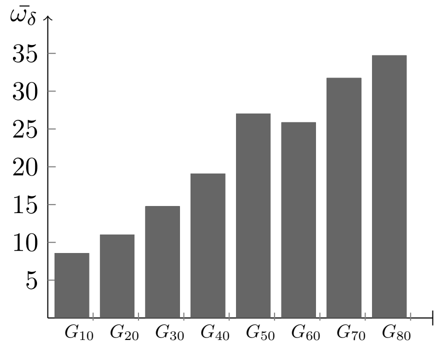


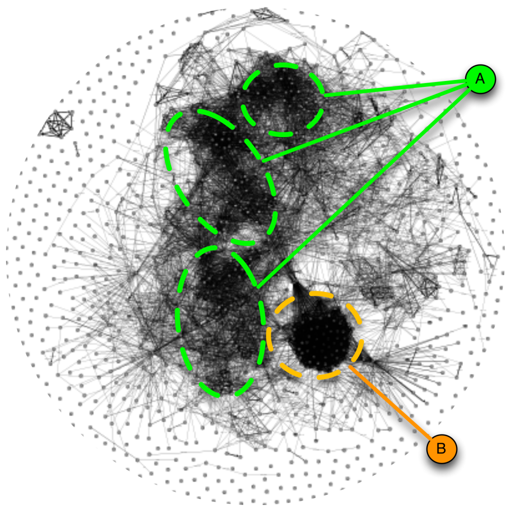
Figure 4.4: The Trend of Coupling Weight versus Number of Clones

Summary: *Based on our analysis on* ADEMPIERE, *we observe a significantly ranked correlation between domain-based coupling and the number of clone fragments.*

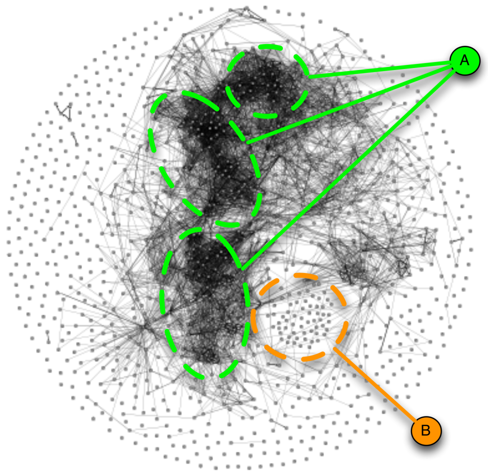
4.6.4 Visual Comparison

To visualize the relationships between domain based coupling and code clones the comparison graphs (Figure 4.5b) are created by the following steps: First, the domain-based coupling graph is generated (Figure 4.5a) with a given threshold of $\omega > 0.1$. The value of the threshold is derived by a heuristic approach based on the distribution of UIC pairs in the graph to filter out weak couplings. Second, all the edges with no code clones, $\xi = 0$ are filtered out from the graph. The result is presented in Figure 4.5b. The remaining edges show the UIC pairs with both strong domain-based coupling and code clones.

The comparison between these graphs in Figure 4.5 shows that from four clusters in the domain-based coupling graph, three of them a have high number of clones (tagged with A), and there is one cluster with no clones (tagged with B). The qualitative analysis of this cluster shows that it is composed of 76 UICs



(a) Domain-based coupling graph



(b) Filtered graph by code clones

Legend: Nodes are the UICs of ADEMPIERE in both graphs. Top: Edges represent domain-based coupling between UICs with $\omega > 0.1$. Bottom: Edges show that there are both code clones and domain-based coupling.

Figure 4.5: Domain-based coupling vs clones

which implement the multilingual aspect of ADEMPIERE. These UICs are generated at runtime based on the predefined business rules and there is no code behind these UICs.

4.7 Threats to Validity

The code clone patterns and behaviour might be influenced by the inherent features of a particular programming language. Similarly, the distribution and patterns of clones may vary from system to system. We aimed to address these issues by selecting a case study system which typifies a majority of enterprise systems. However, further studies are required with diversified experimental settings to draw generalized conclusions. In this study, the quality of derived domain variables is dependent on authors' knowledge about the system. We addressed this issue by cross-checking the derived domain variables with the help descriptions of the system.

4.8 Related Work

Studies show that domain knowledge is useful in understanding the functionality of the code [96, 131]. Domain knowledge has been incorporated into reverse engineering and software maintenance by a number of researchers. Michail [107] discussed the possibilities of using the application GUI to guide browsing and searching of its source code. Rugaber [152] showed how domain knowledge can be useful in program comprehension. Riebisch [139] introduced feature models based on domain analysis to supporting software evolution. Moreover, mapping domain concepts to source code has been investigated by number of researchers [25, 137, 103, 138]. Also, user cognitive abilities and their understanding from the system has been used to develop reverse engineering tools and methods [173, 174, 172], later this method has been proposed to support software change processes [183]. Given the potentials of domain information, Aryani *et al.* [15, 14] recently showed how domain information can be used to identify dependencies among software components to support change propagation. These methods mainly aim to assist programmers in activities of understanding and changing the source code. Since the primary objective of these methods is to incorporate domain knowledge into the development environment, these are difficult to use by consultants, managers, or expert users who have little knowledge about the source code. In contrast, our proposed approach only relies on domain information visible and understandable to the end users, without any requirement on software engineering or programming knowledge.

Clone detection, on the other hand, is not a new topic and there have been a great many tools and techniques available in the literature [147]. Some of these clone detection techniques are measuring textual similarity, finding common subsequences of tokens, finding the similar sub-trees in Abstract Syntax Tree (AST), comparing matrix values computed for source code blocks and finding isomorphic sub-graphs [147]. Although there are a wide variety of clone detection tools and techniques, these tools lack in generalized application to detect clones from hybrid systems with the source code of different languages. This exploratory study examines one potential application of domain information to predict clones in the program source code without even accessing the source code.

4.9 Summary

In this study, we examined the relationships between domain based coupling and code clones. Unlike the traditional clone detection approaches, it evaluates the feasibility of using domain-based coupling to complement the existing clone detection tools by providing the probability of the existence of code clones between software components. The results based on one of the largest open source enterprise systems, ADempiere, demonstrate that code clones could be predicted using solely domain information with more than 90% precision. In addition, we presented how domain-based coupling can be used to give efficient and precise suggestions about code clones to software maintainers. The likelihood of finding clones in such suggestions is more than 88%. The co-existence of clones at the source code level and coupling at the domain level is an indication that code clones are related to domain-based coupling. Thus, clones are likely to have negative impacts by introducing higher coupling at the domain level. This study explores the impacts of clones from a new perspective based on information beyond the source code. For a more comprehensive understanding of the impacts of clones, we study the comparative stability of cloned and non-cloned code within a uniform framework in the next chapter.

Chapter 5

Analyzing Comparative Stability of Cloned and Non-Cloned Code within Uniform Framework

While we studied the stability of clones using fine-grained change types (Chapter 3) and then their impacts even at the domain level (Chapter 4), there remain many other important perspectives to assess the impacts of clones. Many studies have investigated different aspects of the impacts of clones on software systems. Stability is a well-investigated perspective to analyze the impacts of clones on software systems. However, existing studies greatly differ in conclusions regarding the comparative stability of cloned and non-cloned code. One important point is that studies are based on different experimental settings such as subject systems, clone detection tools, and the metrics measured. To address this issue we need to evaluate the comparative stability of cloned and non-cloned code within uniform comparison framework. In our earlier study [121], we comprehensively assessed the stability of clones considering eight stability metrics within a uniform framework. However, the study does not evaluate the stability metrics for the subject systems used in the original studies with experimental settings different from the original experimental settings. This study extends and complements our earlier study [121] by investigating the comparative stability of cloned and non-cloned code based on the original subject systems and new experimental settings different from the original studies. We also compare the results from new experimental settings with the stability results from the systematic replication of the original experiments to evaluate the comparative stability of cloned and non-cloned code. The findings of this study have been partly published in EMSE 2017[1].

The following sections of the chapter are organized as follows: Section 5.1 represents the motivation, context and research questions of the study. Section 5.2 outlines our uniform framework for the comparative stability analysis of cloned and non-cloned code. Section 5.3 describes the experimental settings and briefly describe the stability metrics for the study. Section 5.4 represents our experimental results by answering the research questions. Potential threats to the validity of our study findings are discussed in Section 5.5. Section 5.6 discusses earlier research works related to our empirical study. Finally, Section 5.7 summarizes our findings to conclude the chapter.

[1] M. Mondal, M. S. Rahman, C. K. Roy, K. A. Schneider, Is cloned code really stable?, Empirical Software Engineering (EMSE), pp. 78, July 2017. DOI: 10.1007/s10664-017-9528-y

5.1 Introduction

The impact of clones is of great importance in software maintenance and evolution. The most common claims against code clones are that clones may increase software maintenance efforts due to frequent changes and clones may also introduce bugs or inconsistencies due to inconsistent changes. When a cloned fragment is changed it is assumed that all the cloned copies of the code fragment should also be changed consistently. This is likely to increase maintenance efforts in the software systems and affect the stability of the software systems. Thus, comparative stability of cloned and non-cloned code has been a well-investigated perspective in assessing the impacts of clones on software maintenance. Here, the hypothesis is that if clones appear to be less stable than non-cloned code, clones are likely to require higher maintenance effort and cost than non-cloned code. However, as mentioned earlier, the studies on the comparative stability of cloned and non-cloned code have made contradictory conclusions.

A number of studies [76, 83, 153, 18, 56, 24, 133, 84, 74, 51] have identified some positive impacts of code clones. On the other hand, there are strong empirical evidences [99, 97, 94, 21, 69, 100, 135] of negative impacts of clones. Different studies investigated different aspects of clone stability. In addition, the studies vary in experimental settings such as clone detection tools and subject systems. Thus, differences in experimental settings might have contributed to the contradictory findings of the earlier studies on clone stability.

Addressing the important issue we introduced the concept of the uniform framework in our earlier studies [110, 122]. The key objective of this framework is to carry out analysis of comparative stability of cloned and non-cloned avoiding possible biases introduced by the differences in experimental settings. The concept of the uniform framework for the analysis of clone stability has been extended and applied for other studies [123, 121]. Our recent work [121] incorporates seven stability analysis methods implementing eight stability metrics for a comprehensive analysis of stability of code clones. However, our previous study [121] does not investigate the stability metrics for the subject systems used in original studies with different experimental settings used in the original studies. In this study, we present an empirical study by extending our uniform framework for the comparative analysis of stability of cloned and non-cloned code. We apply additional clone detection tool NiCad on the subject systems used in original studies. We evaluate five different stability metrics used in four earlier studies to analyze the comparative stability of cloned and non-cloned code. We systematically replicate the original studies with their original experimental settings (clone detection tools and subject systems).

We evaluate the comparative stability of cloned and non-cloned code by answering the research questions as in Table 5.1. Based on our analysis of stability within the uniform framework we have the following findings:

- The results of our overall analysis of stability of clones show that clones tend to be more stable than non-cloned code. However, the stability scenarios vary with experimental settings. Thus, although clones appear to be more stable in the higher number of cases, it is hard to draw a firm conclusion on the comparative stability of clones and non-cloned code.

Table 5.1: STABILITY MEASUREMENT METRICS AND THEIR IMPLICATIONS

Research Questions	**Stability Metrics Analyzed**
Q1. Do clones change more frequently than non-cloned code?	*Modification frequency (MF)* [56]
Q2. Do clones have higher probability of modification than non-cloned code?	*Modification Probability (MP)* [49]
Q3. Which code is comparatively older, cloned or non-cloned code?	*Average Last Change Dates (ALCD)* [86]
Q4. Do cloned methods have higher impact of changes than non-cloned method?	*Impact* [97]
Q5. Do clones have higher likelihood of changes than non-cloned code?	*Likelihood* [97]
Q6. To what extent stabilities of different types of clones are different?	*Clone-type based Analysis*

- Our study shows that Type 1 and Type 3 clones are comparatively less stable than Type 2 clones. Thus, Type 1 and Type 3 clone may need to be prioritized for management.
- Comparative stability of clone and non-cloned code varies with systems, stability metrics measured and the experimental settings. In most cases the overall conclusions regarding clone stability from our extended study agree with the findings from replication studies on the original systems. Thus, it shows that development strategy for a particular software system and the evolution of the system over time may have an influence on the comparative stability of cloned and non-cloned code.

5.2 Uniform Framework for Stability Analysis

Different earlier studies analyzed the comparative stability of cloned and non-cloned code from different perspectives. The studies have different conclusions regarding the comparative stability of cloned and non-cloned code. Notably, the studies differ in the experimental settings such as clone detection tools used, subject systems analyzed, and the stability metrics measured. Thus, differences in experimental settings might have contributed to the contradictory findings of the earlier studies on clone stability. In this study, we investigate the stability metrics within a uniform framework by applying a different clone detection tool (NiCad) on the subject systems from the original studies. We compare the results from our new experimental settings with the results from our systematic replication of the original studies with original experimental settings.

Our framework is implemented in Java with MySQL for the back-end database. We implement four different methodologies with five stability metrics. As we store the data for the candidate systems in the database, processed change and clone information from the candidate subject systems can be shared among the methodologies where applicable. We analyze the stability metrics for different clone types on the subject systems used in the corresponding original studies. For systematic replication of the selected previous methodologies, we use clone detection tools CCFinderX, iClones and Simian as used in the original studies.

5.2.1 Metrics Analyzed

To analyze the comparative stability of cloned and non-cloned code we implement four methodologies and compare five different stability metrics. We briefly discuss the stability metrics we investigated in the following sections. Our definitions and equations in this section are taken from our earlier study [121]:

(1) Modification frequency (MF): Hotta *et al.* [56] measure the modification frequency of cloned (MF_c) and non-cloned (MF_n) code using the following equations:

$$MF_c = \frac{\sum_{r \epsilon R} MC_c(r)}{|R|} \times \frac{\sum_{r \epsilon R} LOC(r)}{\sum_{r \epsilon R} LOC_c(r)} \tag{5.1}$$

$$MF_n = \frac{\sum_{r \epsilon R} MC_n(r)}{|R|} \times \frac{\sum_{r \epsilon R} LOC(r)}{\sum_{r \epsilon R} LOC_n(r)} \tag{5.2}$$

In Equation 5.1 and Equation 5.2, R is the number of revisions of any particular candidate system. For a given revision r, the terms $MC_c(r)$ and $MC_n(r)$ are the number of modifications in the cloned and non-cloned code regions respectively between revisions r and $(r+1)$. Here, $LOC(r)$ is the system size in Lines of Code (LOC) for revision r and the size of cloned and non-cloned LOCs for revision r are represented by $LOC_c(r)$ and $LOC_n(r)$ respectively. Modification frequency is calculated by considering a block of consecutive modified lines as a single modification count.

(2) Modification Probability (MP): Modification Probability (MP) proposed by Göde and Harder [49] calculates the probabilities of modification of cloned and non-cloned code with respect to addition, deletion and modification to tokens. Incremental clone detection tool iClones [50] was used in the original study. Modification probabilities of cloned (MP_c) and Non-cloned (MP_n) code are calculated by the following equations:

$$MP_c = \frac{\sum_{r \epsilon R} A_c(r) + D_c(r) + C_c(r)}{\sum_{r \epsilon R} Tok_c(r)} \tag{5.3}$$

$$MP_n = \frac{\sum_{r \epsilon R} A_n(r) + D_n(r) + C_n(r)}{\sum_{r \epsilon R} Tok_n(r)} \tag{5.4}$$

In Equation 5.3 and Equation 5.4, R is the set of all revisions of the candidate subject system. $A_c(r)$, $D_c(r)$, and $C_c(r)$ are the total number of tokens added, deleted and changed respectively in the cloned regions of revision r. Similarly, $A_n(r)$, $D_n(r)$, and $C_n(r)$ are the total number of tokens added, deleted and modified in the non-cloned regions of revision r. Here, $Tok_c(r)$ and $Tok_n(r)$ represents total number of tokens in cloned and non-cloned code respectively in revision r. Equation 5.3 and Equation 5.4 basically determine the ratios of the modified tokens to the total number of tokens in cloned and non-cloned regions respectively which we measure as modification probabilities. However, the original study by Göde and Harder [49] termed these modification probabilities MP_c and MP_n as overall instability of cloned and non-cloned code.

(3) Average Last Change Date (ALCD): Krinke [86] introduced the stability metric Average Last Change Date for cloned and non-cloned code to analyze the comparative stability. If clones have average last change dates earlier than that of non-cloned code, it can be said that cloned code is comparatively older than non-cloned code and thus more stable. The average last change dates of cloned ($ALCD_c$) and non-cloned ($ALCD_n$) are calculated by taking the average date-difference between the last change date of the selected code and the last revision date. Average last change dates for cloned ($ALCD_c$) and non-cloned ($ALCD_n$) code are calculated using the following equations:

$$ALCD_c = ODCL + \frac{\sum_{l \epsilon CL} (DATE(l) - ODCL)}{|CL|} \tag{5.5}$$

$$ALCD_n = ODNL + \frac{\sum_{l \epsilon NL} (DATE(l) - ODNL)}{|NL|} \tag{5.6}$$

In Equation 5.5 and Equation 5.6, $ODCL$ and $ODNL$ are respectively the oldest change dates of cloned and non-cloned lines in the last revision of the candidate subject system. CL and NL are the sets of cloned and non-cloned lines in the last revision. $DATE(l)$ is the change date corresponding to the source code line l. Change Dates are extracted by applying the SVN *blame* command on the last revision of the subject systems.

In this study, Krinke analyzed three open-source Java systems using Simian [3] clone detector considering only Type 1 clones. Kirnke's study [86] concluded that cloned code is more stable than non-cloned code. In our study, we apply NiCad clone detection tool and analyze the stability with respect to Type 1, Type 2 and Type 3 clones.

(4) Impact: Lozano and Wermelinger [97] introduced the stability metric *impact* to evaluate the impacts of clones on software maintenance. This metric measures the extent to which components (methods) of a software system is affected (get changed) when a particular method is changed. The impact of a method m is defined by the average percentage of the system that gets changed whenever m changes during its evolution. The impacts are measured for both cloned and non-cloned period of each methods lifespan. Impacts of cloned and non-cloned methods are calculated by the average number of methods changed in the commits where method m gets changed. Impacts of cloned (ICC) and non-cloned (INC) code are calculated using the following equations:

$$ICC = \frac{\sum_{m \epsilon M_c} Impact_c(m) + \sum_{m \epsilon M_{sc}} Impact_c(m)}{|M_c| + |M_{sc}|} \tag{5.7}$$

$$Impact_c(m) = \frac{\sum_{c \epsilon CCP(m)} CCM(c)}{|CCP(m)|} \tag{5.8}$$

$$INC = \frac{\sum_{m \epsilon M_n} Impact_{nc}(m) + \sum_{m \epsilon M_{sc}} Impact_{nc}(m)}{|M_n| + |M_{sc}|} \tag{5.9}$$

$$Impact_{nc}(m) = \frac{\sum_{c \epsilon CNP(m)} CCM(c)}{|CNP(m)|} \tag{5.10}$$

In Equation 5.7 and Equation 5.8, M_c, M_{sc}, and M_n are the sets of always cloned, sometimes cloned and never cloned methods. $Impact_c(m)$ is the impact of the cloned period of method m. $CCP(m)$ is the set of all commits where m was changed during its cloned period. $CCM(c)$ is the count of methods changed in commit c. We also calculate the average impact of never cloned methods and the non-cloned periods of sometimes cloned methods. We term this impact as the impact of non-cloned code (INC). INC is calculated according to Equation 5.9 and Equation 5.10. In Equation 5.9 and Equation 5.10, $Impact_{nc}(m)$ is the impact of the non-cloned period of method m. $CNP(m)$ is the set of all commits where m was changed during its non-cloned period.

In the original study, Lozano and Wermelinger [97] analyzed impact at method level granularity using the clone detection tool CCFinderX [1]. They separated methods as always cloned, never cloned and sometimes cloned categories by analyzing their evolution. They also considered fully and partially cloned methods in their analysis. They performed the origin analysis of methods in consecutive revisions to map methods in successive revisions. We calculate the average impact of always cloned methods and the cloned periods of sometimes cloned methods. We term this impact as the impact of cloned code (ICC).

(5) Likelihood: Lozano and Wermelinger in the same study [97], analyzed the metric *likelihood* of methods. The likelihood of change of a method m during its cloned period (or non-cloned period) is the ratio between the number of changes to m and the total number of changes to the system (all methods) during the cloned period (or the non-cloned period). We calculate the likelihood of cloned (LCC) and non-cloned code (LNC). LCC is the average likelihood considering always cloned methods and the cloned periods of the sometimes cloned methods. We calculate LCC according to Equation 5.11 and Equation 5.12.

$$LCC = \frac{\sum_{m \epsilon M_c} Likelihood_c(m) + \sum_{m \epsilon M_{sc}} Likelihood_c(m)}{|M_c| + |M_{sc}|} \tag{5.11}$$

$$Likelihood_c(m) = \frac{\sum_{c \epsilon CCP(m)} NCM(m,c)}{\sum_{c \epsilon CCP(m)} \sum_{m_i \epsilon M} NCM(m_i,c)} \tag{5.12}$$

$$LNC = \frac{\sum_{m \epsilon M_n} Likelihood_{nc}(m) + \sum_{m \epsilon M_{sc}} Likelihood_{non-cloned}(m)}{|M_n| + |M_{sc}|} \tag{5.13}$$

$$Likelihood_{nc}(m) = \frac{\sum_{c \epsilon CNP(m)} NCM(m,c)}{\sum_{c \epsilon CNP(m)} \sum_{m_i \epsilon M} NCM(m_i,c)} \tag{5.14}$$

In Equation 5.11 and Equation 5.12, $Likelihood_c(m)$ is the likelihood of the method m during its cloned period. $NCM(m, c)$ is the number of changes to the method m in commit c. M is the set of all methods. We also calculate LNC, the average likelihood considering never cloned methods and the non-cloned periods of the sometimes cloned methods, according to Equation 5.13 and Equation 5.14. In Equation 5.14, $Likelihood_{nc}(m)$ is the likelihood of the method m during its non-cloned period. Lozano and Wermelinger performed this study [97] on five open source Java systems and concluded that cloned methods are more likely to be modified compared to the non-cloned methods. Also, cloned methods seem to increase maintenance efforts considerably.

5.3 Experimental setup

5.3.1 Subject Systems

Table 5.2 shows the list of subject systems used in our study. For our empirical study, we selected 16 subject systems used in the original studies on stability analysis. The subject systems are from diverse application domains, different size, and length of evolution histories and the systems are implemented with Java and C++.

5.3.2 Code Extraction

We extracted all the revisions of the subject systems from the corresponding SVN repositories. We created local mirrors of the corresponding repositories to speed up the extraction of source code revisions for each of the subject systems studied. As our analysis is based on the changes to the source code only, we extract the source code files only to optimize the source code extraction process. For all the metrics except the average last change date, we need to process all the source code revisions in the range as listed in Table 5.2. For Krinke's [86] methodology we only need to process the last revision of the source code and the change dates for source code from the repository.

5.3.3 Preprocessing

To avoid any bias in the measurement of stability metrics due to formatting differences in the source code, we apply preprocessing steps on the source code revisions. First, we remove all blank lines and comments from the source code. Then we normalize the source code by shifting isolated left or right braces to the end of the previous source code line. We do not apply any preprocessing on the source code for the calculation of Average Last Change Date proposed by Krinke [86] because this methodology works on the output of the SVN blame command. The output of the blame command is based on the original file in the repository. However, in this case we simply ignore the output of the blame command for blank lines and comments.

Table 5.2: SUBJECT SYSTEMS

	Systems	Application Domains	LOC	Revisions
Java	Adserverbeans	Ad Server	5008	125
	DatabaseToUML	Database Modeling Tool	7063	60
	EclEmma	Code Coverage Tool	8617	1736
	Freecol	Game	87930	6000
	OpenYmsg	Messenger	8821	304
	Squirrel	SQL Client	332635	6737
	Threecam	3D CAM Application	2658	14
	ArgoUML	UML Modeling Application	157573	19915
	JEdit	Text Editor	75967	6000
	Ganttproject	Project Management	58694	2629
	JBoss	Application Server	146383	3000
	Columba	Email Client	93084	465
C++	FileZilla	FTP Client	105251	7634
	Gamescanner	Game Tracker	18374	457
	Tritonn	Database Engine	23033	100
	Winmerge	File Merging Tool	99231	7618

LOC = LOC in Last Revision

5.3.4 Change Detection

We detect source code changes in successive revisions by applying UNIX *diff* command on two corresponding versions of each of the source code files. We list the files changed in each revision from the SVN repository information. We apply *diff* to only the files changed between two successive revisions to speed up the change extraction process. We store change information as the addition, deletion and changes to lines regarding corresponding lines numbers of associated files. We count changes to clones and non-cloned code for each revision by determining how many added, deleted or changed lines belong to cloned and non-cloned code respectively. This information is used for the calculation of stability metrics.

5.3.5 Clone Detection

For this study we use hybrid clone detection tool NiCad [144] for our stability analysis. NiCad detects both exact (Type 1) and near-miss (Type 2 and Type 3) clones with high accuracy [145]. Our settings for NiCad clone detector is shown in Table 5.3. For our systematic replication of the original studies we use CCFinderX

Table 5.3: NiCad SETTINGS FOR THE STUDY

Parameters	Values
Minimum Size	5 lines
Maximum Size	500 lines
Granularity	Method Level
Threshold	0% (Type 1, Type 2), 30% (Type 3)
Identifier Renaming	blindrename (Type 2, Type 3)

[1], iClones [50] and Simian [3] for the corresponding methodologies as these tools were used in the original studies. For CCFinderX we detect clones of minimum 30 tokens with default settings for other parameters. For Simian, we use the default configuration that defines 6 lines as the minimum clone size. For, iClones we also use the default configuration with 100 tokens for the minimum clone size.

5.3.6 Detection of Methods and Method Genealogy

We apply CTAGS [2] to detect methods from each revision of the source code files. A method is identified by file name, method name, signature, start line, end line, class name (Java) and package name (Java). As in change detection, we apply CTAGS only on the files added or changed in a particular revision as determined from the information extracted from the corresponding SVN repository. For unchanged files, we simply reuse the method information extracted in from the revision immediately preceding the current revision. Method information is saved for each revision and then we map methods across the revisions by analyzing method genealogies by the procedure as follows.

When we have all the methods detected from all revisions of the source code for a subject system, we analyze method genealogies using the technique proposed by Lozano and Wermelinger [97]. We assign unique IDs for each method identified and alive in many different revisions of a subject system. This approach matches the method signature to map methods between successive revisions. In case of unmatched signature due to changes to method signature or newly added methods, text similarity-based search is carried out to map methods. We use Strike A Match [4] algorithm to determine textual similarity. Methods that do not find a match even with text-based matching are treated as newly added methods.

5.3.7 Mapping Changes

Once we have all change and method information extracted from the source code revisions, changes are mapped to methods based on the line numbers associated with change information. The line-based changed information is also used to map changes to cloned and non-cloned methods in each revision.

5.3.8 Calculation of the Metrics

When we have all the information related to methods, clones, and changes in revisions, we calculate the stability metrics Modification Frequency, Modification Probability, Average Last Change Date, Impact and Likelihood for cloned and non-cloned code for the subject systems selected from the corresponding original studies. We calculated the metrics both for NiCad and the other clone detection tools used in the original study. We also carry out clone-type centric analysis of the comparative stability with clones from NiCad tool as NiCad can detect Type 1, Type 2 and Type 3 clones.

5.4 Experimental Results

This section represents the results of our empirical study regarding the comparative stability of cloned and non-cloned code. We compare the values for stability metrics selected from the earlier studies. We apply clone detection tool NiCad on the subject systems used in the original studies. We compare stability of cloned and non-cloned code regarding different clone types (Type 1, Type 2 and Type 3). We also systematically replicated the original studies with original experimental settings. This gives us the opportunity to compare our results with the results from original experimental settings.

Comparison Method: For each of the metrics analyzed, we define decision points based on the corresponding metric values for cloned and non-cloned code. Each decision point represents whether cloned code is less stable than non-cloned code (and vice versa). For example, for a particular subject system and particular clone type, the comparative metric values for cloned and non-cloned code is interpreted as a decision and represented by a decision point. The proportional distribution of decision points determines our conclusion on the comparative stability of cloned and non-cloned code. Here, we group decision points into two categories. *Category 1* represents the decision points where *"Clones are more stable than non-cloned code"* and *Category 2* represents the decision points where *"Clones are less stable than non-cloned code"*.

Replication of the Original Studies: We have replicated the original studies using our uniform framework. For the purpose of replication, we used the same clone detection tools and the subject systems used in the original studies. One important purpose of the replication is to determine whether our implemented uniform framework produces experimental results that are equivalent to the results of the original studies. If the results are equivalent, it ensures the reliability of our uniform framework for investigating clone stability. We compare the findings of our replicated experiments with the original setup with the corresponding findings from our extended study using NiCad clone detection tool.

5.4.1 Answer to RQ1: Do clones change more frequently than non-cloned code?

To answer this research question, we compare the comparative modification frequencies of cloned and non-cloned as proposed by Hotta *et al.* [56]. Modification frequencies of cloned (MF_c) and non-cloned code (MF_n) are calculated using the Equation 5.1 and Equation 5.2 as defined in Section 5.2.1. The metric *modification frequency* measures the frequency of changes to code region (consecutive lines of code). Code regions with lower modification frequency are more stable than code with higher modification frequency. We implement Hotta *et al.*'s [56] methodology using NiCad clone detector and subject systems used in the original studies. We also systematically replicate the original study with the original experimental settings for comparison. We present our results of comparative modification frequencies of cloned and non-cloned code in Table 5.4, Table 5.5 and Table 5.6.

Table 5.4 represents the modification frequencies of cloned and non-cloned code for seven subject systems used in the original study by Hotta *et al.* [56]. We apply NiCad [144] tool to detect Type 1 Type 2 and Type 3 clones. We compare the corresponding modification frequencies for cloned and non-cloned code to assign decision for each decision point. Table 5.4 shows the modification frequencies by considering all types of clones together. Here, for all seven subject systems the modification frequency of cloned code is less than the modification frequency of non-cloned code (*i.e.,* MF_c<MF_n) and thus we assign the decision symbol '$\oplus$' for all seven decision points. These results show that *cloned code is more stable than non-cloned code* which agrees with the original findings of Hotta *et al.* [56].

Now, Table 5.5 shows the comparative modification frequencies of cloned and non-cloned code regarding Type 1 , Type 2 and Type 3 clones. We used NiCad to detect individual types of clones separately. Here, for seven subject systems, there are seven decision points for each clone types. For Type 1 clones, five decision points (71.42% of seven decision points) belong to Category 1 indicating that clones are more stable than non-cloned code. However, for two decision points (EclEmma and ThreeCam) the modification frequencies of clones are more than that of non-cloned code. For these two systems (28.57%), clones are less stable compared to non-cloned code regarding Type 1 clones. Again, for all the decision points for Type 2 and Type 3 clones, the modification frequencies of cloned code are less than that of non-cloned code. Thus, our clone-type centric results show that clones are more stable compared to non-cloned code.

Results from the systematic replication of the original study: We replicated the original study of Hotta *et al.* [56] using Simian [3] clone detection tool as it was used in the original study. We used the default configurations for Simian as in the original study. This setting considers six lines of code as minimum clone size. We measured the modification frequencies of cloned (MF_c) and non-cloned (MF_n) code for the same subject systems used in the original study. Table Table 5.6 shows the modification frequencies of cloned and non-cloned code for the systems considering all types of clones. For all the subject systems except Gamescanner our findings agree with the overall conclusion of Hotta *et al.* that modification frequencies for cloned code are less than the modification frequencies of non-cloned code. These imply that cloned code is more stable than non-cloned code. Due to some small variations in the experimental settings such as the

Table 5.4: MODIFICATION FREQUENCIES BY HOTTA ET AL.'S METHODOLOGY WITH NiCad

		NiCad-Combined			
	Systems	MF_c	MF_n	Remark	#Revisions
Java	Adserverbeans	43.5907	56.3898	⊕	125
	DatabaseToUML	1.1011	13.72	⊕	60
	EclEmma	9.8875	15.0365	⊕	1736
	Freecol	7.5802	14.3646	⊕	6000
	OpenYmsg	9.9336	18.0713	⊕	304
	Squirrel	12.2158	17.3608	⊕	6737
	Threecam	8.5607	10.6203	⊕	14

MF_c, MF_n= Modification Frequency of Cloned and Non-cloned Code Respectively

$\oplus$= MF_c <MF_n (Category 1, CLONES MORE STABLE)

$\ominus$= MF_c >MF_n (Category 2, CLONES LESS STABLE)

Table 5.5: MODIFICATION FREQUENCIES BY HOTTA et al.'s METHODOLOGY FOR DIFFERENT CLONE TYPES WITH NiCad

	Type 1			**Type 2**			**Type 3**		
Systems	MF_c	MF_n	*Rem*	MF_c	MF_n	*Rem*	MF_c	MF_n	*Rem*
Adserverbeans	11.7013	55.3062	⊕	31.5992	55.2106	⊕	50.4705	54.8611	⊕
DatabaseToUML	0.0	11.7686	⊕	1.9653	12.6223	⊕	0.0	12.8991	⊕
EclEmma	36.9608	19.6209	⊖	6.6908	14.55	⊕	10.1772	14.8862	⊕
Freecol	3.8082	13.9128	⊕	8.7449	13.8467	⊕	7.9482	14.2779	⊕
OpenYmsg	4.2721	17.2446	⊕	11.1221	17.3044	⊕	9.7049	17.8842	⊕
Squirrel	12.0657	16.2987	⊕	11.6221	16.2025	⊕	12.2365	16.8504	⊕
Threecam	86.441	10.206	⊖	0.0	10.6146	⊕	3.5007	10.8499	⊕

MF_c, MF_n= Modification Frequencies of Cloned and Non-cloned Code respectively

$\oplus$= MF_c <MF_n (Category 1, CLONES MORE STABLE)

$\ominus$= MF_c >MF_n (Category 2, CLONES LESS STABLE)

Table 5.6: MODIFICATION FREQUENCIES BY HOTTA et al.'s METHODOLOGY WITH ORIGINAL SETUP

		Simian-Combined			
	Systems	MF_c	MF_n	Remark	#Revisions
Java	Adserverbeans	32.0335	60.0428	$\oplus$	125
	DatabaseToUML	2.9124	11.8236	$\oplus$	60
	EclEmma	15.7983	28.8905	$\oplus$	1736
	Freecol	6.1282	14.6388	$\oplus$	6000
	OpenYmsg	27.4103	16.8265	$\ominus$	304
	Squirrel	9.8247	19.3574	$\oplus$	6737
	Threecam	4.0774	10.7737	$\oplus$	14
C++	FileZilla	10.7072	12.2153	$\oplus$	7634
	Gamescanner	23.6345	22.3518	$\ominus$	457
	Tritonn	77.8368	113.9316	$\oplus$	100
	Winmerge	5.1693	12.7347	$\oplus$	7618

MF_c= *Modification Frequency of Cloned Code*

MF_n= *Modification Frequency of Non-Cloned Code*

$\oplus$= MF_c <MF_n *(Category 1, CLONES MORE STABLE)*

$\ominus$= MF_c >MF_n *(Category 2, CLONES LESS STABLE)* number of revisions analyzed, the values of the metrics are not comparable by absolute values. However, our conclusions for subject systems regarding the modification frequencies of cloned and non-cloned code agree with the findings of the original study in most cases.

Summary: Our results regarding the modification frequency show that clones are more stable than non-cloned code. This conclusion also holds for Type 1, Type 2 and Type 3 clones. Our findings agree with the overall conclusion of the original study.

5.4.2 Answer to RQ2: Do clones have higher probability of modification than non-cloned code?

Modification probability [121] (originally defined as *overall instability* [49]) measures the proportion of cloned and non-cloned tokens get modified per commit operation. Here, code region with higher modification probability is said to be less stable. To measure modification probability of cloned (MP_c) and non-cloned (MP_n) code we use Equation 5.3 and Equation 5.4 respectively as defined in Section 5.2.1. Modification probability is calculated considering addition, deletion, and modification of tokens in a code region. For our measurement of modification probability, we apply NiCad clone detection tool to detect Type 1 Type 2 and Type 3 clones. We used subject systems from the original study by Göde and Harder [49]. We also systematically replicate the original study using the original experimental settings (subject systems, clone detection tools, and settings) to compare our findings. Our experimental results are shown in Table 5.7 and Table 5.8. The results from our replication of the original study are shown in Table 5.9.

First, we compare the comparative modification probability of cloned and non-cloned code for NiCad considering all types of clones as shown in Table 5.7. Here, for both the subject systems ArgoUML and Squirrel, the modification probabilities of clones is less than the modification probabilities of non-cloned code (*i.e.,* $MP_c<MP_n$ and so the decision symbol '$\oplus$'). Thus, when we consider all clone types combined, cloned code exhibits higher stability than non-cloned code. These findings agree with the conclusion from the original study that clones are more stable.

Again, Table 5.8 shows the comparative modification probabilities for cloned and non-cloned code for Type 1, Type 2 and Type 3 clones for NiCad clone detector. From our experimental results based on clone types, we observe that for both ArgoUML and Squirrel the modification probabilities of cloned code is less than that of non-cloned code for all clone types (*i.e.,* $MP_c<MP_n$ and so the decision symbol '$\oplus$'). Thus, our clone-type centric analysis shows that clones are more stable compared to non-cloned code. This conclusion also agrees with the overall conclusion of the original study by Göde and Harder [49].

Results from the systematic replication of the original study: We replicated the study by Göde and Harder [49] with the same subject systems and the clone detection tool iClones [50] used in the original study. We used the default configuration settings for iClones to detect clones of all Types (Type 1, Type 2

Table 5.7: MODIFICATION PROBABILITY BY GÖDE et al.'s METHODOLOGY WITH NiCad

		NiCad-Combined			
	Systems	MP_c	MP_n	Remark	#Revisions
Java	ArgoUML	0.00013825775	0.00015787818	⊕	15000
	Squirrel	0.00008689886	0.00009804694	⊕	6737

MP_c, MP_n are Modification Probability of Cloned and Non-Cloned Code Respectively

⊕= $MP_c < MP_n$ (Category 1, CLONES MORE STABLE)

⊖= $MP_c > MP_n$ (Category 2, CLONES LESS STABLE)

Table 5.8: MODIFICATION PROBABILITY BY GÖDE et al.'s METHODOLOGY WITH NiCad AND ORIGINAL SYSTEMS

	Type 1			**Type 2**			**Type 3**		
Systems	MP_c	MP_n	*Rem*	MP_c	MP_n	*Rem*	MP_c	MP_n	*Rem*
ArgoUML	0.0001328	0.0001553	⊕	0.0001237	0.0001528	⊕	0.0001118	0.0001363	⊕
Squirrel	0.0000886	0.0000957	⊕	0.0000769	0.0000963	⊕	0.0000859	0.0000973	⊕

MP_c, MP_n are Modification Probability of Cloned and Non-Cloned Code Respectively

⊕= $MP_c < MP_n$ (Category 1, CLONES MORE STABLE)

⊖= $MP_c > MP_n$ (Category 2, CLONES LESS STABLE)

Table 5.9: MODIFICATION PROBABILITY BY THE METHODOLOGY OF GÖDE AND HARDER WITH ORIGINAL SETUP

		iClones-Combined			
	Systems	MP_c	MP_n	Remark	#Revisions
Java	ArgoUML	0.000166443	0.000153663	⊖	15000
	Squirrel	0.000093629	0.000095191	⊕	6737

MP_c, MP_n are Modification Probability of Cloned and Non-Cloned Code Respectively

⊕= $MP_c < MP_n$ (Category 1, CLONES MORE STABLE)

⊖= $MP_c > MP_n$ (Category 2, CLONES LESS STABLE)

and Type 3) for a minimum size of 100 tokens. We measure modification probabilities for cloned (MP_c) and non-cloned (MP_n) code for the subject systems ArgoUML and Squirrel used in the original study. Our results in Table 5.9 show that for Squirrel the modification probability of cloned code is lower than the modification probability of non-cloned code (*i.e.,* the decision symbol '$\oplus$'). This implies that cloned code is more stable than non-cloned code which agrees with the original findings by Göde and Harder. However, for ArgoUML the modification probability of cloned code is slightly higher than the modification probability of non-cloned code (*i.e.,* the decision symbol '$\ominus$'). This finding for ArgoUML differs from the original study. We have used the most updated version of iClones available which is different from the version used in the original study. Differences in clone detection results may have significant impact on the stability analysis as mentioned by different existing studies [97, 100, 56].

Summary: Our results of the empirical evaluation of the comparative modification probability of cloned and non-cloned code show that clones tend to have lower modification probability compared to non-cloned code for both cases when all clone types are combined or considered separately. Thus, clones are more stable than non-cloned code. However, from our systematic replication, finding regarding one of the two subject systems differ from the original study possibly due to minor variations in experimental settings.

5.4.3 Answer to RQ3: Which code is comparatively older, cloned or non-cloned code?

To analyze the stability metric *Average Last Change Date (ALCD)* proposed by Kirnke [86] we applied NiCad Clone detector on the last revision of each subject systems used in the original study. We measure the comparative values for average last change dates for cloned ($ALCD_c$) and non-cloned ($ALCD_n$) code for all clone types combined and also for considering Type 1, Type 2 and Type 3 clones separately. $ALCD_c$ and $ALCD_n$ is calculated according the Equation 5.5 and Equation 5.6 as defined in Section 5.2.1. For the calculation of the average last change dates, SVN *blame* command is applied on the last revision of the source code to get the last change dates of the source code lines. The average is calculated based on the date differences from the last date of change as discussed in Section 5.2.1. We present our results for combined clone types and for individual clone types in Table 5.10 and Table 5.11 respectively. We also replicate the original study by Krinke using Simian clone detector and the subject systems from the original study. Table 5.12 shows our results of the replicated study with original settings.

Table 5.10 shows the comparative values for $ALCD_c$ and $ALCD_n$ for all clone types combined for NiCad clone detector. Here, the average last change dates for cloned and non-cloned code for ArgoUML is 1-APR-2007 and 6-APR-2007 respectively. This shows that cloned code is comparatively older than non-cloned code. Again, for JEdit we observe that average last change date for clones (8-JAN-2007) is newer than that of non-cloned code (30-MAY-2006). This implies that non-cloned code in JEdit is older than cloned code. Interestingly, for all clone types combined (using NiCad), both ArgoUML and JEdit exhibit stability scenarios opposite to the corresponding results from the original study.

Now, in Table 5.11 we present the clone-type centric stability results for NiCad clone detector regarding average last change dates. For ArgoUML, the average last change dates for Type 2 and Type 3 clones are older than that of non-cloned code. For Type 1 clones in ArgoUML, cloned code has the newer average last change date (27-MAR-2008) than that of non-cloned code (02-APR-2007). Again, for JEidt we observe that for Type 2 and Type 3 clones, non-cloned code is older than cloned code while Type 1 clones exhibit the opposite. Thus, based on our results on the original systems, we can not make concrete decision on the comparative stability of cloned and non-cloned code. However, we observe that for both ArgoUML and JEdit, the overall (NiCad-combined) comparative stability results are same as the results for Type 2 and Type 3 clones. While for both subject systems, results for Type 1 clones do not agree with the results based on all clone types combined. Thus, the distribution of the number of clones of each type in the subject systems and how those clones evolve during software evolution may influence the stability results.

Results from the systematic Replication of the original study: To evaluate our implementation of Krinke's [86] methodology to measure average last change date, we analyzed the same subject systems and used the text-based clone detection tool Simian as in the original study. We used the default settings for Simian with minimum clone size of 6 lines. For ArgoUML, we analyzed revision 19915 (18995 in the original study) and for JEdit we analyzed revision 24443 (19285 in the original study). Krinke's methodology applies SVN *blame* command on the last revision of a system to retrieve the last change dates for source code lines. Thus we have analyzed updated revisions than the revisions used in the original studies. Our experimental results for ArgoUML and JEdit are shown in Table 5.12. Our results show that for ArgoUML the average last change date ($ALCD_c$) for cloned code (5-JUL-2007) is newer than the average last change date (1-APR-2007) for non-cloned code ($ALCD_n$). This implies that clones are younger than non-clone code. This finding agrees with the original study. However, for JEdit the average last change date for cloned code (21-JUL-2006) is newer than the average last change date (3-JUN-2006) for non-cloned code. This finding does not agree with the overall conclusion of the original study by Krinke. For JEdit we analyzed a considerable number of more revisions which might have introduced differences in our findings for JEdit.

Again, when we compare our NiCad-combined results (Table 5.10) with results from original settings (Table 5.12), for both NiCad and Simian, cloned code is newer than non-cloned code. Thus, clones are less stable in JEDit which is opposite to the original findings for JEdit. For ArgoUML on the other hand, clones appear to be more stable in case of NiCad clone detector while it is opposite for Simian clone detector. Thus, we cannot make a decision on the comparative stability based on our findings regarding average last change date. However, we observe that stability scenarios may be influenced by the experimental settings and the subject systems.

Table 5.10: AVERAGE LAST CHANGE DATES BY KRINKE'S METHODOLOGY WITH NiCad CLONE DETECTOR

		NiCad-Combined			
	Systems	$ALCD_c$	$ALCD_n$	Remark	Revision
Java	ArgoUML	1-APR-2007	6-APR-2007	⊕	19915
	JEdit	8-JAN-2007	30-MAY-2006	⊖	24443

$ALCD_c$, $ALCD_n$= Average Last Change Date of Cloned and Non-cloned code respectively

⊕= $ALCD_c$ is older than $ALCD_n$ (Category 1, CLONES MORE STABLE)

⊖= $ALCD_c$ is newer than $ALCD_n$ (Category 2, CLONES LESS STABLE)

Table 5.11: AVERAGE LAST CHANGE DATES FOR DIFFERENT TYPES OF CLONES BY KRINKE'S METHODOLOGY WITH NiCad AND ORIGINAL SYSTEMS

	Type 1			**Type 2**			**Type 3**		
Systems	$ALCD_c$	$ALCD_n$	*Rem*	$ALCD_c$	$ALCD_n$	*Rem*	$ALCD_c$	$ALCD_n$	*Rem*
ArgoUML	27-MAR-2008	02-APR-2007	⊖	31-DEC-2006	07-APR-2007	⊕	09-MAR-2007	06-APR-2007	⊕
JEdit	22-NOV-2004	04-JUN-2006	⊕	03-JUL-2007	02-JUN-2006	⊖	31-DEC-2006	31-MAY-2006	⊖

$ALCD_c$, $ALCD_n$ = Average Last Change Date for Cloned and Non-cloned Code respectively

⊕= $ALCD_c$ is older than $ALCD_n$ (Category 1, CLONES MORE STABLE)

⊖= $ALCD_c$ is newer than $ALCD_n$ (Category 2, CLONES LESS STABLE)

Summary: Although we cannot make a firm conclusion on the comparative stability based on our results regarding average last change dates, we observe that comparative stability of cloned and non-cloned code may vary with experimental settings such as subject systems, clone detection tools and clone types.

5.4.4 Answer to RQ4: Do cloned methods have higher impact of changes than non-cloned method?

The stability metric *impact* proposed by Lozano and Wermelinger [97] measures the extent to which changes to cloned or non-cloned method affects other program entities (methods). *Impact* is calculated by counting the number of other methods need to change in consequence to changes to a cloned or non-cloned method. The comparative impacts of cloned (ICC) and non-cloned (ICN) methods are calculated using Equation 5.8 and Equation 5.10 respectively as defined in Section 5.2.1. We apply NiCad clone detector on each revision of the subject systems. we calculate the impact of cloned and non-cloned methods for Type 1, Type 2 and

Table 5.12: AVERAGE LAST CHANGE DATES BY KRINKE'S METHODOLOGY WITH ORIGINAL SETUP

		Simian-Combined			
	Systems	$ALCD_c$	$ALCD_n$	Remark	Revision
Java	ArgoUML	5-JUL-2007	1-APR-2007	$\ominus$	19915
	JEdit	21-JUL-2006	3-JUN-2006	$\ominus$	24443

$ALCD_c$, $ALCD_n$= Average Last Change Date of Cloned and Non-cloned Code Respectively

$\oplus$= $ALCD_c$ is older than $ALCD_n$ (Category 1, CLONES MORE STABLE)

$\ominus$= $ALCD_c$ is newer than $ALCD_n$ (Category 2, CLONES LESS STABLE)

Type 3 clones and for all clone types combined. We also calculate the impact using the original experimental settings from our systematic replication.

We present our results for combined clone types and for individual clone types in Table 5.13 and Table 5.14 respectively. We also replicate the original study using CCFinderX clone detector and the subject systems from the original study. Table 5.15 shows our results of the replicated study with original settings.

Table 5.13 shows the comparative impacts for cloned (ICC) and non-cloned (INC) for combined clone type for NiCad. Here, we observe that for all the subject systems Freecol, Ganttproject, and JBoss the impacts of cloned methods are higher than that of non-cloned methods *i.e.,* clones are less stable than non-cloned code (presented by '$\ominus$'). This result agrees with our findings from the original experimental settings as in Table 5.15.

Now, regarding individual clone types (in Table 5.14), we observe that for Type 1 clones in JBoss and Ganttproject, cloned methods have higher impacts than non-cloned methods ($ICC > INC$, *i.e.,* $\ominus$). However, for Type 1 clones in Freecol non-cloned methods exhibit higher impacts than cloned methods ($ICC < INC$, *i.e.,* $\oplus$). We observe impact scenario for Type 2 clones same as Type 1 clones. However, for Type 3 clones the impact for cloned methods is higher than non-cloned methods for all three subject systems. Thus, for 7 out of 9 cases (3 systems x 3 clone types) for comparative impacts, we observe that cloned methods tend to have higher impacts than non-cloned methods.

Results from the systematic replication of the original study: We evaluated the comparative impacts for our implementation of the study proposed by Lozano and Wermelinger [97] using subject systems and the same clone detection tool (CCFinderX) used in the original study. Table Table 5.15 shows the *impact* of cloned and non-cloned code for three subject systems used in the original study. Our results show that for all three subject systems the impact of cloned code (ICC) is greater than the impact of non-cloned code (INC). This suggests that clones are less stable than non-cloned code which agrees with the original study by Lozano and Wermelinger [97].

Table 5.13: IMPACT OF CLONED AND NON-CLONED CODE BY THE METHODOLOGY OF LOZANO AND WERMELINGER WITH NiCad CLONE DETECTOR

		NiCad-Combined			
	Systems	*ICC*	*INC*	Remark	#Revisions
Java	Freecol	0.00909026	0.00834031	⊖	6000
	Ganttproject	0.01631868	0.01089798	⊖	2629
	JBoss	0.01105989	0.01076336	⊖	3000

ICC, INC= Impact of Cloned and Non-cloned Code Respectively

⊕= *ICC <INC (Category 1, CLONES MORE STABLE)*

⊖= *ICC >INC (Category 2, CLONES LESS STABLE)*

Table 5.14: IMPACT OF CLONED AND NON-CLONED CODE BY THE METHODOLOGY OF LOZANO AND WERMELINGER WITH NiCad FOR DIFFERENT CLONE TYPES

	Type 1			**Type 2**			**Type 3**		
Systems	*ICC*	*INC*	*Rem*	*ICC*	*INC*	*Rem*	*ICC*	*INC*	*Rem*
Freecol	0.0060009	0.0076238	⊕	0.0070030	0.007684	⊕	0.0087800	0.008228	⊖
GanttProject	0.015905	0.014155	⊖	0.013858	0.015669	⊕	0.022679	0.018802	⊖
JBoss	0.0119205	0.008057	⊖	0.009775	0.008429	⊖	0.010701	0.009438	⊖

ICC, INC= Impact of Cloned and Non-cloned Code Respectively

⊕= *ICC <INC (Category 1, CLONES MORE STABLE)*

⊖= *ICC >INC (Category 2, CLONES LESS STABLE)*

Table 5.15: IMPACT OF CLONED AND NON-CLONED CODE BY THE METHODOLOGY OF LOZANO AND WERMELINGER WITH ORIGINAL SETUP

		CCFinderX-Combined			
	Systems	*ICC*	*INC*	Remark	#Revisions
Java	Freecol	0.002602425	0.002317386	⊖	6000
	Ganttproject	0.010625655	0.009936604	⊖	2629
	JBoss	0.005176758	0.004745167	⊖	3000

ICC, INC= Impact of Cloned and Non-cloned Code Respectively

⊕= *ICC <INC (Category 1, CLONES MORE STABLE)*

⊖= *ICC >INC (Category 2, CLONES LESS STABLE)*

Summary: Regarding the metric *impact*, our experimental results show that the comparative impact of clones is higher than that of non-cloned code. Thus, clones tend to be less stable compared to non-cloned code and this conclusion agrees with the findings from the original study.

5.4.5 Answer to RQ5: Do clones have higher likelihood of changes than non-cloned code?

The stability metric *Likelihood* proposed by Lozano and Wermelinger [97] measures the probability of change for a particular cloned or non-cloned method. The higher the *Likelihood* of change the less stable a method is. The comparative likelihood of cloned (LCC)and non-cloned (LCN) methods are calculated using Equation 5.12 and Equation 5.14 respectively as defined in Section 5.2.1. We apply NiCad clone detector on each version of the subject systems. We calculate the impact of cloned and non-cloned methods for Type 1, Type2 and Type 3 clones and for all clone types combined. We also calculate the likelihood using the original experimental settings from our systematic replication.

We present our results for combined clone types and for individual clone types in Table 5.16 and Table 5.17 respectively. We also replicate the original study using CCFinderX clone detector and the subject systems from the original study. Table 5.18 shows our results of the replicated study with original settings.

Table 5.16 shows the comparative likelihood for cloned (LCC) and non-cloned (LNC) for combined clone type for NiCad. Here, we observe that for all the subject systems Freecol, Ganttproject and JBoss the likelihood of clone is higher than that of non-cloned code *i.e.,* clones are less stable than non-cloned code ('$\ominus$'). This result agrees with our findings from the original experimental settings as in Table 5.18.

Now, for individual clone types as in Table 5.17, we observe that for all clone types and for all subject systems except for Type 1 and Type 2 clones in JBoss, clones exhibit a higher likelihood of changes than non-cloned code. This shows that clones tend to be less stable compared to non-cloned code. Here, for 7 out of 9 cases (3 systems x 3 clone types) for comparative likelihood, we observe that cloned code tend to have a higher likelihood of changes than non-cloned code. So, our results show that clones are less stable.

Results from systematic replication of the original study: Again, Table Table 5.18 shows the *likelihood* of the cloned and non-cloned code. The results show that for all three subject systems, the likelihood of the cloned code (LCC) is greater than the likelihood of the non-cloned code (LNC). This implies that clones are less stable than non-cloned code which agrees with the findings from the original study.

Summary: Regarding the metric *likelihood*, our experimental results show that the comparative likelihood of change to clones is higher than that of non-cloned code. Thus, clones tend to be less stable and this conclusion agrees with the findings from the original study.

Table 5.16: LIKELIHOOD OF CHANGES OF CLONED AND NON-CLONED METHODS BY THE METHODOLOGY OF LOZANO AND WERMELINGER WITH NiCad CLONE DETECTOR

		NiCad-Combined			
	Systems	*LCC*	*LNC*	Remark	#Revisions
Java	Freecol	0.01182758	0.00372643	⊖	6000
	Ganttproject	0.00620728	0.00329211	⊖	2629
	JBoss	0.01276509	0.00976206	⊖	3000

LCC, LNC= Likelihood of Cloned and Non-cloned Code Respectively

⊕= LCC <LNC (Category 1, CLONES MORE STABLE)

⊖= LCC >LNC (Category 2, CLONES LESS STABLE)

Table 5.17: LIKELIHOOD OF CLONED AND NON-CLONED CODE BY THE METHODOLOGY OF LOZANO AND WERMELINGER WITH NiCad FOR DIFFERENT CLONE TYPES

	Type 1			**Type 2**			**Type 3**		
Systems	*LCC*	*LNC*	*Rem*	*LCC*	*LNC*	*Rem*	*LCC*	*LNC*	*Rem*
Freecol	0.007089	0.003062	⊖	0.006179	0.003018	⊖	0.011433	0.003707	⊖
GanttProject	0.003865	0.001849	⊖	0.009034	0.001002	⊖	0.007280	0.002393	⊖
JBoss	0.0031003	0.004838	⊕	0.003199	0.005447	⊕	0.010426	0.008852	⊖

LCC, LNC= Impact of Cloned and Non-cloned Code Respectively

⊕= LCC <LNC (Category 1, CLONES MORE STABLE)

⊖= LCC >LNC (Category 2, CLONES LESS STABLE)

Table 5.18: LIKELIHOOD OF CHANGES OF CLONED AND NON-CLONED METHODS BY THE METHODOLOGY OF LOZANO AND WERMELINGER WITH ORIGINAL SETUP

		CCFinderX-Combined			
	Systems	*LCC*	*LNC*	Remark	#Revisions
Java	Freecol	0.007500568	0.002713309	⊖	6000
	Ganttproject	0.00421473	0.002518339	⊖	2629
	JBoss	0.008984112	0.004929813	⊖	3000

LCC, LNC= Likelihood of Cloned and Non-cloned Code Respectively

⊕= LCC <LNC (Category 1, CLONES MORE STABLE)

⊖= LCC >LNC (Category 2, CLONES LESS STABLE)

5.4.6 Answer to RQ6: To what extent stabilities of different types of clones are different?

For clone-type centric analysis, we consider the comparative values for cloned and non-cloned code for each clone type and for each stability metrics and for each subject systems analyzed. For example, in Table 5.5 we have modification frequencies for cloned and non-cloned code for each of the clone types. When we consider Type 1 clones we have 7 decision points (one for each subject system). As shown in Table 5.5, 5 out of 7 decision points regarding Type 1 clones show that clones are more stable ($\oplus$) and for the remaining two (2) cases (EclEmma and Threecam) clones are less stable ($\ominus$). Similarly, we consider the decision points for each of the stability metrics and for each of the subject systems for Type 1 clones. For modification probability, the decision points for both ArgoUML and Squirrel show that clones are more stable regarding Type 1 clones. Counting in the same way we have 10 out of 17 (58.82%) decision points showing that clones are more stable compared to non-clones (7 out of 17 decision points *i.e.,* 41.17%).

Now for Type 2 clones, 13 out of 17 (76.47%) decision points show that clones are more stable than non-cloned code. Again, the stability scenario for Type 3 clones is same as Type 1 clones with 10 out of 17 (58.82%) decision points showing that clones are more stable compared to non-cloned code. The comparative clone-type centric stability scenarios are shown in Figure 5.1. This shows that clones tend to be more stable compared to non-cloned code. Type 2 clones tend to be more stable compared to Type 1 and Type 3 clones.

Summary: Our clone-type centric analysis shows that clones tend to be more stable than non-cloned code and Type 1 and Type 3 clones are comparatively less stable compared to Type 2 clones.

5.4.7 Overall Analysis

When we consider all clone types combined we have two possible clone cases *i.e.,* for NiCad combined and the combined results from original studies. Then for NiCad-combined results, 10 out of 17 (58.82%) decision points show that clones are more stable. On the other hand, 7 out of 17 (41.17%) decision points for the original studies show that clones are more stable. Thus, when we aggregate the combined type results for all the metrics, our results with NiCad show the opposite stability scenarios compared to the results from original settings. This shows that even for the same subject systems, the differences in clone detection tools may result in different stability decision. Again, we observe that even for same clone detection tool and same subject system, stability scenario may differ based on the stability metrics. For example, for Type 1 and Type 2 clones with NiCad on JBoss and Freecol the comparative stability results is just opposite for *impact* (in Table 5.14) and *likelihood* (in Table 5.17). Thus, although we observe that clones tend to be more stable, the stability scenario may vary with experimental settings, the underlying development strategy and change patterns of the software systems during evolution. Again, we observe that despite new experimental

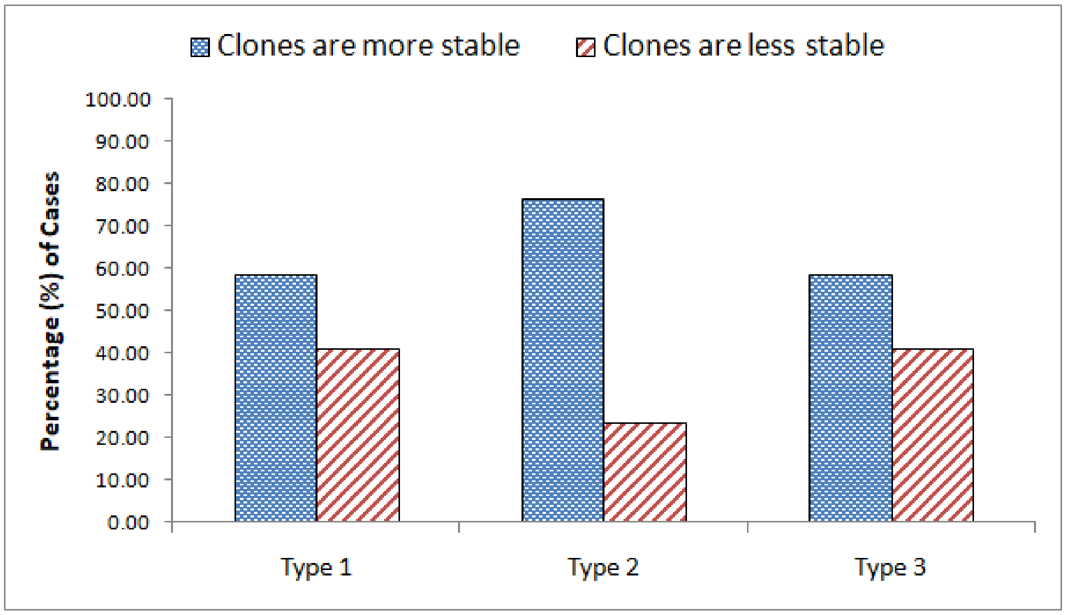


Figure 5.1: Overall Comparative stability of clones and non-cloned code for different clone types.

settings (using NiCad) the findings agree in most cases with the results from the original studies. This is an indication that the overall stability scenarios might also be system dependent *i.e.,* the specific design decisions, distribution and evolution of clones might have influences on the overall stability of clones in a particular software system. Thus, we cannot draw a general conclusion on the comparative stability of cloned and non-cloned code. However, we see that clones may have negative impacts on software systems and it varies system to system. Thus, software developers should be aware of code clones and clones need to be managed as it is also suggested by earlier research.

5.5 Threats to Validity

This study investigates the comparative stability of cloned and non-cloned code with a uniform comparison framework. Stability is one of the widely-investigated aspects of the impacts of clones. Here, the underlining assumption is that if clones are less stable *i.e.,* change more frequently as compared to non-cloned code, clones are likely to require more maintenance efforts and cost. However, code cloning and the impacts of clones might be influenced by many parameters such as specific design decision, programmers preferences, time constraints of software projects. This study considers only the changes or evolution of clones visible through source code analysis. We used systems from the original studies those are from diverse application domains, system size and with thousands of revisions to compensate the limitations.

We use NiCad for extended analysis of the original stability metrics on the subject systems used in original studies. NiCad detects Type 1, Type 2 and Type 3 clones that facilitates the clone-type centric analysis. We also apply the tools used in the original studies for systematic replication of the original studies. Stability analysis results may vary with changes in configurations of the tools and other experimental settings. This is known as confounding configuration problem as defined by Wang *et al.* [180]. NiCad settings we used in our study are considered to be standard and have been used in other studies [135, 122]. We used defaults settings for the tools CCFinderX, iClones and Simian used in the original studies.

As our analysis is based on the subject systems used in original studies, the majority of the subject systems in Java and C++. The inclusion of other systems may help more generalization of the findings. However, the scope of this study is to evaluate the original metrics with the original subject systems. We compare the results from the original studies with our NiCad-based analysis.

Again, there are other important perspectives of analyzing the impacts of clones on software systems such as bugs or inconsistent evolution. However, this study focuses on the comparative stability and we investigate how different evolutionary features such as stability and coupling are related to bugs in Chapter 6 and Chapter 7 respectively.

5.6 Related Work

There are a large body of research work investigating the impacts of clones on software maintenance and evolution. The studies focused primarily on evaluating the impacts of clones in terms of the degree of consistency in comparative evolution, measuring stability to study how frequently a code region is modified and the likelihood of introducing bugs during evolution.

One of the widely studied perspectives of evaluating the impacts of clone is *stability* [84, 56, 122]. Krinke [84] proposed a measure for the comparative stability of cloned and non-cloned code. His approach counts the modification in terms of the number of lines added, deleted and modified in cloned and non-cloned code. The more the number of changes to a code region, the less stable the code is. His study concluded that clone code is more stable than non-cloned code. Krinke's other study [86] measures the average last change date of the cloned and non-cloned code. The average last change dates are calculated by the average change date of the code measured from the modification history in the source code repository with respect to last revision. Mondal *et al.* later introduced the metric *average age* that measures how long a coned or non-cloned code remains unchanged on an average. This study concludes that clone code is older meaning more stable than non-cloned code. Gde and Harder [49] extended Krinke's [84] approach with token-based incremental clone detection tool iClones and experimented with different settings of clone detection parameters, clone types and their changes. This study agrees with the findings of Krinke [84] that cloned code is more stable than non-cloned code but this does not hold in case of deletion.

Hotta *et al.* [56], on the other hand, measure the stability of cloned and non-cloned code in terms of *modification frequency*. Their study concludes that cloned code has lower modification frequency and thus more stable than non-cloned code. Their approach counts the number of blocks (consecutive lines) modified in cloned and non-cloned code. The average modification counts per revision in cloned and non-cloned code are then used to measure the modification frequency. The less the modification frequency, the more stable the code region is.

Lozano *et al.* [99] investigated whether clones are harmful or not. Their study concludes that cloned code tend to be more frequently modified than non-cloned code. To assess the impacts of clones on software maintenance Lozano and Wermelinger [97] also investigated the impacts of clones on software maintenance. Their study shows that although the *likelihood* (the ratio of the number of change to an entity to the total number of changes in the system) in cloned code is not very different than in non-cloned code, in some cases the *impact* (percentage of the system affected by a change) of cloned methods is greater than that of non-cloned methods. Their study also suggests that the presence of clones may decrease the changeability of the code entity containing clones. Lozano and Wermelinger [100] also investigated the instability of cloned methods and concluded that clones tend to be less stable than non-cloned code.

Given the contradictory outcomes from existing studies possibly due to differences in experimental settings, Mondal *et al.* [122] carried out a comprehensive stability analysis within a uniform evaluation framework considering existing (taken from [56, 84, 86]) and their proposed metrics to measure stability. Their study concluded that there is no firm conclusion on the stability of cloned and non-cloned code, rather, the comparative stability varies with programming languages, clone types and overall system development strategies. Mondal *et al.* [111] also introduced the metrics *change dispersion* that measures how changes are scattered over the respective code region. Higher dispersion of changes are an indicator of code instability. In a recent study (Mondal *et al.* [121]), we comprehensively evaluate eight exiting stability metrics from seven existing methodologies. We replicate the original study to compare our results with the findings from original experimental settings. However, the study does not carry out comparative analysis on the subject systems with experimental settings beyond the original settings. This study extends and complements our previous study by applying new clone detection tool on the original subject systems for comparison of results with the results from original settings. This study also carries out clone-type centric analysis of the comparative stability of the clone and non-cloned code on the subject systems used in the original studies.

There are other studies [120, 60, 21, 22] on the impacts of clone on software systems with respect to bugs and inconsistencies. However, this study particularly focuses on the evaluation comparative of stability of cloned and non-cloned code with original experimental settings.

5.7 Summary

Existing studies of the comparative impacts of clones on software systems have resulted in contradictory outcomes. The differences in conclusions regarding the comparative stability of cloned and non-cloned code might have been influenced by the differences in the experimental settings in the existing studies. In this empirical study, we analyze the comparative stability of cloned and non-cloned code within a uniform framework. We focus on the evaluation of five stability metrics from four different existing methodologies. We systematically replicate each of the four methodologies using original experimental settings (subject systems and clone detection tools). This gives us the opportunity to compare the stability results from the original settings with the results from our extended study using NiCad clone detector on the original subject systems.

The key objective of the stability analysis is that if clones appear to be less stable compared to non-cloned code, we can decide that clone have negative impacts on the software systems.

From our overall analysis with respect to all the metrics, we observe that our findings with NiCad generally agree with the findings from the original experimental settings. Again, this study shows that the stability scenarios vary with experimental settings. Thus, it is hard to have a firm conclusion on the comparative stability of cloned and non-cloned code and the stability may vary with a particular experimental setting.

Our type-centric analysis suggests that clones tend to be more stable compared to non-cloned code. However, Type 1 clones tend to be more unstable compared to Type 2 and Type 3 clones. Thus, Type 1 clones may need to be prioritized in clone management tasks such as clone refactoring and tracking. As Type 1 clones are exact copies, corresponding changes more likely to be propagated to all exact copies of a code fragment. This may have contributed to the Type 1 clones to be more unstable compared to Type 2 and Type 3 clones. Our findings are important to assess the comparative stability of cloned and non-cloned code. These findings have the potentials to help making clone management decisions for better maintenance of clones.

From our analysis of the comparative stability of cloned and non-cloned code in this chapter and in Chapter 4, we observe that the impacts of clones might be influenced by many parameters. However, clones often exhibit higher instability compared to non-cloned code. To have a deeper understanding of the importance of the stability of clones, we further investigate whether stability of clones is related to bug-proneness, one of the key concerns related to clones. We present our empirical study on the relationships between stability and bug-proneness of clones in the next chapter.

Chapter 6

Analyzing the Relationships between Stability and Bug-proneness of Clones

We have studied the comparative stability of cloned and non-cloned code in Chapter 3 and Chapter 5. Although the comparative stability of cloned and non-cloned code varies with systems, studies show that clones often tend to be less stable compared to non-cloned code. The large body of existing research on the stability of clones shows the importance of the stability of clones. However, it is yet to know whether instability of clones has any further impacts on the software systems beyond the intuitive one, increased maintenance efforts due to frequent changes. Thus, it is important to investigate whether stability is related to other aspects of the impacts of clones on the software systems such as bugs. Frequent changes to clones may increase the likelihood of missing change propagation to the co-change candidates leading to inconsistencies or bugs. However, none of the existing studies investigate the relationships between stability and bug-proneness of clones. In this chapter, we present an empirical study that analyzes the relationships between stability and bug-proneness of clones using fine-grained change types. The findings of this study have been published in SCAM 2017[1].

We identify bug-fix commits by analyzing the commit messages from software repositories. We then identify the clones those are changed in the bug-fix commits as bug-prone clones. We then compare the stability of buggy and non-buggy clones considering the fine-grained syntactic change types and their significance. Our experimental results based on five open-source Java systems of different size and application domains show that (1) stability and bug-proneness of code clones are related and this relationship is statistically significant, (2) for both exact (Type 1) and near-miss (Type 2 and Type 3) clones, buggy clones tend to have higher frequency of changes than non-buggy clones, (3) the bug-proneness of Type 2 and Type 3 clones tend to be strongly related with their stability compared to Type 1 clones, and (4) the relation between the stability and the bug-proneness of clones with respect to fine-grained change types is likely to be influenced by the changes of low to medium significance. We believe that our findings are important and potentially useful in identifying and prioritizing candidate clones for management.

[1]M. S. Rahman and C. K. Roy, "On the Relationships between Stability and Bug-proneness of Code Clones: An Empirical Study", In Proceedings of the 17th IEEE International Working Conference on Software Code Analysis and Manipulation (SCAM 2017), 2017, pp. 131–140.

The rest of the chapter is organized as follows: Section 6.1 presents the background and motivation of the study. Section 6.2 defines and explains key terminology in context of our study. Section 6.3 briefly describes the taxonomy of changes used in this study. Section 6.4 represents the experimental settings and steps used in the study including the subject systems used, change extraction and classification procedure, and the metrics we measure. Section 6.5 represents the experimental results. Potential threats to the validity of the study is presented in section Section 6.6. Section 6.7 discusses the related works followed by the conclusion in Section 6.8.

6.1 Introduction

Code reuse by copying a code fragment and pasting it with or without modification results in duplicate copies of exact or similar code fragments in the code base. These exact or similar copies of code fragments are known as code clones. As clones constitute a significant proportion (between 7% and 23% [151]) of code bases of software systems, many studies investigated the impact of clones on software systems. A number of studies [56, 84, 86, 83, 133] show that clones are not harmful rather clones can be beneficial for the software systems [74]. Although clones have some obvious benefits like increasing productivity due to faster software development by code reuse, a good number of studies [71, 97, 100, 22, 34, 95, 94, 135] report instances of strong empirical evidence and conclude that clones have negative impacts on software systems. Thus, code clone is considered as one of the serious code smells.

One well-known claim against code clones is that code clones may introduce bugs or inconsistencies [71] in the software systems if clones are not changed consistently during software evolution. Moreover, if a code fragment with unidentified bugs is cloned, hidden bugs may also propagate to the new cloned fragments [94]. Given the important concerns related to clones, there are a great many studies on the bug-proneness of clones. Some studies focused on the identification and localization of bugs [95, 94, 65]. Some studies investigated the comparative bug-proneness of different types of clones [116, 185, 178] while some studies focused on analyzing the features or characteristics of clones related to bugs [163]. A number of studies also analyzed the bug-proneness of clones from the perspective of inconsistencies due to late propagation [18, 22, 171].

Stability refers to the extent to which a code fragment remain unchanged. Less stable clones change more frequently and thus require comparatively more maintenance efforts than more stable clones. Stability of clones is one of the most widely investigated aspects of the impacts of clones on software systems [84, 98, 100, 122]. Although the existing studies have significant findings with respect to different aspects of the bug-proneness of clones, none of the existing studies investigated whether or to what extent the stability of clones is related to the bug-proneness. Again, there are instances of strong empirical evidence that inconsistent changes to clones and missing change propagation or late propagation are strongly related to the bug-proneness of clones [71, 18, 22, 171]. As it is obvious that different types of syntactic changes have different impacts on change propagation and missing change propagation to clones are related to bugs and inconsistencies, it is important to investigate whether the types of syntactic changes code clones evolve through contribute to their bug-proneness behavior.

Considering the differences in the impacts of different change types, this study carries out an empirical evaluation of the relationship between the stability and the bug-proneness of clones. We consider a comprehensive taxonomy of source code change proposed by Fluri and Gall [43]. This taxonomy assigns a level of significance to each type of source code change. We extract fine-grained changes from all successive revisions of software systems using *ChangeDistiller* [43] which employs Abstract Syntax Tree (AST) differencing between two consecutive revisions of source code files. Extracted and classified changes from *ChangeDistiller* are mapped to the source code entities and stored for further analysis. We detect both exact (Type 1) and near-miss (Type 2 and Type 3) clones of all the revisions of the systems using the hybrid clone detection tool NiCad [144]. The classified changes are then mapped to the cloned and non-cloned code. We then measure the comparative stability of buggy and non-buggy clones to evaluate the relationship between the stability and the bug-proneness of different types of clones.

In particular, we represent the findings of our empirical study by answering the following research questions:

RQ1 *Is there any relationship between the stability and the bug-proneness of code clones? If yes, is the relationship significant?*

RQ2 *To what extent the association between the bug-proneness and stability of different types of clones are different?*

RQ3 *Do the frequencies of changes of different significance levels differently affect the bug-proneness of clones?*

In this study of analyzing the relationships between the stability and the bug-proneness of clones using fine-grained change information from software repositories, we make the following important contributions:

(1) We investigate the relationships between the stability of clones and their bug-proneness. Although the stability of clones is one of the most widely studied aspects of the impacts of clones on software systems, none of the existing studies investigated whether or to what extent the stability of clones is related to the bug-proneness. Clone-induced inconsistencies and bugs are of important concern in software development and maintenance. Our investigation aims at revealing the relationships between stability and bug-proneness of clones. This is important whether stability is an influencing factor for clones to be bug-prone and vice-versa.

(2) Different types of clones may have a different degree of association with their corresponding stability. We carry out a clone type-centric analysis of the relationship between the stability and the bug-proneness of clones.

(3) Although different types of syntactic changes have different impacts on the software systems, none of the existing studies consider fine-grained change types while investigating the bug-proneness of clones. We

consider fine-grained syntactic change types and their levels of change significance for in-depth analysis of the relationships between the stability and bug-proneness of clones.

We analyze the bug-proneness of clones from a new perspective. The study aims to relate the stability of clones as a potential influencing factor to the bug-proneness of clones. The findings of the study have the potentials to help in identifying and prioritizing clones for clone management activities such as clone refactoring and tracking.

6.2 Terminology

Change Significance: Fluri and Gall [43] in their taxonomy of fine-grained source code change, assign a level of *significance* to each of the change types. The *significance* level defines the extent of the impacts of a change *i.e.,* the extent of changing system functionality and the likelihood of a change to affect other related entities for the required changed propagation. The higher the significance level, the higher impacts of the change will be on the related source code elements. Fluri and Gall define four different levels *(low, medium, high, crucial)* of significance for fine-grained syntactic change types. Here, *crucial* is the highest level of change significance and *low* is the lowest level of change significance.

6.3 Taxonomy of Software Change at Method Level

Software systems evolve through different kinds of changes. For our study, we consider the taxonomy of change proposed by Fluri and Gall [43] which is a comprehensive taxonomy for fine-grained source code change and it defines the significance levels of changes. As our clone analysis is at method level granularity, we consider only the method level changes in the taxonomy proposed by Fluri and Gall. The taxonomy of changes used in this empirical study is presented in Table 6.1.

Changes to individual methods are referred to as method-level changes. Changes to methods are further divided into two groups, changes to method declaration and changes to the method body, based on where the changes occur in the methods. Changes to method declaration part (*method signature*) include changes in accessibility, overridability, method renaming, parameter change and changes to return type. Changes to parameters include addition, deletion, renaming, reordering and parameter type change. Changes to *method body* comprise changes to statements and structure statements (*e.g.,* loop, branching). Statements might be added, deleted, modified and reordered. Each of these fine-grained change types is assigned a level of significance based on their likelihood of affecting other code entities and the extent they modify the functionality of the system as proposed by Fluri and Gall [43].

Table 6.1: TAXONOMY OF METHOD-LEVEL CHANGES WITH SIGNIFICANCE (adapted from [43])

Part	Change Group	Change Type	Significance
Method Declaration	Accessibility	Accessibility Increase (MAI)	Medium
		Accessibility Decrease (MAD)	High
	Overridability	Add Method Overridability *final* (AMO)	Crucial
		Delete Method Overridability (DMO)	Low
	Parameter	Parameter Insert (PI)	Crucial
		Parameter Delete (PD)	Crucial
		Parameter Ordering (PO)	Crucial
		Parameter Renaming (PR)	Medium
		Parameter Type Change (PTC)	Crucial
	Method Name	Method Renaming (MR)	High
	Return Type	Return Type Insert (RTI)	Crucial
		Return Type Delete (RTD)	Crucial
		Return Data Type Change (RDC)	Crucial
Method Body	Statement	Statement Insert (SI)	Medium
		Statement Delete (SD)	Medium
		Statement Update (SU)	Low
		Statement Re-ordering (SO)	Low
	Structure Statement	Condition Expression Change (CEC)	Medium
		Statement Parent Change (SPC)	Medium
		Alternative- part (*else*) Insert (API)	Medium
		Alternative- part (*else*) Delete (APD)	Medium

Table 6.2: SUBJECT SYSTEMS

Systems	Type	Size (LOC)	#Revision
DNSJava	DNS Protocol	20831	1679
JabRef	Bibliography Manager	153952	3718
Carol	Driver Application	13213	2237
Ant-Contrib	Web Server	79434	177
OpenYMSG	Open Messenger	8821	233

6.4 Experimental Setup

This section describes the experimental setup of our empirical study including the subject systems used, methodologies for identifying buggy commits, change extraction and classification, clone detection and the measurement of comparative stabilities of the buggy and non-buggy clones. The following subsections describe the details of the experimental settings of our study.

6.4.1 Subject Systems

This study is based on five open source systems implemented in Java with diversified size and application domain. Table 6.2 briefly represents the features of the software systems including the application domain, size in lines of code (LOC) and the total number of revisions. The size of the systems represents the lines of code (LOC) in the last revision of the systems counted after removal of comments and pretty-printing.

6.4.2 Detecting Bug-fix Commits

To identify bug-fix commits of a candidate system, we extract the SVN commit messages by applying *SVN log* command. A commit message describes the objective of the associated commit and thus can be used to infer whether the commit is a bug-fix commit. We apply the heuristics proposed by Mockus and Votta [108] on the commit messages to automatically identify bug-fix commits. This approach for identifying bug-fix commits have been used in different earlier studies [22, 116, 120, 21]. This technique identifies bug-fix commits based on the occurrence of keywords in the commit message from a predefined set. For example, if a commit message contains the word "bug", it will be classified as a bug-fix commit. This heuristic-based approach may sometimes results in false positives due to incorrect classification of commits as bug-fix commits. However, earlier study [22] shows that this approach can identify bug-fix commits with 87% accuracy. Once the bug-fix commits are listed, we identify the bug-fix commits where the cloned fragments have changed. When any clone fragment is changed in a bug-fix commit, it is reasonable to infer that changes to that clone are necessary

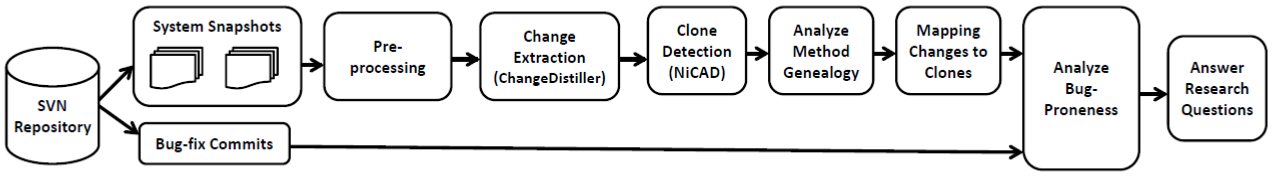


Figure 6.1: Overall Analysis Process

to fix the bug. We analyze and identify all the cloned methods that are related to bug-fixes. We then analyze the stability considering fine-grained change types associated with each of the bug-related clone fragments to measure the extent of the relationship between stability and bug-proneness.

6.4.3 Experimental Steps

We carry out our experimental analysis through some processing steps. Figure 6.1 shows the schematic diagram of the overall experimental analysis process. We briefly describe the experimental steps as follows:

Preprocessing

For our study, we first extract all the revisions of the subject systems from their corresponding *SVN* repositories [7]. As our analysis focuses on the changes to source code only, we remove comments from the source files. Pretty-printing of the source files are then carried out to eliminate the formatting differences using the tool *ArtisticStyle* [6]. We extract the file modification history using *SVN diff* command to list added, modified and deleted files in successive revisions. This information is used to exclude the unchanged files during change analysis to speed up the process.

Method Extraction and Origin Analysis

To analyze the changes to cloned methods throughout all the revisions, we extract method information from the successive revisions of the source code. We store the method information (file path, package name, class name, signature, start line, end line) in a database to use for mapping changes to the corresponding methods. Again, SVN keeps track of the files that are added, deleted or modified and the history of changes to individual file content are preserved as the modification of lines. This line-level change information is not sufficient to describe the evolution of source code entities at higher granularity levels such as classes or methods. As a result, to map changes to methods throughout the development cycle, we need to map the methods across the revisions. Therefore, we carry out origin analysis of the methods on the revisions of the systems. We used the same approach for origin analysis as presented in the study by Lazano and Wermelinger [97]. Here, if a method is relocated in the same file or if the method signature is changed, the method is mapped by string comparison with the candidate methods in the new file using *Strike A Match* algorithm [4]. The origin mapping information is used to map the classified changes and cloning information back to the corresponding methods.

Change Extraction and Classification

The change analysis system in this study is implemented in Java based on the change extraction and classification core of *ChangeDistiller* [43]. The system imports copies of changed files from successive versions and uses JDT API of Eclipse for the extraction of methods and extracting the differences between the copies of each of the files in any two successive revisions. The details of change extraction and classification are as follows:

- *Change Extraction:* Code changes are extracted using *ChangeDistiller* classifier. *ChangeDistiller* extracts changes by taking differences between two versions of ASTs of the same file. The differences are represented as a sequence of tree-edit operations. The generic operations contain insert, delete, move and update operations on the nodes in the AST. The tree-edit operations encoded as edit scripts are then processed by *ChangeDistiller* to classify extracted changes to fine-grained change types. We have customized the *ChangeDistiller* classifier to suit for analyzing local repository exported from SVN. To extract source code changes, two successive versions of the same file are selected from the source repository and then are passed to the differencing engine of the change classifier. The extracted changes are then passed to the classifier for classification. The process is repeated for all changed files (identified by SVN *diff*) and for all the revisions of the subject systems.

- *Change Classification:* Changes extracted by AST-differencing of two successive revisions of source code files are classified into fined-grained changes to source code entities. Changes are classified according to the defined taxonomy and are assigned the corresponding levels of significance. For the analysis, classified changes are mapped to the corresponding source code entities based on the information extracted during the origin analysis.

Mapping Change Data

After classification, the classified changes are mapped to their corresponding source code entities (methods) with the help of extracted origin mapping information. We preserve the extracted, classified and mapped changes into a database to measure metrics for the bug-proneness of clones at the method level granularity.

Clone Detection

There are a great many clone detection tools and techniques [3, 50, 73, 64, 157, 170, 176, 177, 167, 168] available including a recent one using deep learning [182]. However, for this study, we use the hybrid clone detection tool NiCad [144, 36]. NiCad is reported to have a higher level of precision and recall [145] and supports the detection of both exact (Type 1) and near-miss (Type 2 and Type 3) clones. We run NiCad on all revisions of the subject systems to detect clones at method level granularity. We then store the clone information in the database. Table 6.3 lists the parameter settings for NiCad used for this study.

Table 6.3: NiCad SETTINGS FOR THE STUDY

Parameters	Values
Minimum Size	5 lines
Maximum Size	500 lines
Granularity	Method
Threshold	0% (Type 1, Type 2), 30% (Type 3)
Identifier Renaming	blindrename (Type 2, Type 3)

Mapping Changes to Clones

We use the extracted method genealogies to map method information to the detected clones in each revision of the software systems. We also map classified fine-grained changes to the clones in each revision. Using the list of identified bug-fix commits we separate the list of the buggy and non-buggy clones. Methods that were changed in any of the bug-fix commits are considered as buggy clones while other cloned methods are considered as non-buggy clones.

6.4.4 Measuring the Change Frequencies

Once the mapped change information for buggy and non-buggy clones are available, we measure the stability (frequency of changes) for buggy and non-buggy clones. We measure stability with respect to different clone types considering the different levels of significance of fine-grained syntactic changes. We measure the frequency of changes as follows:

Let R_b be the set of bug-fix commits and $S = \{low, medium, high, crucial\}$ be the set of different levels of change significance. Let M_b and M_n are the sets of buggy and non-buggy cloned methods respectively. Then, we measure the stability (frequency of changes) of the buggy and non-buggy clones using the following two equations:

(i) The frequency of changes to buggy cloned methods (CF_b) is measured as

$$CF_b = \frac{\sum_{m_i \epsilon M_b, s \epsilon S} CC_s(m_i)}{\sum_{m_i \epsilon M_b} AVGLOC(m_i)} \tag{6.1}$$

(ii) The frequency of changes to non-buggy cloned methods (CF_n) is measured as

$$CF_n = \frac{\sum_{m_i \epsilon M_n, s \epsilon S} CC_s(m_i)}{\sum_{m_i \epsilon M_n} AVGLOC(m_i)} \tag{6.2}$$

Table 6.4: COMPARATIVE FREQUENCIES OF CHANGES OF DIFFERENT LEVELS OF SIGNIFICANCE FOR BUGGY AND NON-BUGGY CLONES OF ALL TYPES.

Change Significance→	Low		Medium		High		Crucial		Overall	
Systems ↓	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n
DNSJava	1.6546	0.5145	1.4196	0.5177	0.1471	0.1150	0.0723	0.0254	3.2936	1.1726
JabRef	0.7213	0.3496	0.4198	0.1768	0.0033	0.0031	0.0267	0.0163	1.1710	0.5458
Carol	0.7505	0.2740	0.6515	0.2542	0.0178	0.0153	0.0323	0.0248	1.4521	0.5684
Ant-Contrib	0.4325	0.2010	0.3249	0.1484	0.0172	0.0031	0.0038	0.0371	0.7785	0.3896
OpenYmsg	0.4211	0.2280	0.4044	0.0909	0.0069	0.0000	0.0014	0.0014	0.8338	0.3803
***p*-value**	**0.0366**		0.0601		0.2113		0.7565		0.0601	
Cohen's *d*	1.3129		1.1964		0.2016		0.2828		1.1651	
PS	0.82		0.80		0.55		0.58		0.80	

CF_b=Frequency of Changes in Buggy Cloned Methods

CF_n=Frequency of Changes in Non-Buggy Cloned Methods, PS=Probability of Superiority

In Equation 6.1 and Equation 6.2,

- $CC_s(m_i)$ represents the total number of changes to a cloned method m_i with change significance level s where $s \epsilon S$.

- $AVGLOC(m_i) = \frac{\sum_{r \epsilon R_{m_i}} LOC_r(m_i)}{|R_{m_i}|}$ represents the average size (LOC) of a cloned method. Here, $LOC_r(m_i)$ refers to the size (LOC) of the cloned method in revision $r \epsilon R_{m_i}$ where R_{m_i} is the set of revisions a method m_i was cloned.

- A cloned method is considered buggy (*i.e.,* $m_i \epsilon M_b$) if $R_{m_i} \cap R_b \neq \phi$, *i.e.,* the method m_i has changed in at least one of the bug-fix commits. Otherwise, the method is a non-buggy (*i.e.,* $m_i \epsilon M_n$) cloned method.

Here, the change frequencies of the buggy and non-buggy clones simply represent the average number of changes per line of code in buggy and non-buggy cloned methods respectively. The lower the frequency of changes of a cloned fragment, the higher will be the stability of that clone fragment.

6.5 Results

We present our experimental results in the following subsections to answer the research questions we defined in Section 6.1.

Answer to RQ1*: Is there any relationship between the stability and the bug-proneness of code clones? If yes, is the relationship significant?*

Importance: Stability of clones is one of the most widely studied aspects of analyzing the impacts of clones on software systems. Studies [97, 100, 122, 121] show that clones are often comparatively less stable meaning that clones change more frequently than non-cloned code. It is intuitive that the more frequently a code fragment is changed, the more maintenance efforts it will require. Again, clone fragments changing more frequently are likely to increase the chance of missing change propagations and thus may introduce inconsistencies or bugs in the software systems. Consequently, it is important to investigate whether and to what extent the stability of clones are related to their bug-proneness.

Methodology: To analyze the relationships between the stability and the bug-proneness of clones, we measure the frequencies of changes for buggy and non-buggy clones as described in Section 6.4.4 for each of the subject systems. First, we determine the list of bug-fix commits by analyzing the commit messages from SVN repositories (Section 6.4.2). Then we identify the changed cloned methods from a selected clone type. Then we count the number of changes of different significance levels for each of the methods selected. We also count the average size (LOC) of each of the cloned methods in the list. We measure the total number of changes and the total code size (LOC) for both buggy and non-buggy cloned methods. Then we measure the corresponding frequencies of changes as the ratio of total number changes to total code size in LOC for both buggy and non-buggy clones for comparison. We measure the frequency of changes regarding each level of significance of change. We consider all clones without differentiating their types for answering RQ1.

Table 6.4 shows the frequencies of changes for buggy and non-buggy clones for different levels of change significance. Here, we observe that change frequencies for buggy clones for all levels of change significance and for all the systems are higher (*i.e.,* $CF_b > CF_n$) than the corresponding frequencies of changes of non-buggy clones (except for the *crucial* significance in OpenYmsg). This shows that buggy clones tend to be less stable than non-buggy clones. To analyze whether there is any statistically significant relationship between the stability and bug-proneness of clones, we carry out Mann-Whitney Wilcoxon (MWW) test (two-tailed, at < 0.05) [5]. From Table 6.4, we see that the p-value for *low* level of change significance is 0.0366 which is less than 0.05. This implies that the frequency of changes for buggy clones is significantly higher than the frequency of changes in non-buggy clones regarding changes of *low* significance. However, for other levels of change significance (*medium, high and crucial*), although the values of frequencies of changes for buggy clones are higher than that of non-buggy clones, the p-values are greater than 0.05. This indicates that change frequencies for buggy and non-buggy clones are not significantly different for those cases.

Again, the Mann-Whitney Wilcoxon test examines whether the findings are likely due to chance and may not alone fully express the magnitude of differences found. Thus, we also calculate the *effect size* to analyze the magnitude of differences in stability of buggy and non-buggy clones. Effect size calculates the standardized mean difference between two data sets. We measure the effect size (Cohen's d [35]) from the comparative frequencies of changes to buggy and non-buggy clones as shown in Table 6.4. We see that the values of the effect size for both *low* and *medium* significance levels are 1.3129 and 1.1964 respectively which belong to 'large' category as they are above 0.8. The effect size for *high* and *crucial* significance levels are 'small' with values 0.2016 and 0.2828 respectively. For easier interpretation of the effect size, we also convert the effect size values to the probability of superiority or 'Common Language Effect Size' [106] as shown in Table 6.4. The probability of superiority value for *low* significance level is 0.82 meaning that if selected randomly, there is 82% chance that buggy clones will have higher change frequency than non-buggy clones. This probability values are lower for changes of *high* (0.55) and *crucial* (0.58) significance and closer to 0.5 (0.5 refers to the equal likelihood of having higher change frequencies for buggy and non-buggy clones). We also analyzed the comparative stability of buggy and non-buggy clones by aggregating changes of all levels of significance (shown in the 'overall' column). Here, we observe that $CF_b > CF_n$ for all the subject systems and the effect size is large (1.1651). The probability of superiority is 0.8 *i.e.,* in 80% cases buggy clones likely to have a higher frequency of changes than non-buggy clones. Thus, buggy clones tend to be less stable than the non-buggy clones.

Summary- *According to our experimental results, buggy clones tend to have a higher frequency of changes compared to non-buggy clones* i.e., *buggy clones are less stable than non-buggy clones. Thus, the stability and the bug-proneness of clones are related and this relationship is statistically significant for changes of low significance level.*

Answer to RQ2: *To what extent the bug-proneness of different types of clones are associated with their stability?*

Importance: Different types of clones exhibit different degree of bug-proneness [116]. Similarly, the stability of different types of clones may vary with the changes of different levels of significance. Thus, an in-depth analysis of the relationships between the stability of clones and their bug-proneness should also investigate the degree of the relationships regarding different clones types. This clone-type centric analysis is likely to reveal how the relationship between stability and bug-proneness of clones varies for different types of clones. This analysis is important to identify what types of clones to be comparatively more bug-prone and need to be prioritized in clone management activities.

Table 6.5: COMPARATIVE FREQUENCIES OF CHANGES OF DIFFERENT LEVELS OF SIGNIFICANCE FOR BUGGY AND NON-BUGGY CLONES OF TYPE 1.

Change Significance→	Low		Medium		High		Crucial		Overall	
Systems ↓	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n
DNSJava	1.6438	0.4373	1.4155	0.4373	0.0918	0.0435	0.0787	0.0239	3.2297	0.9422
JabRef	0.6710	0.3409	0.3861	0.1571	0.0021	0.0026	0.0190	0.0082	1.0783	0.5088
Carol	0.3235	0.2102	0.5202	0.1564	0.0087	0.0587	0.0393	0.0391	0.8918	0.4643
Ant-Contrib	0.4299	0.2571	0.3277	0.1959	0.0162	0.0041	0.0011	0.0490	0.7749	0.5061
OpenYmsg	0.4422	0.2262	0.2802	0.0524	0.0103	0.0000	0.0000	0.0024	0.7326	0.2810
***p*-value**	0.0601		0.0601		0.6745		0.8337		0.0601	
Cohen's *d*	1.0492		1.1062		0.1226		0.1147		1.0379	
PS	0.76		0.78		0.52		0.52		0.76	

CF_b=Frequency of Changes in Buggy Cloned Methods

CF_n=Frequency of Changes in Non-Buggy Cloned Methods, PS=Probability of Superiority

Table 6.6: COMPARATIVE FREQUENCIES OF CHANGES OF DIFFERENT LEVELS OF SIGNIFICANCE FOR BUGGY AND NON-BUGGY CLONES OF TYPE 2.

Change Significance→	Low		Medium		High		Crucial		Overall	
Systems ↓	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n
DNSJava	1.0859	0.3979	0.9594	0.3650	0.1186	0.1885	0.0422	0.0299	2.2061	0.9814
JabRef	0.8192	0.5251	0.4398	0.2590	0.0000	0.0035	0.0751	0.0987	1.3341	0.8863
Carol	1.3994	0.2267	1.0694	0.2713	0.0293	0.0167	0.0146	0.0056	2.5127	0.5204
Ant-Contrib	0.4200	0.0455	0.3800	0.1667	0.0000	0.0000	0.0000	0.0606	0.8000	0.2727
OpenYmsg	0.4211	0.2880	0.4044	0.0909	0.0069	0.0000	0.0014	0.0014	0.8338	0.3803
***p*-value**	**0.0366**		**0.0121**		1.000		0.6031		0.0949	
Cohen's *d*	1.6283		1.6916		-0.1567		-0.3414		1.5519	
PS	0.87		0.88		0.55		0.58		0.87	

CF_b=Frequency of Changes in Buggy Cloned Methods

CF_n=Frequency of Changes in Non-Buggy Cloned Methods, PS=Probability of Superiority

Table 6.7: COMPARATIVE FREQUENCIES OF CHANGES OF DIFFERENT LEVELS OF SIGNIFICANCE FOR BUGGY AND NON-BUGGY CLONES OF TYPE 3.

Change Significance→	Low		Medium		High		Crucial		Overall	
Systems ↓	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n	CF_b	CF_n
DNSJava	1.2390	0.7305	0.9372	0.8774	0.3268	0.3449	0.0590	0.0292	2.5620	1.9820
JabRef	0.9494	0.3227	0.5978	0.1779	0.0039	0.0027	0.0357	0.0277	1.5869	0.5310
Carol	0.7698	0.2874	0.6624	0.2558	0.0186	0.0135	0.0312	0.0233	1.4820	0.5800
Ant-Contrib	0.3735	0.1159	0.3550	0.0454	0.0078	0.0000	0.0029	0.0202	0.7391	0.1814
OpenYmsg	0.3918	0.2783	0.5623	0.0569	0.0000	0.0000	0.0000	0.0000	0.9541	0.3353
***p*-value**	**0.0366**		0.0949		0.6744		0.4654		0.0949	
Cohen's *d*	1.2916		1.1945		-0.0054		0.2903		1.0386	
PS	0.82		0.80		0.50		0.55		0.76	

CF_b=Frequency of Changes in Buggy Cloned Methods

CF_n=Frequency of Changes in Non-Buggy Cloned Methods, PS=Probability of Superiority

Methodology: To answer RQ2, we analyze the comparative frequencies of changes of different levels of significance for buggy and non-buggy clones considering clone types separately. We measure the frequency of changes for buggy and non-buggy clones in the same way as in RQ1. However, we measure the stability metrics for distinct types of clones (Type 1, Type 2 and Type 3) separately. The results for the comparative frequencies of changes of buggy and and non-buggy clone for Type 1, Type 2 and Type 3 clone are presented in Table 6.5, Table 6.6, Table 6.7 respectively.

For Type 1 clones (Table 6.5), the frequency of changes for buggy clones are higher than that of non-buggy clones for all subject systems when we consider changes of *low* and *medium* significance (*i.e.,* $CF_b > CF_n$). However, for changes of *high* significance in JabRef and Carol and for changes of *crucial* significance in Ant-Contrib and OpenYmsg we observe $CF_b < CF_n$. We observe that for changes of *low* and *medium* significance, the likelihood of buggy Type 1 clones to have higher frequencies than non-buggy Type 1 clones are 76% and 78% respectively. However, we do not observe any statistically significant relationship between the stability and the bug-proneness for Type 1 clones.

For Type 2 clones (Table 6.6), we observe that the frequency of changes for buggy clones are higher than that of non-buggy clones for all the subject systems regarding changes of *low* and *medium* significance. For comparative frequencies of the buggy and non-buggy Type 2 clones with changes of *low* and *medium* significance, we get the p-values 0.0366 and 0.0121 respectively from Mann-Whitney Wilcoxon (MWW) test (two-tailed, at <0.05). These p-values imply that the frequencies of changes in buggy Type 2 clones are significantly higher than the frequency of changes in non-buggy Type 2 clones regarding changes of *low* and *medium* significance. In addition, for changes of *low* and *medium* significance, the values of effect size are 'large' (1.6283 and 1.6916 respectively) with the corresponding likelihood of 87% and 88% that buggy Type 2 clones have a higher frequency of changes than non-buggy Type 2 clones. However, we did not observe any statistically significant differences between CF_b and CF_n for changes of *high* and *crucial* significance.

As shown in Table 6.7, the frequencies of changes for Type 3 buggy clones are higher than that of non-buggy Type 3 clones for all the subject systems regarding changes of *low* and *medium* significance. For Type 3 clones with changes of *low* significance, the p-value for Mann-Whitney Wilcoxon (MWW) test (two-tailed, at <0.05) is 0.0366. This implies that for changes of *low* significance, the frequency of changes in buggy Type 3 clones are significantly higher than that of non-buggy Type 3 clones. The effect size d for comparative frequencies for changes for buggy and non-buggy Type 3 clones is also 'large' (1.2916) for changes of *low* significance. For *low* significance, buggy Type 3 clones have 82% likelihood of having a higher frequency of changes than non-buggy Type 3 clones.

Figure 6.2 represents the comparative likelihood (%) of buggy clones to have higher frequencies for changes than non-buggy clones with respect to different clones types. We use the probability of superiority from the effect size analysis for comparative change frequencies of the buggy and non-buggy clones. We consider probabilities of superiority for all four levels of change significance with respect to each type of clones. From Figure 6.2 we see that for Type 1 clones, the probability of superiority values regarding changes of *low* and *medium* significance are 76% and 78% respectively. On the other hand Type 2 clones have the highest likelihood (87% and 88% respectively) of buggy clones to have a higher frequency of changes than non-buggy clones regarding changes of *low* and *medium* significance. Type 3 clones also have 82% and 88% likelihood of buggy clones to have a higher frequency of changes than non-buggy clones. However, for all clone types while considering the changes of *high* and *crucial* significance, the likelihood of buggy and non-buggy clones are closer to 50%. This indicates that the probability of buggy and non-buggy clones to have higher frequency of changes are similar.

From the above clone type based analysis, we observe that although the frequency of changes tend to be higher in buggy clones than in the non-buggy clones for both exact (Type 1) and near-miss(Type 2 and Type 3) clones, the bug-proneness of Type 2 and Type 3 clones have stronger relationships with their stability. The significance of the relationship between the bug-proneness of Type 2 and Type 3 clones are mostly influenced by changes of *low* and *medium* significance.

Summary- *Our results on clone-type centric analysis show that the bug-proneness of Type 2 and Type 3 clones are significantly related to the stability of clones and this relationship is mostly influenced by the changes of low and medium significance. The bug-proneness of Type 1 clones does not exhibit a significant relationship with the stability.*

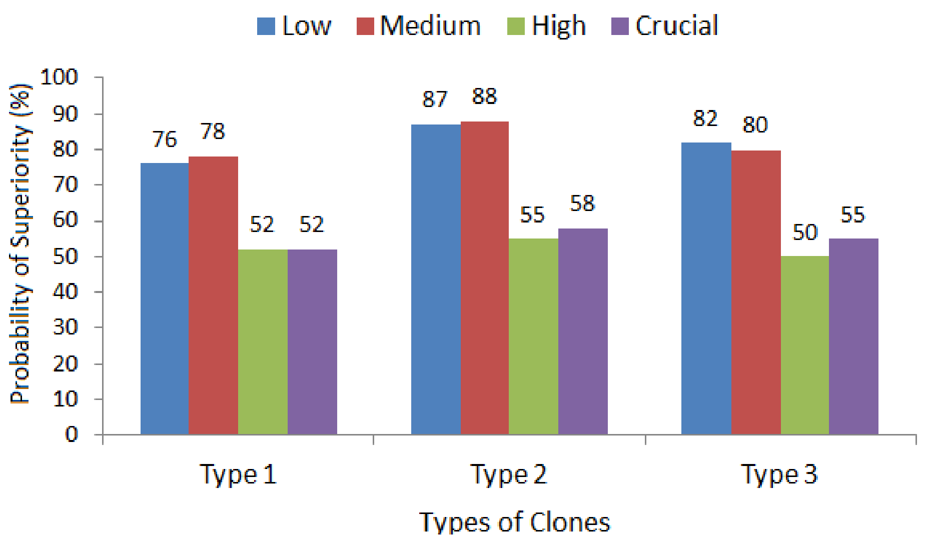


Figure 6.2: Comparison of change frequency of buggy and non-buggy clones based on different clone types.

Answer to RQ3: *Do the frequencies of changes of different significance levels differently affect the bug-proneness of clones?*

Importance: Changes of different levels of significance have different impacts on software systems both in terms of the modification of the system functionality and the extents of required change propagation. Thus, for better understanding the relationships between bug-proneness and stability of clones, we need to investigate how changes of different levels of significance influence this relationship. Therefore, we analyze the comparative stability of buggy and non-buggy clones with respect to each of the four levels of change significance.

Methodology: To answer RQ3, we consider the comparative frequencies of changes for the buggy and non-buggy clones from Table 6.5, Table 6.6 and Table 6.7 regarding each type of change significance. For changes of *low* significance and for all types (Type 1 , Type 2 and Type 3) of clones, buggy clones have higher frequencies of changes than non-buggy clones. However, for changes of *low* significance, the comparative frequencies of changes for buggy and non-buggy clones are statistically significant for only for Type 2 and Type 3 clones. For changes of *medium* significance, all types of clones tend to have higher frequencies of changes for buggy clones compared to non-buggy clones. However, only Type 2 clones show a statistically significant difference in the frequencies of changes in buggy and non-buggy clones. For changes of *high* and *crucial* significance, none of the clone types exhibit any significant differences between the frequency of changes in buggy and non-buggy clones. Thus, changes of *low* and *medium* significance likely to influence the relationship between the stability and the bug-proneness of clones.

Figure 6.3 represents the comparative likelihood (%) of buggy clones to have higher frequencies for changes than non-buggy clones with respect to different levels of changes significance. We use the probability of superiority from the effect size analysis for comparative change frequencies of the buggy and non-buggy clones.

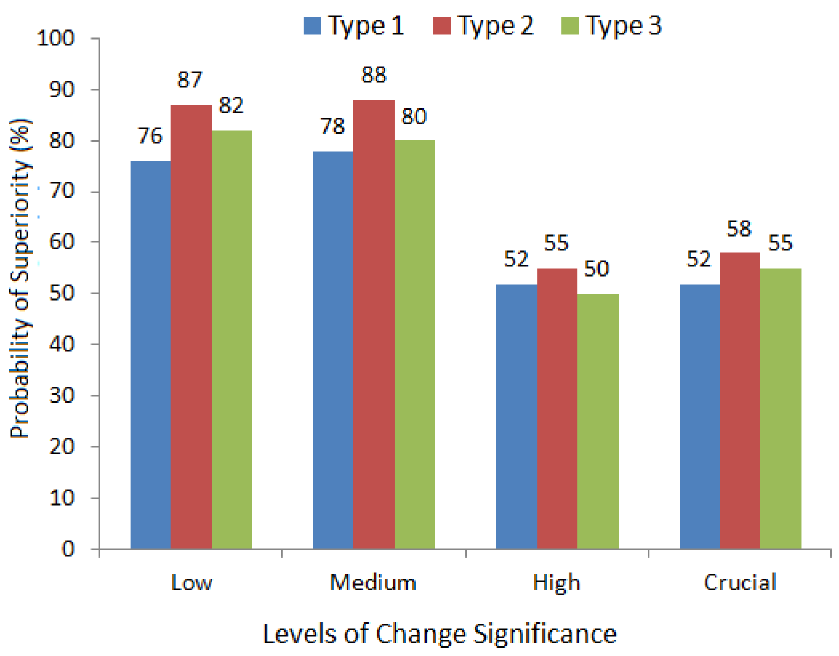


Figure 6.3: Likelihood (%) buggy clones having higher frequencies of changes than non-buggy clones regarding different levels of change significance.

From Figure 6.3 we see that for changes of *low* significance, the probability of superiority values regarding Type 1, Type 2 and Type 3 clones are 76%, 87% and 82% respectively. On the other hand, for changes of *medium* significance, the probability of superiority values regarding Type 1, Type 2 and Type 3 clones are 78%, 88%, and 80% respectively. These imply that the frequencies of changes to buggy clones are comparatively higher than that of non-buggy clones regarding changes of *low* and *medium* significance for all clone types. However, Type 2 clones have the highest probability values for buggy clones to have a higher frequency of changes than non-buggy clones for changes of *low* and *medium* significance. For changes of *high* and *crucial* significance, none of the clone types exhibit higher likelihood of buggy clones to have a higher frequencies of changes than non-buggy clones. Here, the negative value for d indicates the direction of mean difference.

Type 2 clones have the highest likelihood (87% and 88% respectively) of buggy clones to have a higher frequency of changes than non-buggy clones regarding changes of *low* and *medium* significance. Type 3 clones also have 82% and 88% likelihood of buggy clones to have a higher frequency of changes than non-buggy clones. However, for all clone types while considering the changes of *high* and *crucial* significance, the likelihood of buggy and non-buggy are closer to 50% which indicates that the probability of buggy and non-buggy clones to have a higher frequency of changes are similar.

Summary- *The bug-proneness of clones tend to exhibit stronger relationship with the stability regarding the changes of low and medium significance. Thus, the relationship between the stability and the bug-proneness of clones is most likely to be influenced by changes of low and medium significance.*

6.6 Threats to Validity

The change analysis in this study is based on the change extraction and classification engine of the *ChangeDistiller*. Although *ChangeDistiller* is reported to have good performance, the validity of the outcomes of the study is dependent upon the accuracy of the core classifier used.

The detection of bug-fix commits in our study is based on the technique proposed by Mockus and Votta [108]. Same technique has also been used in several earlier studies [22, 116, 120, 21, 60]. Although this heuristics based technique may result in some incorrect identification of bug-fix commits, an earlier study by Barbour *et al.* [22] shows that the probability is low. Their study reports that the accuracy of the technique to identify bug-fix commits is 87%.

We used NiCad, a hybrid clone detection tool which detects Type 1, Type 2 and Type 3 clones with high precision and recall. Again, different settings for clone detection tool may result in different stability scenarios because of the variations in clone detection results. This is termed as *confounding configuration choice problem* [180]. However, the NiCad settings we used are considered standard [151] and are close enough to the optimal configuration settings identified in recent study [180] for NiCad to detect clones in Java systems. This is likely to mitigate the potential adverse effects of the configuration settings on our findings.

We selected subject systems with diversified size, number of revisions, length of evolution and application domain to avoid potential biasing. However, due to the limitation of existing change classifier our study is limited to Java systems only. Although Java is considered to be a widely used language with a comprehensive set of language features, the inclusion of subject systems of other languages might help in more generalization of the findings.

6.7 Related Work

Many studies have investigated the bug-proneness of cloned code from different perspectives. Wagner *et al.* [178] investigated the relationship between Type 3 clones and faults. This study shows that a considerable proportion (17%) of Type 3 clones are associated with faults. Mondal *et al.* [116] investigated the bug-proneness of different types of clones and reported that Type 3 clones are more bug-prone compared to Type 1 and Type 2 clones. Although this study gives important insights regarding the bug-proneness of clones of different types, the study does not consider fine-grained change information for the analysis as we did.

Mondal *et al.* in a recent study [120] observed that change-prone clones are not necessarily to be bug-prone. They measured change frequency as the number of commits a particular clone fragment was changed before a given bug-fix commit. Our study, on the other hand, considers fine-grained change types and their significance. Our change frequency metrics consider the actual number of fine-grained changes per line of cloned code. Thus, our analysis of relationships between stability and bug-proneness is different from the study by Mondal *et al.* [120].

Li *et al.* [95] proposed the tool CP-Miner that uses data mining techniques to detect copy-baste clones and bugs in large-scale software systems. Li and Ernst [94] studied bug-proneness of clones and developed a tool called CBCD based on their study. For a given buggy code fragment, CBCD searches semantically identical copies of the given code fragment. Inoue *et al.* [58] developed a tool called CloneInspector to detect inconsistent changes to code clones and associated latent bugs in software systems.

Barbour *et al.* [22] examined the late propagation of clones (Type 1 and Type 2) and investigated the extent to which different types of late propagation are related to bugs and inconsistencies in software systems. Aversano *et al.* [18] reported that clones evolve consistently in most cases while late propagation may introduce faults. Thummalapenta *et al.* [171] show that clones with late propagation are more bug-prone than others.

Xie *et al.* [185] studied the fault-proneness of clones with respect to mutation and migration of clones during evolution. They concluded that both mutation of a clone group to Type 2 or Type 3 clones and increasing distance between clone fragments in a clone group increase the risk for faults. Steidl and Göde [163] proposed a machine learning approach to investigate how different features of clones can be used to automatically identify incomplete bug-fixes. Göde and Koschke [51] reported that changes to clones are mostly infrequent and a small proportion (14.8%) of changes are unintentionally inconsistent.

Sajnani *et al.* [156] analyzed the comparative bug-proneness of cloned and on-cloned code on 31 Java projects. They report that the density of bugs is comparatively less in clones than non-cloned code. Rahman *et al.* [133] observed that clones may be less defect-prone than non-cloned code which is opposite to the findings of the study by Juergens *et al.* [69].

The existing studies reveal important insights regarding the techniques for the detection [95, 65] of bug-prone clones, identifying features and types of bugs [65] and analyzing bug-proneness of different types of clones [116] from the perspective of consistent evolution. However, none of the existing studies consider fine-grained syntactic changes despite the types of changes mostly define the extent of change propagation to related cloned and non-cloned fragments. In this study, we extract fine-grained change information and analyze the relationship between the stability and the bug-proneness of clones. We evaluate stability as a potential influencing factor to the bug-proneness of clones and analyze the bug-proneness of Type 1, Type 2 and Type 3 clones.

6.8 Summary

This study investigates the relationships between the stability and the bug-proneness of clones considering fine-grained syntactic change types and their significance extracted using ChangeDistiller. We automatically identify the bug-fix commits by analyzing the commit messages from the SVN repositories. We detect Type 1, Type 2 and Type 3 clones using the hybrid cloned detection tool NiCad. Clones that change in any of the bug-fix commits are identified as buggy clones. Our experimental results on five diverse open source Java systems show that buggy clones tend to have a higher frequency of changes compared to non-buggy clones. This relationship is often significant especially when changes of *low* and *medium* significance are considered. Again, the stabilities of Type 2 and Type 3 clones have comparatively stronger associations with the bug-proneness than Type 1 clones. The relationships between the stability and bug-proneness of different types of clones are more likely influenced by the changes of *low* and *medium* significance. Our findings regarding the relationships between the stability and bug-proneness of clones have the potential to identify bug-prone clones based on their change-proneness. Thus, the correspondence between the stability and bug-proneness of clones might be useful to identify and to prioritize important clones for management. As bug-proneness of clones is of great concern, we further investigate another important evolutionary feature of clones, the co-change coupling and its relationships with the bug-proneness. We present an empirical study on the impacts of co-change coupling on the bug-proneness of clones in the next chapter.

Chapter 7
Analyzing the impacts of Co-change Coupling on the Bug-proneness of Code Clones

In the previous chapter (Chapter 6) we presented our study on the relationships between stability and the bug proneness of clones where we observe that stability of clones is related to the bug-proneness. Our empirical results show that clones that change more frequently tend to be more bug-prone as compared to infrequently changed (more stable) clones. Bug-proneness is an important concern related to the impacts of clones on software systems. In this chapter, we investigate *co-change coupling*, another important evolutionary feature of clones to evaluate whether it contributes to the bug-proneness of clones. When a cloned fragment is coupled with higher number of other code fragments, it intuitively increases the complexity and cost of change propagation. Moreover, the higher the number of co-change candidates, the higher might be the likelihood of missing change propagation that may introduce inconsistencies. Although there are many different studies investigating the bug-proneness of different types of clones, none of the existing studies investigate whether the bug-proneness of clones are related to the degree of co-change coupling. In this chapter, we empirically evaluate the impacts of co-change coupling on the bug-proneness of clones with respect to different types of clones and their couplings with clones and non-clones.

The rest of the chapter is organized as follows: Section 7.1 outlines the context and motivation of the study and the research questions. Section 7.2 represents the experimental settings and steps used in the study including the subject systems used, change extraction and classification procedure, and the metrics we measure. Section 7.3 represents the experimental results. Potential threats to the validity of the study are represented in section Section 7.4. Section 7.5 discusses the related works followed by the summary in Section 7.6.

7.1 Introduction

Many studies investigated the impacts of clones on software systems. Some studies [56, 84, 86, 83, 133] show that clones are not harmful rather clones can be beneficial for the software systems [74]. Although clones have some obvious benefits like increasing productivity due to faster software development by code reuse, a good number of studies [69, 97, 100, 22, 34, 95, 94, 135] report strong empirical evidences and conclude that clones have negative impacts on software systems. Thus, code clone is considered as one of the serious code smells. One well-known claim against code clones is that code clones if evolve inconsistently may introduce bugs in the software systems [69]. Also, if a code fragment with unidentified bugs is cloned, hidden bugs may also propagate to the new cloned fragments [94]. Given the serious concerns related to clones, there are a great many studies on the bug-proneness of clones. Some studies focused on the identification and localization of bugs [95, 94, 65] while some studies investigated the comparative bug-proneness of different types of clones [116, 185, 178]. Another important focus of the existing studies is to analyze the features or characteristics of clones related to bugs [163]. A number of studies also analyzed the bug-proneness of clones from the perspective of inconsistencies due to late propagation [18, 22, 171].

Inconsistent evolution of clones due to missing change propagation is considered to be a primary reason for clones to introduce bugs in software systems [69]. Thus, it is important to identify the evolutionary characteristics of clones that contribute to the inconsistencies or bug-proneness of clones. When a cloned fragment co-evolves with a comparatively large number of other related code fragments, it is likely to impose relatively higher comprehension overhead for the developers to identify and evaluate the co-change candidates and to propagate required changes to the co-change candidate. Consequently, this may increase the likelihood of missing some of the required changes to the co-change candidates. This, in turn, may result in unintended inconsistencies or bugs in software systems. Again, clones may co-change with other clone fragments from the same *clone class* (set of two or more code fragments that are clones of each other), clone fragments from other clone classes or even with non-cloned code fragments [113]. The types, volume, and complexity of required changes associated with each evolving clone fragment may vary depending on the number of related co-changing entities and the types of relationships among the code fragments. Therefore, it is important to investigate whether and how evolutionary (co-change) dependencies of clones affect their bug-proneness.

A program entity (*e.g.,* method) is said to have *co-change coupling* if the entity co-changes with one or more other program entities during evolution. We measure co-change coupling of a method as *the average number of methods co-changed per revision* with a given method during evolution. The value of co-change coupling of a method represents the extent of co-change dependency of that particular method on other related methods. Thus, the higher the value of co-change coupling, the higher will be the number of co-change candidates and so the volume and complexity of change propagation. So, the co-change coupling of clones is likely to be an influencing factor for clones to experience bugs due to evolving inconsistently.

Although co-change coupling is an important evolutionary characteristic for clones, none of the existing studies investigated the impact of co-changes coupling on the bug-proneness of clones. Selim *et al.* [158] used survival analysis to study the impacts of clones on software defects and this study partially investigated the impact of change coupling considering only the number of clone siblings. Their case study on two software systems using two clone detection tools (CCFinder [1] and Simian [3]) concluded that bug-proneness of clones might be system dependent. They considered only Type 1 and Type 2 clones and did not carry out the comparative analysis of the bug-proneness of different types of clones as we did. Moreover, this study investigates the relationships between the co-change coupling and the bug-proneness of clones considering different types of coupling relationships.

In this empirical study, we measure co-change couplings of the buggy and non-buggy clones and investigate the impacts of co-change coupling on the bug-proneness of clones. We extract changes from all successive revisions of software systems using *ChangeDistiller* [43] which uses Abstract Syntax Tree (AST) differencing between two consecutive revisions of source code files. Extracted changes from *ChangeDistiller* are mapped to the source code entities and stored for further analysis. We detect both exact (Type 1) and near-miss (Type 2 and Type 3) clones of all the revisions of the systems using the hybrid clone detection tool NiCad [144]. The changes are then mapped to the cloned and non-cloned code. We detect big-fix commits by analyzing the commits messages for SVN repositories. Cloned methods changed in bug-fix commits are considered as buggy cloned methods. We measure the comparative co-change coupling of the buggy and non-buggy clones to evaluate the impacts of co-change coupling on the bug-proneness of different types of clones.

In particular, we represent the findings of our empirical study by answering the following research questions:

RQ1 *Does the degree of co-change coupling affect the bug-proneness of code clones?*

RQ2 *Do different types of clones exhibit different degree of associations between the co-change coupling and their bug-proneness?*

RQ3 *Which of the co-change coupling types in clones are more vulnerable to bugs?*

In this study of analyzing the impacts of co-change coupling on the bug-proneness of code clones, we make the following important contributions:

(1) We define and measure the co-change coupling of clones. We investigate whether the degree of co-change coupling is related to the bug-proneness of clones by comparing the coupling values for buggy and non-buggy clones.

(2) We carry out clone-type based analysis to investigate how the bug-proneness of different types of clones are affected by the degree of co-change couplings. This analysis gives insights into the bug-proneness of different types of clones and how the bug-proneness is associated with the degree of co-change coupling.

(3) We analyze how different types of co-change coupling are associated with the bug-proneness of clones.

We analyze the bug-proneness of clones from a new perspective. The study aims to relate the degree of co-change coupling of clones as a potential influencing factor to the bug-proneness of clones. The findings of the study have the potentials to help in identifying and prioritizing clones for clone management activities such as clone refactoring and tracking.

7.2 Experimental Setup

This section describes the experimental setup of our empirical study including the subject systems used, methodologies for identifying buggy commits, change extraction, clone detection and the measurement of

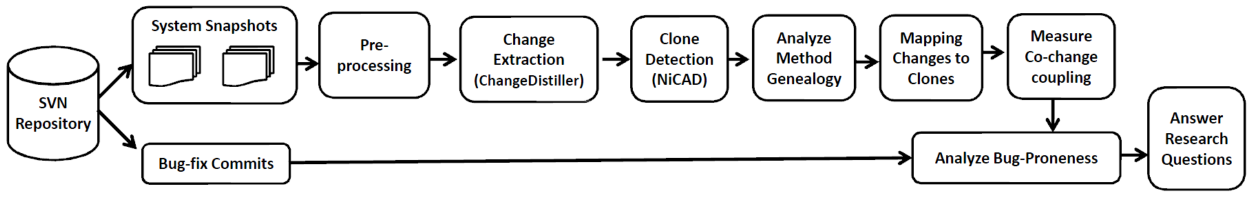


Figure 7.1: Overall Process of Bug-Proneness Analysis

co-change coupling of the buggy and non-buggy clones. We follow the same or similar approaches as in Chapter 6 for preprocessing, clone detection, change extraction and the detection of bug-fix commits. However, we in this study we consider the co-change information only and do not distinguish between change types. The following subsections describe the details of the experimental settings of our study.

7.2.1 Subject Systems

This study is based on six open source systems implemented in Java with diversified size and application domains. Table 7.1 represents the features the software systems including the application domain, size in lines of code (LOC) and the number of revisions. The size of the systems represents the lines of code (LOC) in the last revision of the systems counted after removal of comments and pretty-printing.

7.2.2 Detecting Bug-fix Commits

To identify bug-fix commits of a candidate system, we apply the heuristics proposed by Mockus and Votta [108] on the commit messages to automatically identify bug-fix commits. For this we first extract the SVN commit messages by applying *SVN log* command. A commit message describes the objectives of the associated commit and thus can be used to infer whether the commit is a bug-fix commit. This technique identifies bug-fix commits based on the occurrence of keywords in the commit message from a predefined set. For example, if a commit message contains the word "bug", it will be classified as a bug-fix commit. This heuristics-based approach for identifying bug-fix commits have been used in different earlier studies [22, 116, 120, 21]. This heuristic-based approach may sometimes result in false positives due to incorrect classification of commits as bug-fix commits. However, earlier study [22] shows that this approach can identify bug-fix commits with 87% accuracy. Once the bug-fix commits are listed, we identify the bug-fix commits where the cloned fragments have changed. When any clone fragment is changed in a bug-fix commit, it is reasonable to infer that changes to that clone are necessary to fix the bug. We analyze and identify all the cloned methods that are related to bug-fixes. We then measure the co-change coupling values for buggy and non-buggy cloned methods to evaluate the impacts of co-change coupling on the bug-proneness of clones.

7.2.3 Experimental Steps

We follow some experimental steps to carry out our empirical analysis. Figure 7.1 shows the schematic diagram of the overall experimental analysis process. The experimental steps are briefly described below:

Preprocessing

For our study, we apply some preprocessing steps on the source code. First, we extract all the revisions of the subject systems from their corresponding *SVN*[1] repositories. Our analysis focus on the changes to source code only. Thus, we remove comments from the source files. Pretty-printing of the source files are then carried out to eliminate the formatting differences using the tool *ArtisticStyle.*[2] We extract the file modification history using *SVN diff* command to list added, modified and deleted files in successive revisions. This information is used to exclude the unchanged files during change analysis to speed up the process.

Table 7.1: SUBJECT SYSTEMS

Systems	Type	Size (LOC)	#Rev
DNSJava	DNS Protocol	20831	1679
JabRef	Bibliography Manager	153952	3718
Carol	Driver Application	13213	2237
Ant-Contrib	Web Server	79434	177
OpenYMSG	Open Messenger	8821	233
Squirrel	SQL Client	332635	6737

Method Extraction and Origin Analysis

For our analysis, we extract method information from the successive revisions of the source code. We store the method information (file path, package name, class name, signature, start line, end line) in a database to use for mapping changes to the corresponding methods. Again, SVN keeps track of the files that are added, deleted or modified and the history of changes to individual file content are preserved as the modification of lines. This line-level change information is not sufficient to describe the evolution of source code entities at higher granularity levels such as classes or methods. As a result, to map changes to methods throughout the development cycle, we need to map the methods across the revisions. Therefore, we carry out origin analysis of the methods on the revisions of the systems. We used same the approach for origin analysis as presented in study by Lazano and Wermelinger [97]. Here, if a method is relocated in the same file or if the method

[1]https:/sourceforge.net/
[2]http://astyle.sourceforge.net/

Table 7.2: NiCad SETTINGS FOR THE STUDY

Parameters	Values
Minimum Size	5 lines
Maximum Size	500 lines
Granularity	Method
Threshold	0% (Type 1, Type 2), 30% (Type 3)
Identifier Renaming	blindrename (Type 2, Type 3)

signature is changed, the method is mapped by string comparison with the candidate methods in the new file using *Strike A Match* algorithm [4]. The origin mapping information is used to map the extracted changes and cloning information back to the corresponding methods.

Change Extraction

The change extraction system in this study is implemented in Java based on the change extraction and classification core of *ChangeDistiller* [43]. We have customized the *ChangeDistiller* classifier to analyze local repository exported from SVN. To extract source code changes, two successive versions of the same file are selected from the source repository and then are passed to the differencing engine of the change classifier. The extracted changes are then passed to the classifier for classification. The process is repeated for all changed files (identified by SVN *diff*) and for all the revisions of the subject systems. Changes extracted from two successive revisions of source code files are classified into fined-grained changes to source code entities.

Mapping Change Data

The extracted changes are mapped to their corresponding source code entities (methods) using the origin mapping information for the associated entities. We preserve the extracted, classified and mapped changes into a database to measure metrics for the bug-proneness of clones at the method level granularity.

Clone Detection

For this study, we detect clones using the hybrid clone detection tool NiCad [144]. NiCad is reported to a have higher level of precision and recall [145] and supports the detection of both exact (Type 1) and near-miss (Type 2 and Type 3) clones. We run NiCad on all revisions of the subject systems to detect clones at method (function) level granularity. The clone detection results are then processed to store the clone information in the database. Table 7.2 lists the parameter settings for NiCad used for this study.

Mapping Changes to Clones

We use the extracted method genealogies to map method information to the detected clones in each revision of the software systems. We also map the changes to the clones in each revision. Using the list of identified bug-fix commits we separate the list of the buggy and non-buggy clones. Methods that were changed in any of the bug-fix commits are considered as buggy clones while other cloned methods are considered as non-buggy clones.

7.2.4 Measuring the Co-change Coupling

To calculate the co-change coupling metrics for buggy and non-buggy cloned methods, we first identify the list of all cloned methods of a particular clone type changed using the clone information and the change history extracted from the SVN repositories of the corresponding software systems. Then, for each changed cloned method we determine the list of revisions in which each particular method was changed. We also identify the list of bug-fix revisions for each of the selected cloned methods. If a cloned method changes in at least one of the bug-fix commits (revisions), we consider that method a buggy cloned method. Otherwise, we consider the method as a non-buggy cloned method.

Now for each cloned method changed, we check the change history for all the revisions the selected method was changed. For each revision, we determine how many (1) cloned methods from the same clone class (2) cloned methods from different clone class, and (3) non-clone methods were co-changed with the selected cloned method. Adding the number of co-changed clones both from the same and different clone classes gives the total number of clones co-changed with the selected cloned method. Now, dividing these values by the number of revisions a particular cloned method were co-changed, we get the values of *co-change coupling.* We measure co-change coupling for a cloned method regarding the co-changes with clones from the same clone class, clones from different clone classes, total clones, and total non-clones. Now, we take the average of the coupling values separately for buggy and non-buggy clones to analyze the relationships between co-change couplings and the bug-proneness of different types of clones.

To formally define the co-change coupling metric, let us consider R_b be the set of big-fix commits, M_b and M_n be the set of buggy and non-buggy cloned methods respectively. Then, for a method m_i changed in the set of revisions R_{m_i}=$\{r_1, r_2, ..., r_n\}$, we measure the co-change coupling of m_i as

$$CC\left(m_i\right) = \frac{\sum_{r_j \epsilon R_{m_i}} C_{r_j}\left(m_i\right)}{|R_{m_i}|} \tag{7.1}$$

The value of co-change coupling $CC\left(m_i\right)$ in Equation 7.1 represents the average number of co-changed methods per revision for a given method m_i. Here, $C_{r_j}\left(m_i\right)$ is the co-change coupling of a cloned method m_i in revision $r_j \epsilon R_{m_i}$ *i.e.,* the number of methods co-changed with method m_i in revision r_j. For measuring the co-change coupling regarding a particular type of coupling (*e.g.,* clones from same clone class or different clone class, non-clones) we refine the co-changed method list accordingly.

Table 7.3: COMPARATIVE CO-CHANGE COUPLINGS FOR BUGGY AND NON-BUGGY CLONES OF ALL TYPES.

Coupling Types→	Same Class		Different Class		All Clones		Non-clones		All Methods	
Systems ↓	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n
DNSJava	0.9497	0.7378	2.0538	4.6865	3.0035	2.4882	6.4013	3.5062	9.4000	5.9944
JabRef	0.0937	0.0651	0.7713	1.2620	0.8651	0.8218	7.4284	5.0158	8.2935	5.8376
Carol	0.9414	0.4247	2.5579	4.9882	3.4994	2.1947	7.5695	4.2216	11.0689	6.4163
Ant-Contrib	0.0000	0.2857	2.1041	0.0000	2.1041	0.2857	5.6339	0.5714	7.7380	0.8571
OpenYmsg	0.3222	0.2352	0.7138	0.8703	1.0360	0.6960	9.4583	6.9411	10.4944	7.6372
Squirrel	0.4672	0.4138	1.6348	5.1201	2.1020	1.8148	5.2365	4.7425	7.3386	6.5573
p*-value	0.6892		0.5754		0.2301		**0.0203**		**0.0083**	
Cohen's *d*	0.2762		-0.6801		0.6673		1.4387		1.7543	
PS (%)	58		69		69		84		90	

CC_b, CC_n=Co-change Coupling in Buggy and Non-buggy Cloned Methods respectively

PS=Probability of Superiority, *= Two-tailed test at <0.05

Now, for the system level metrics for co-change couplings for buggy and non-buggy clones, we measure the average coupling per method represented by the following two equations:

(i) The co-change coupling for buggy cloned methods (CC_b) is measured as

$$CC_b = \frac{\sum_{m_i \epsilon M_b} CC\left(m_i\right)}{|M_b|} \tag{7.2}$$

(ii) The co-change coupling for non-buggy cloned methods (CC_n) is measured as

$$CC_n = \frac{\sum_{m_i \epsilon M_n} CC\left(m_i\right)}{|M_n|} \tag{7.3}$$

In Equation 7.2 and Equation 7.3,

- $CC\left(m_i\right)$ represents the co-change coupling for a cloned method m_i.
- A cloned method is considered buggy (*i.e.,* $m_i \epsilon M_b$) if $R_{m_i} \cap R_b \neq \phi$, *i.e.,* the method m_i has changed in at least one of the bug-fix commits. Otherwise, the method is a non-buggy (*i.e.,* $m_i \epsilon M_n$) cloned method.

7.3 Results

We analyze thousands of revisions of six open source software systems of diverse size and application domains. We present our experimental results in the following subsections to answer the research questions we defined in Section 7.1 regarding the impacts of co-change couplings on the bug-proneness of different types of clones.

Answer to RQ1: *Does the degree of co-change coupling affect the bug-proneness of code clones?*

Importance: Existing studies [69, 100] report that when clones are evolved inconsistently (*e.g.,* because of missing required change propagations to the co-change candidates), they may introduce bugs in software systems. When a cloned fragment is changed, the corresponding changes may need to be propagated to other clones in the same clone class, to clones in other clone classes and even to related non-clone code fragments [114]. Thus, the more methods (cloned and non-cloned) a cloned fragment is coupled with, the more will be the extent and the complexity of required change propagation. As a result, if a clone fragment has higher co-change coupling it is likely to add more comprehension challenges to the developer to change all related cloned and non-clone co-change candidates consistently. Consequently, the highly coupled clone may have a higher likelihood of missing required change propagation to co-change candidates. So, it is important to investigate if and to what extent co-change couplings of clones affects the bug-proneness of clones.

Methodology: To answer the first research question (RQ1), we first calculate the values of co-change couplings for buggy and non-buggy clones for all the subjects systems. We measure the co-change coupling of each clone with other clones from the same clone class, with clones from different clone classes and coupling with non-cloned methods using the process described in Section 7.2.4. We also measure the co-change coupling values regarding all clones and for all methods (cloned and non-cloned). Table 7.3 shows the data regarding the co-change couplings for the six subject systems considering all clones types. We compare the corresponding co-change coupling values for buggy and non-buggy clones to analyze the impacts of co-change couplings on the bug-proneness.

As shown in Table 7.3, when we consider the co-change couplings of clones with other methods (both cloned and non-cloned shown in column 'All Methods'), for all six subject systems the value for co-change couplings for buggy-clones are comparatively higher than the corresponding values for non-buggy clones. This implies that the co-change coupling of clones likely to have impacts on the bug-proneness of clones. To analyze whether the degree of co-change couplings for buggy and non-buggy clones are significantly different, we carry out Mann-Whitney Wilcoxon (MWW) [5] test for the coupling values of the buggy and non-buggy clones. The p-value from the test is 0.0083 (two-tailed test, at <0.05) which is very small compared to 0.05. This implies that *there is a significant association between the co-change coupling and the bug-proneness of clones.*

Again, the Mann-Whitney Wilcoxon test examines if the findings are likely due to chance and may not alone fully express the magnitude of differences found. Thus, we also calculate the *effect size* to analyze the magnitude of differences in co-change coupling for buggy and non-buggy clones. Effect size calculates the standardized mean difference between two data sets. We measure the effect size (*Cohen's d* [35]) from the comparative co-change coupling values for buggy and non-buggy clones. We see that the values of the effect size for buggy and non-buggy clones considering couplings with both cloned and non-cloned methods is 1.7543 which belongs to the 'large' category as it is above 0.8. For easier interpretation of the effect size, we also convert the effect size value to the probability of superiority or the *Common Language Effect Size* [106] as shown in Table 7.3. The probability of superiority value for comparative co-change couplings for buggy and non-buggy clones is 90% (considering couplings with both clones and non-clones) meaning that if selected randomly, there is 90% chance that buggy clones will have higher co-change couplings than non-buggy clones. Thus, buggy-clones tend to have higher co-change coupling compared to non-buggy clones.

Again, regarding the couplings with non-clones as in Table 7.3, buggy-clones have higher co-change couplings than non-buggy clones for all six subject systems. This shows that co-change couplings of buggy-clones with non-clones are likely to be higher than that of non-buggy clones. To analyze if the co-change couplings of the buggy and non-buggy clones with non-clones are significantly different, we carry out Mann-Whitney Wilcoxon (MWW) test for the corresponding coupling values of the buggy and non-buggy clones. The p-value for the statistical significance test is 0.0203 (two-tailed test, at <0.05) which is smaller than 0.05. Thus, the co-change couplings of buggy clones are significantly higher than that of non-buggy clones considering the couplings with non-clones. The effect size for coupling with non-clones is 1.4387 (large) which refers to the probability of superiority of 84%. This indicates that for randomly selected cases there is 84% chance that buggy-clones will have higher co-change couplings than non-buggy clones regarding coupling with non-clones.

Now, considering co-change couplings with all clones (clones from both same and different clone class in column 'All Clones') as shown in Table 7.3, buggy clones exhibit higher couplings than non-buggy clones for all six subject systems. However, the differences in corresponding coupling values for buggy and non-buggy clones are not statistically significant (p-value is 0.2301, which is greater than 0.05). Again, regarding the co-change coupling of clones with clones from the same clone class, buggy clones have higher couplings than non-buggy clones for all subject systems (except for Ant-contrib). For couplings with clones from different clone class, we observe a quite opposite scenario compared to couplings with clones from same clone class. Here, five of the six subject systems except the Ant-Contrib exhibit higher coupling for non-buggy clones than buggy clones. The differences are not statistically significant (p-value 0.5754, greater than 0.05). The corresponding coupling values show that couplings with clones from different clone class may not have significant influence on the clones to be bug-prone.

Summary- *Buggy-clones tend to have higher co-change couplings compared to non-buggy clones. Thus, co-change coupling and bug-proneness of clones are related. This association between the co-change coupling and the bug-proneness of clones is likely to be dominated by the couplings of clones with non-clones.*

Table 7.4: COMPARATIVE CO-CHANGE COUPLINGS FOR BUGGY AND NON-BUGGY TYPE 1 CLONES.

Coupling Types→	Same Class		Different Class		All Clones		Non-clones		All Methods	
Systems ↓	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n
DNSJava	1.0039	0.7352	1.8692	5.5585	2.8731	2.6046	6.0121	2.8221	8.8853	5.4267
JabRef	8.3333	0.0045	0.0559	0.1154	0.0567	0.0750	8.1702	5.9776	8.2270	6.0527
Carol	0.5909	0.2285	0.0000	4.3636	0.5909	1.6000	11.4090	5.5000	12.0000	7.1000
Ant-Contrib	0.0000	0.1538	0.1372	0.0000	0.1372	0.1538	7.9379	0.7692	8.0751	0.9230
OpenYmsg	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	8.7777	7.0156	8.7777	7.0156
Squirrel	0.0544	0.0514	0.1316	0.5641	0.1860	0.1904	5.9080	5.4397	6.0941	5.6301
p*-value	0.6892		0.5755		0.8729		**0.0203**		**0.0131**	
Cohen's *d*	0.6284		-0.7569		-0.1183		1.5795		1.5758	
PS	66		69		78		87		87	

CC_b, CC_n=Co-change Coupling in Buggy and Non-buggy Cloned Methods respectively

PS=Probability of Superiority, *= Two-tailed test at <0.05

Answer to RQ2: *Do different types of clones exhibit different degree of associations between the co-change coupling and their bug-proneness?*

Importance: Different types of clones have a different degree of syntactic and semantic similarities and differences. Studies [116, 60, 120] show that different types of clones experience different trends in bug-proneness during evolution. In RQ1, we observe that co-change couplings of clones are significantly related with the bug-proneness. Now, it is important to investigate whether and to what extent the bug-proneness of different types of clones are different. If the co-change couplings of a particular type of clone exhibit strong association with the bug-proneness, co-change coupling is likely to be important to make management decisions for those clones. Thus, we carry out clone-type centric analysis of the bug-proneness regarding the co-change coupling.

Methodology: We consider different types of clones separately to calculate the co-change couplings for buggy and non-buggy clones for all the subject systems. As in RQ1, we measure the coupling values regarding couplings with clones (same clone class, different clone class and combined) and non-clones. We also measure couplings for all methods (clones and non-clones combined). Table 7.4, Table 7.5 and Table 7.6 represent the comparative coupling values for buggy and non-buggy clones for Type 1, Type 2 and Type 3 clones respectively.

As shown in Table 7.4, for all methods (cloned and non-cloned combined) buggy clones have higher values of co-change couplings than non-buggy clones. This indicates that for Type 1 clones, buggy-clones tend to

Table 7.5: COMPARATIVE CO-CHANGE COUPLINGS FOR BUGGY AND NON-BUGGY TYPE 2 CLONES.

Coupling Types→	Same Class		Different Class		All Clones		Non-clones		All Methods	
Systems ↓	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n
DNSJava	0.3814	0.8306	0.3939	0.3571	0.7753	0.9112	8.1235	6.4091	8.8989	7.3204
JabRef	0.2011	0.2368	0.0333	1.0689	0.2344	0.6447	8.6828	6.9241	8.9172	7.5688
Carol	0.0289	0.2986	1.5279	3.7536	1.5569	1.1534	14.1589	6.0577	15.7158	7.2112
Ant-Contrib	0.0000	0.3333	0.0000	0.0000	0.0000	0.3333	1.3333	0.6666	1.3333	1.0000
OpenYmsg	0.1000	0.4285	0.0000	0.0000	0.1000	0.4285	13.6500	3.7142	13.7500	4.1428
Squirrel	0.7127	0.5596	0.4454	3.2297	1.1581	1.1294	6.3651	6.3545	7.5233	7.4839
p*-value	0.1285		0.5755		0.6892		0.0930		0.0658	
Cohen's *d*	-0.8550		-0.7984		-0.2529		0.9764		0.8813	
PS	74		71		58		76		74	

CC_b, CC_n=Co-change Coupling in Buggy and Non-buggy Cloned Methods respectively

PS=Probability of Superiority, *= Two-tailed test at <0.05

have higher co-change couplings than non-buggy clones. From Mann-Whitney Wilcoxon (MWW) [5] test we see that the difference between co-change couplings for buggy and non-buggy clones are statistically significant (p-value 0.0131, less than 0.05). Thus, *there is a significant association between the co-change coupling and the bug-proneness of Type 1 clones.* Here, the effect size is large (1.5758) with probability of superiority is 87% meaning that for randomly selected cases, there is 87% chance that buggy clones will have a higher co-change couplings than non-buggy clones. Again, considering couplings with non-clones we observe that buggy Type 1 clones have higher co-change couplings than non-buggy Type 1 clones in all subject systems. The small p-value (0.0203, less than 0.05) for Mann-Whitney Wilcoxon (MWW) test implies that the differences between the co-change coupling for buggy and non-buggy Type 1 clones are statistically significant with large effect size (1.5795) and the probability of superiority of 87%. Thus, *for Type 1 clones, the degree of co-change couplings with non-clones has a significant association with the bug-proneness.* Now for couplings with clones (same clone class, different clones class and combined), we do not observe statistically significant differences between buggy and non-buggy Type 1 clones.

Table 7.6 represents the comparative couplings for buggy and non-buggy Type 3 clones. Regarding all methods (cloned and non-cloned combined), buggy Type 3 clones have higher values of co-change couplings than non-buggy Type 3 clones for all subject systems. The differences between the co-change couplings for buggy and non-buggy Type 3 clones are statistically significant (p-value is 0.0083, much smaller than 0.05)

Table 7.6: COMPARATIVE CO-CHANGE COUPLINGS FOR BUGGY AND NON-BUGGY TYPE 3 CLONES.

Coupling Types→	Same Class		Different Class		All Clones		Non-clones		All Methods	
Systems ↓	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n	CC_b	CC_n
DNSJava	0.4544	0.2867	1.0014	1.3259	1.4558	0.6845	9.0237	6.4944	10.4795	7.1789
JabRef	0.1914	0.1347	0.9390	1.5160	1.1305	1.1818	7.2411	5.8057	8.3716	6.9876
Carol	0.9323	0.3872	2.5269	2.9301	3.4592	1.5468	7.8856	4.8219	11.3448	6.3687
Ant-Contrib	0.0000	0.3333	2.4083	0.0000	2.4083	0.3333	6.6208	0.6666	9.0291	1.0000
OpenYmsg	0.8200	0.4500	0.1666	0.8500	0.9866	0.8750	9.9733	7.3250	10.9600	8.2000
Squirrel	0.4921	0.3941	1.6407	4.7330	2.1329	1.6716	5.5202	5.0282	7.6531	6.6998
p*-value	0.2983		0.9362		0.1285		**0.0455**		**0.0083**	
Cohen's *d*	0.5709		-0.3273		1.1666		1.340		1.6990	
PS	64		58		80		82		88	

CC_b, CC_n=Co-change Coupling in Buggy and Non-buggy Cloned Methods respectively

PS=Probability of Superiority, *= Two-tailed test at <0.05

with large effect size (1.6990) and a probability of superiority of 88%. Thus, *there is a significant relationship between co-change coupling and the bug-proneness for Type 3 clones*. Again, considering couplings with non-clones buggy Type 3 clones have higher co-change couplings than non-buggy clones for all the subject systems. The differences of corresponding co-change couplings for buggy and non-buggy clones are statistically significant (marginally, with p-value 0.0455) with large effect size (1.340) and 82% probability of superiority. Thus, *couplings with non-clones have a significant association with the bug-proneness of Type 3 clones.* Again, when we consider couplings with all clones (same and different clone classes combined), although differences are not statistically significant (p-value is 0.1285), buggy clones have higher coupling values than non-buggy clones for all the systems. Also, we do not observe significant differences in couplings for buggy and non-buggy clones if we separately consider couplings with clones from same and different clone classes.

Table 7.5 represents the comparative couplings for buggy and non-buggy Type 2 clones. Although we observe significant relationships between the co-change coupling and bug-proneness for Type 1 and Type 3 clones (for couplings with clones and for clones and non-clones combined), we do not see any statistically significant relationship between co-change couplings of Type 2 clones and the bug-proneness. This result for Type 2 clones might be because Type 2 clones are structurally exact with differences in identifier names and data types. Type 2 clones are likely to be created because of the reuse of templates, and thus may evolve independently. However, the couplings for buggy clones tend to have higher values compared to non-buggy clones regarding

Table 7.7: COMPARATIVE COUPLINGS OF BUGGY AND NON-BUGGY CLONES REGARDING DIFFERENT TYPES OF CO-CHANGE COUPLINGS.

Coupling Types→	Same Class			Different Class			All Clones			Non-clones			All Methods		
Systems ↓	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3
DNSJava	⊕	⊕	⊕	⊕	⊕	⊖	⊕	⊖	⊕	⊕	⊕	⊕	⊕	⊕	⊕
JabRef	⊕	⊕	⊕	⊖	⊖	⊖	⊖	⊖	⊖	⊕	⊕	⊕	⊕	⊕	⊕
Carol	⊕	⊕	⊕	⊖	⊖	⊖	⊖	⊖	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Ant-Contrib	⊖	⊖	⊖	⊕	⊙	⊕	⊖	⊖	⊕	⊕	⊕	⊕	⊕	⊕	⊕
OpenYmsg	⊕	⊙	⊕	⊙	⊙	⊖	⊙	⊖	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Squirrel	⊕	⊕	⊕	⊖	⊖	⊖	⊖	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕

⊕=cases where coupling in buggy clones is higher than the coupling in non-buggy clones ($CC_b > CC_n$)

⊖=cases where coupling in buggy clones is smaller than the coupling in non-buggy clones ($CC_b < CC_n$)

⊙=cases where frequency of changes in buggy and non-buggy clones are same ($CC_b = CC_n$)

couplings with non-clones and for couplings with clones and non-clones combined.

Summary- *Type 1 and Type 3 clones exhibit significant relationships between co-change couplings and the bug-proneness regarding couplings with non-clones and clones and non-clones combined. However, Type 2 clones do not show significant association between the co-change coupling and their bug-proneness.*

Answer to RQ3: *Which of the co-change coupling types in clones are more vulnerable to bugs?*

Importance: Clones may co-evolve with both other clones and non-cloned code fragments. A cloned fragment from one clone class may co-change with other clone fragments from the same clone class, with clones from different clone classes and with other non-cloned code fragments [114]. The higher the number of different other cloned and non-cloned fragments co-change with a cloned fragment, the volume and complexity of change propagation are likely to increase. This, in turn, may add comprehension overhead for the developers and may increase the likelihood of missing change propagation leading to inconsistencies or bugs in the system. Thus, it is important to investigate how the co-change couplings of different types (with clones and non-clones) are related to the bug-proneness.

Methodology: To analyze the bug-proneness of clones regarding the different types of coupling (with clones and non-clones) scenarios, we compare the corresponding coupling values for buggy and non-buggy clones. We summarize the coupling data for Type 1 Type 2 and Type 3 clones from Table 7.4, Table 7.5 and Table 7.6 respectively in Table 7.7. For each type of coupling, we get 18 cases (6 subject systems x 3

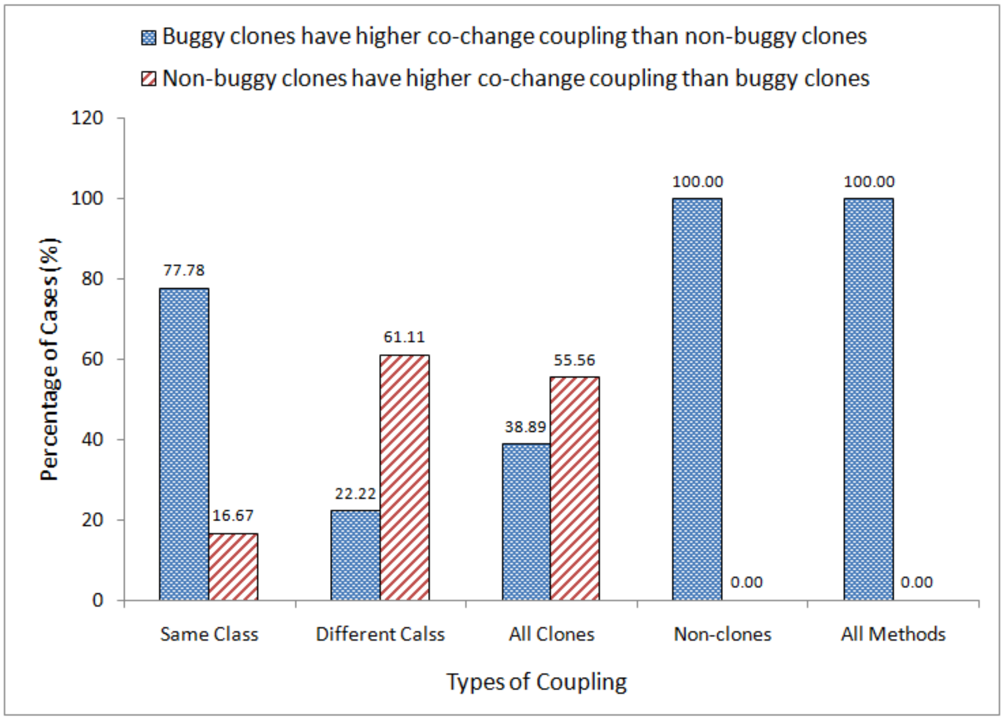


Figure 7.2: Comparative Couplings of buggy and non-buggy clones for different types of coupling

clone types) of comparative co-change couplings. Based on the comparative coupling values for buggy and non-buggy clones, we represent each case using one of the three different symbols $\oplus$, $\ominus$ and $\odot$ where the symbols indicate if the coupling value for buggy clones is greater than, less than or equal to coupling value for non-buggy clones respectively. Then, we observe what percentage of cases (out of 18 cases for each coupling type) show the higher or lower values of co-change coupling for buggy clones compared to non-buggy clones. We represent the comparative percentage values in Figure 7.2.

Now, from Table 7.7 and Figure 7.2 we see that for couplings with all methods (clones and non-clones), the co-change couplings for buggy clones are greater than that of non-buggy clones for 100% (18/18) cases. The same scenario is observed for couplings of clones only with non-clones where buggy clones have higher couplings for all 18 cases. Thus, when coupling with both clones and non-clones and couplings with only non-clones are considered, buggy clones have higher couplings compared to non-buggy clones for all clone types.

For, couplings with clones we observe that in 7 out of 18 (38.89%) cases, buggy-clones have higher couplings than non-buggy clones. When we consider couplings with clones from the same clone class and different clone classes separately, the percentages of cases where buggy clones have higher couplings than non-buggy clones are 77.78% (14/18) and 22.22% (4/18) respectively. When couplings with both clones and non-clones are considered, we see that buggy-clones tend to have higher couplings than non-buggy clones. However, from the summary in Table 7.7 and from the statistical analysis results in Table 7.4, Table 7.5 and Table 7.6, we observe that couplings of clones with non-clones have significant relation with bug-proneness while for coupling with clones from same clone class buggy clones tend have higher couplings than non-buggy clones. However, couplings of clones with clones from different clone classes tend to exhibit higher coupling values for non-buggy clones than buggy clones. Thus, we can conclude that *the relationship between co-change coupling and the bug-proneness of clones are mostly dominated by the couplings of clones with non-clones.* For co-change couplings with clones, the coupling of clones from the same clone class likely to contribute to this relationship which is reflected in the statistically significant relationships between couplings and the bug-proneness of clones (Type 1 and Type 3) when couplings with both clones and non-clones are considered. However, for Type 2 clones, although the coupling differences for buggy and non-buggy clones are not significantly different, buggy clones tend to have higher couplings than non-buggy clones.

Summary- *The association between the co-change coupling and the bug-proneness of clones are likely to be influenced by the couplings of clones with non-clones. When couplings with both clones and non-clones are considered, the observed association between the co-change coupling and the bug-proneness is likely to be influenced by the couplings of clones with non-clones and coupling with clones from the same clone class.*

7.3.1 Significance of the Findings

From the results of our empirical study, we observe that co-change couplings have significant impacts on the bug-proneness of clones. Our findings are important from different perspectives of managing software clones. First, clones are known to be a serious code smell and there are considerable proportions (7%-23% [151]) of cloned code in software systems. Clones, when evolve inconsistently, may introduce bugs [69]. Results from our empirical study show that there is a statistically significant relationship between the co-change coupling and the bug-proneness of clones. Thus, bug-proneness of clones are related to the degree of co-change coupling. Consequently, the degree of co-change coupling extracted from the analysis of the evolution is likely to be an important feature to identify bug-prone clones. As we observe that degree of co-change coupling is significantly associated to bug-proneness, clones with higher couplings are likely to experience bugs. Therefore, the degree of coupling can be useful to select and prioritize buggy clones for management by refactoring and tracking. Second, our findings show that Type 1 and Type 3 clones have a stronger association between the co-change coupling and their bug-proneness compared to Type 2 clones. This finding can help in making management decisions for different types of clones based on the degree of co-change couplings. Third, from our findings, we observe that the couplings of clones with non-clones are more vulnerable to bugs while co-change couplings of clones with clones from same clone class tend to have higher associations with bug-proneness. Thus, for the management of bug-prone clones, more emphasis might be given on the degree of co-change couplings with non-clones and couplings with clones from same clone class. From the findings, we see that our study reveals co-change coupling as an important feature of buggy clones and give useful insights regarding how the degree of co-change coupling can be useful for selecting and prioritizing clones for management especially the clones those are bug-prone.

7.4 Threats to Validity

For our study, we used NiCad, a hybrid clone detection tool which detects both exact (Type 1) and near-miss (Type 2, and Type 3) clones with high precision and recall. Different settings for clone detection tools might affect the study results because of the variations in clone detection results. This is termed as *confounding configuration choice problem* [180]. However, the NiCad settings we used are considered standard [151] and are close enough to the optimal configuration settings identified in recent study [180] for NiCad to detect clones in Java systems. This is likely to mitigate the potential adverse effects of the configuration settings on our findings.

We selected subject systems with diversified size, number of revisions, length of evolution and application domain to avoid potential biasing. However, due to the limitation of existing change classifier our study is limited to Java systems only. Although Java is considered to be a widely used language with a comprehensive set of language features for software development, the inclusion of subject systems of other languages might help in more generalization of the findings.

7.5 Related Work

Many studies have investigated the bug-proneness of cloned code from different perspectives. Wagner *et al.* [178] in an industrial case study investigated the relationships between Type 3 clones and faults. This study shows that a considerable proportion (17%) of Type 3 clones are associated with faults. Mondal *et al.* [116] in their empirical study on the bug-proneness of different types of clones report that Type 3 clones are more bug-prone compared to Type 1 and Type 2 clones. However, Type 3 clones related to bug-fixes have a higher likelihood of evolving consistently compared to Type 1 and Type 2 clones. Mondal *et al.* in a recent study [118] defined clone evolution patterns related to bug-propagation through code cloning. Their study reports that near-miss (Type 2 and Type 3) clones are more prone to bug-propagation compared to exact (Type 1) clones. These studies give important insights for making clone management decision regarding the bug-proneness of clones of different types. However, we focus on evaluating co-change coupling to evaluate whether it has impacts on the bug-proneness of clones.

Li *et al.* [95] proposed the tool CP-Miner that uses data mining techniques to detect copy-baste clones and bugs in large-scale software systems. This tool focuses mostly on identifying inconsistencies due to missing change propagation to copy-pasted clone fragments. Li and Ernst [94] conducted an empirical study on the bug-proneness of clones and developed a tool called CBCD based on their study. For a given buggy code fragment, CBCD searches semantically identical copies of the given code fragment. Jiang *et al.* [65] introduced the notion of context-based inconsistencies and proposed an approach for detecting clone related bugs. They used DECKARD [64] clone detector and categorized the detected bugs and style issues for two open-source software systems (Linux Kernel and Eclipse) to explore diverse characteristics of clone-related bugs. Inoue *et al.* [58] developed a tool called CloneInspector to detect inconsistent changes to code clones and associated latent bugs in software systems.

Barbour *et al.* [22] examined the late propagation of clones (Type 1 and Type 2) and investigated the extent to which different types of late propagation are related to bugs and inconsistencies in software systems. Aversano *et al.* [18] investigated the evolution of clones in two software systems reporting that clones evolve consistently in most cases while late propagation may introduce faults. Thummalapenta *et al.* [171] show that clones with late propagation are more bug-prone than others.

Xie *et al.* [185] studied the fault-proneness of clones with respect to mutation and migration of clones during evolution. Their investigation on three open-source software systems concludes that both mutation of a clone group to Type 2 or Type 3 clones and increasing distance between clone fragments in a clone group increase the risk for faults. Steidl and Göde [163] proposed a machine learning approach to investigate how different features of clones can be used to automatically identify incomplete bug-fixes. Göde and Koschke [51] analyzed two mature software systems and reported that changes to clones are mostly infrequent and a small proportion (14.8%) of changes are unintentionally inconsistent. Selim *et al.* [158] used survival analysis to study the impacts of clones on software defects and concluded that defect-proneness of clones are system dependent.

Chatterji *et al.* [33] studied the effects of code clones on software maintainability with respect to bugs. Their study shows that fixing cloned bugs may be significantly difficult than fixing non-cloned bugs and the developers when aware of clones perform better in fixing bugs. Chatterji *et al.* in another study [34] investigated how clone information can support developers to localize bugs. Sajnani *et al.* [156] analyzed the comparative bug-proneness of cloned and on-cloned code on 31 Java projects. They report that the density of bugs is comparatively less in clones than non-cloned code. Rahman *et al.* [133] in their empirical study found that clones may be less defect-prone than non-cloned code which is opposite to the findings of the study by Juergens *et al.* [69]. Islam and Zibran [59] investigated the vulnerabilities associated with different types of clones and non-cloned code on 97 software systems. This study reports that due to the larger code proportion non-cloned code are related to more vulnerabilities and Type 3 clones have the highest association with vulnerabilities.

The existing studies reveal important insights regarding the techniques for the detection [95, 65] of bug-prone clones, identifying features and types of bugs [65] and analyzing bug-proneness of different types of clones [116] from the perspective of consistent evolution. However, none of the existing studies consider different types of co-change coupling with clones and non-clones to analyze bug-proneness of clones. In this study, we investigate the relationship between the co-change coupling and the bug-proneness of clones. We evaluate co-change coupling as a potential influencing factor to the bug-proneness of clones. We consider co-change couplings with clones from same and different clone class and non-cloned methods to analyze the bug-proneness of Type 1, Type 2 and Type 3 clones.

7.6 Summary

In this study, we investigate the impacts of co-change coupling on the bug-proneness of different types of clones. We measure the degree of co-change coupling of the buggy and and non-buggy cloned methods regarding both exact (Type 1) and near-miss (Type 2 and Type 3) clones. We investigate the impacts of co-change coupling on the bug-proneness of clones by separately considering the co-change couplings of clones with clones and non-clones.

We automatically identify the bug-fix commits by analyzing the commit messages from the SVN repositories. We detect Type 1, Type 2 and Type 3 clones using the hybrid cloned detection tool NiCad. Changes are extracted using ChangeDistiller classifier. Clones that change in any of the bug-fix commits are identified as buggy clones. By measuring the comparative coupling values of the buggy and non-buggy clones for six diverse open source Java systems, we observe that the co-change coupling and the bug-proneness are related. Our experimental results show that buggy clones tend to have higher coupling compared to non-buggy clones and this relationship is often significant.

In this empirical study, our investigation of the relationships between the co-change coupling and the bug-proneness of clones helps in having deeper insights into clone evolution. The relationship between co-change coupling and the bug-proneness might be a useful indicator whether co-change coupling is an influencing factor for clones to be bug-prone. Thus, the information regarding the correspondence between the co-change coupling and bug-proneness of clones might be useful to identify and to prioritize important clones for management.

The studies we presented up to this chapter comprehensively investigate the impacts of clones on software systems. We investigated the comparative stability of cloned and non-cloned code from new perspectives (*e.g.,* change types Chapter 3) and within a uniform framework (Chapter 5) for comparison without possible bias. Then we studied the impacts of clones from the perspective beyond source code (*e.g.,* domain-based coupling in Chapter 3). In addition, we investigated two important evolutionary characteristics of clones, stability (Chapter 6) and co-change coupling (Chapter 7). These studies address our research goals in Phase 1. We observe from our studies in Phase 1 that although we can not always draw a firm conclusion on the impacts of clones on software systems, clones may have negative impacts on the software systems. Thus, developers need to manage clones properly to take advantages of cloning while avoiding the negative impacts of code clones as recommended by the researchers [148]. So, the developers need support tools and techniques to identify the co-changes candidates for clones to ensure the consistent evolution of clones. Phase 2 of our research addresses this objective as presented in the next two chapters. In Chapter 8, we present a clustering-based technique to minimize the impacts of atypical commits on the detection of evolutionary coupling. Then, in Chapter 9 we propose an evolutionary coupling based change impact analysis technique for clones.

Chapter 8

Analyzing the Effects of Atypical Commits on the Detection of Evolutionary Coupling

In Chapter 3, Chapter 4 and Chapter 5 we investigated the impacts of clones on software systems from different perspectives. Then, in Chapter 6 and Chapter 7 we empirically evaluated two important evolutionary features of clones (stability and co-change coupling) to assess whether they are related to the bug-proneness of clones. Our empirical results suggest that clones often tend to have negative impacts on software systems. However, the results of the comparative impacts of cloned and non-cloned code may vary depending on the subject systems, the tools and the parameters settings used in the studies. Despite the variations in the outcomes of the studies on the impacts of clones on the software systems, researchers agree that clones need to be managed [149]. One of the key objectives of clone management is to ensure the consistent evolution of clones. This requires the identification of candidate clones for change propagation when a cloned fragment is changed.

Detection of evolutionary coupling is a promising technique to discover dependencies among the evolving program entities. When two or more program entities change together during software evolution, they are said to have evolutionary coupling. Evolutionary coupling is measured by mining association rules based on the frequency of change and co-change of entities extracted from the commit history from the software repositories. Thus, when different groups of unrelated entities are changed in the same commit, it may introduce false associations among the co-changed program entities. This is most likely to occur when an excessively large number of entities are changed in an atypical commit (*i.e.,* extra-large commit).

In this chapter, we present a clustering-based technique to split atypical commits or transactions based on the change history in non-atypical commits. We split atypical commits into a set of pseudo commits to remove possible false associations. We comparatively evaluate whether automatic transaction splitting contributes to the improvement of accuracy in detecting evolutionary coupling. Our experimental results on five diverse subject systems written in Java show that transaction splitting has the potential to minimize the impacts of atypical commits. The proposed approach improves the precision and the recall of the detection of evolutionary coupling as compared to the conventional approach.

The rest of the chapter is organized as follows: Section 8.2 defines and explains related key terminologies. Section 8.3 outlines the important motivations of the study. Section 8.4 represents the experimental settings including the subject systems used, change extraction and classification procedure, and the metrics we measure. Section 8.5 represents the experimental results. Potential threats to the validity of the study are presented in section Section 8.6. Section 8.7 discusses the related works followed by the conclusion in Section 8.8.

8.1 Introduction

Software systems evolve through various changes. Due to the underlying relationships and dependencies among the components of the software systems, changes in one component may introduce changes to other components in a software system. Identification of software components related to a candidate component for change, known as Change Impact Analysis (CIA) [13], is an important task in software maintenance and evolution. However, due to the increasing complexities in architecture and dependencies among the components in modern software systems, change impact analysis has been a challenging task for the software developers and the maintainers. A significant amount of research efforts over the past few decades in CIA has produced a wide spectrum of methods and tools by using different techniques including but not limited to static analysis [92, 130], dynamic analysis [12, 127, 128], Mining Software Repositories (MSR) [189, 72, 31], and Information Retrieval (IR) [32].

Detection of evolutionary coupling is a useful technique to discover dependency relationships among the evolving software components. When two or more software entities co-change (change together in same commits) frequently, the entities are said to have evolutionary coupling, also known as logical coupling. Evolutionary couplings among software entities are represented by association rules [189]. Two well-known metrics, *support* and *confidence* for the association rules are used to quantify the strength of the evolutionary coupling among the associated entities. Both *support* and *confidence* are measured in terms of the frequencies of co-change and the frequencies of changes to individual entities by mining change history from software repositories. Higher values for *support* and *confidence* for association rules indicate stronger evolutionary coupling among the associated entities.

Evolutionary coupling represents the dependencies among co-changed program entities from the perspective that the developers change related entities together and these changes are then reflected in the software repository by a single commit operation. Thus, the history of co-change of the evolving entities in the software repository is likely to represent the logical coupling among the co-changed entities. Conventionally, evolutionary coupling is detected by leveraging the co-change history assuming that history of co-change represents the existence of coupling among the associated entities. Any history of co-change thus may have some degree of contributions to the strength of the evolutionary coupling.

However, software systems may undergo major refactoring activities during the evolution due to significant changes in requirements, technology, and other infrastructures. Such refactoring activities may result in changes to an excessively large number of entities in the software systems in a single commit, introducing an *atypical commit* in the evolution history. So, instead of a small group of related entities to co-change as in regular or non-atypical commits, different groups of related entities are likely to co-change in an atypical commit. Moreover, developers may choose either to commit a single set of logically coupled entities or multiple groups of logically coupled entities in a single larger commit *i.e.,* an atypical commit. Thus, the assumption regarding evolutionary coupling that *"related entities change together"* may be highly sensitive to the commit preference of the developers or whether there involve some major refactoring activities. Consequently, a larger number of entities from more than one co-evolving groups of entities may change in a single commit introducing a considerable amount of false positives in the detection of evolutionary coupling.

Although most of the commits (about 75%) involve a fairly small number of changed entities per commit, other commits might be excessively large introducing atypical commits [10]. Atypical commits may occur as a results of tangled changes [78] that involve changes to unrelated or loosely related entities [53]. These larger commits (atypical) can introduce a considerable proportion of false associations. From our analysis, we observe that a major proportion (26.70%) of the changed methods in the software systems are changed exclusively in larger commits while another (28.28%) of the methods are associated with both atypical and non-atypical commits. Thus, atypical commits are associated with the change history of more than half of the changed methods. This may have significant negative effects on the results of studies involving evolution analysis of source code entities.

Given the importance of the effects of atypical commits, Lozano *et al.* [97] opted to discard a portion (2.5%) of the large commits in their analysis of code clone evolution. Zimmermann *et al.* [189], on the other hand, excluded any transaction larger than 30 in their analysis of evolutionary coupling. However, such exclusion of atypical commits may result in loss of significant proportion of co-change history of the associated entities. Although, larger commits may look like outliers, they are important [54]. As atypical commits comprise of a major portion of changed entities, we need to preserve the useful change information regarding true couplings while filtering out the false coupling introduced by the entities appearing in the atypical transactions. Thus, instead of simply discarding atypical commits (either fully or partially), we need a solution to extract true coupling from atypical commits.

Mondal *et al.* [112] proposed a metric, *Significance*, alternative to the well-known metrics *support* and *confidence* of association rules to improve the accuracy of the detection of evolutionary coupling. This approach partially addressed the effects of atypical commits by assigning less weight to the co-change appearing in larger commits. Although this approach addresses the effects of atypical commits, the proposed metric is not intended to separate true couplings from false couplings introduced by atypical commits. Instead, this approach assigns less weight to all couplings introduced by larger commits.

In this study, we introduce an approach to mitigate the effects of atypical commits on the detection of evolutionary coupling. This approach applies DBSCAN [42] clustering algorithm on the atypical transactions to split the atypical transactions into smaller transactions of methods those are likely to be truly coupled. We define a distance function based on the frequency of co-change of the methods in the change history. The more frequently a pair of methods co-changed in commit history, they less their distance will be. Based on this distance function, the clustering algorithm splits atypical transactions into smaller transactions. In case of co-change of a pair of methods in an atypical commit, we consider it as a co-change only if both methods belong to the same cluster obtained from the associated atypical commits. We then extract evolutionary coupling considering the clusters obtained from the atypical commits. We also extract evolutionary coupling using the conventional approach which discards commits larger than a constant threshold to detect evolutionary coupling. We empirically evaluate comparative performance of the proposed approach and the conventional approach.

In particular, we represent our findings by answering the following research questions:

RQ1 *To what extent atypical commits affect the accurate detection of evolutionary coupling?*

RQ2 *Can we predict future co-change candidates more accurately by evolutionary coupling detected using automatic splitting of atypical commits?*

Our experimental results based on the analysis of thousands of commits (revisions) of five diverse subject systems written in Java show that *we can improve the precision and recall of the conventional approach for detection of evolutionary coupling by clustering-based splitting of atypical commits.*

We can summarize the key findings as-

- Atypical commits contain a significant proportion of the changed entities in the software systems. Around 26.70% of the changed methods are exclusively associated with atypical commits whereas another 28.28% of the changed methods are associated with both atypical and non-atypical commits. Thus, atypical commits can significantly affect the detection of evolutionary coupling because of the full or partial associations with more than half of the changed methods in the software systems.

- The proposed approach demonstrates a considerable improvement in precision in predicting co-change candidates based on evolutionary coupling without compromising overall recall.

Thus, our empirical results demonstrate that by using clustering-based automatic splitting of atypical transactions can considerably minimize the negative effects of atypical commits on the detection of evolutionary coupling. The proposed approach yields improved precision in detecting co-change candidates based on the evolutionary coupling.

8.2 Terminology

Transaction A *transaction* refers to a set of changes submitted by a developer to the version archive in a single commit or revision [189]. A transaction may represent descriptive change information associated with a commit including the types of changes, affected methods, classes and files and so on. However, in our study, we are only concerned about what methods are changed in a particular commit. Thus, by *transaction* we refer to the set of methods changed in a particular revision. We assign a unique ID to each method and thus a transaction in this study is simply represented by a set of IDs of methods changed in a given commit or revision.

Atypical Commits Atypical commits are commits or revisions that involve changes to an unusually higher number of source code entities. Such commits are believed to be caused by major structural changes to software systems [97] rather than changes to a small group of related entities. Existing Study by Alali *et al.* [10] shows that although most of the commits (75%) are fairly small, there exists larger to excessively larger commits in the software systems. They measure commits with respect to the number of files changed, the number of lines changed, and the number of hunks (contiguous blocks of lines) changed in a particular commit. However, as our study focuses on the evolutionary coupling at the method level granularity, we define atypical commits as the commits involving changes to an excessively higher number of methods.

Evolutionary Coupling When two or more entities (e.g, files, classes, methods) in a software system co-change frequently (in many revisions) during evolution, these entities are said to have evolutionary coupling which is also known as change coupling [47]. The existence of evolutionary coupling among the software entities is an indication of dependencies among the associated entities and changes to one such entity may introduce corresponding changes to other coupled entities in the software system. As evolutionary coupling reveals the underlying relationships among software components, it helps in Change Impact Analysis (CIA) by identifying co-change candidates when a related entity is changed during evolution [189].

Association Rule An association rule is an expression of the form $X \Rightarrow Y$ where X is called the antecedent and Y is called the consequent of the association rule. X and Y are sets of one or more software entities. In the context of our study, an association rule is interpreted as *"if method X is changed in a revision, Y is likely to be changed in the same revision"*. The strength of this likelihood is expressed in terms of the *support* and *confidence* of the association rule defined below.

Support is defined as the number of commits (revisions) an entity or a set of entities were changed [189]. *Confidence* of any association rule $X \Rightarrow Y$ is defined as,

$confidence(X \Rightarrow Y) = support(X, Y)/support(X)$

This represents the probability that consequent Y will change given that antecedent X is changed in the same revision.

Clustering Clustering is the process of organizing objects into groups such that member in a group are similar in some way [42]. Clustering is an important technique in data mining for classification of objects by an unsupervised learning process.

DBSCAN Clustering DBSCAN is a density-based clustering technique that effectively finds all clusters independent of the size, shape and cluster locations [42]. This algorithm defines two important parameters *epsilon neighborhood (e)* defining the maximum neighborhood distance within a cluster and the *minimum points* required to form a cluster.

Distance function for DBSCAN clustering For clustering the methods appearing in an atypical transaction, we need to define distance function for the DBSCAN clustering algorithm. Here, by the distance, we refer to how frequently a pair of methods has co-changed during their evolution. The more frequently a pair of methods co-change, the closer the methods are. Thus, a frequently co-changed method pair has closer distance than an infrequently co-changed pair of methods. Thus the distance is inversely proportional to the frequency of co-change of any pair of methods. For any pair of methods M_1 and M_2 changed in the set of commits or revisions $R_1 = \{r_{11}, r_{12}, \ldots, r_{1m}\}$ and $R_2 = \{r_{21}, r_{22}, \ldots, r_{2n}\}$ respectively, we define the distance function formally considering three possible cases (C_1, C_2 and C_3) as,

C_1**:** When $|R_1 \cap R_2| >= 1$ *and* ($|R_1| > 1$ OR $|R_2| > 1$)

$$dist(M_1, M_2) = \frac{1}{|R_1 \cap R_2|} \tag{8.1}$$

Here, methods M_1 and M_2 each changed multiple times and they co-changed more than once during evolution. We consider their distance as the reciprocal of the frequency of their co-change.

C_2**:** When $|R_1 \cap R_2| = 1$, $|R_1| = 1$ *and* $|R_2| = 1$

$$dist(M_1, M_2) = eps \tag{8.2}$$

Here, methods M_1 and M_2 each changed only once and they co-changed in the same commit during evolution. We assign their distance larger than maximum neighborhood distance for items to be in same cluster.

C_3**:** When $|R_1 \cap R_2| = 0$, $|R_1| >= 1$ *and* $|R_2| >= 1$

$$dist(M_1, M_2) = MAX \tag{8.3}$$

When methods M_1 and M_2 never co-changed, we assign their distance a value large enough to keep them in different clusters *i.e.,* a value greater than the maximum possible distance within a cluster.

Here, *eps* is the maximum limit of neighborhood distance for items (methods) to belong to the same clusters. The normal range of the values for the distance function is $0 < dist(M_1, M_2) <= 1$. The value of MAX is any value > 1 which forces a pair of methods never co-changed in any commit not to belong to the same cluster.

Justification for the distance function: Let A, B and C are three co-changed methods where method A changed in five (5) revisions $\{r1, r2, r4, r5, r6\}$ while method B changed in six (6) revisions $\{r1, r2, r3, r5, r7, r9\}$ and method C changed in four (4) revisions $\{r7, r8, r10, r11\}$. Other two methods D and E changed only in revisions $\{r12\}$ while another method F changed in revision $\{r14\}$ respectively. Here, methods A and B co-changed in three (3) revisions $\{r1, r2, r5\}$ and methods B and C co-changed only in one (1) revision $\{r7\}$ while methods A and C never co-changed.

Here the $dist(A, B) = 1/3 = 0.33$, $dist(B, C) = 1/1 = 1.0$, $dist(A, C) = MAX$. Here, methods A and B co-changed more frequently than method pair (B, C). Thus method pair (A, B) has smaller distance than (B, C). Again, method pair (A, C) never co-changed having MAX distance. If $eps = 0.5$, methods A and B may belong to same cluster while method pair (B, C) and (A, C) cannot belong to the same cluster. Again each of the methods in the pair (D, E) co-changed in their only revision of change $\{r12\}$. Here, $dist(D, E) = eps = 0.5$ which allow D and E to belong to the same cluster. Although, both method pairs (B, C) and (D, E) co-change in only one revision each, unlike methods D and E the methods B and C changed in several other revisions. Thus, the distance function assigns higher distance to (B, C) than (D, E) which is logical. Again, neither of the method pairs (D, F) and (E, F) co-changed in any revision. So, the distance for both method pairs are assigned to MAX.

We set two parameters for the DBSCAN clustering algorithm, $eps = 0.5$ and the minimum number of two (2) points (methods) are required to form a cluster. The value $eps = 0.5$ specifies that for methods changed in more than one revisions to be in the same cluster must co-change at least twice *i.e.,* the distance is $1/2 = 0.5$. However, for methods changed only in one revision where they co-changed, we allow them to form a cluster by assigning the distance equal to eps.

8.3 Motivation

Many studies show the effectiveness of evolutionary coupling as a useful representation of dependency relationships among the software entities [189, 129, 115, 114]. Evolutionary coupling can be detected from the software change history and it does not require computationally expensive program analysis as in static and dynamic analysis (analysis of runtime information) approaches [12, 127, 128] for change impact analysis. However, the accuracy of the conventional approaches for the detection of evolutionary coupling is very low [189]. This is likely to limit the application of the evolutionary coupling in practice for dependency analysis similar to the less precise static analysis approaches [92, 130].

The conventional approach [189] for detecting evolutionary coupling considers only the frequency of co-change. Thus, any occurrence of co-change will have some degree of contribution to the detected evolutionary coupling. However, earlier study [10] shows that about 25% of the revisions can be excessively large (atypical commits). Such commits may usually be introduced by major refactoring activities. Developers' preference to commit more than on related groups of entities may also introduce larger commits. Whatever might be the reasons, atypical commits likely to introduce false associations among the co-evolving program entities. This results in an excessive number of association rules which may, in turn, make the evolutionary coupling less useful in practice causing the suggested list of change candidates to be too long for the developers to support the change propagation process.

For example, if there are N methods changed in a revision, each method will have an association to some extents with the remaining (N-1) co-changed methods. This commit may have contributions to $2{*}n_{C_2} = 2 * \frac{n!}{2! * (n-2)!}$ association rules. Again, from the analysis we see that an atypical commit can be as large as containing hundreds of co-changed methods. Given that a significant proportion of entities are associated with atypical commits, exclusion of atypical commits from evolution analysis may miss a significant portion of the change history. Thus, we need techniques to extract the useful coupling information from the atypical commits while filtering out the false associations. In this study, we propose a solution to the above-mentioned problem using clustering-based splitting of atypical commits into smaller transactions representing groups of entities those are likely to have true evolutionary coupling.

8.4 Experimental Setup

This section outlines the experimental settings for different components and steps of our empirical study including preprocessing of subject systems, change extraction and detection of evolutionary couplings and measurement and comparison of the accuracies.

8.4.1 Subject Systems

This study is based on five open source systems implemented in Java. Subject systems studied are of diversified size, evolution history and application domain. Table 8.1 briefly represents the features the software systems including the application domain, length of evolution history, size in lines of code (LOC) and the total number of revisions studied. The size of the systems represents the lines of code (LOC) in the last revision of the systems studied counted after removal of comments and pretty printing.

Table 8.1: SUBJECT SYSTEMS

Systems	Type	Size (LOC)	Evolution Period	#Revision Studied
DNSJava	DNS Protocol	20831	SEP 1998 - FEB 2013	1679
FreeCol	Game	78894	JAN 2002 - JAN 2013	5000
JabRef	Bibliography manager	13213	JUN 2002 - SEP 2013	3718
OpenYMSG	Open Messenger	8821	MAR 2007 - MAR 2013	304
Squirrel	SQL Client	332635	JUN 2001 - JAN 2013	6737

8.4.2 Experimental Steps

Preprocessing

For the proposed analysis, we first extract all the revisions of the subject systems from their corresponding SVN repository. We analyze the changes to source code only and thus we remove the comments. Based on the file modification history extracted using *SVN diff* we list added, modified and deleted files in successive revisions. This helps to exclude the unchanged files during change analysis to speed up the process.

Method Extraction and Origin Analysis

For change analysis, we extract method information from the successive revisions and store in a database. To map changes to methods throughout the development cycle, we need to map the methods across the revisions. Therefore, we carry out origin analysis (mapping methods across revisions) [97] of the methods and preserve mapping information in a database. This information is used to map the classified changes back to the corresponding methods.

Change Extraction

Code changes are extracted using *ChangeDistiller* classifier. *ChangeDistiller* extracts changes by taking differences between two versions of ASTs of the same file. The differences are represented as tree-edit operations and encoded as edit scripts. *ChangeDistiller* classify extracted changes to fine-grained change types. We have customized the *ChangeDistiller* classifier to suit for analyzing local repository exported from SVN. To extract source code changes, two successive versions of the same file are selected from the source repository and then are passed to the differencing engine of the change classifier. The process is repeated for all changed files (identified by SVN *diff*) and for all the revisions of the subject systems.

Mapping Changes to Methods

We carry out origin analysis [97] of the methods across all the revisions. Extracted and classified changes are mapped to their corresponding source code entities (methods) with the help of extracted origin mapping information for the associated entities. We preserve the extracted, classified and mapped changes into a database to measure evolutionary couplings.

Determining the Atypical Threshold

There is no standard threshold that identifies a commit to be an atypical commit or not. Atypical commits are the commits involving changes to an excessive number of entities as compared to regular non-atypical commits. To separate the atypical commits from non-atypical commits we define atypical threshold based on the classification of commits by Alali *et al.* [10]. They studied the trends and characteristics of how developers commit source code. Their study suggests that majority (75%) of the commits are fairly smaller. However, there might be larger commits that may involve even hundreds of program entities as we also observed in our analysis. Alali *et al.* [10] proposed a descriptive statistics based classification of commits as shown in Figure 8.1.

Alali *et al.* [10] classified the commits into five categories based on the number of the entities changed. A 5-point summary is used to classify the commits. Here, Q_0 is the minimum value and Q_1, Q_2, Q_3 are the

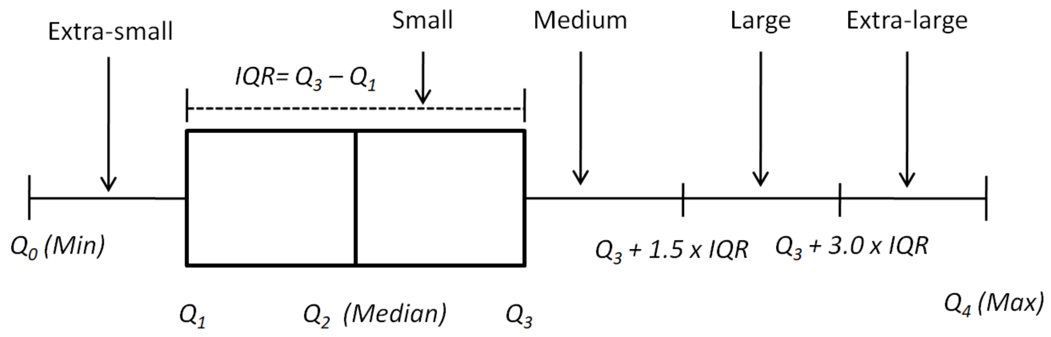


Figure 8.1: Classification of commits based on commit-size [10]

first quartile, median and the third quartile representing 25%, 50% and 75% of data respectively. $Q_3 - Q_1$ represents the Inter Quartile Range (IQR) covering the data from Q_1 to Q_3. Two other points are defined at $Q_3 + 1.5 \times IQR$ and $Q_3 + 3.0 \times IQR$ to group the mild and extreme outliers. Here, the commits within the range Q_0 to Q_1 are considered *extra-small* commits whereas the commits belonging to Q_1 to Q_3 and Q_3 to $Q_3 + 1.5 \times IQR$ are considered as *small* and *medium* respectively. Commits within the range $Q_3 + 1.5 \times IQR$ to $Q_3 + 3.0 \times IQR$ are *large* commits while the rest of the commits within $Q_3 + 3.0 \times IQR$ to Q_4 are *extra-large* commits. We consider the *extra-large* commits as atypical commits because those commits are excessively large compared to the typical or regular commits in the software repository.

Implementation of Clustering Algorithm

We use the *Apache Commons*[1] Java library for the implementation of the DBSCAN clustering algorithm. We define (Section 8.2) and implement the distance function that measures the distance between the points (methods) based on the frequency of co-change of the associated methods.

For clustering, we extract change history from the repository. We determine the number of changed methods in each revision. Based on the individual commit size we calculate the average number of changed methods per revisions. We then define the atypical threshold which determines a revision to be atypical if the number of changed methods exceeds the atypical threshold. This way we list the atypical commits for any given system.

Now, for each changed methods in an atypical commit, we list the revisions (commits) where the corresponding methods have changed. Each method along with the list of changed revisions act as a data point for clustering. We define a distance function that determines the distance between a pair of changed methods as presented in Section 8.2. We also set the maximum radius of the neighborhood of any cluster $eps = 0.5$ and minimum two data points required to form a cluster. Based on the distance function, the clustering algorithm returns the clustering result as a set of non-overlapping clusters of the list of methods in a given atypical commits. After clustering, instead of using the original transaction we use the set of transactions obtained

[1]commons.apache.org/proper/commons-math

from clustering the atypical transaction for the detection of evolutionary coupling. For the proposed approach for the detection of evolutionary coupling, if a pair of methods co-changed in an atypical commit, they will be considered as a co-changed method pair only if both methods belong to the same cluster obtained from the splitting of the associated atypical transaction. This is likely to discard a major portion of false associations induced by the atypical commits. However, for the conventional approach, we filter out atypical transactions based on a threshold to detect evolutionary coupling.

8.5 Results

In this section, we discuss the results of our study by answering the research questions we defined in Section 8.1. We evaluated our research questions based the analysis of six subject systems of diversified size, application domain and length of the evolution period. We first investigate the extent to which atypical commits affect the detection of evolutionary coupling. Then we perform a comparative evaluation of the conventional approach and our proposed approach by measuring the precision and recall in predicting future co-change candidates. The following sections represent our findings regarding our research questions:

Answer to RQ1: *To what extent atypical commits affect the detection of evolutionary coupling?*

In this research question, we investigate the extent of how atypical commits affect the detection of the evolutionary coupling. As our study investigates the evolutionary coupling at method level granularity, we empirically measure what proportions of changed methods in software systems are associated with atypical commits. To measure this, we mine change history from the source code repository to determine the number of methods changed in each commit. We then determine the threshold for a commit or revision to be considered as atypical commits as described in Section 8.4.2. Based on the threshold we identify the atypical commits and then we measure what proportion of methods are associated with atypical, non-atypical and both types of commits. This gives us important insights regarding the impacts of atypical commits on the detection of evolutionary coupling.

Table 8.2 represents the proportional distribution of methods changed exclusively in atypical and non-atypical commits and changed in both atypical and non-atypical commits. From the table, we see that around 17.96% to 33.70% with an average of 26.70% of the changed methods are changed exclusively in atypical commits. In addition, another 28.28% of methods on an average are associated with both atypical and non-atypical commits. Thus, we cannot simply exclude the atypical commits while analyzing the evolution of the software systems as more than one-fourth of the changed methods are exclusively associated with atypical commits. Similarly, atypical commits are also contributing to the change history of another 28.28% of methods changed in both atypical and non-atypical commits. This demonstrates that instead of discarding the atypical commits we need techniques that make use of the important change history of methods associated with atypical commits while mitigating the possible negative effects of atypical commits on the detection of evolutionary coupling.

Table 8.2: DISTRIBUTION OF CHANGED METHODS IN ATYPICAL AND NON-ATYPICAL COMMITS

Systems	TCM	MCA	%	MCN	%	Both	%
DNSJava	1653	505	30.55	641	38.78	507	30.67
FreeCol	3390	609	17.96	1307	38.55	1474	43.48
JabRef	2576	868	33.70	882	34.24	826	32.06
OpenYMSG	347	84	24.21	208	59.94	55	15.85
Squirrel	9253	2502	27.04	4960	53.60	1791	19.35
Average (%)	-	-	**26.70**	-	**45.02**	-	**28.28**

TCM= Total Number of Changed Methods

MCA=Number of Methods Changed in Atypical Commits

MCN=Number of Methods Changed in Non-atypical Commits

Both= Methods Changed in both Atypical and Non-atypical Commits

Summary- Our analysis shows that around 26.70% of the changed methods are exclusively associated with atypical commits whereas another 28.28% of the changed methods are associated with both atypical and non-atypical commits. Thus, *atypical commits can affect the detection of evolutionary coupling because more than half of the changed methods in the software systems are fully or partially associations with atypical commits.*

Answer to RQ2: Can we predict future co-change candidates more accurately by evolutionary coupling extracted using automatic splitting of atypical transactions?

This research question investigates how well the proposed approach can predict change candidates as compared to the conventional approach. This evaluates how the automatic splitting of atypical commits contributes the accuracy of the detection of evolutionary coupling. Given that there is no benchmark dataset to evaluate the association rules, manual investigation of the association rules even a considerable proportion of the huge rule set likely to be impractical. Thus, we make use of the change history from later revisions to evaluate the accuracy of the evolutionary coupling extracted from earlier revisions. In our approach, the evaluation of the evolutionary couplings for any commit c is done by analyzing the evolution history from 1 to $c - 1$. This technique is considered to be a variant of *n-fold cross-validation technique* and it has been used in many other existing studies [114, 189].

To evaluate the comparative accuracy in prediction of co-change candidates, we successively examine all the revisions. For each method M changed in commit c we have the following measurements:

- *True Co-change Candidates (TCC_m):* The list of all changed methods in commits c excluding method M is considered as the True Co-change Candidate (TCC_m) for the method M.

- *Predicted Co-change Candidates (PCC_m):* We determine the predicted co-change candidates for method M in commit c by examining the change history for commits 1 to $c-1$. Any method that co-changed with method M in one or more commits from 1 to $c-1$ are said to have evolutionary coupling with method M. We consider all such co-changed methods for method M as the Predicted Co-change Candidates (PCC_m) for method M for commit c.

- *Correctly Predicted Co-change Candidates ($CPCC_m$):* Now, if we see any M_p method in True Co-change Candidates (TCC_m) appearing in Predicted Co-change Candidates (PCC_m), then we consider M_p as a correctly predicted co-change candidate for method M. We represent set of all correctly predicted candidate methods as Correctly Predicted Co-change Candidates ($CPCC_m$). This is basically the intersection of TCC_m and PCC_m *i.e.,* ($TCC_m \cap PCC_m$).

Now, for each method we can measure the *precision* for method M in commit c as the percentage of correct prediction with respect to the total number of predicted co-change candidates methods for M. We measure the precision by the following equation:

$$P = \frac{|CPCC_m| \times 100}{|PCC_m|} \tag{8.4}$$

Now, for each method we can measure the *recall* for method M in commit c as the percentage of correct prediction with respect to the total number of true co-change candidates methods for M. We measure the recall by the following equation:

$$R = \frac{|CPCC_m| \times 100}{|TCC_m|} \tag{8.5}$$

We measure precision and recall for each method and for each successive revisions using Equation 8.4 and Equation 8.5. Then we take the average of the precision and recall values to have the average precision and recall for our proposed approach and the conventional approach for the detection of evolutionary coupling. We present the comparative results for precision and recall in Table 8.3.

As shown in Table 8.3, the precision values for all the subject systems for proposed approach is higher than that of the conventional approach. The '+' sign preceding the changes in precision and recall shows the cases where the proposed approach performs better than the conventional approach and the '-' sign shows the opposite. The improvement in precision for our proposed approach varies from 7.64% for JabRef to as high as 40.43% in DNSJava with an average improvement of 24.01%. These results show that our proposed clustering-based approach for splitting atypical commit can considerably improve the precision of the detection of the evolutionary coupling. In other words, our proposed approach can help in predicting co-change candidates based on evolutionary couplings with better accuracy.

Table 8.3: COMPARATIVE PRECISION AND RECALL FOR THE CONVENTIONAL AND THE PROPOSED APPROACH

	Precision			Recall		
Systems ↓	P_c	P_p	CHG_P(%)	R_c	R_p	CHG_R (%)
DNSJava	12.16	20.43	+ 40.47	79.19	80.30	+ 1.38
FreeCol	3.51	5.26	+ 33.27	70.83	73.17	+ 3.19
JabRef	6.89	7.46	+ 7.64	71.13	73.12	+ 2.72
OpenYmsg	7.87	9.85	+ 20.10	62.94	62.57	- 0.59
Squirrel	13.84	16.99	+ 18.54	59.10	59.74	+ 1.07
Average			**+ 24.01**			**+ 1.56**

P_c, P_p=Average Precision for Conventional and Proposed approach respectively

R_c, R_p= Average Precision for Conventional and Proposed approach respectively

CHG_P, CHG_R= Changes (%) in Precision and Recall for two approaches respectively

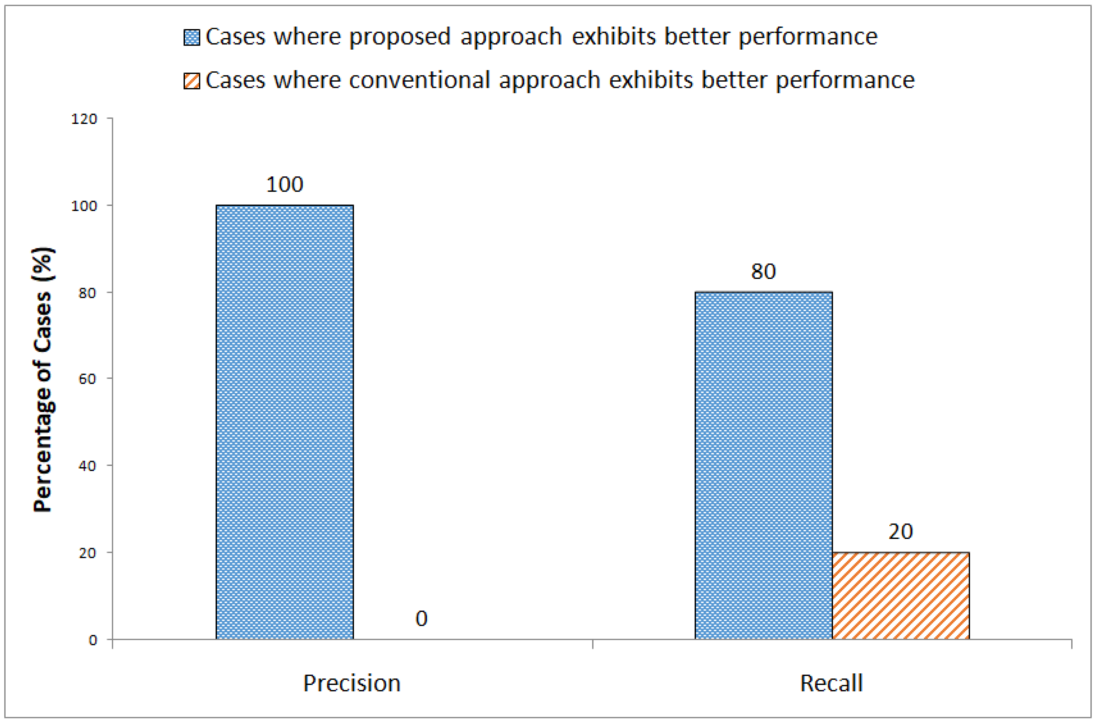


Figure 8.2: Comparative Overall Precision and Recall of Conventional and Proposed Approaches

Now, for recall we observe in Table 8.3 that our proposed approach yields better recall over the conventional approach for all the subject systems except OpenYmsg. However, our proposed approach have 1.56% overall improvement in recall in predicting co-change candidates. Our experimental results show that our proposed approach considerably improves precision in predicting co-change candidates without compromising recall. Figure 8.2 represents the comparative performance summary for both approaches in terms of precision and recall. Here, the proposed approach outperforms in terms of precision in all cases where in 80% of cases (4 out of 5 systems) the recall of proposed approach is better than that of convention approach. Thus, our proposed approach has the potential to detect evolutionary coupling more accurately by minimizing the effects of atypical commits.

Summary- Our findings show that evolutionary coupling detected by using automatic splitting of atypical commits considerably improve the precision in predicting co-change candidates with slightly improved recall compared to the conventional approach.

8.6 Threats to Validity

Performance of the clustering is very important for the results of the proposed empirical study. We chose the density-based DBSCAN clustering algorithm [42] that is suitable for the context of the study as we need to cluster the methods based on their degree of closeness (frequency of co-change) into an unknown number of clusters. We use Apache library for the clustering which is known to be a reliable and widely used implementation of the clustering algorithms.

We selected subject systems with diversified size, number of revisions, length of evolution and application domain to avoid potential biasing. Due to the limitations of existing change classifier, ChangeDistiller we consider Java systems only. The inclusion of subject systems of other languages might help in more generalization of the findings. However, our only focus on this study is to investigate whether and how clustering-based transaction splitting contributes to the improvement in detecting atypical commits.

8.7 Related Work

A large number of existing studies explored the potentials of evolutionary couplings to discover dependencies among the evolving software components. The relationships among the program entities are commonly represented as association rules proposed by Agrawal *et al.* [8]. Zimmermann *et al.* [189] applied data mining to software repositories to identify related components as change candidates. This study investigated the effectiveness of association rules at different levels of granularities (variables, methods, class, files). The proposed tool ROSE supports programmers by recommending potential change candidates associated with a particular change. This study, although for different levels granularity, mines association rules based only on co-change information.

Mondal *et al.* [112] proposed *significance*, a new measure to represent the degree of evolutionary coupling among the co-changed software entities as an improvement of the detection accuracy of the conventional evolutionary coupling. Here, the hypothesis is that the larger the co-changed method groups a method evolve through the less is the likelihood of the method being related to the method in that group. However, whether one or more groups of functionally related methods appear in a single commit might be subject to the developers choice. This study partially addresses the problem of atypical commits but does not aim to separate true associations of entities from false association in large commits. Again, this study does not propose any standard threshold on the significance to be representative of the existence of dependency relation and a limited empirical validation of the measure was presented.

Mondal *et al.* in another study [115] proposed an improvement in accuracy for detecting evolutionary coupling by measuring *change correspondence*. The key idea here is that if two co-changed methods have common identifiers changed in the corresponding changed parts of the methods, they are likely to be related. However, the similarity in identifiers in two different methods may not always reflect the dependency unless those identifiers are associated with same or similar syntactic and semantic contexts. The authors also have investigated the application of evolutionary coupling to identify and rank potential change candidates for cloned code in other studies [114, 124, 115].

Kagdi *et al.* [72], showed that integrating conceptual couplings to the evolutionary coupling can significantly improve the accuracy of detecting couplings among the evolving software artifacts. This study considers source code artifacts as documents and measures document similarity using IR technique to infer on the dependency between the co-evolving source code entities. Their study exemplifies the potentials of using information beyond the frequency of change or co-change of software entities to improve the accuracy of the conventional evolutionary coupling.

Kirinuki *et al.* [77], proposed an approach to avoid the atypical commits by early informing the developer about unrelated changes involving multiple tasks (they termed as tangled change). This technique suggests developer to split commits likely to introduce tangled changes based on the past change history. This study also shows the importance of dealing with the atypical commits. However, it do not provide solutions to existing atypical commits as our approach does. Several other studies [53, 54] also outline the importance of the effects of atypical commits. Thus, it is important to minimize the effects of atypical commits on the detection of evolutionary coupling.

In this study, we use clustering technique to split atypical commits into smaller transactions of related entities to minimize the impacts of atypical commits. This eventually results in improved accuracy of the detection of couplings among the software entities. Our empirical results show that the proposed approach considerably improve the precision of conventional evolutionary coupling with a small improvement in overall recall.

8.8 Summary

In this chapter, we present an empirical study using a clustering-based technique to split atypical transactions to minimize the effects of atypical commits on the detection of evolutionary coupling. We comparatively evaluate how automatic transaction splitting contributes to the improvement of accuracy in detecting evolutionary coupling. Our proposed approach applies DBSCAN clustering algorithm on the atypical transaction to split the atypical transactions in smaller transactions of methods those are likely to be truly coupled. We define a distance function based on the frequency of co-change of the methods in the change history. The more frequently a pair of methods co-changed in commit history, the less their distance will be. Our results from the empirical study based on the analysis of thousands of revisions of five diverse subject systems written in Java show that we can considerably improve the precision of the conventional approach for detection of evolutionary coupling by the clustering-based splitting of atypical commits. This approach quantifies the possible effects in terms of the proportion of entities affected by atypical commits. The proposed clustering-based splitting technique has the potential to complement the existing approaches for improving the detection accuracy of evolutionary coupling. The splitting approach is independent of the programming language as it considers only the frequency of the co-changed entities for clustering purpose. In the next chapter, we apply this clustering-based technique and we propose an evolutionary coupling based technique for change impact analysis of clones.

Chapter 9

A Change-type Based Approach for Detecting Evolutionary Coupling of Code Clones

In Chapter 8, we proposed a clustering-based technique to split extra-large (atypical) transactions into smaller pseudo transactions. This minimizes the effects of atypical commits on the detection of evolutionary coupling. In this chapter, we propose the concept of typed evolutionary coupling. We use syntactic change information in detecting evolutionary coupling of clones. The conventional approaches detect evolutionary coupling by mining association rules based on change histories from software repositories. Two well-known metrics, *support* and *confidence* for the association rules represent the strength of the evolutionary coupling among the associated entities. However, these metrics only consider the change frequencies of the co-changed entities. The conventional approaches count change or co-change frequencies of the software entities from the change history in software repositories and do not consider whether those changes are syntactically relevant to represent dependencies among the associated entities. This most likely affects the accuracy of the conventional approaches. Moreover, because of considering only the change frequencies, the conventional approaches may also fail to identify evolutionary coupling among the infrequently co-changed program entities. In this study, we propose *typed evolutionary coupling* that uses change history and the fine-grained syntactic change types extracted using *ChangeDistiller* to detect evolutionary coupling of clones. For each pair of co-changed clones, we mine association rules for all distinct pairs of change types to measure aggregated coupling values. We present an empirical study to comparatively evaluate the accuracy of the proposed approach. Our experimental results based on the analysis of thousands of revisions of six diverse subject systems written in Java show that our approach demonstrates improved recall over the conventional approach for the detection of evolutionary coupling for clones.

The rest of the chapter is organized as follows: Section 9.2 defines and explains key terminologies in context of our study. Section 9.3 outlines the important motivations of the study with examples. Section 9.4 briefly describes the taxonomy of changes used in this study. Section 9.5 represents the experimental settings and the steps used in the study including subject systems used, change extraction and classification procedure, and the metrics we measure. Section 9.6 represents the experimental results. Potential threats to the validity of the study are presented in section Section 9.7. Section 9.8 discusses the related works followed by the conclusion in Section 9.9.

9.1 Introduction

Software systems evolve through various changes. Due to the underlying relationships and dependencies among the components of the software systems, changes in one component may introduce changes to other components in a software system. Identification of software components related to a candidate component for change, known as Change Impact Analysis (CIA) [13], is an important task in software maintenance and evolution. Change impact analysis has been a challenging task for the software developers and the maintainers especially because of the increasing complexities in architecture and dependencies among the components in modern software systems. A significant amount of research efforts over the past few decades in CIA has produced a wide spectrum of methods and tools by using different techniques including but not limited to static analysis [92, 130] and dynamic analysis [12, 127, 128], Mining Software Repositories (MSR) [189, 72, 31] and Information Retrieval (IR) [32].

When a program entity is changed, the dependencies among the program components define both what other entities need to be changed and how the related entities should be changed to preserve both syntactic and semantic integrity of the system. Consequently, if we examine the fine-grained information on how the related entities have changed, it is likely that we will be able to comprehend the reasons why the entities have changed *i.e.,* their underlying relationships. Detection of evolutionary coupling is an important way to discover dependency relationships among the evolving software components. Evolutionary coupling does not depend on expensive source code analysis and it uses the change history. Evolutionary coupling is likely to detect dependencies among software components that are not visible to static or dynamic analysis. This feature of evolutionary coupling makes it suitable for detecting evolutionary dependencies among clones as clones may need to co-change to maintain consistencies despite having no syntactic or semantic dependencies expressing the data-flow and the control-flow of the software systems. However, the conventional approaches [189] for the detection of evolutionary coupling are based only on the co-change information (frequency) and it does not consider the way the entities have changed syntactically. Thus, conventional approaches for detection of evolutionary coupling may incorrectly report co-changed entities as related while missing important pairs of dependent but infrequently co-changed entities.

Again, from our manual analysis, we observe that majority of the co-evolving entities do not change very frequently (once or twice). However, as the conventional approaches consider only the frequencies of change, the more the entities co-change (change together) the more reliable the evolutionary coupling will be to reveal the dependency between the co-changed entities. Considering this, a number studies [30, 186] ignore infrequently co-changed entities and thus are likely to miss to identify related items because of discarding a major portion of the candidate entities. However, from our manual analysis we see that entities those have co-changed only once (support=1) may have true coupling among them. Discarding rules for lower support not only ignores the majority of the association rules but may also fail to identify important couplings among the infrequently co-changed entities.

In this study, we propose *typed evolutionary coupling* that extracts fine-grained change types using ChangeDistiller [43]. It then combines the change information with the extracted frequencies (number of revisions in which the entities were changed or co-changed) of changes as in conventional approaches for the detection of evolutionary coupling. This approach capitalizes the strengths of the conventional approach and aims to identify the dependencies among both frequently or infrequently co-changed program entities. We mine association rules using conventional approach and measure support and confidence. We extract fine-grained syntactic change types and then detect evolutionary coupling for each distinct pairs of syntactic change types. For each conventional association rule, we will have a set of typed association rules based on how many distinct types of changes the entities have evolved through. From these rules, we measure the equivalent coupling value (confidence) for each of the conventional association rules. To the best of our knowledge, our study is the first to augment fine-grained syntactic change types with the conventional change history for the detection of evolutionary coupling among software entities.

In this study, we describe the concept of *typed evolutionary coupling* and present an empirical study to evaluate whether we can improve the detection accuracy of evolutionary coupling using fine-grained syntactic change types. In particular, we present our findings by answering the following research questions:

RQ1 *Can we detect evolutionary coupling of clones more accurately using fine-grained syntactic changes compared to conventional approach?*

RQ2 *How do the comparative accuracies of typed evolutionary coupling and conventional evolutionary coupling in predicting co-change candidates differ for different clone types?*

Our experimental results based on the analysis of thousands of revisions of six diverse subject systems written in Java show that *we can considerably improve the recall of the conventional approach for detection of evolutionary coupling by augmenting fine-grained change types with the change frequencies.*

9.2 Terminology

Evolutionary Coupling- When two or more entities (e.g, files, classes, methods) in a software system co-change frequently (in many revisions) during evolution, these entities are said to have evolutionary coupling which is also known as change coupling [47]. The existence of evolutionary coupling among the software entities is an indication of dependencies among the associated entities and changes to one such entity may introduce corresponding changes to other coupled entities in the software system. As evolutionary coupling reveals the underlying relationships among software components, it helps in Change Impact Analysis (CIA) by identifying co-change candidates when a related entity is changed during evolution [189].

Association Rule- An association rule is an expression of the form $X \Rightarrow Y$ where X is called the antecedent and Y is called the consequent of the association rule. X and Y are sets of one or more software entities. In the context of our study an association rule is interpreted as *"if method X is changed in a revision, Y is likely to be changed in the same revision"*. The strength of this likelihood is expressed in terms of the *support* and *confidence* of the association rule defined below.

Support and Confidence- *Support* is defined as the number of commits (revisions) an entity or a set of entities were changed [189]. For example, let two co-changed methods $M1$ and $M2$ where the set of revisions method $M1$ was changed in is $R1 = \{3, 4, 7, 15\}$. On the other hand, set of revisions method $M2$ was changed in is $R2 = \{4, 6, 7, 15, 17\}$. Here, $support(M1) = |R1| = 4$, the cardinality of $R1$. Similarly, $support(M2) = |R2| = 5$. However, $support(M1, M2) = |R1 \cap R2| = |\{4, 7, 15\}| = 3$, the number of revisions $M1$ and $M2$ have co-changed and thus $support(M1, M2) = support(M2, M1)$. For co-changed methods $M1$ and $M2$, we can have two association rules: $M1 \Rightarrow M2$ and $M2 \Rightarrow M1$. Again, support of an association rule is defined as the number of revisions (commits) where the antecedent and consequent changed together (i.e, co-changed). For an association rule $X \Rightarrow Y$,

$$support(X \Rightarrow Y) = support(X, Y)$$

In our example, $support(M1 \Rightarrow M2) = support(M1, M2) = 3$. *Confidence* of any association rule $X \Rightarrow Y$ is defined as,

$$confidence(X \Rightarrow Y) = support(X, Y)/support(X)$$

This represents the probability that consequent Y will change given that antecedent X is changed in the same revision. Here, $confidence(M1 \Rightarrow M2) = support(M1, M2)/support(M1) = 3/4 = 0.75$ and $confidence(M2 \Rightarrow M1) = 3/5 = 0.6$.

Typed Evolutionary Coupling- In this study, we propose the measure of typed evolutionary coupling at the method level. It refers to the evolutionary coupling of two co-changed methods X and Y with respect to a pair of syntactic change types (T1, T2) where in a particular revision X has at least a change of type T1 while Y has at least a change of type T2. Typed evolutionary coupling considers not only the information that an entity has changed but also takes into account 'what' that change is. This gives better insight into the discovery of the underlying relationships among the program entities. Typed evolutionary coupling examines the probability of a coupled entity to have a particular type of change given that another entity has a given type of change in the same revision. Our hypothesis is that the incorporation of the knowledge of particular types of changes will reveal the dependency relationships among the software entities more precisely.

Typed Association Rule- For two given co-changed entities X and Y with corresponding types of changes T1 and T2 respectively, typed association rule $X_{T1} \Rightarrow Y_{T2}$ represents the typed evolutionary coupling between the antecedent X and consequent Y. In the context of our study, a typed association rule $X_{T1} \Rightarrow Y_{T2}$ is interpreted as *"if a method X has a change of T1 in a revision, Y is likely to have a change of type T2 in the same revision"*. We describe the strength of a typed association rule in terms of *typed support* and *typed confidence* defined below.

Typed Support and Confidence- *Typed support* is defined as the number of commits (revisions) an entity or a set of entities were changed with a particular change type. Given two co-changed methods $M1$ and $M2$, the set of revisions method $M1$ was changed with change type $T1$ is $R1_{T1} = \{4, 5, 8, 11\}$. On the other hand, set of revisions method $M2$ was changed with change type $T2$ is $R2_{T2} = \{4, 6, 8, 10, 12\}$. Here, $\mathit{support}(M1_{T1}) = |R1_{T1}| = 4$, the cardinality of $R1_{T1}$. Similarly, $\mathit{support}(M2_{T2}) = |R2_{T2}| = 5$. However, $\mathit{support}(M1_{T1}, M2_{T2}) = |R1_{T1} \cap R2_{T2}| = |\{4, 8\}| = 2$, the number of revisions $M1$ and $M2$ have co-changed with change types $T1$ and $T2$ respectively and thus $\mathit{support}(M1_{T1}, M2_{T2}) = \mathit{support}(M2_{T2}, M1_{T1})$. For co-changed methods $M1$ and $M2$ with respective change types $T1$ and $T2$, we can have two typed association rules: $M1_{T1} \Rightarrow M2_{T2}$ and $M2_{T2} \Rightarrow M1_{T1}$. Again, support of a typed association rule is defined as the number of revisions (commits) where the antecedent and consequent changed together (i.e, co-changed) for a given pair of change types (T1,T2). For an association rule $X_{T1} \Rightarrow Y_{T2}$,
$\mathit{support}(X_{T1} \Rightarrow Y_{T2}) = \mathit{support}(X_{T1}, Y_{T2})$
In our example, $\mathit{support}(M1_{T1} \Rightarrow M2_{T2}) = \mathit{support}(M1_{T1}, M2_{T2}) = 2$. *Confidence* of any association rule $X_{T1} \Rightarrow Y_{T2}$ is defined as,

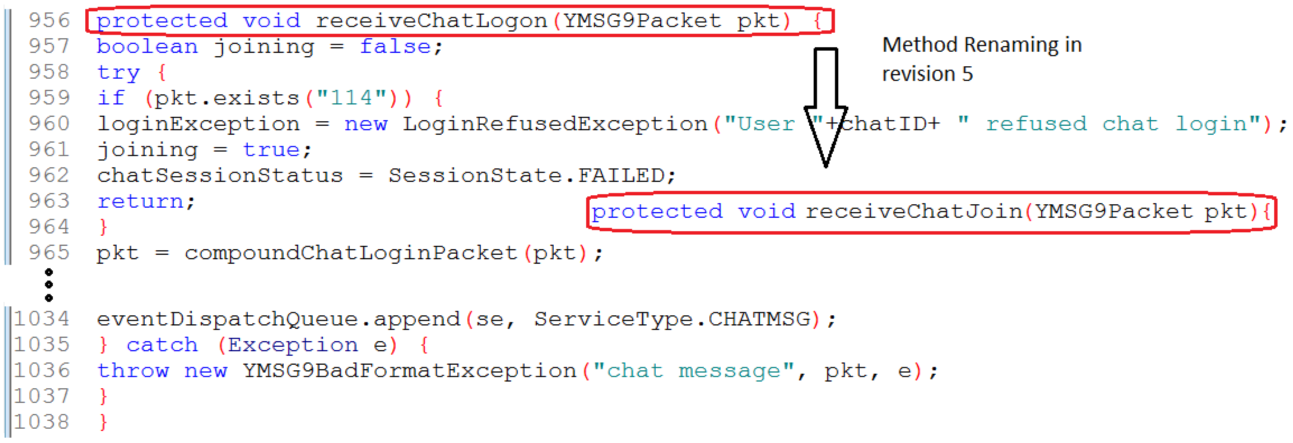
$$confidence(X_{T1} \Rightarrow Y_{T2}) = \frac{support(X_{T1}, Y_{T2})}{support(X_{T1})}$$

This represents the probability that consequent Y will have a change of type $T2$ given that antecedent X has changed with a change type $T1$ in the same revision. Here, $confidence(M1_{T1} \Rightarrow M2_{T2}) = \mathit{support}(M1_{T1}, M2_{T2})/\mathit{support}(M1) = 2/4 = 0.5$ and $confidence(M2_{T2} \Rightarrow M1_{T1}) = 2/5 = 0.4$.

Higher values of typed support and confidence indicate stronger dependency relationships between the associated entity for the given change types.

9.3 Motivation

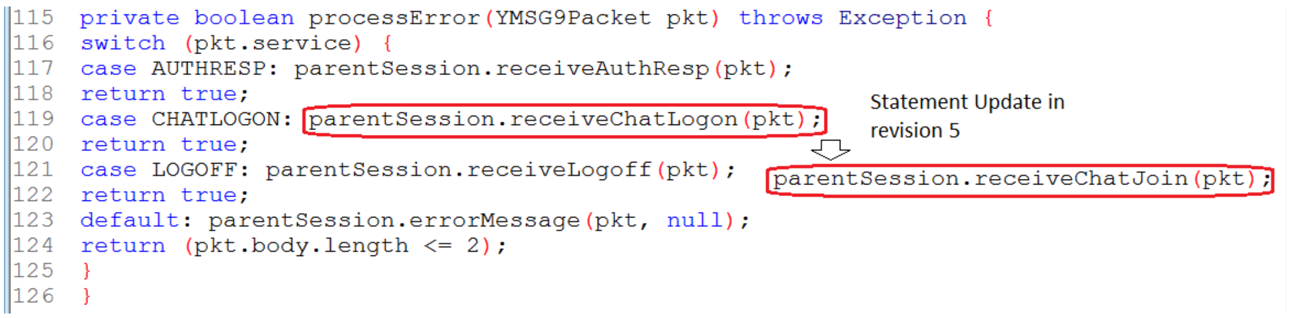
Many studies show the effectiveness of evolutionary coupling as a useful representation of dependency relationships among the software entities [189, 129, 115, 141]. Evolutionary coupling can be detected from the software change history and it does not require computationally expensive program analysis as in static and dynamic analysis (analysis of runtime information) approaches for change impact analysis. However, the accuracy of the conventional approaches for the detection of evolutionary coupling is very low [189]. This may limit the application of the evolutionary coupling in practice for dependency analysis similar to the less precise static analysis approaches [92, 130]. Conventional approaches for the detection of evolutionary coupling consider only 'which' entities have co-changed in a particular revision and they do not consider 'how' the entities have changed by considering 'what' actually those changes are. But, it is intuitive that the types of syntactic changes that any pair of entities went through during evolution can give us better insight to predict the dependencies between those entities. For example, if we observe that *"whenever a parameter is inserted into the definition of method A, a statement was updated in method B"* then it is reasonable to assume that



```
protected void receiveChatLogon(YMSG9Packet pkt) {
boolean joining = false;
try {
if (pkt.exists("114")) {
loginException = new LoginRefusedException("User "+chatID+ " refused chat login");
joining = true;
chatSessionStatus = SessionState.FAILED;
return;
}
pkt = compoundChatLoginPacket(pkt);
⋮
eventDispatchQueue.append(se, ServiceType.CHATMSG);
} catch (Exception e) {
throw new YMSG9BadFormatException("chat message", pkt, e);
}
}
```

```
protected void receiveChatJoin(YMSG9Packet pkt){
```

Figure 9.1: Method **receiveChatLogon()** renamed to **receiveChatJoin()** in **Session.java** in revision 5 of OpenYMSG



```
private boolean processError(YMSG9Packet pkt) throws Exception {
switch (pkt.service) {
case AUTHRESP: parentSession.receiveAuthResp(pkt);
return true;
case CHATLOGON: parentSession.receiveChatLogon(pkt);
return true;
case LOGOFF: parentSession.receiveLogoff(pkt);
return true;
default: parentSession.errorMessage(pkt, null);
return (pkt.body.length <= 2);
}
}
```

```
parentSession.receiveChatJoin(pkt);
```

Figure 9.2: Statement update in method **processError()** in **InputThread.java** in revision 5 to update call to renamed method **receiveChatJoin()** in OpenYMSG

method B is a caller of method A and thus methods A and B are related and they have evolutionary coupling.

Example: In file `/src/org/openymsg/network/Session.java` in OpenYMSG, we observe that the method `receiveChatLogon()` (let be X) was changed in four revisions {3, 5, 17, 44}. Another method `processError()` (let be Y) in file `/src/org/openymsg/network/InputThread.java` was changed only in revision {5}. For these co-changed method pair, the conventional approach for evolutionary coupling gives an association rule $X \Rightarrow Y$ with confidence of 1/4=0.25 which indicates a very weak evolutionary coupling. However, form our manual code analysis we see that method `processError()` (in Figure 9.2) is a caller method of co-changed method `receiveChatLogon()` (in Figure 9.1) and they are related.

Again, the confidence for the association rule $Y \Rightarrow X$ is 1/1=1.0 which indicates a very strong evolutionary coupling. From our analysis, we see that if method Y is changed method X is not necessarily required to change despite the high confidence of the association rule $Y \Rightarrow X$. Therefore, *association rules from conventional approaches for evolutionary coupling may fail to identify the true dependency relationship among the co-changed entities and thus may introduce 'false negatives' and 'false positives'.*

From our change type based analysis for the two above co-changed method, we see that method `receiveChatLogon()` was changed with four distinct change types *(Method Renaming, Statement Insert, Statement Update, Statement Delete)* in revisions 3, 5, 17 and 44. Method `processError()`, on the other hand, was changed with three distinct change types *(Statement Insert, Statement Update, Statement Delete)* in revision 5. These two methods were co-changed only in revision 5 where method `receiveChatLogon()` was renamed to `receiveChatJoin()` and method `processError()` had to update the statement containing a call to `receiveChatLogon()`.

Here, we get 4x3=12 typed association rules of the form $X_{tx} \Rightarrow Y_{ty}$ and another 12 rules of the form $Y_{ty} \Rightarrow X_{tx}$ where tx and ty are change types for methods X and Y respectively. In this case, we are mostly interested in the typed association rule $X_{tx} \Rightarrow Y_{ty}$ where tx=*Method Renaming* and ty=*Statement Update*. This is because ordered pair (*Method Renaming, Statement Update*) is a dependency sensitive change type pair (illustrated in Section 9.5.3) meaning that method renaming in method X and statement update in a co-changed method Y likely to indicate an existence of dependency relation between methods X and Y. We observe that the confidence for the typed association rule $X_{MethodRenaming} \Rightarrow Y_{StatementUpdate}$ is 1.0 which indicates a very strong evolutionary coupling between X and Y which is just opposite to the conventional evolutionary coupling. Again, the typed association rules $Y_{StatementUpdate} \Rightarrow X_{MethodRenaming}$ may not be a true representative of the dependency relation because the pair of change types (*Statement Update, Method Renaming*) is not dependency sensitive change pair. This example shows that typed evolutionary coupling is capable of identifying dependency relation between co-evolving components where the conventional evolutionary coupling may fail to discover the dependency relation precisely. Based on the change types and clone relationships between a pair of methods, we calculate the typed equivalent confidence of the conventional association rules (details in Section 9.5.4) for the co-changed methods to identify and evaluate the evolutionary coupling.

Our example shows that conventional approach for evolutionary coupling may fail to identify dependencies correctly as they consider only the fact of co-change no matter what are those changes. Typed evolutionary coupling, on the other hand, considers the types of changes to predict the reasons behind the changes in addition to the fact that the concerned methods have co-changed. This is likely to contribute to the improvement in accuracy of the detection of evolutionary coupling.

9.4 Taxonomy of Software Change at Method Level

Software systems evolve through different kinds of changes. Each type of change refers to a particular syntactic context of the change. Again, each change type affects the software systems from different functional and structural contexts. For our study, we consider the taxonomy of change proposed by Fluri and Gall [43]. As our analysis of evolutionary coupling is at method level granularity, we consider only the method-level changes in the taxonomy proposed by Fluri and Gall.

Changes to individual methods are referred to as method-level change. Changes to methods are further divided into two groups, method declaration and method body, based on where the changes occur in the methods. Changes to method declaration part (*method signature*) include changes in accessibility, overridability, method renaming, parameter change and changes to return type. Changes to *method body*, on the other hand, include changes to statements and structure statements (*e.g.,* loop, branching). Each of these fine-grained change types is assigned a level of *significance* based on their likelihood of affecting other code entities and the extent they modify the functionality of the system [43]. This study uses the same taxonomy of method level changes as described in previous chapters.

Table 9.1: SUBJECT SYSTEMS

Systems	Type	Size (LOC)	Evolution Period	#Revision Studied
Ant-Contrib	Web Server	79434	JUL 2006 - MAR 2009	177
Carol	Driver Application	13213	JUN 2002 - SEP 2013	2237
DNSJava	DNS Protocol	20831	SEP 1998 - FEB 2013	1679
JabRef	Bibliography manager	13213	JUN 2002 - SEP 2013	3718
OpenYMSG	Open Messenger	8821	MAR 2007 - MAR 2013	304
Squirrel	SQL Client	332635	JUN 2001 - JAN 2013	6737

9.5 Experimental Setup

This section outlines the experimental settings for different components and steps of our empirical study. The steps include the preprocessing of the subject systems, change extraction and classification procedure, the detection of both conventional and typed evolutionary couplings, and the comparison of the precision and recall.

9.5.1 Subject Systems

This study is based on six open source systems implemented in Java with diversified size, evolution history and application domain. Table 9.1 briefly represents the features the software systems including the application domain, length of evolution history, size in lines of code (LOC) and the total number of revisions. The size of the systems represents the lines of code (LOC) in the last revision of the systems counted after removal of comments and pretty printing.

Table 9.2: NiCad SETTINGS FOR THE STUDY

Parameters	Values
Minimum Size	5 lines
Maximum Size	500 lines
Granularity	Method
Threshold	0% (Type 1, Type 2), 30% (Type 3)
Identifier Renaming	blindrename (Type 2, Type 3)

9.5.2 Experimental Steps

Preprocessing of Code

For our study, we apply some preprocessing steps on the source code. First, we extract all the revisions of the subject systems from their corresponding *SVN*[1] repositories. Our analysis focus on the changes to source code only. Thus, we remove comments from the source files. Pretty-printing of the source files are then carried out to eliminate the formatting differences using the tool *ArtisticStyle.*[2] We extract the file modification history using *SVN diff* command to list added, modified and deleted files in successive revisions. This information is used to exclude the unchanged files during change analysis to speed up the process.

Method Extraction and Origin Analysis

For our analysis, we extract method information from the successive revisions of the source code. We store the method information (file path, package name, class name, signature, start line, end line) in a database to use for mapping changes to the corresponding methods. Again, SVN keeps track of the files that are added, deleted or modified and the history of changes to individual file content are preserved as the modification of lines. This line-level change information is not sufficient to describe the evolution of source code entities at higher granularity levels such as classes or methods. As a result, to map changes to methods throughout the development cycle, we need to map the methods across the revisions. Therefore, we carry out origin analysis of the methods on the revisions of the systems. We used the same approach for origin analysis as presented in the study by Lazano and Wermelinger [97]. Here, if a method is relocated in the same file or if the method signature is changed, the method is mapped by string comparison with the candidate methods in the new file using *Strike A Match* algorithm [4]. The origin mapping information is used to map the extracted changes and cloning information back to the corresponding methods.

[1]https:/sourceforge.net/
[2]http://astyle.sourceforge.net/

Change Extraction

We used change extraction and classification core of *ChangeDistiller* [43]. The details of change extraction and classification are as follows: We have customized the *ChangeDistiller* classifier to analyze local repository exported from SVN. To extract source code changes, two successive versions of the same file are selected from the source repository and then are passed to the differencing engine of the change classifier. The extracted changes are then passed to the classifier for classification. The process is repeated for all changed files (identified by SVN *diff*) and for all the revisions of the subject systems. Changes extracted from two successive revisions of source code files are classified into fined-grained changes to source code entities.

Code Clone Detection

For this study, we detect clones using the hybrid clone detection tool NiCad [144]. NiCad is reported to have a higher level of precision and recall [145] and supports the detection of both exact (Type 1) and near-miss (Type 2 and Type 3) clones. We run NiCad on all revisions of the subject systems to detect clones at method (function) level granularity. The clone detection results are then processed to store the clone information in the database. Table 9.2 lists the parameter settings for NiCad used for this study.

Mapping Change Data

After classification, the classified changes are mapped to their corresponding source code entities (methods) with the help of extracted origin mapping information for the associated entities. We preserve the extracted, classified and mapped changes into a database to measure evolutionary couplings.

9.5.3 Dependency Sensitive Change Type Pairs

We define *dependency sensitive* change type pair as an ordered pair of syntactic change types (t_x, t_y). For any co-changed method pair (X, Y), if there is a change of type t_x in X and a change of type t_y in method Y in the same revision, we call the change type pair (t_x, t_y) dependency sensitive when (t_x, t_y) is likely to indicate an existence of dependencies. Here, $(t_x, t_y) \neq (t_y, t_x)$. We represent the corresponding typed association rule as $X_{tx} \Rightarrow Y_{ty}$. Based on our taxonomy of method level changes, we identified ten (10) pairs of change types as *dependency sensitive*. We further illustrate and exemplify the above definition as follows:

Method Renaming(MR) and Statement Update (SU)

If a method X is renamed in a particular revision and another method Y requires a change of type 'Statement Update' in the same revision then we assume that Y might be a caller of method X. If the corresponding typed changes are frequent for the associated methods (X and Y) for revisions where X and Y are co-changed then this is an indication that methods X and Y are dependent. We mine the following association rule for

Table 9.3: DEPENDENCY SENSITIVE CHANGE TYPES AND CORRESPONDING TYPED ASSOCIATION RULES

Type	Antecedent Change Types	Consequent Change Types	Typed Association Rules
Call Based Dependency	Method Renaming (MR)	Statement Update (SU)	$X_{MR} \Rightarrow Y_{SU}$
	Parameter Insert (PI)	Statement Update (SU)	$X_{PI} \Rightarrow Y_{SU}$
	Parameter Delete (PD)	Statement Update (SU)	$X_{PD} \Rightarrow Y_{SU}$
	Parameter Ordering (PO)	Statement Update (SU)	$X_{PO} \Rightarrow Y_{SU}$
	Parameter Type Change (PTC)	Statement Update (SU)	$X_{PTC} \Rightarrow Y_{SU}$
	Return Type Insert (RTI)	Statement Update (SU) or Statement Insert (SI)	$X_{RTI} \Rightarrow Y_{SU}$ or $X_{RTI} \Rightarrow Y_{SI}$
	Return Type Delete (RTD)	Statement Update (SU) or Statement Delete (SD)	$X_{RTD} \Rightarrow Y_{SU}$ or $X_{RTD} \Rightarrow Y_{SD}$
	Return Data Type Change (RDC)	Statement Update (SU)	$X_{RDC} \Rightarrow Y_{SU}$
	Accessibility Decrease (MAD)	Statement Update(SU) or Statement Delete (SD)	$X_{MAD} \Rightarrow Y_{SU}$ or $X_{MAD} \Rightarrow Y_{SD}$
	Add Method Overridability *final* (AMO)	Statement Update (SU) or Statement Delete (SD)	$X_{AMO} \Rightarrow Y_{SU}$ or $X_{AMO} \Rightarrow Y_{SD}$
Others*	Any method-level change type T_x	Any method-level change type T_y	$X_{T_x} \Rightarrow Y_{T_y}$, where $T_x = T_y$

*=Dependencies due to object-oriented hierarchy, shared global entities, duplicate (cloned) code.

any co-changed method pairs (X,Y) with respect to ordered change pair (MR, SU) to measure the strength of the dependency between X and Y :

$$X_{MR} \Rightarrow Y_{SU}$$

Parameter Insert (PI) and Statement Update (SU)

If a new parameter is inserted in method X, any method Y calling X needs to update the statement(s) where X is called inside method Y. Thus we assume that typed association rule for a co-changed method pair (X,Y) where X has a *Parameter Insert* and Y has a *Statement Update* change is likely to represent the dependency relation between method X and Y. We denote this association rule as,

$$X_{PI} \Rightarrow Y_{SU}$$

Parameter Delete (PD) and Statement Update (SU)

If a parameter is deleted in method X, any method Y calling X needs to update the statement(s) where X is called inside method Y. Thus we assume that typed association rule for a co-changed method pair (X,Y) where X has a *Parameter Delete* and Y has a *Statement Update* is likely to represent the dependency relation between method X and Y. The confidence of the association rule indicates the strength of such dependency relation. We denote this association rule as,

$$X_{PD} \Rightarrow Y_{SU}$$

Parameter Ordering (PO) and Statement Update (SU)

If there is a change in the order of parameters in method X definition, any method Y calling X needs to update the statement(s) where X is called inside method Y. Thus we assume that typed association rule for a co-changed method pair (X,Y) where X has a *Parameter Ordering* change and Y has a *Statement Update* is likely to represent the dependency relation between method X and Y. The confidence of the association rule indicates the strength of such dependency relation. We denote this association rule as,

$$X_{PO} \Rightarrow Y_{SU}$$

Parameter Type Change (PTC) and Statement Update (SU)

When the data type of a parameter in method X is changed, any method Y calling method X needs to update the statement(s) either where X is called inside method Y to update the data type by type casting or in the declaration of the actual parameter or in the statements assigning value to the actual parameter. Thus we assume that typed association rule for a co-changed method pair (X,Y) where X has a *Parameter Type Change* and Y has a *Statement Update* is likely to represent the dependency relation between method X and Y. The confidence of the association rule indicates the strength of such dependency relation. We denote this association rule as,

$$X_{PTC} \Rightarrow Y_{SU}$$

Return Type Insert (RTI) and Statement Update (SU) or Statement Insert (SI)

If return type is inserted into the definition of a method X which was previously not returning any value (void), any method Y calling method X might need to update statement(s) to assign the return value of X or insert statement(s) to make use of the return value of the method X. For such cases we assume that typed association rules for a co-changed method pair (X, Y) where X has a *Return Type Insert* and Y has either a *Statement Update* or a *Statement Insert* is likely to represent the dependency relation between method X and Y. We denote these association rules as,

$X_{RTI} \Rightarrow Y_{SU}$ and $X_{RTI} \Rightarrow Y_{SI}$

Return Type Delete (RTD) and Statement Update (SU) or Statement Delete(SD)

If in the definition of method X, return type is deleted, any method Y calling method X might need to update statement(s) that were receiving the return value of X or delete statement(s) making use of the return value of the method X. For such cases we assume that typed association rules for a co-changed method pair (X,Y) where X has a *Return Type Delete* and Y has either a *Statement Update* or a *Statement Delete* is likely to represent the dependency relation between method X and Y. We denote these association rules as,

$X_{RTD} \Rightarrow Y_{SU}$ and $X_{RTD} \Rightarrow Y_{SD}$

Return Data Type Change (RDC) and Statement Update (SU)

If for any method X the return type is changed a different data type, any method Y calling method X might need to update the statement(s) associated with calling method X. For such cases our assumption is that typed association rules for a co-changed method pair (X,Y) where X has a *Return Data Type Change* and Y has a *Statement Update* is likely to represent the dependency relation between method X and Y. We denote this association rule as,

$$X_{RDC} \Rightarrow Y_{SU}$$

Method Accessibility Decrease (MAD) and Statement Update (SU) or Statement Delete (SD)

If the accessibility of a method X is decreased (e.g., public to private) any method Y calling X may loose access to the called method X and thus method Y need either to delete the statement(s) calling X or to update the statement(s) to modify the way X is accessed (e.g, through objects). Here we assume that typed association rules for a co-changed method pair (X,Y) where X has a *Method Accessibility Decrease* and Y has a *Statement Update* or *Statement Delete* is likely to represent the dependency relation between method X and Y. We denote these association rules as,

$X_{MAD} \Rightarrow Y_{SU}$ and $X_{MAD} \Rightarrow Y_{SD}$

Add Method Overridability(AMO) and Statement Update (SU) or Statement Delete(SD)

$X_{AMO} => Y_{SU}$ and $X_{AMO} => Y_{SD}$ If the overridability is restricted by adding *final* keyword, any method calling a overridden method either update the statement containing a call to overridden method or If the overridability of a method method X is restricted by adding *final* keyword, any method Y calling a method overriding X may need either to delete the statement(s) calling overridden method of X or to update the statement(s) to modify the statement accordingly. Here we assume that typed association rules for a co-changed method pair (X,Y) where X has a *Add Method Overridability* and Y has a *Statement Update* or *Statement Delete* is likely to represent the dependency relation between method X and Y. We denote these association rules as,

$$X_{AMO} \Rightarrow Y_{SU} \quad \text{and} \quad X_{AMO} \Rightarrow Y_{SD}$$

However, in addition to the call based relationships, methods may have dependencies due to other reasons such as sharing of global data or calling common methods. Again, as our aim is to detect evolutionary coupling of clones, the above change type pairs aim for identifying dependencies between clones from different clone class. For clones from the same clone class, the co-change is likely to be influenced by the requirement for the consistent evolution of clones. Thus, methods those are clone to each other are likely to evolve through similar types of changes. Thus, we use typed association rules of same type change pairs (*i.e.,* $t_x = t_y$) to detect evolutionary couplings among such methods. Here, we measure the extent to which associated methods evolve consistently through same types of changes.

9.5.4 Calculation of Typed Equivalent Confidence for Conventional Association Rules

We calculate and assign a typed equivalent confidence for each conventional association rule based on our analysis on typed evolutionary coupling for the associated cloned method pair. The value of the typed equivalent confidence determines the strength of dependency between methods X and Y regarding typed evolutionary coupling. For a co-changed cloned method pair X and Y with the sets of distinct change types T_X and T_Y respectively, we measure the equivalent confidence for the association rule $X \Rightarrow Y$ considering the following cases:

Case 1: Pair of clone fragments from same clone class: For a co-changed cloned method pair X and Y from same clone class, it is expected that both of the clone fragments should change consistently. Thus we measure the typed equivalent confidence of the association rule $X \Rightarrow Y$ as the average of all confidence from typed association rules with same change types for both of the cloned methods. We express typed equivalent confidence of the association rule $X \Rightarrow Y$ by the following equation:

$$\begin{aligned} Confidence_{TE}(X \Rightarrow Y) = Average(Confidence(X_{t_x} \Rightarrow Y_{ty})) \\ \mid \ \forall(t_x, t_y), t_x \in T_X, t_y \in T_Y, t_x = t_y \end{aligned}$$

Here, T_X and T_Y are the sets of distinct types of changes for method X and Y respectively and any change type pair (t_x, t_y) is such that $t_x = t_y$. In case of cloned method pair without any same types of changes in the history, we take average of the confidence values of the associated typed rules.

Case 2: Pair of clone fragments from different clone class: For a co-changed cloned method pair from different clone class, we consider two possible scenarios. First, if there exists dependency sensitive pair of change types in co-change history for the cloned method pair, it indicates a possible syntactic dependency. So, we measure the typed equivalent confidence as the average confidence from all the typed association rules with dependency sensitive pair of change types. We express it as,

$$Confidence_{TE}(X \Rightarrow Y) = Average(Confidence(X_{t_x} \Rightarrow Y_{ty})) \\ \mid \forall (t_x, t_y), t_x \in T_X, t_y \in T_Y, isDS(t_x, t_y) = true$$

Here, any change type pair (t_x, t_y) is a dependency sensitive change pair (*i.e., $isDS(t_x, t_y) = true$*). The function $isDS(t_x, t_y)$ returns *true* if the change type pair is dependency sensitive, and *false* otherwise.

Secondly, if there is no dependency sensitive pair of change types in co-change history, we can assume that co-change of a pair of cloned methods from different clone class might be because of the existence of common code context (*e.g.,* due to sharing of global data, call to common methods). In such scenarios, we measure the typed equivalent confidence in the same way for a pair of clones from same clone class (Case 1).

9.5.5 Measurement of Accuracy

Given that there is no benchmark dataset to evaluate the association rules, manual investigation of the association rules even a considerable proportion of the huge rule set likely to be impractical. Thus, we make use of the change history from later revisions to evaluate the accuracy of the evolutionary coupling extracted from the earlier revisions as used in Chapter 8. In our approach, to the evolutionary couplings for any commit c is done by analyzing the evolution history from 1 to $c - 1$. This technique is considered to be a variant of *n-fold cross validation technique* and it has been used in many other existing studies [114, 189].

To evaluate the comparative accuracy in prediction of co-change candidates for clones, we successively examine all the revisions. For each cloned method M changed in commit c we have the following measurements:

- *True Co-change Candidates (TCC_m):* The list of all cloned methods changed in commits c excluding cloned method M is considered as the True Co-change Candidate (TCC_m) for the method M.

- *Predicted Co-change Candidates (PCC_m):* We determine the predicted co-change candidates for cloned method M in commit c by examining the change history for commits 1 to $c-1$. Any cloned method that co-changed with method M in one or more commits from 1 to $c-1$ are said to have evolutionary coupling with method M. We consider all such cloned methods co-changed with M as the Predicted Co-change Candidates (PCC_m) for cloned method M for commit c. The selection of co-changed methods is based on a threshold value for the support and confidence values and association rules extracted by mining change history from revision 1 to $c-1$.

- *Correctly Predicted Co-change Candidates ($CPCC_m$):* Now, if we see any cloned method M_p in True Co-change Candidates (TCC_m) appearing in Predicted Co-change Candidates (PCC_m), then we consider M_p as a correctly predicted co-change candidate for cloned method M. We represent the set of all correctly predicted candidate methods as Correctly Predicted Co-change Candidates ($CPCC_m$). This is basically the intersection of TCC_m and PCC_m *i.e.,* ($TCC_m \cap PCC_m$).

Now, for each method we can measure the *precision* for method M in commit c as the percentage of correct prediction with respect to the total number of predicted co-change candidates for method M. We measure the precision by the following equation:

$$P = \frac{|CPCC_m| \times 100}{|PCC_m|} \tag{9.1}$$

Now, for each cloned method we can measure the *recall* for method M in commit c as the percentage of correct prediction with respect to the total number of true co-change candidates methods for M. We measure the recall by the following equation:

$$R = \frac{|CPCC_m| \times 100}{|TCC_m|} \tag{9.2}$$

We measure precision and recall for each method and for each successive revisions using Equation 9.1 and Equation 9.2. Then we take the average of the precision and recall values to have the average precision and recall for our proposed approach and the conventional approach for the detection of evolutionary coupling of clones.

9.6 Results

In this section, we discuss the results of our study by answering the research questions we defined in Section 9.1. We evaluated our research questions based on our analysis of six subject systems of diversified size, application domain and length of the evolution period. We then perform a comparative evaluation of conventional approach and our proposed approach by measuring average precision and average recall.

9.6.1 Answer to RQ1: Can we detect evolutionary coupling of clones more accurately using fine-grained syntactic changes compared to conventional approach?

In this research question, we compare the comparative accuracies of conventional and typed evolutionary coupling in terms of average precision and recall. Here, we measure the average precision and recall discussed in Section 9.5.5. The comparative precision and recall values for both conventional and typed evolutionary coupling regarding all the subject systems are represented in Table 9.4. For this research question, we do not differentiate among the clone types and consider all types of clones.

Table 9.4: COMPARATIVE AVERAGE PRECISION AND RECALL FOR CONVENTIONAL AND TYPED EVOLUTIONARY COUPLINGS

Systems	Average Precision (%)		Recall (%)	
	Conventional (P_{CC})	Typed (P_{TC})	Conventional (R_{CC})	Typed (R_{TC})
Ant-Contrib	74.10	62.50	80.92	85.47
Carol	49.85	57.72	73.48	77.23
DNSJava	41.21	34.45	70.94	81.89
JabRef	28.40	20.15	51.54	58.72
OpenYMSG	36.26	25.13	54.16	66.15
Squirrel	34.31	36.47	56.29	62.99

As shown in Table 9.4, the average precision for conventional evolutionary coupling is higher than that of the typed evolutionary coupling in most cases. For the subject systems Carol and Squirrel, the average precision for conventional evolutionary coupling is higher than that of typed evolutionary coupling (Figure 9.3). However, for other subject systems the precision for conventional evolutionary coupling is higher than that of typed evolutionary coupling. From Mann-Whitney Wilcoxon (MWW) [5] test we see that the difference between average precision for conventional and typed evolutionary coupling are not statistically significant (p-value 0.68916, at $p<0.05$, two-tailed test). However, for the comparative values of the precisions for subject systems we observe that conventional evolutionary coupling tend to have a higher precision compared to typed evolutionary coupling.

Again, in Table 9.4 and Figure 9.4, we observe that average recall values for typed evolutionary coupling is higher than that of the conventional approach for all the subject systems. This shows that proposed typed evolutionary coupling is capable of detecting couplings which the conventional approach may fail to detect. Recall is very important in detecting co-change candidates for clones because missing any related clones to update consistently with the other related entities may introduce bugs in the system. Although we do not observe statistically significant differences (p-value 0.23014, at $p<0.05$, two-tailed test), we observe that typed evolutionary coupling tend to have higher recall compared to conventional coupling.

The results show the potentials and effectiveness of typed evolutionary coupling as an approach to detect co-change candidates for clones. Thus, our experimental results suggest that typed evolutionary coupling outperforms conventional evolutionary coupling in terms of recall in identifying co-change candidates for clones. As recall is very important while identifying coupled components as change candidates, the improvement in recall by our proposed approach shows the potentials of augmenting fine-grained change information for detecting co-change candidates for clones.

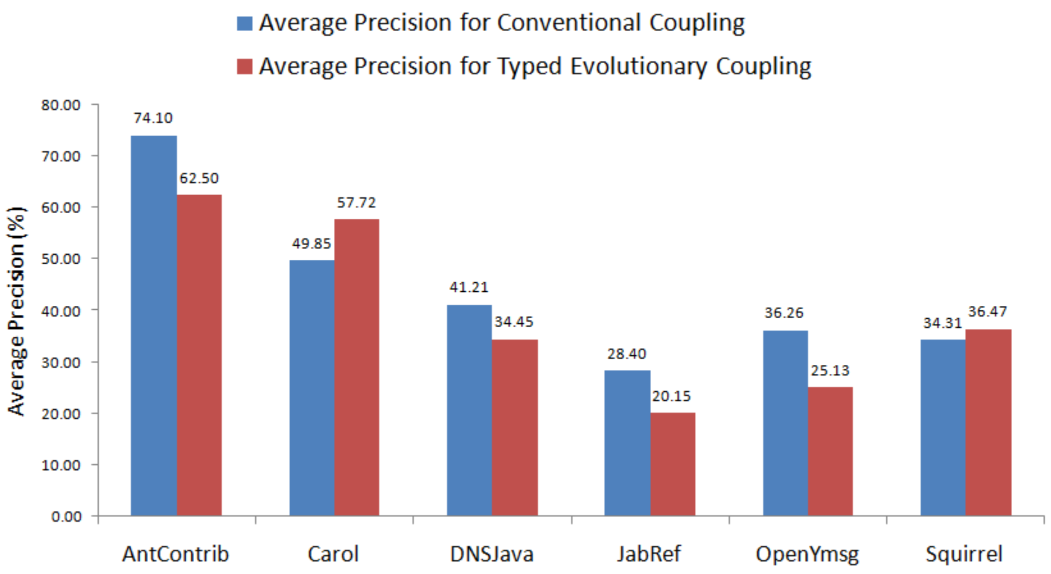


Figure 9.3: Comparative Average Precision (%) of conventional EC and Typed EC

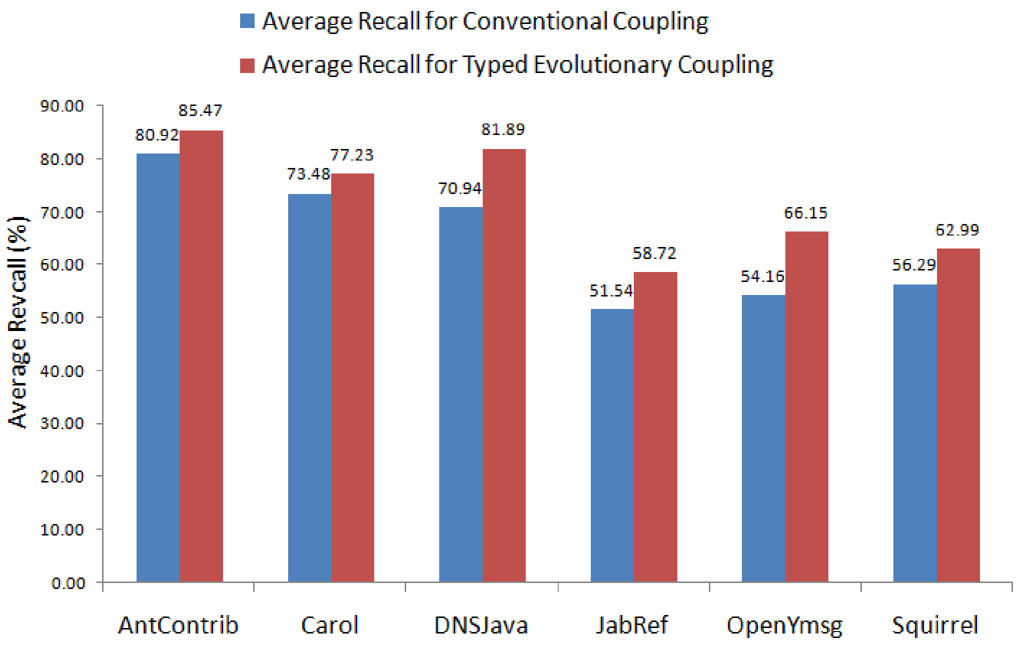


Figure 9.4: Comparative Average Recall (%) of Conventional EC and Typed EC

Table 9.5: AVERAGE PRECISION (%) FOR DIFFERENT CLONE TYPES FOR CONVENTIONAL AND TYPED EVOLUTIONARY COUPLING

Clone Types→	Type 1		Type 2		Type 3	
Systems ↓	P_{CC}	P_{TC}	P_{CC}	P_{TC}	P_{CC}	P_{TC}
Ant-contrib	80.76	63.12	100	100	68.75	70
Carol	100	100	89.21	93.93	50.77	56.73
DNSJava	41.44	36.05	74.44	75.0	51.88	56.21
JabRef	30.62	20.93	75.83	82.5	41.61	32.02
OpenYMSG	66.66	66.66	100	100	63.88	32.94
Squirrel	42.38	45.39	54.77	75.74	36.08	38.05
Average	60.31	55.36	82.37	87.86	52.16	47.65

P_{CC}=Average Precision (%) for conventional evolutionary coupling

P_{TC}=Average Precision (%) for typed evolutionary coupling

Summary- Our findings show that typed evolutionary coupling by incorporating actual syntactic change information can increase the recall over conventional evolutionary coupling.

9.6.2 Answer to RQ2: How do the comparative accuracies of typed evolutionary coupling and conventional evolutionary coupling in predicting co-change candidates differ for different clone types?

In this research question, we examine how the accuracy values for conventional and typed evolutionary coupling are different for different types of clones. Here, one of the primary objectives is to empirically evaluate whether the accuracy (precision and recall) for the conventional and typed evolutionary coupling are consistent for different types of clones. This will give us important insights into the generalization of the potential applicability of the proposed approach to detect evolutionary coupling for different types of clones. For this research question, we consider each type of clones separately to mine association rules for cloned methods in successive revisions. The comparative precision and recall for individual clone types are shown in Table 9.5 and Table 9.6 respectively.

In Table 9.5, we observe that for Type 1 clones the comparative average precision for conventional coupling is greater than or equal to that of typed evolutionary coupling except for Squirrel. This shows that the average precision for conventional coupling tends to be higher than that of typed evolutionary coupling for Type 1 clones. However, for Type 2 clones, typed evolutionary coupling exhibits higher or equal precision in predicting

Table 9.6: AVERAGE RECALL (%) FOR DIFFERENT CLONE TYPES FOR CONVENTIONAL AND TYPED EVOLUTIONARY COUPLING

Clone Types→	Type 1		Type 2		Type 3	
Systems ↓	R_{CC}	R_{TC}	R_{CC}	R_{TC}	R_{CC}	R_{TC}
Ant-contrib	86.25	87.5	100	100	74.83	80.33
Carol	100	100	90.38	100	73.44	74.40
DNSJava	71.94	78.64	95.55	91.66	69.83	87.58
JabRef	51.55	59.24	81.83	87.66	55.04	59.96
OpenYMSG	100	100	75.0	100.0	66.66	67.04
Squirrel	64.98	68.06	73.00	75.97	57.69	64.21
Average	79.12	82.24	85.96	92.54	66.25	72.25

R_{CC}=Average Recall (%) for conventional evolutionary coupling

R_{TC}=Average Recall (%) for typed evolutionary coupling

co-change candidates compared to the conventional evolutionary coupling for all six subject systems. Similarly, for Type 3 clones, in four out of six systems typed evolutionary coupling gives better precision compared to conventional evolutionary coupling. The average precision for different types of clones is shown in Figure 9.5. From Mann-Whitney Wilcoxon (MWW) [5] test, the p-values for comparative precision values for Type 1, Type 2 and Type 3 clones are 0.8103, 0.6891, and 0.6891 (at $p<0.05$, two-tailed test) respectively. So, the difference between average precision for conventional and typed evolutionary coupling are not significantly different. However, we observe that typed evolutionary coupling tends to have higher precision values for Type 2 and Type 3 clones and comparable precision for Type 3 clones.

Table 9.6 shows the comparative average recall for conventional and typed evolutionary coupling for Type

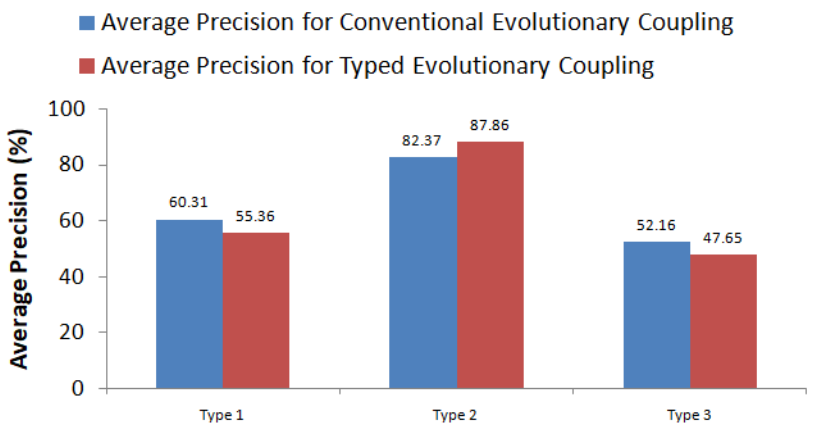
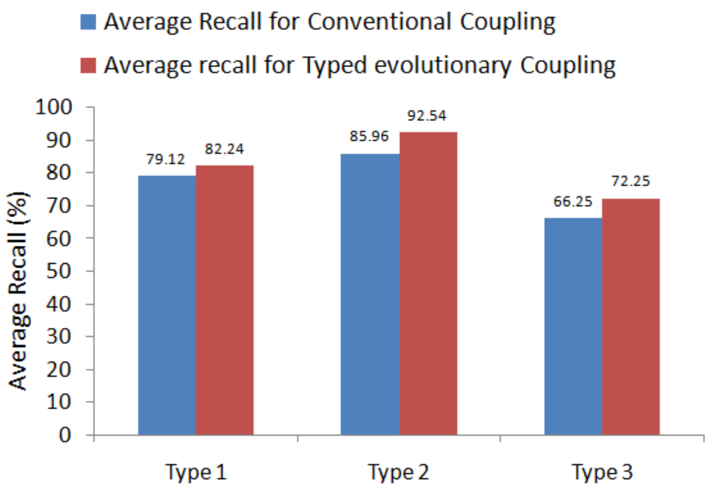


Figure 9.5: Comparative Precision of conventional EC and typed EC for different clone Types

Figure 9.6: Comparative Recall of conventional EC and typed EC for different clone Types



1, Type 2, and Type 3 clones. For Type 1 clones, the typed evolutionary coupling has a higher recall for all six subject systems compared to conventional evolutionary coupling. For Type 2 clones, we observe the similar trend except for DNSJava where conventional coupling has a slightly higher average recall. For Type 3 clones, typed evolutionary coupling has higher average recall compared to the conventional coupling for all the subject systems. From Mann-Whitney Wilcoxon (MWW) [5] test, the p-values for comparative recall values for Type 1, Type 2 and Type 3 clones are 0.8103, 0.2627, and 0.3788 (at $p<0.05$, two-tailed test) respectively. So, we do not observe significant differences between average recall for conventional and typed evolutionary coupling. Figure 9.6 shows the comparative average recall for all types of clones where the average recall for typed evolutionary coupling is higher than that of conventional evolutionary coupling in all cases. This is an indication that typed evolutionary coupling tends to exhibit higher average recall in predicting co-change candidates for clones. Thus, our results show that typed evolutionary coupling has the potential to better predict the co-change candidates for clones which conventional coupling may not be able to identify.

Summary- Our results from the clone-type centric analysis show that the typed evolutionary coupling tends to have a higher recall than conventional coupling with comparable precision.

9.7 Threats to Validity

The change analysis in this study is based on the change extraction and classification engine of the *ChangeDistiller*. The validity of the outcomes of the study might be dependent upon the accuracy of the core classifier used. However, *ChangeDistiller* is a well-known change classifier based on a comprehensive taxonomy of change types and also reported to have good performance [43].

We selected subject systems with diversified size, number of revisions, length of evolution and application domain to avoid potential biasing. Due to the limitations of existing change classifier we consider Java systems only. The inclusion of subject systems of other languages might help in more generalization of the findings.

9.8 Related Work

A large number of existing studies explored the potentials of evolutionary couplings to discover dependencies among the evolving software components. The relationships among the program entities are commonly represented as association rules proposed by Agarwal *et al.* [8]

Zimmermann *et al.* [189] applied data mining to software repositories to identify related components as change candidates. This study investigated the effectiveness of association rules at different levels of granularities (variables, methods, class, files). The proposed tool ROSE supports programmers by recommending potential change candidates associated with a particular change. This study, although for different levels granularity, mines association rules based only on co-change information.

Mondal *et al.* [112] proposed *significance*, a new measure to represent the degree of evolutionary coupling among the co-changed software entities as an improvement of the detection accuracy of the conventional evolutionary coupling. Here, the hypothesis is that the larger the co-changed method groups a method evolve through the less is the likelihood of the method being related to other methods in that group. However, whether one or more groups of functionally related methods appear in a single commit might be subject to the developers choice. Again, this study does not propose any standard threshold on significance to be representative of the existence of dependency relation and a limited empirical validation of the measure was presented.

Mondal *et al.* in another study [115] proposed an improvement in accuracy for detecting evolutionary coupling by measuring *change correspondence*. The key idea here is that if two co-changed methods have common identifiers changed in the corresponding changed parts of the methods, they are likely to be related. However, the similarity in identifiers in changed regions of two different methods may not always reflect the dependency unless those regions are syntactically similar. The concept of change correspondence might be useful in identifying related duplicate or cloned methods. However, to better understand the changes we need to know the syntactic types of changes in addition of the identifiers affected. The authors also have investigated the application of evolutionary coupling to identify and rank potential change candidates for cloned code in other studies [114, 124, 115]. However, none of the studies consider fine-grained change types in detecting couplings among the entities.

Kagdi *et al.* [72], showed that integrating conceptual couplings to the evolutionary coupling can significantly improve the accuracy of detecting couplings among the evolving software artifacts. This study considers source code artifacts as documents and measures document similarity using IR technique to infer on the dependency between the co-evolving source code entities. Their study exemplifies the potentials of using information beyond the frequency of change or co-change of software entities to improve the accuracy of the conventional evolutionary coupling. Rolfsnes *et al.* [141] proposed a new algorithm to detect evolutionary coupling. They proposed targeted association rule mining technique for both seen and unseen queries for improved detection of evolutionary coupling. In our study, we mine association rules for each pair of co-changed methods for both proposed and conventional approach [189].

In this study, we augment the fine-grained change type information with the frequencies of change or co-change of co-changing entities to improve the accuracy of the detection of couplings between clones. Our empirical results show that integration of fine-grained change information can improve the accuracy of the detection of evolutionary coupling in terms of recall for clones compared to the conventional approach.

9.9 Summary

This study demonstrates the potential benefits of incorporating syntactic change information in detecting evolutionary coupling of clones by mining association rules. Our empirical results show that the proposed typed evolutionary coupling exhibits higher recall than conventional evolutionary coupling. This approach takes advantages of the key concept behind the conventional approach that *"entities those co-change frequently are likely to be related"* and strengthens it with the knowledge of real syntactic change types for improved accuracy. In this study, we have used the fine-grained change information available at the method level. However, changes at class level (*e.g.,* renaming a field, change in the class hierarchy) might also introduce changes to related methods. Inclusion of changes at other granularity levels in the detection of typed evolutionary coupling would be an interesting area for future research.

Chapter 10
Conclusion

Clones comprise a significant proportion of the code base and the impact of clones on software maintenance has been a great concern. Although it is believed that code cloning speeds up software development and facilitates the reuse of mature and tested code, clones are often accused of introducing maintenance challenges by making consistent changes more difficult leading to the introduction of bugs, propagation of existing bugs and thus resulting in increased maintenance efforts. Given the negative impacts of clones, researchers agree that clones need to be managed. However, to manage clones we need to identify the characteristics of clones and their impacts on software maintenance and evolution. In our research, we comprehensively investigate the impact of clones from new perspectives and we propose an evolutionary coupling based change impact analysis technique for clones.

The rest of the chapter is organized as follows: Section 10.1 provides a summary of our research. Section 10.2 lists the key contributions of the thesis. Section 10.3 discusses the limitations of this thesis. Finally, Section 10.4 outlines some future research directions.

10.1 Research Summary

In this thesis, we broadly focus on two key goals: (1) comprehensive assessment of the impacts of clones on software systems and then (2) developing a solution to support clone management by supporting the identification of co-change candidates for clones. We address these two objectives in two thematic phases of research. Phase 1 comprises of investigating the impacts of clones from new perspectives to understand, evaluate the problem and formulate the background to devise a possible solution. Phase 2, on the other hand, focuses on developing solution for the problem investigated in Phase 1. We summarize all our research regarding this thesis as follows:

Assessing the impacts of clones: To comprehensively assess the impacts of clones on software maintenance and evolution, we investigate the impacts of clones from new perspectives. In Chapter 3, we measure comparative *stability* of cloned and non-cloned code by considering the fine-grained syntactic change types and their significance. As different syntactic change types have different impacts on the components related to a changed component, this study investigates the comparative stability of cloned and non-cloned code considering syntactic change types and their significance. In Chapter 4 we carry out an exploratory study to evaluate the impacts of code clones at the domain level. We investigate the relationships between domain-level coupling among the user interface components (UIC) and code clones. Earlier studies show that domain level coupling corresponds to dependencies at code level such as conceptual and evolutionary coupling [14, 48]. From our study, we observe a strong correspondence between code clones and domain-based coupling. Thus, the existence of clones corresponds to the existence of couplings at domain-level as well as code level. This shows that code clones corresponds to coupling and thus clones likely to have negative impacts on the software systems. In Chapter 5, we evaluate the comparative stability of cloned and non-cloned code within a uniform framework employing different clone detection tools and subject systems used in the original studies. We evaluate the metrics for the impacts of clones in their original experimental settings (tools, parameters, and subject systems) to comprehensively assess the impacts of clones avoiding possible biases in experimental settings. Then we take a deeper look into whether and how different evolutionary characteristics of clones such as stability and co-change coupling contribute to the bug-proneness, one of the major claims against clones. In Chapter 6 we investigate the relationships between the stability of clones and their bug-proneness. The objective is to evaluate whether instability of clones contributes to their bug-proneness. Our experimental results show that stability of clones are related to bug-proneness and buggy clones tend to be less stable compared to non-buggy clones. In Chapter 7, we investigate the relationships between bug-proneness and co-change coupling of clones. The objective is to examine whether and how the degree co-change coupling of clones are related to their bug-proneness. Our experimental results show that bug-proneness of clones are significantly related to the co-change coupling. Our analysis in Chapter 6 and Chapter 7 gives important insights regarding two evolutionary characteristics of clones and our findings have the potential to support the identification and management of bug-prone clones.

Developing evolutionary coupling based CIA approach for clones: After our comprehensive analysis of the impacts of clones and examining two important evolutionary characteristics of clones as the potential influencing factors for clones to be bug-prone, we focus on how we can support managing clones. We aim to support Change Impact Analysis (CIA) of clones to support clone management. Identification of co-change candidates is important during clone evolution to ensure the consistent evolution of clones. However, the identification of co-change candidates for clones is a challenging task as the dependencies among clones may not be identified by traditional static or dynamic analysis. Evolutionary coupling has been found to be a useful technique to identify evolutionary dependencies among clones. However, evolutionary coupling may suffer from low accuracies as it is based only on the change histories. In addition, excessively large commits may affect the accuracy of the detecting the evolutionary coupling. In Chapter 8 we evaluate the impact of atypical (extra-large) commits on the detection of evolutionary coupling and proposed a clustering-based automatic splitting of atypical commits to smaller pseudo commits of related entities. Our proposed approach considerably improve the precision of the detection of evolutionary coupling with slightly improve in the overall recall. In Chapter 9 we proposed typed evolutionary coupling for identifying co-change candidates for clones. Our proposed considers fine-grained syntactic change types in detecting evolutionary coupling for clones. Our experimental results show that typed evolutionary coupling can support more accurate detection of evolutionary couplings for clones.

10.2 Contributions

In our research, we address two specific broader goals through series of empirical studies. Our findings are important regarding the impacts of clones on software systems. In this thesis we make the following research contributions:

- **Using classified syntactic changes in the analysis of stability of clones:** We analyze the comparative stability of cloned and non-cloned code using fine-grained syntactic changes from ChangeDistiller. As different syntactic changes have different impacts on software systems, use of syntactically classified changes in stability analysis provides more practical analysis of the comparative stability of cloned and non-cloned code. This research has been published in the proceedings of **SCAM 2014** [135].

- **Assessing the impact of clones at domain level**: In an exploratory study, we investigate the relationships between domain-based coupling and code clones. Our study results show that there is strong correspondence between the existence of domain-based coupling and code clones among the user interface components. As high coupling is not a desirable software attribute, the high likelihood of co-existence of domain level coupling and code clones likely to be an indication of negative impacts of clones on software systems. We assess the impacts from beyond the source code *i.e.,* at the domain level. Our work has been published in the proceedings of **ICSE 2013** (NIER Track) [134].

- **Analyzing clone stability using uniform framework:** Given the contradictory outcomes from the earlier studies on the comparative stability of cloned and non-cloned code, we carry out stability analysis of clones within a uniform framework to avoid possible biases due to the differences in experimental settings. We systematically replicate the original stability analysis methodologies to compare the stability results with that of original experimental settings. This research has been published as part of a journal paper published in the Empirical Software Engineering **EMSE 2017** [121]. The key concept of this uniform framework is originated from our earlier studies related to the analysis of comparative stability of cloned and non-cloned code published in **ICPC 2011** [110] and **ACM SAC 2012** [122].

- **Investigating the relationship between stability and bug-proneness of clones:** In an empirical study, we investigate whether stability and bug-proneness are related. Our study results suggest that there exist significant relationships between stability and bug-proneness of clones. We analyze the stability using fine-grained syntactic change types. This study gives important insights regarding stability as an influencing factor for clones to be bug-prone. Our research has been published in the proceedings of **SCAM 2017** [136].

- **Investigating the relationships between the co-change coupling and bug-proneness of clones:** In this study, we investigate the relationships between the degree of co-change coupling and the bug-proneness of clones. Our findings suggest that degree of co-change coupling is significantly related to the bug-proneness of clones. Clones tend to be more bug-prone when they have stronger co-change couplings with non-clones. This study identifies co-change coupling as an important evolutionary characteristic of clones to be a useful indicator for clones to be bug-prone.

- **Analyzing the effects of atypical commits on the detection of evolutionary coupling:** We proposed a clustering-based automatic splitting of atypical commits to improving the detection accuracy of evolutionary coupling. Our empirical results show that our approach considerably improves the prediction accuracy of co-change candidates over the conventional approach for the detection of evolutionary coupling.

- **Typed evolutionary coupling of clones:** We introduce the notion of typed evolutionary coupling. The proposed typed evolutionary coupling incorporates fine-grained syntactic change types in the detection of evolutionary coupling. Types of syntactic changes in addition to co-changes history have the potential to detect evolutionary coupling more accurately. We leverage the syntactic change information to detect evolutionary couplings of clones more accurately. Our empirical results show that our proposed approach can help identifying co-change candidate clones with increased accuracy.

In conclusion, our research analyzes the impacts of clones, identify important evolutionary characteristics of clones to be bug-prone and proposes evolutionary coupling-based improved technique to support change impact analysis of clones. Our research contributions are important for making clone management decisions and guiding the consistent evolution of clones.

10.3 Threats to Validity

We have mentioned the limitations of the individual study in the corresponding chapters. Here, we discuss some of the limitations of the thesis in general. Our studies are limited to the syntactic (Type 1, Type 2 and Type 3) clones only and the analysis of semantic (Type 4) clones are not within the scope of this thesis. In the most of our studies, we used fine-grained syntactic change types. We used ChangeDistiller for as the classifier to extract classified changes. Due to the limitation of ChangeDistiller most of our studies are based on Java systems. Although Java is is a widely used language, the inclusion of systems of other programming languages might help more generalization of our findings.

The analysis in our research depends on different tools, software platforms, and source code libraries. Our analysis process is thus sensitive to the limitations of the underlying tools and resources. However, we have carefully chosen the tools and resources that are widely used and reported to have reliable performance. Again, our analysis is based on the open-source subject systems. However, the design and evolution patterns of commercial software systems may be different. It would be interesting to investigate commercial software systems as well. However, we selected subject systems with diverse size, application domains, and the evolution history. Our studies are limited to open-source software systems as we have limited access to commercial software systems for analysis due to undisclosed business interests of the concerned organizations.

10.4 Future Research

Clones are of important concern in the context of software maintenance and evolution. Based on our research we have identified some potential areas in clone evolution to further explore. We consider the followings as our future research directions:

Change Impact analysis of Type 4 clones: Our research in this thesis encompasses the analysis of syntactic (Type 1, Type 2 and Type 3) clones only. However, Type 4 clones by definition [142, 149] are functionally similar code fragments with different syntactic implementations. Thus, the approaches to identifying dependency and evolution patterns in syntactic clones may not be applicable to semantic (Type 4) clones. It is important to identify semantic similarity and dependency between code entities to facilitate code reuse and support software maintenance. However, the measurement of the semantic similarity is still an interesting research problem with limited success [160, 46, 159]. Again, to ensure consistent evolution of clones, it is necessary to propagate corresponding changes to candidate clones. Since Type 4 clones are syntactically different, determining equivalent changes for Type 4 co-change candidates during change propagation would be a challenging task. Application of deep learning on a very large clone data set has the potential to determine the corresponding changes for Type 4 clones.

Application of Machine Learning for clone evolution analysis: Recently, Machine Learning (ML) is getting increasingly popular in solving many software engineering problems [89, 90, 184]. Application of machine learning has been found to be useful in detection and analysis of clones [182, 62, 159, 169]. Given the huge body of open-source software projects, machine learning algorithms can learn different insights from the existing projects which are difficult to identify by conventional static and dynamic analysis. For example, in our studies, we analyzed the impacts of clones on software systems and we also investigated some evolutionary characteristics (stability and co-change coupling) of clones to determine whether and how these evolutionary properties are related to bugs. We observe that the impacts of clones on software systems vary from system to system and thus it is hard to draw a firm conclusion. So, we need to uncover the hidden relationships between the features (based on source code and their evolution) of the software systems to understand and determine the factors influencing the impacts of clones. Application of machine learning has the potentials to discover such hidden relationships. Thus, we can apply machine learning to identify the hidden features and patterns of clone evolution to guide clone management activities such as clone detection, refactoring, and tracking.

Increasing language support for change classifier: Changes are important in software evolution and maintenance [161]. From our studies, we observe that fine-grained syntactic changes are important for the evolution analysis for clones or even non-cloned code. However, currently we have classifier (ChangeDistiller) for Java programming language only [43]. This change classification is based on the AST-differencing of source code files. It would be useful to develop change classifiers for other programming languages and integrate into the IDEs for automatic extraction of classified changes for the evolution analysis of clones.

An integrated framework for clone analysis: Code clone analysis is a time and resource intensive work. Unfortunately, there are very limited opportunities to have access to sharable research data and frameworks to support the ongoing research on code clones[109]. These limitations are also likely to contribute to the contradictory outcomes of clone analysis results. It is thus important to develop a sharable open-access framework for research data, tools, and benchmark data to be available for research community [165, 166, 143, 146].

Bibliography

[1] *CCFinderX*, accessed on July 3, 2016. `http://www.ccfinder.net/ccfinderxos.html`.

[2] *CTAGS*, accessed on July 3, 2016. `http://ctags.sourceforge.net/`.

[3] *Simian*, accessed on July 3, 2016. `http://www.redhillconsulting.com.au/products/simian/`.

[4] *Strike A Match*, last accessed on June 30, 2014. `http://www.catalysoft.com/articles/strikeamatch.html`.

[5] *Mann-Whitney Wilcoxon (MWW) Test*, last accessed on March 05, 2018. `http://www.socscistatistics.com/tests/mannwhitney/Default2.aspx`.

[6] *ArtisticStyle*, last accessed on May 14, 2017. `https://sourceforge.net/projects/astyle/`.

[7] *SourceForge*, last accessed on September 10, 2017. `https://sourceforge.net/`.

[8] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proc. SIGMOD*, pages 207–216, 1993.

[9] Samuel Ajila. Software maintenance: an approach to impact analysis of objects change. *Software Pract. Exper.*, 25:1155–1181, 1995.

[10] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. What's a typical commit? a characterization of open source software repositories. In *Proc. ICPC*, pages 182–191, 2008.

[11] Aryani Amir, Perin Fabrizio, Lungu Mircea, Mahmood Abdun Naser, and Nierstrasz Oscar. Predicting dependences using domain-based coupling. *Journal of Software: Evolution and Process*, 26(1):50–76, 2014.

[12] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proc. ICSE*, pages 432–441, 2005.

[13] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[14] A. Aryani, F. Perin, M. Lungu, A. N. Mahmood, and O. Nierstrasz. Can we predict dependencies using domain information? In *Proc. WCRE*, pages 55–64, 2011.

[15] Amir Aryani, Ian D. Peake, and Margaret Hamilton. Domain-based change propagation analysis: An enterprise system case study. In *Proc. ICSM*, pages 1–9, 2010.

[16] Amir Aryani, Ian D. Peake, Margaret Hamilton, Heinz Schmidt, and Michael Winikoff. Change propagation analysis using domain information. In *Proc. ASWEC*, pages 34–43, 2009.

[17] Muhammad Asaduzzaman, Chanchal K. Roy, and Kevin A. Schneider. Viscad: flexible code clone analysis support for nicad. In *Proc. IWSC*, pages 77–78, 2011.

[18] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *Proc. CSMR*, pages 81–90, 2007.

[19] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: A control call graph based technique. In *Proc. APSEC*, pages 167–175, 2005.

[20] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. WCRE*, pages 86 –95, 1995.

[21] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proc. ICSM*, pages 273–282, 2011.

[22] L. Barbour, F. Khomh, and Y. Zou. An empirical study of faults in late propagation clone genealogies. *Journal of Soft. Evol. and Proc*, 25:1139–1165, May 2013.

[23] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proc. ICSM*, pages 368–377, 1998.

[24] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *Proc. WCRE*, pages 85–94, 2009.

[25] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proc. ICSE*, pages 482–498, 1993.

[26] Thierry Bodhuin and Maria Tortorella. A tool for static and dynamic model extraction and impact analysis. In *Proc. CSMR*, pages 193–, 2005.

[27] Ben Breech, Mike Tegtmeyer, and Lori Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *Proc. ICSM*, pages 55–65, 2006.

[28] Lionel C. Briand, Juergen Wuest, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proc. ICSM*, pages 475–482, 1999.

[29] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Proc. METRICS*, pages 9 pp. –29, 2005.

[30] Gerardo Canfora, Michele Ceccarelli, Luigi Cerulo, and Massimiliano Di Penta. Using multivariate time series and association rules to detect logical change coupling: an empirical study. In *Proc. ICSM*, 2010.

[31] Gerardo Canfora and Luigi Cerulo. Fine grained indexing of software repositories to support impact analysis. In *Proc. MSR*, pages 105–111, 2006.

[32] Gerardo Canfora and Luigi Cerulo. Jimpa: An eclipse plug-in for impact analysis. In *Proc. CSMR*, pages 341–342, 2006.

[33] D. Chatterji, J. C. Carver, N. A. Kraft, and J. Harder. Effects of cloned code on software maintainability: A replicated developer study. In *Proc. WCRE*, pages 112–121, 2013.

[34] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, and N. A. Kraft. Measuring the eficacy of code clone information in a bug localization task: An empirical study. In *Proc. ESEM*, pages 20–29, 2011.

[35] J. Cohen. *Statistical power analysis for the behavioral sciences (2nd ed.).* 1988.

[36] J. R. Cordy and C. K. Roy. The nicad clone detector. In *Proc. ICPC*, pages 219–220, 2011.

[37] James R. Cordy and Chanchal K. Roy. The NiCad clone detector. In *Proc. ICPC*, pages 219–220, 2011.

[38] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *Proc. ICSM*, pages 169–178, 2009.

[39] E. Duala-Ekoko and M. Robillard. Clonetracker. In *Proc. ICSE*, pages 843–846, 2008.

[40] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. ICSM*, pages 109–118, 1999'.

[41] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proc. ICSM*, pages 109–118, 1999.

[42] Martin Ester, Hans-Peter Kriegel, Jrg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. 96(34):226–231, 1996.

[43] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proc. ICPC*, pages 35–45, 2006.

[44] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000.

[45] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[46] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proc. ICSE*, pages 321–330, 2008.

[47] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proc ICSM*, pages 190–198, 1998.

[48] M. Gethers, A. Aryani, and D. Poshyvanyk. Combining conceptual and domain-based couplings to detect database and code dependencies. In *Proc. SCAM*, pages 144–153, 2012.

[49] N. Göde and J. Harder. Clone stability. In *Proc. CSMR*, pages 65–74, 2011.

[50] Nils Göde and Rainer Koschke. Incremental clone detection. In *Proc. CSMR*, pages 219 –228, 2009.

[51] Nils Göde and Rainer Koschke. Frequency and risks of changes to clones. In *Proc. ICSE*, pages 311–320, 2011.

[52] Nils Göde and Rainer Koschke. Studying clone evolution using incremental clone detection. *Journal of Software: Evolution and Process*, 25(2):165–192, 2013.

[53] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proc. MSR*, pages 121–130, 2013.

[54] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: A taxonomical study of large commits. In *Proc. MSR*, pages 99–108, 2008.

[55] Michael A. Hoffman. Automated impact analysis of object-oriented software systems. In *Proc. OOPSLA*, pages 72–73, 2003.

[56] Keisuke Hotta, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: An empirical study on open source software. In *Proc. IWPSE*, pages 73–82, 2010.

[57] Keisuke Hotta, Yui Sasaki, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto. An empirical study on the impact of duplicate code. *Adv. Soft. Eng.*, 2012:5:5–5:5, January 2012.

[58] 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 *IWSC*, pages 94–95, 2012.

[59] J. F. Islam, M. Mondal, and C. K. Roy. Bug replication in code clones: An empirical study. In *Proc. SANER*, volume 1, pages 68–78, 2016.

[60] Judith F. Islam, Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. A comparative study of software bugs in clone and non-clone code. In *Proc. SEKE*, pages 436–443, 2017.

[61] Judith F. Islam, Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. Comparing software bugs in clone and non-clone code: An empirical study. *International Journal of SEKE*, 27(09n10):1507–1527, 2017.

[62] S. Jadon. Code clones detection using machine learning technique: Support vector machine. In *Proc. ICCCA*, pages 399–303, 2016.

[63] Mohammad-Amin Jashki, Reza Zafarani, and Ebrahim Bagheri. Towards a more efficient static software change impact analysis method. In *Proc. PASTE*, pages 84–90, 2008.

[64] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. ICSE*, pages 96–105, 2007.

[65] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proc. FSE*, pages 55–64, 2007.

[66] J. H. Johnson. Substring matching for clone detection and change tracking. In *In Proc. ICSM*, pages 120–126, 1994.

[67] J. Howard Johnson. Visualizing textual redundancy in legacy source. In *Proc. CASCON*, pages 32–41, 1994.

[68] J.H. Johnson. Identifying redundancy in source code using fingerprints. In *Proc. CASCON*, pages 171–183, 1993.

[69] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. ICSE*, pages 485–495, 2009.

[70] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. ICSE*, pages 485–495, 2009.

[71] Elmar Jürgens, Florian Deissenboeck, Martin Feilkas, Benjamin Hummel, Bernhard Schätz, Stefan Wagner, Christoph Domann, and Jonathan Streit. Can clone detection support quality assessments of requirements specifications? In *Proc. ICSE*, pages 79–88, 2010.

[72] Huzefa H. Kagdi, Malcom Gethers, and Denys Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering*, 18(5):933–969, 2013.

[73] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE TSE*, 28:654–670, 2002.

[74] Cory Kapser and Michael W. Godfrey. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[75] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.

[76] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proc. FSE*, pages 187–196, 2005.

[77] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto. Splitting commits via past code changes. In *Proc. APSEC*, pages 129–136, 2016.

[78] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! are you committing tangled changes? In *Proc. ICPC*, pages 262–265, 2014.

[79] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Static Analysis*, pages 40–56, 2001.

[80] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In Patrick Cousot, editor, *Static Analysis*, pages 40–56, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[81] K. Kontogiannis, R. De Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. 3(1/2):79–108, June 1996.

[82] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proc. WCRE*, pages 253 –262, 2006.

[83] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proc. WCRE*, pages 170–178, 2007.

[84] J. Krinke. Is cloned code more stable than non-cloned code? In *Proc. SCAM*, pages 57–66, 2008.

[85] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. WCRE*, pages 301–309, 2001.

[86] Jens Krinke. Is cloned code older than non-cloned code? In *Proc. IWSC*, pages 28–33, 2011.

[87] G. P. Krishnan and N. Tsantalis. Unification and refactoring of clones. In *Proc. CSMR-WCRE*, pages 104–113, 2014.

[88] Daniel E Krutz and Emad Shihab. Cccd: Concolic code clone detection. In *Proc. WCRE*, pages 489–490, 2013.

[89] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Proc. ASE*, pages 476–481, 2015.

[90] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *Proc. ICPC*, pages 218–229, 2017.

[91] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proc. ICSE*, pages 308–318, 2003.

[92] Michelle Lee, A. Jefferson Offutt, and Roger T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proc. TOOLS*, pages 61–70, 2000.

[93] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23:613–646, 2012.

[94] J. Li and M. D. Ernst. CBCD: Cloned buggy code detector. In *Proc. ICSE*, pages 310–320, June 2012.

[95] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE TSE*, 32:176–192, 2006.

[96] Mikael Lindvall and Kristian Sandahl. Traceability aspects of impact analysis in object-oriented systems. *Journal of Soft. Maintenance: Res. and Practice*, 10(1):37–57, 1998.

[97] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proc. ICSM*, pages 227–236, 2008.

[98] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proc. ICSM*, pages 227–236, 2008.

[99] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Proc. MSR*, pages 18–21, 2007.

[100] Angela Lozano and Michel Wermelinger. Tracking clones' imprint. In *Proc. IWSC*, pages 65–72, 2010.

[101] M. Mandal, C. K. Roy, and K. A. Schneider. Automatic ranking of clones for refactoring through mining association rules. In *Proc. CSMR-WCRE*, pages 114–123, 2014.

[102] A. Marcus and J.I. Maletic. Identification of high-level concept clones in source code. In *Proc. ASE*, pages 107–114, 2001.

[103] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proc. IWPC*, pages 33 – 42, 2005.

[104] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *In Proc. ICSE*, pages 107–, 2001.

[105] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. ICSM*, pages 244–253, 1996.

[106] K. O. R McGraw and S. P. Wong. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *SCP*, 11:470–495, 2009.

[107] A. Michail. Browsing and searching source code of applications written using a gui framework. In *Proc. ICSE*, pages 327 –337, 2002.

[108] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proc. ICSM*, pages 120–130, 2000.

[109] M. Mondal. *Analyzing Clone Evolution for Identifying Important Clones for Management.* PhD thesis, University of Saskatchewan, Canada, 2017.

[110] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider. An empirical study of the impacts of clones in software maintenance. In *Proc. ICPC*, pages 242–245, 2011.

[111] M. Mondal, C. K. Roy, and K. A. Schneider. Dispersion of changes in cloned and non-cloned code. In *Proc. IWSC*, pages 29–35, 2012.

[112] M. Mondal, C. K. Roy, and K. A. Schneider. Improving the detection accuracy of evolutionary coupling. In *Proc. ICPC*, pages 223–226, 2013.

[113] M. Mondal, C. K. Roy, and K. A. Schneider. Automatic identification of important clones for refactoring and tracking. In *Proc. SCAM*, pages 11–20, 2014.

[114] M. Mondal, C. K. Roy, and K. A. Schneider. A fine-grained analysis on the evolutionary coupling of cloned code. In *Proc. ICSME*, pages 51–60, 2014.

[115] M. Mondal, C. K. Roy, and K. A. Schneider. Improving the detection accuracy of evolutionary coupling by measuring change correspondence. In *Proc. CSMR-WCRE*, pages 358–362, 2014.

[116] M. Mondal, C. K. Roy, and K. A. Schneider. A comparative study on the bug-proneness of different types of code clones. In *Proc. ICSME*, pages 91–100, 2015.

[117] 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 *Proc. SANER*, pages 484–488, 2015.

[118] M. Mondal, C. K. Roy, and K. A. Schneider. Bug propagation through code cloning: An empirical study. In *Proc. ICSME*, pages 227–237, 2017.

[119] M. Mondal, C. K. Roy, and K. A. Schneider. Does cloned code increase maintenance effort? In *Proc. IWSC*, pages 1–7, 2017.

[120] M. Mondal, C. K. Roy, and K. A. Schneider. Identifying code clones having high possibilities of containing bugs. In *Proc. ICPC*, pages 99–109, May 2017.

[121] Manishankar Mondal, Md Saidur Rahman, Chanchal K. Roy, and Kevin A. Schneider. Is cloned code really stable? *Empirical Software Engineering*, Jul 2017.

[122] Manishankar Mondal, Chanchal K. Roy, Md. Saidur Rahman, Ripon K. Saha, Jens Krinke, and Kevin A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proc. ACM SAC*, pages 1227–1234, 2012.

[123] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. An empirical study on clone stability. *SIGAPP App. Comp. Rev.*, 12(3):20–36, 2012.

[124] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. Prediction and ranking of co-change candidates for clones. In *Proc. MSR*, pages 32–41, 2014.

[125] Manishankar Mondal, Chanchal K. Roy, and Kevin A. Schneider. A comparative study on the intensity and harmfulness of late propagation in near-miss code clones. *Software Quality Journal*, 24(4):883–915, 2016.

[126] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen. Scalable and incremental clone detection for evolving software. In *Proc. ICSM*, pages 491–494, 2009.

[127] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. FSE*, pages 128–137, 2003.

[128] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proc. ICSE*, pages 491–500, 2004.

[129] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *Proc. ASE*, pages 268–278, 2013.

[130] M. Petrenko and V. Rajlich. Variable granularity for improving precision of impact analysis. In *Proc. ICPC*, pages 10 –19, 2009.

[131] M. Petrenko, V. Rajlich, and R. Vanciu. Partial domain comprehension in software evolution and maintenance. In *Proc. ICPC*, pages 13 –22, 2008.

[132] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, 2009.

[133] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *Proc. MSR 2010*, pages 72–81, 2010.

[134] M. S. Rahman, A. Aryani, C. K. Roy, and F. Perin. On the relationships between domain-based coupling and code clones: An exploratory study. In *Proc. ICSE-NIER*, pages 1265–1268, 2013.

[135] M. S. Rahman and C. K. Roy. A change-type based empirical study on the stability of cloned code. In *Proc. SCAM*, pages 31–40, 2014.

[136] M. S. Rahman and C. K. Roy. On the relationships between stability and bug-proneness of code clones: An empirical study. In *Proc. SCAM*, pages 131–140, 2017.

[137] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proc. IWPC*, pages 271 – 278, 2002.

[138] D. Ratiu and F. Deissenboeck. From reality to programs and (not quite) back again. In *Proc. ICPC*, pages 91 –102, 2007.

[139] M. Riebisch. Supporting evolutionary development by feature models and traceability links. In *Proc. ECBS*, pages 370 – 377, 2004.

[140] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *Proc. WCRE*, pages 100 – 109, 2004.

[141] T. Rolfsnes, S. D. Alesio, R. Behjati, L. Moonen, and D. W. Binkley. Generalizing the analysis of evolutionary coupling for software change impact analysis. In *Proc. SANER*, pages 201–212, 2016.

[142] C. K. Roy. Detection and analysis of near-miss software clones. In *Proc. ICSM*, pages 447–450, 2009.

[143] C. K. Roy. Large scale clone detection, analysis, and benchmarking: An evolutionary perspective (keynote). In *Proc. IWSC)*, pages 1–1, 2018.

[144] C. K. Roy and J. R. Cordy. NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proc. ICPC*, pages 172 –181, 2008.

[145] C. K. Roy and J. R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proc. ICSTW*, pages 157–166, 2009.

[146] C. K. Roy and J. R. Cordy. Benchmarks for software clone detection: A ten-year retrospective. In *Proc. SANER*, pages 26–37, 2018.

[147] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Comp. Prog.*, 74:470–495, 2009.

[148] C. K. Roy, M. F. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *Proc. CSMR-WCRE*, pages 18–33, 2014.

[149] C. K. Roy, M. F. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *proc. CSMR-WCRE*, pages 18–33, 2014.

[150] Chanchal K. Roy and James R. Cordy. A survey on software clone detection research. *School Of Computing TR 2007-541, Queens University*, page 115, 2007.

[151] Chanchal K. Roy and James R. Cordy. Near-miss function clones in open source software: An empirical study. *Journal of Soft. Maintenance*, 22(3):165–189, 2010.

[152] Spencer Rugaber. The use of domain knowledge in program understanding. *Annals of Soft. Engg.*, 9:143–192, 2000.

[153] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *Proc. SCAM*, pages 87–96, 2010.

[154] R. K. Saha, C. K. Roy, and K. A. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proc. ICSM*, pages 293–302, 2011.

[155] Ripon K. Saha, Chanchal K. Roy, and Kevin A. Schneider. Visualizing the evolution of code clones. In *Proc. IWSC*, pages 71–72, 2011.

[156] H. Sajnani, V. Saini, and C. V. Lopes. A comparative study of bug patterns in java cloned and non-cloned code. In *Proc. SCAM*, pages 21–30, 2014.

[157] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proc. ICSE*, pages 1157–1168, 2016.

[158] 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 *Proc. WCRE*, pages 13–21, 2010.

[159] A. Sheneamer and J. Kalita. Semantic clone detection using machine learning. In *Proc. ICMLA*, pages 1024–1028, 2016.

[160] Abdullah Sheneamer, Swarup Roy, and Jugal Kalita. A detection framework for semantic code clones and obfuscated code. *Expert Systems with Applications*, 97:405 – 420, 2018.

[161] Quinten David Soetens, Romain Robbes, and Serge Demeyer. Changes as first-class citizens: A research perspective on modern software tooling. *ACM Comput. Surv.*, 50(2):18:1–18:38, April 2017.

[162] D. Steidl and N. Göde. Feature-based detection of bugs in clones. In *Proc. IWSC*, pages 76–82, 2013.

[163] Daniela Steidl and Nils Göde. Feature-based detection of bugs in clones. In *Proc. IWSC*, pages 76–82, 2013.

[164] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. Change impact analysis based on a taxonomy of change types. In *Proc. COMPSAC*, pages 373–382, 2010.

[165] 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 *Proc. ICSME*, pages 476–480, 2014.

[166] J. Svajlenko and C. K. Roy. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In *Proc. ICSME*, pages 596–600, 2016.

[167] J. Svajlenko and C. K. Roy. Fast and flexible large-scale clone detection with cloneworks. In *Proc. ICSE-C*, pages 27–30, 2017.

[168] Jeffrey Svajlenko and Chanchai K. Roy. Fast, scalable and user-guided clone detection. In *Proc. ICSE*, pages 352–353, 2018.

[169] Jeffrey Svajlenko and Chanchal K. Roy. A machine learning based approach for evaluating clone detection tools for a generalized and accurate precision. *International Journal of SEKE*, 26(09n10):1399–1429, 2016.

[170] Jeffrey Svajlenko and Chanchal K. Roy. Cloneworks: A fast and flexible large-scale near-miss clone detection tool. In *Proc. ICSE-C*, pages 177–179, 2017.

[171] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Soft. Engg.*, 15(1):1–34, 2010.

[172] S. R. Tilley. Domain-retargetable reverse engineering. iii. layered modeling. In *Proc. ICSM*, pages 52–61, 1995.

[173] S. R. Tilley, H. Muller, M. J. Whitney, and K. Wong. Domain-retargetable reverse engineering. In *Proc. ICSM*, pages 142 –151, 1993.

[174] S.R. Tilley. Domain-retargetable reverse engineering. ii. personalized user interfaces. In *Proc. ICSM*, pages 336 –342, 1994.

[175] Nikolaos Tsantalis, Davood Mazinanian, and Giri P. Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11):1055–1090, November 2015.

[176] M. S. Uddin, C. K. Roy, and K. A. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *Proc. ICPC*, pages 236–238, 2013.

[177] Md. Sharif Uddin, Chanchal K. Roy, Kevin A. Schneider, and Abram Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *Proc. WCRE*, pages 13–22, 2011.

[178] S. Wagner, A. Abdulkhaleq, K. Kaya, and A. Paar. On the relationship of inconsistent software clones and faults: An empirical study. In *Proc. SANER*, pages 79–89, 2016.

[179] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. Ccaligner: A token based large-gap clone detector. In *Proc. ICSE*, pages 1066–1077, 2018.

[180] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proc. FSE*, pages 455–465, 2013.

[181] W. Wang and M. W. Godfrey. Recommending clones for refactoring using design, context, and history. In *Proc. ICSME*, pages 331–340, 2014.

[182] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proc. ASE*, pages 87–98, 2016.

[183] K. Wong. On inserting program understanding technology into the software change process. In *Proc. IWPC*, pages 90 –99, 1996.

[184] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Xiaohu Yang. Elblocker: Predicting blocking bugs with ensemble imbalance learning. *Information and Software Technology Journal (IST)*, 61:93–106, 2015.

[185] S. Xie, F. Khomh, and Y. Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *Proc. MSR*, pages 149–158, 2013.

[186] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574 – 586, Sept. 2004.

[187] Sai Zhang, Yu Lin, Zhongxian Gu, and Jianjun Zhao. Effective identification of failure-inducing changes: a hybrid approach. In *Proc. PASTE*, pages 77–83, 2008.

[188] M.F. Zibran and C.K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *Proc. SCAM*, pages 105–114, 2011.

[189] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429 – 445, June 2005.

[190] Thomas Zimmermann and Peter Weissgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. MSR*, pages 2–6, 2004.

Appendix A

List of Publications from the Thesis

1. M. Mondal, M. S. Rahman, C. K. Roy, K. A. Schneider, Is cloned code really stable?, Empirical Software Engineering (EMSE), pp. 78, July 2017. DOI: 10.1007/s10664-017-9528-y

2. M. S. Rahman and C. K. Roy, "On the Relationships between Stability and Bug-proneness of Code Clones: An Empirical Study", In Proceedings of the 17th IEEE International Working Conference on Software Code Analysis and Manipulation (SCAM 2017), 2017, pp. 131 140.

3. M. S. Rahman, Chanchal K. Roy. "A Change Type-Based Empirical Study on the Stability of Cloned Code". In Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2014, pp. 31-40.

4. M. S. Rahman, Amir Aryani, Chanchal K. Roy, Febrizio Perin. "On the relationships between Domain-Based Coupling and Code Clones: An Exploratory Study". In proceedings of the International Conference on Software Engineering (ICSE NIER Track), 2013, pp. 1265-1268.

5. 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 of the 27th ACM Symposium on Applied Computing (ACM SAC, Software Engineering Track), pp.1227-1234, 2012. (Best Paper Award)

6. M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, K. A. Schneider, An Empirical Study of the Impacts of Clones in Software Maintenance, In proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC), pp. 242-245, 2011.