Detection and Analysis of Near-Miss Clone Genealogies

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

By
Ripon K. Saha

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

It is believed that identical or similar code fragments in source code, also known as code clones, have an impact on software maintenance. A clone genealogy shows how a group of clone fragments evolve with the evolution of the associated software system, and thus may provide important insights on the maintenance implications of those clone fragments. Considering the importance of studying the evolution of code clones, many studies have been conducted on this topic. However, after a decade of active research, there has been a marked lack of progress in understanding the evolution of near-miss software clones, especially where statements have been added, deleted, or modified in the copied fragments. Given that there are a significant amount of near-miss clones in the software systems, we believe that without studying the evolution of near-miss clones, one cannot have a complete picture of the clone evolution. In this thesis, we have advanced the state-of-the-art in the evolution of clone research in the context of both exact and near-miss software clones. First, we performed a large-scale empirical study to extend the existing knowledge about the evolution of exact and renamed clones where identifiers have been modified in the copied fragments. Second, we have developed a framework, gCad that can automatically extract both exact and near-miss clone genealogies across multiple versions of a program and identify their change patterns reasonably fast while maintaining high precision and recall. Third, in order to gain a broader perspective of clone evolution, we extended gCad to calculate various evolutionary metrics, and performed an in-depth empirical study on the evolution of both exact and near-miss clones in six open source software systems of two different programming languages with respect to five research questions. We discovered several interesting evolutionary phenomena of near-miss clones which either contradict with previous findings or are new. Finally, we further improved gCad, and investigated a wide range of attributes and metrics derived from both the clones themselves and their evolution histories to identify certain attributes, which developers often use to remove clones in the real world. We believe that our new insights in the evolution of near-miss clones, and about how developers approach and remove duplication, will play an important role in understanding the maintenance implications of clones and will help design better clone management systems.

Acknowledgements

First of all, I would like to express my heart-felt and most sincere gratitude to my respected supervisors Dr. Chanchal K. Roy and Dr. Kevin A. Schneider for their constant guidance, advice, encouragement and extraordinary patience during this thesis work. Without them, this work would have been impossible.

I would like to thank Dr. Nadeem Jamali, Dr. Gordon McCalla, and Dr. Michael Bradley for their willingness to take part in the advisement and evaluation of my thesis work.

Thanks to all of the members of the Software Research Lab with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Minhaz Fahim Zibran, Muhammad Asaduzzaman, Md. Sharif Uddin, Mohammad Asif Ashraf Khan, Md. Saidur Rahman, Khalid Billah, and Manishankar Mondal.

I am grateful to Department of Computer Science, the University of Saskatchewan for their generous financial support through scholarships, awards and bursaries that helped me to concentrate more deeply on my thesis work.

I thank the anonymous reviewers for their valuable comments and suggestions in improving the papers produced from this thesis.

I would like to thank all of my friends and other staff members of the Department of Computer Science who have helped me in one way or another along the way. In particular I would like to thank Janice Thompson, Gwen Lancaster, Maureen Desjardins, and Heather Webb.

I express my gratefulness to my family members and relatives especially, my mother Sreelekha Saha, my father Dilip Saha, my brother Avigit Saha, my uncle Nil Ratan Saha, my aunt Mukty Saha, and my friends Pronab Hira, Shashanka Mondal, Tushar Podder, and Naresh Biswas, who did not get the share of my time that they deserved.

Invariably, acknowledgements always miss someone important. For those that I have not listed explicitly, thank you for being a part of this thesis and helping me grow as a person and a researcher

I dedicate this thesis to my mother Sreelekha Saha, whose selfless support and inspiration has always been with me at each and every step of my life.

Contents

List of Tables

List of Figures

List of Abbreviations

CC	Consistent Change
CCG	Consistently Changed Genealogy
CGE	Clone Genealogy Extractor
IC	Inconsistent Change
ICG	Inconsistently Changed Genealogy
LCS	Longest Common Subsequence
LOC	Lines of Code
SD	Standard Deviation
SLOC	Source lines of code

Chapter 1

Introduction

1.1 Motivation

In the twenty first century, software has become one of the most important products and industries in the world. However, software is still expensive. With the advancement of technology, the cost of hardware has consistently decreased, while the cost of software, in many instances, has increased. There are two fundamental reasons behind this. First, software development and maintenance are human dependent as opposed to hardware that is now largely driven by a mechanized industry [44]. Second, for software development it has been found that maintenance and evolution are critical activities from the cost perspective unlike other industries where development cost is the primary concern.

In software engineering, the term *software maintenance* refers to the modification of a software product after delivery to correct faults, to improve performance or other attributes. Studies show that up to 80% of total effort on software is spent on software maintenance [2]. Therefore, researchers are trying to decrease the maintenance cost by improving tool and language support, and reducing the attributes of code that may hamper maintenance tasks. Like many other factors, it is widely believed that identical or similar code fragments in source code, also known as code clones, have an impact on software maintenance. Consequently, the detection and analysis of code clones has attracted considerable attention from the software engineering research community in recent years.

Programmers often copy code fragments and then paste them with or without modifications during software development. A group of code fragments that are exactly similar to each other but may have some differences in comments or formatting is called a Type-1 or *exact* clone class. If they have some differences in the name of identifiers in addition to some variations in comments and formatting, they are called Type-2 clones. The clone fragments of a Type-3 clone class have further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments. Both Type-2 and Type-3 clone classes are known as *near-miss* clones. Previous studies have shown that systems contain duplicate code or clones in amounts ranging from 5-15% of the code-base [86] to as high as 50% [85]. While clones can be useful in many ways [52, 56], their presence may add difficulties during software maintenance. For example, if a bug is detected in a clone fragment, all the fragments similar to it should be investigated to check for the same bug. Similarly, when enhancing or adapting a piece of code, duplicate

fragments can multiply the work to be done [67]. Furthermore, inconsistent changes/updates to cloned code are frequent and may lead to severe unexpected behavior [49]. Therefore, code clones are considered as "bad smells" of a software system [7, 49].

Since researchers have presented evidence that code clones have both positive [56, 96] and negative [84] consequences for maintenance activities, but in general, code clones are neither good nor bad. It is also not possible or practical to eliminate certain clone classes from a software system [56]. Therefore, the identification and management of software clones, and the evaluation of their impact has become an essential part of software maintenance. Although there are some disagreements among the findings about the degree of harmfulness of clones, researchers agree that knowing the evolution of clones throughout a system's history is important for properly comprehending and managing the system's clones [40].

1.2 Problem Statement

Studying the evolution of code clones is essential not only to understand whether clones are harmful or not, but also to understand the diverse behaviour of their changeability, which could be useful for developing a clone management system. However, current knowledge about the evolution of code clones is mostly limited to Type-1 and Type-2 clones. By surveying the most up-to-date literature, we identified the following research problems:

- Previous studies were highly influenced by the idea that clones are harmful and can be removed through refactoring [31]. This notion was first challenged by the work of Kim et al. [56]. However, further evidence is required to confirm their findings as they analyzed two small Java systems only. They also speculated that the selected systems might not have captured the characteristics of larger systems and thus, further empirical evaluations need to be carried out for larger systems written in different programming languages.
- Despite a decade of active research, there has been a marked lack of progress in understanding the evolution of Type-3 clones. However, literature shows that there are a significant number of Type-3 clones in software systems [89]. Therefore, it is important to characterize the evolution of Type-3 clones accurately.
- We still do not know which attributes developers choose to remove clones from a system. Discovering these attributes could be useful in selecting clones for removal, especially in terms of managing clones or build a clone management system.

1.3 Contributions of the Thesis

In this thesis, we have advanced the state-of-the-art in the evolution of clone research in the context of both exact (Type-1) and near-miss (Type-2 and Type-3) software clones. There are mainly four contributions of

this thesis:

1. We have conducted a large-scale empirical study on the evolution of code clones at release level to see whether the findings reported by Kim et al. [56] also hold for other systems or not. We also extended the study in many ways such as understanding the effects of programming languages and system's size on the evolution of clones, and got useful results.

2. We have proposed an automatic framework for accurately extracting and classifying both exact (Type-1) and near-miss (Type-2 and Type-3) clone genealogies. In order to evaluate the proposed framework, we implemented a prototype, gCad that can extract and classify near-miss clone genealogies reasonably fast while maintaining high precision and recall.

3. We have extended the state-of-the-art knowledge regarding the evolution of code clones by adding to the understanding of the evolution of Type-3 clones. This study shows how current results about the evolution of code clones vary for the inclusion of Type-3 clones. We also have unveiled some evolutionary phenomena, especially for Type-3 clones, such as frequent inconsistent changes to Type-3 clones, transition from one clone class type to another type, e.g. from Type-2 to Type-3 clone class that could be useful for building a better clone management system.

4. We have retrospectively investigated the relevant attributes of code clones, which developers prefer to use when removing a clone class from a system. We have identified several attributes that are directly or indirectly related to the decision making process while removing clones. We believe our findings will be helpful for creating a 'hot list' of clones to be considered for refactoring.

1.4 Related Publications

A couple of parts of this thesis have been previously published. I have co-authored several publications which are related to this thesis as well during my thesis work. This section lists these publications. For the first three publications, I was the primary author and conducted the research under the supervision of Chanchal K. Roy and Kevin A. Schneider.

- Chapter 3 has been published in the proceedings of the Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, (SCAM'10) co-authored with Muhammad Asaduzzaman, Minhaz Fahim Zibran, Chanchal K. Roy, and Kevin A. Schneider [91].

- Chapter 4 has been published in the proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11) co-authored with Chanchal K. Roy, and Kevin A. Schneider [92].

- A position paper also has been published in the proceedings of the 5th international workshop on Software clones (IWSC'11) co-authored with Chanchal K. Roy, and Kevin A. Schneider [93], which is mentioned in Chapter 7.

The co-authored publications are -

- Minhaz Fahim Zibran, **Ripon K. Saha**, Muhammad Asaduzzaman, and Chanchal K. Roy
  Analyzing and forecasting near-miss clones in evolving software: An empirical study [104]
  16th IEEE International Conference on Engineering of Complex Computer Systems, 2011.

- Manishankar Mondal, Md. Saidur Rahman, **Ripon K. Saha**, Chanchal K. Roy, Jens Krinke, and Kevin A. Schneider
  An Empirical Study of the Impacts of Clones in Software Maintenance [77]
  Student Research Symposium Track of the 19th IEEE International Conference on Program Comprehension, 2011.

- 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 [78]
  Software Engineering Track of the 27th ACM Symposium on Applied Computing, 2012. (to appear)

1.5 Outline of the Thesis

In this chapter, we motivated the research problem in software clone detection and analysis, especially the studies of clone genealogies, and described the contributions of the thesis. The remaining chapters of the thesis are organized as follows:

In Chapter 2, we introduce the relevant terminology and outline the related research which will form the foundation we build upon in this thesis.

Chapter 3 presents an empirical study of evaluating clone genealogies in 17 open source systems at release level to see how they change and to find if the evolution of code clones is affected by programming languages, and size of programs.

In Chapter 4 we introduce an automatic framework, gCad, for extracting and classifying both exact (Type-1) and near-miss (Type-2 and Type-3) clone genealogies. We provide the details of the method along with evaluation strategies that demonstrates the effectiveness of gCad.

Chapter 5 presents an empirical study in which we investigate the evolutionary phenomena of Type-3 clones, especially in terms of clone genealogies.

In Chapter 6 we investigated the relevant attributes, which developers prefer to use when removing clones from a system.

Finally, Chapter 7 concludes the thesis along with some directions for future research.

Chapter 2

Background and Related Work

This chapter provides background information relevant to the research of this thesis. Since this thesis is concerned with developing techniques and tools to better understand the evolution of code clones, we begin with the description of code clones, reasons of cloning, and the state-of-art tools/techniques to detect clones. In the later part of this chapter, we describe the clone evolution model, existing methods to extract clone genealogies, and related works of clone evolution.

2.1 Code Clones

Code clones are similar or identical code fragments in source code often created through copy and paste programming practices of the developers. However, clones are also created accidentally while developing the same concept in different places [1]. Two code fragments similar to each other form a *clone pair*. A *clone class* is a group of clone fragments that are similar to each other. Therefore, a clone class may have two or more number of code fragments where each pair of code fragments forms a clone pair. There are mainly four types of clone classes based on their degree of textual, syntactical, and semantic similarity among clone fragments.

Type-1: A group of code fragments that are exactly similar to each other but may have some differences in comments or formatting is called a Type-1 clone class. These clone classes are also known as *exact* clone class. Figure 2.1 presents an example of Type-1 clone class.

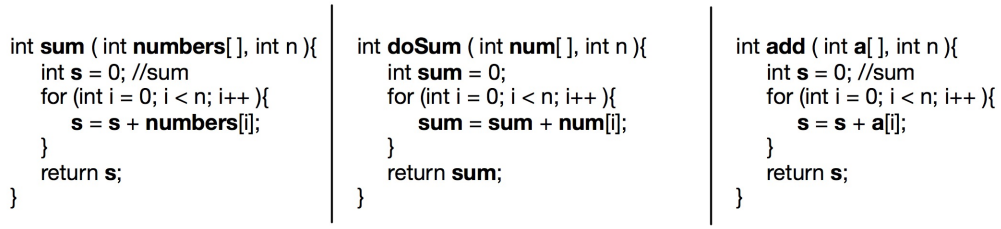
Type-2: The code fragments of a Type-2 clone class have some differences in the name of identifiers. In addition, they may have some variations in comments and formatting. Therefore, they have exactly similar syntactic structures as shown in Figure 2.2.

```
int sum ( int numbers[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

```
int sum ( int numbers[ ], int n ){
    int s = 0;
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

```
int sum ( int numbers[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

Figure 2.1: A Type-1 clone class having three clone fragments

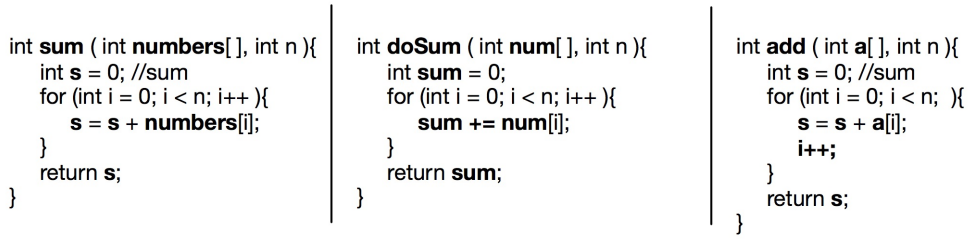


```
int sum ( int numbers[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

```
int doSum ( int num[ ], int n ){
    int sum = 0;
    for (int i = 0; i < n; i++ ){
        sum = sum + num[i];
    }
    return sum;
}
```

```
int add ( int a[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n; i++ ){
        s = s + a[i];
    }
    return s;
}
```

Figure 2.2: A Type-2 clone class having three clone fragments



```
int sum ( int numbers[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

```
int doSum ( int num[ ], int n ){
    int sum = 0;
    for (int i = 0; i < n; i++ ){
        sum += num[i];
    }
    return sum;
}
```

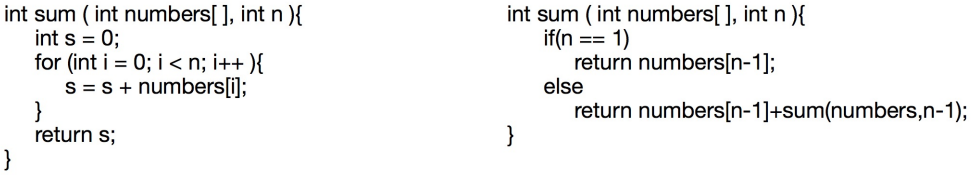
```
int add ( int a[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n;  ){
        s = s + a[i];
        i++;
    }
    return s;
}
```

Figure 2.3: A Type-3 clone class having three clone fragments

Type-3: The clone fragments of a Type-3 clone class have further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments. Figure 2.3 presents an example of Type-3 clone class. Both Type-2 and Type-3 clone classes are known as *near-miss* clone classes.

Type-4: A group of code fragments that are intended to do the same function regardless of their textual or syntactical similarity are called Type-4 or *semantic* clone class. Figure 2.4 presents an example of semantic clone class having two different code fragments both of which are intended to calculate the factorial of a number using loop and recursion.

Previous research shows that there exists a significant amount of cloned code in various software systems depending on their domain and origin [45, 67]. By summarizing the findings from various studies [11, 53, 64, 75], it can be said that both open source and industrial software systems have cloned code varying from 5% to 20% of source code. In some cases the proportion can be even higher. For example, in one study Ducasse et al. [28] found that 50% of the entire source code was duplicated in a COBOL system.



```
int sum ( int numbers[ ], int n ){
    int s = 0;
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

```
int sum ( int numbers[ ], int n ){
    if(n == 1)
        return numbers[n-1];
    else
        return numbers[n-1]+sum(numbers,n-1);
}
```

Figure 2.4: A Type-4 clone class having two clone fragments

2.1.1 Reasons

It is believed that developers often copy and paste code snippets for faster development and thus it is one of the main reasons for the creation of clones. However, researchers found many reasons for which developers create clones [6, 11, 18, 47, 51, 55, 75]. Roy and Cordy [86] classified these reasons mainly into four categories. They are as follows:

Development Strategy: Developers often introduce clones in software systems as a part of various development strategies. Reusing existing code through copying and pasting is the simplest form of code cloning. Sometimes clones are also introduced when two software systems of similar functionality are merged to produce a new one.

Maintenance Benefits: Sometimes developers intentionally keep clones in systems to facilitate the maintenance activities. As clone fragments are separate pieces of code snippets, they can evolve independently without affecting each other, which is often required. Sometimes clones are useful to keep the software architecture clean and understandable.

Overcoming Underlying Limitations: Clones are also introduced due to the limitations of both programming languages and programmers [10, 82]. These include lack of reusing mechanism in programming languages, difficulty in understanding large system, developers lack of knowledge in a program domain, and so on.

Cloning by Accidents: Clones may be introduced by accident. For example, two different developers may implement similar concepts in different places, or programmers can unintentionally repeat a common solution for similar problems.

Figure 2.5 represents a detail tree diagram for the reasons of cloning. To get a more comprehensive overview about each of the reasons, please refer to the survey by Roy and Cordy [86].

2.1.2 Impact

In the previous section we have discussed several positive reasons for which developers often intentionally create clones. However, sometimes clones may adversely affect the quality, reusability and maintainability of a software system [49]. In this section we discuss some of the negative impacts of code clones stated in the literature to explain why we need to find and remove them.

Bug Propagation

One of the major threats of code cloning is that it may facilitate propagating bugs if developers copy and paste a buggy code. Although sometimes developers modify the pasted code according to their needs, they may not notice the bug. Furthermore, it is also possible that the programmer may forget to add some statements that are specific to the new pasted segment after the copy-paste operation. Therefore, the probability of bug propagation may increase significantly in the system through cloning [49, 67].

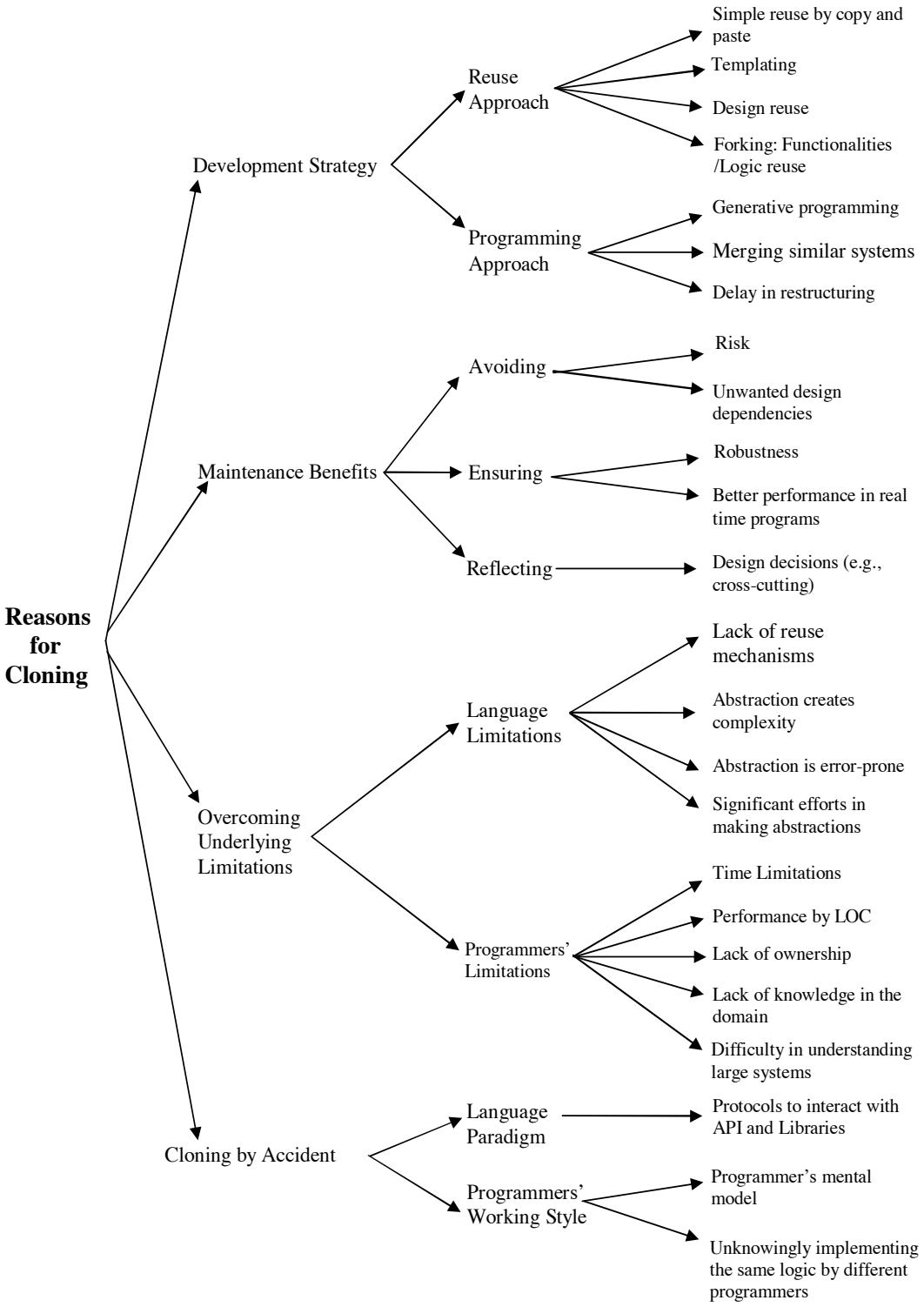


Figure 2.5: Tree diagram of reasons for cloning (taken from [86])

Incomplete Bug Removal

If a bug is detected in a clone fragment, it is highly probable that the same bug is also present in the other fragments similar to it. However, when removing bug in one clone fragment, a developer may forget to fix the same bug in the other clone fragments. It is also possible that the developer is not aware of the presence of similar code fragments of it in anywhere else in the system. Therefore, code clones may be one of the reasons for incomplete bug removal.

Change Propagation

Changing or enhancing a clone fragment can multiply the change effort to be done especially when the same change is needed for the rest of the clone fragments. Therefore, inconsistent changes to code clones may introduce bugs. There is quite a bit of study where researchers tried to understand how clone fragments of a given class actually evolve with respect to the each other during the evolution of software. For example, Kim et at. [56] found that on average 36%-38% of clone classes change consistently. We have also found similar results by conducting a large scale empirical study provided in Chapter 3. We will discuss the existing knowledge about the change patterns of code clones more elaborately in Section 2.2.3.

Program Comprehension

Sometimes code clones increase the required cognitive effort by the developers to comprehend a large program. Since cloning creates multiple occurrences of code fragments in source code, developers or maintenance engineers have to spend a significant amount of time analyzing each fragment separately to understand the differences between them [46].

Software Design

Unnecessarily copying and pasting code also may degrade the design and structure of a software. It also may result in lack of good inheritance or abstraction. As a result it lowers the software quality such as readability and changeability [79].

Resource Requirements

Since cloning creates multiple occurrences of identical or similar fragments, it inflates the size of software. Although program size is not a big issue now a days for most of the domains, large program may be a cause for hardware upgrade in some fields like embedded systems, which usually have small piece of memory. Therefore, cloning has a huge financial impact also.

2.1.3 Detection

In the previous two sections, we have already seen that code clones have both positive and negative consequences on software development and maintenance activities. Therefore, the investigation and analysis of code clones has attracted considerable attention from the software engineering research community in recent years. Detecting clones in source code of a software system is the fundamental part of code clone research since all other related research depend on the result of code clones. Furthermore, if developers or maintenance engineers do not know where the clones exactly are in the software systems, they will not be able to solve the problems caused by clones. Till now researchers have applied a number of techniques to detect clones in software systems. The approaches could be mainly classified as textual approaches, lexical approaches, tree-based/syntactic approaches, graph-based approaches, and metric-based approaches. In this section we briefly describe these approaches.

Textual Approaches

As the name implies, textual approaches consider source code of a program as a sequence of characters. Little or no source code transformation is applied on the source code before the actual comparison. As a result, this approach is independent of programming language, and even works for the source code which is not compilable. Johnson [47] is the first who introduced text-based clone detection. His approach uses *fingerprints* on substrings of the source code to find clones in source code. Manber [73] also used *fingerprints* to identify similar files. Lee et al. [65], and Wettel and Marinescu [99] also applied text-based approaches to find near-miss clones in systems. However, one of the main drawbacks of textual approaches is that it does not work well when the same syntactical structure is represented differently in different places by the developers.

Marcus and Maletic [74] applied latent semantic indexing (LSI) to source text to find high level concept clones in the source code. However, they only considered comments and identifiers instead of entire source code.

NiCad [87] is also a text-based approach but takes advantage of tree-based structural analysis based on lightweight parsing to implement flexible pretty-printing, code normalization, source transformation and code filtering. Thus it eliminates the aforementioned drawback of the textual approach. Recently, Uddin et al. [98] have performed a study with a modified version of NiCad by incorporating a text similarity measurement technique called simhash [16], which was found effective in fast detection of both exact and near-miss clones.

Lexical Approaches

Lexical approaches consider program's source code as a sequence of tokens. A token is an atomic unit of source code for a given programming language. Therefore, in this approach, first a sequence of tokens is generated from the source code of the program. Then the token sequence is searched for duplicated subsequences and

the associated original code is returned as clones. Since a token is the smallest unit of source code, the lexical approaches are best suited when changes between clone fragments are small such as identifier renaming. Dup [6], CCFinder [51], iClones [36] are some of the examples of token-based clone detectors.

Tree-based Approaches

In this approach, at first the source code of a program is converted into parse trees or abstract syntax trees. Then various tree matching techniques are used to detect clones by finding similar subtrees. Since tree-based representations of programs are insensitive to formatting, and comparatively less sensitive to programming style, it can detect some clones which are not usually detected by text-based or token-based approaches. Also the precision of tree-based approaches is higher compared to text-based and token-based approaches. However, this approach is language dependent, and requires syntactically correct program. Furthermore, the time complexity of tree-based approaches is higher than that of text-based and token-based approaches. Baxter et al.'s CloneDr [11], Jiang et al.'s Deckard [45], Koschke et al.'s cpdetector [59] are some of the examples of tree-based clone detection tools.

Graph-based Approaches

In this approach, the source code of a program is usually represented as a program dependency graph (PDG). The PDG represents a program as a graph in which the nodes are statements and predicate expressions, and the edges represent control and data dependencies among the vertices [30]. Therefore, in the PDG representation, source code are independent of their sequence of statements, and thus this approach is more robust for simple modifications of code clones such as reordering of lines. Then the clones can be searched by finding isomorphic subgraphs [60]. The main limitations of PDG-based approaches are the same as the limitations of tree-based approaches. This approach is program language dependent, requires syntactically correct program, and has high time complexity.

Metrics-based Approaches

Metrics-based approaches calculate a number of metrics for code fragments at a certain level of granularity. A level of granularity could be functions/methods, classes, or any syntactic unit. Once the metrics for all code fragments of a given syntactic unit are computed, the algorithm compares the metric values to find the clones. Since calculation of many source code metrics is language dependent, most of the metric-based clone detection tools are also language dependent. Researchers proposed a number of metrics-based approach to detect clones. Mayrand et al. [75] used several metrics such as names, layout, expressions, and simple control flow of functions to identify functions with similar metrics values as code clones. Davey et al. [21] detect exact, parameterized, and near-miss clones by first computing certain features of code blocks and then training neural networks to find similar blocks based on the features. Metrics-based approaches have also been applied to find duplicate web pages or finding clones in web documents [15, 72].

In addition to the aforementioned techniques, there are some clone detection techniques that use a combination of syntactic and semantic characteristics [66] to detect clones in source code. Besides, clone detection is not limited to source code only. Sæbjørnsen et al. [90] is the first who proposed a practical clone detection algorithm for binary executables. Davis and Godfrey [22, 23] developed a tool that can compile C, C++, and Java to assembler, and then perform clone detection on the resulting stream of assembler instructions contained within functions. Deissenboeck et al. [24, 25] presented an approach for the automatic detection of clones in large models since they are used in model-based development of control systems. Nguyen et al. [80] and Pham et al. [83] also proposed some techniques for finding clones in MATLAB/Simulink models. Researchers also proposed some techniques to detect clones in other software artefacts. Domann et al. [26] proposed an approach for detecting clones in requirement specification. Later Juergens et al. [48] applied the approach to study clones in real world requirement specification. Liu et al. [68] and Störrle et al.[94] proposed techniques for detecting clones in UML sequence diagrams and models respectively. To get a more comprehensive description of each approach mentioned in previous subsections and a detail list of clone detection tools, please refer to the survey by Roy and Cordy [86].

2.1.4 Management

It would have been safe if there were no clones in software systems, or we could refactor all the clone classes. However, it is not feasible or possible to refactor all the clone classes from systems. Furthermore, sometimes developers create clones intentionally in order to get several benefits discussed in Section 2.1.1. Therefore, in order to facilitate the software maintenance activities, we have to manage clones properly. One of the primary reasons of clone management is taking the advantages of cloning as much as possible while overcoming the threats posed by them. Researchers have proposed several approaches for managing clones. In this section, we briefly discuss these approaches.

Preventive Clone Management

There is an english proverb - "An ounce of prevention is worth a pound of cure". The main objective of this approach is to minimize the creation of clones as much as possible rather than detecting and removing them afterwards. In favour of this approach, Laguë et al. [64] described two ways of how a clone detection tool can be used in the software development process for avoiding clones. The first way is called *preventive control* where each of the new functions has to be confirmed as a non-clone code snippet before adding it to the system. If the new function is detected as a clone and there is no way to avoid it, the system architect is informed, and the programmer is queried to explain why a clone was created. If the system architect is not convinced by the provided reason, necessary actions must be taken to reuse the original function. The second way is called *problem mining* where all changes submitted to the central source code repository are monitored. If the associated function is a clone, then all the clones associated with it will be presented to the developers. Then the developer will determine whether the change must or must not be propagated to

each of the clones.

Corrective Clone Management

In corrective clone management, the suspicious clones are refactored to reduce potential sources of errors emerging from duplicated code, and to increase the understandability of software systems. Therefore, this approach may be effective for the software systems where clones were not maintained from the beginning. Extracting a clone fragment as a function, and replacing all of its siblings by calling the newly created function is the simplest form of clone refactoring [29, 42, 50, 58]. We will discuss more about the various forms of code clone refactoring in Section 6.7.

Compensative Clone management

Even after taking every step to minimize code clones in software, it is expected that there will be still some clones that are not worth or possible to refactor. In this occasion, the compensative clone management approach tries to facilitate the evolution of this group of clones. Researchers have been trying to develop new tools and technique for years so that developers can manage clones properly with a minimum effort. One of the first attempts towards this approach is *simultaneous editing* [76] that helps developers to make the same changes to all clone fragments of a given clone class at the same time. Therefore, it can prevent inconsistent changes to clone classes. Duala-Ekoko and Robillard [27] have proposed a tool called CloneTracker that can notify developers when developers tend to change a clone fragment, and offers simultaneous editing.

2.1.5 Summary

From the above discussion it is evident that clones have both positive and negative effects during software maintenance. It is also not possible or practical to eliminate all the clone classes from systems. Therefore, we have to manage clones in a cost effective way. However, the success of lowering the negative effects of code clones depend on the decision of which clones should be managed and which ones should be refactored. In order to make a wise decision for selective refactoring, first, we have to know which clones could be harmful from the maintenance perspective. However, without understanding the evolution of code clones, it is difficult to identify suspicious or harmful clones.

2.2 Evolution of Code Clones

Successful software is not static, it is a living entity that grows and evolves through time. Therefore, during the evolution of software, code clones that reside in it also evolve. Studying the evolution of code clones is important to assess the impact of clones since clones may be harmful not only for their presence in software, but also for their unintentional inconsistent evolution. For example, if there is a bug in a cloned code, it is expected that the same bug is also present in other fragments similar to it. Therefore, bugs may only be

partially removed if developers cannot find all the clone fragments or forget to fix all the clone fragments. In this section we discuss the model of clone evolution, different approaches to extract clone genealogies, and other related studies of clone evolution.

2.2.1 Clone Evolution Model

The clone evolution model seeks to represent how code clones change across versions within an epoch in a logical and objective way. First, Kim et al. [56] defined a model of clone genealogy. In this model, a clone genealogy describes how each fragment in a clone class has been changed over versions with respect to other fragments in the same clone class. Based on the change information and the number of fragments in the same clone class in two consecutive versions, Kim et al. identified six change patterns for evolving clones. We adapted their model of clone genealogy in this thesis. Now we briefly discuss the terminology relevant to this clone evolution model, various change patterns, and genealogies.

Version: As this thesis is concerned with the evolution of clones, the analyses involves more than one version. A version is a system's source code at a specific point of time. A version may be a revision or a release.

Clone Lineage: A clone lineage is a directed acyclic graph that describes the evolution history of a clone class from the beginning to the final version of the software system.

Clone Genealogy: A clone genealogy is a single clone lineage or a set of clone lineages that originate from the same clone class.

Change Patterns

Now we briefly discuss the basic concepts of these change patterns. Let cc^i be a clone class in version V_i. If cc^i is mapped to cc^{i+1} in V_{i+1} by a clone genealogy extractor, then the evolution is characterized as follows:

Same: Each clone fragment in cc^i is present in cc^{i+1}, and no additional clone fragment has been introduced in cc^{i+1}.

Add: At least one clone fragment has been introduced in cc^{i+1} that was not part of cc^i.

Delete/Subtract: At least one of the clone fragments of cc^i does not appear in cc^{i+1}.

Consistent Change (*CC*): All clone fragments in the same clone class have been changed consistently, and thus all of them are again part of the same clone class in the next version. However, a clone class may disappear even though it has been changed consistently. This could happen when the fragments become smaller than the minimum clone length set by the subject clone detection tool.

Inconsistent Change (*IC*): All clone fragments in the same class have not been changed consistently. Here we should note that for Type- 3 clones where the addition and deletion of lines are permitted, all the clone fragments of a particular clone class could still form the same clone class in the next version even if one or more fragments of that class have been changed inconsistently. The dissimilarity between the fragments of a clone class usually depends on the heuristics/similarity threshold of the associated clone detection tools.

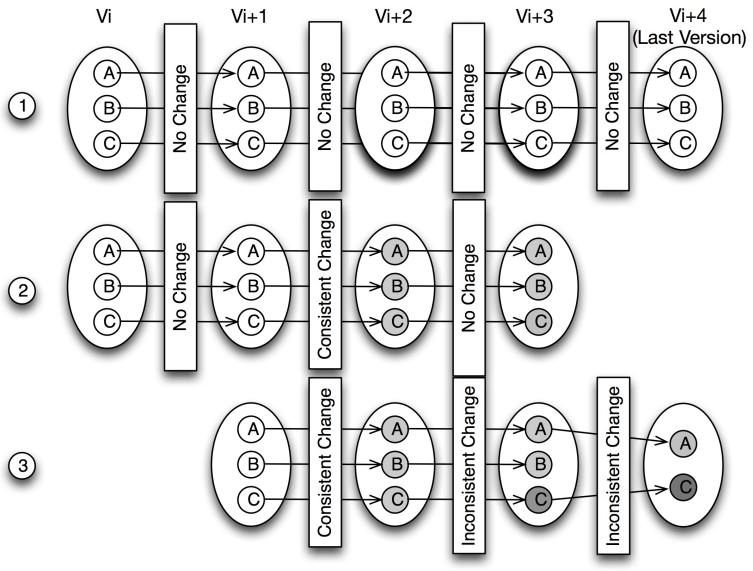


Figure 2.6: Different types of clone genealogies

Types of Genealogies

Based on the aforementioned change patterns and whether a clone class exists in the final version or not, clone genealogies can be categorized into the following groups.

Static Genealogy *(SG)*: Static genealogy refers to those genealogies in which the clone fragments in the associated clone class are propagated through subsequent versions without any textual change. The first genealogy in Figure 2.6 represents a static genealogy since there are no changes of the clone class during the evolution.

Consistently Changed Genealogy *(CCG)*: If a genealogy has any consistent change pattern(s) but does not have any inconsistent change pattern(s) during the evolution, it will be classified as consistently changed genealogy. The second genealogy in Figure 2.6 is an example of *CCG* as there is a consistent change between versions V_{i+1} and V_{i+2}.

Inconsistently Changed Genealogy *(ICG)*: If a genealogy has any inconsistent change pattern(s) throughout the evolution period, it will be classified as inconsistently changed genealogy. The third genealogy in Figure 2.6 is an *ICG* as there is an inconsistent change between versions V_{i+2} and V_{i+3}.

Dead Genealogies *(DG)*: A genealogy is called dead genealogy if its clone class disappear before reaching the final version that we considered. The second genealogy in Figure 2.6 represents a *DG* since the clone class disappears in the final version.

Alive Genealogies *(AG)*: A genealogy is called alive genealogy if the associated clone class is still evolving and thus exist in the final version that we considered. Both the first and the last genealogies in Figure 2.6 are *AG* as they reach up to the final version.

2.2.2 Extraction of Genealogies

Extracting clone genealogies from multiple versions of a program is the fundamental task to study the evolution of code clones. There are mainly four approaches that researchers used for extracting clone genealogies to study the evolution of code clones. In the first approach [56], clones are detected in each of the versions of a program and then they are mapped between consecutive versions to construct clone genealogies. In the second approach [5, 61], clones are detected in the first version of interest and then they are tracked through the subsequent versions to see how they evolve in later versions. The third approach [13] is the combination of the first and the second approaches. Finally, in the fourth approach, clone fragments are mapped during clone detection using the changed information between versions [36, 81]. Although intuitive, each of these approaches has its own advantages and disadvantages. Therefore, researchers used the approach that was best suited for their context of use.

2.2.3 Study of Clone Evolution

Although till now there are some disagreements among the researchers about various aspects of code clones such as whether clones are harmful or not, most of the researchers agree that we need to study the evolution of code clones to better understand their diverse behaviour, and maintain them properly. Therefore, a number of studies have been conducted to understand the evolution of code clones.

Clone Coverage

Laguë et al.[64] first performed the study of clone evolution for evaluating whether a clone detection tool helps software development or not. They analyzed six versions of a large telecommunication system to investigate how function clones evolve with the evolution of the system. More specifically, they checked how many clones were added, modified, and deleted in the next version compared to the previous one. From their study, they observed that a significant number of clones were removed with the evolution of the software but the overall number of clones increased over time. However, they did not address how each fragment in a clone class changed with respect to other clone fragments in the class.

Antoniol et al. [3] proposed a model of cloning using time series to monitor and predict the evolution of clones. They validated their model with several versions of a medium scale software system mSQL, and concluded that time series can predict the clone percentages of subsequent releases with an average error rate below 4%. Later in another study, Antoniol et al. [4] analyzed the evolution of clones in the Linux Kernel. They observed that in the Linux Kernel, many clones are clustered into the subsystems but there are few clone classes that are distributed across the subsystems. However, the overall number of clones over versions was stable as some clones are removed during the evolution process. In a separate study, Godfrey and Tu [38] also found similar results and concluded that cloning is a common and steady practice in the Linux kernel.

Li et al. [67] conducted a study of clone evolution in the Linux Kernel and FreeBSD. They found that

the rate of cloning for both the Linux Kernel and FreeBSD increased gradually over time. In 10 years, the increase of cloning rate in the Linux Kernel was 5% (from 17% to about 22%). Similar observation was also found for FreeBSD. However, when they delved it deeper, they found that the increase rate for few modules, `drivers` and `arch` in the Linux Kernel and `sys` in FreeBSD was actually significant instead of the entire system. Finally, they concluded that this phenomena was due to the extensive support of Linux Kernel for many similar device drivers during that period.

Zibran et al. [104] performed a large empirical study to understand the proportion and evolution of near-miss clones in evolving software systems. I was one of the co-authors of this study. In this study, we used simple regression analysis technique to forecast clone density in future versions of software systems. After extensive quantitative analysis and manual investigation over 1,636 releases of the 18 software systems, we concluded that program languages have significant effect on code cloning, whereas, there exists little or no effect of systems' size on the regularity in the evolution of clone density. We also noticed that with the growth of software systems, as the number of functions increases, the number of both exact and near-miss clone fragments also increases, indicating a very strong positive correlation between them. We also found some common patterns in the evolution of clone density across subsequent versions. For example, we found relatively higher rate of changes in clone densities over early versions of software evolution.

Change Patterns of Code Clones

Kim et al. [56] is the first who first mapped code clones between multiple versions of a program to observe how clones actually evolve from version to version. They define a model of genealogy and classify various changes in clone classes into meaningful patterns described in Section 2.2.1. Based on a case study of two Java systems, they found that on average 36% - 38% of total clones changed consistently. They also observed that many clones in software systems are volatile in nature, and thus an immediate refactoring of such short-lived clones is not required. Furthermore, there exist some long-lived clones in systems which are locally unrefactorable due to the limitations of the underlying programming languages.

Aversano et al. [5] further divided *inconsistent change* pattern into two groups: *independent evolution* and *late propagation*. In *independent evolution*, the clone fragments of a clone class, once changed inconsistently, evolve independently across revisions. On the other hand, in *late propagation*, if a clone class changed inconsistently, the same change is propagated to the rest of the fragments of that class at any time later to make them synchronized again to each other. With these evolution patterns, they performed an empirical study to investigate how clones are maintained when an evolution activity or a bug fixing takes place. In order to find the changes in clone fragments, they extracted the change information of the reference version of interest from the CVS and combined those changes with clone detection result. As of Kim et al. [56] they also manually investigated all the genealogies where changes took place in different *Modification Transactions (MT)* and clone fragments were of different files. From a case study of two Java systems, they concluded that the majority of clones are always maintained consistently. When clones are not changed consistently,

they mostly evolve independently. Later, in an extended version of this study, Thummalapenta et al. [96] found similar results.

Krinke [61] conducted an empirical study on five Java systems to investigate what changes frequently occur to the cloned code. Like Aversano et al. [5] he also considered the clones of first revision in the observation period and found the changes of cloned code by extracting the changes from source code repositories. From the study, he observed that usually half of the changes to clone classes were inconsistent changes. In another study [36], Göde and Koschke found that clones were changed rarely during their lifetime. If they were changed, they tended to be changed inconsistently.

Recently, Göde and Harder [35] analyzed different combinations of consecutive change patterns during the evolution of clones with an intention to identify the unwanted inconsistent changes. From a case study of three open source systems, they found many clones that were changed more than once. They also found few instances of unintentional inconsistent changes. However, they did not find any relationship between the consecutive change patterns and such unwanted inconsistencies.

Stability of Cloned Code

To measure the stability of cloned code, Krinke carried out a case study [62] on five open source software systems considering 200 revisions. He observed that if the dominating factor of deletions is eliminated, it can generally be concluded that cloned code is more stable than non-cloned code, and thus requires less maintenance effort. In another study, Krinke [63] exploited the advantages of the subversion system (SVN) to measure the average last change dates of the cloned and non-cloned code of a subject system. He considered only exact clones, and showed that cloned code is older than non-cloned code on average, which further supports the conclusion of his previous investigation that cloned code is more stable than the non-cloned code.

Göde and Harder [34] replicated and extended Krinke's study [62] using their incremental clone detection technique, iClones [36] to validate the outcome of his study. They supported Krinke by assessing cloned code to be more stable than non-cloned code in general, while this scenario reverses with respect to deletions. Their investigation regarding the impacts of parameters on comparative code stability reveals that, larger clones seem to be more stable with respect to changes while more unstable with respect to additions. They also observed that the general intention behind the deletion of cloned code was not to remove duplication but to perform restructuring and cleanup activities.

In a recent study, Hotta et al. [43], calculated the modification frequencies of cloned and non-cloned code to measure the impact of clones on software maintenance, and concluded that the modification frequency of non-cloned code is higher than that of cloned code, which also implies that cloned code is more stable than non-cloned code.

Although Krinke [62, 63] and Hotta et al. [43] both concluded that cloned code is more stable than the non-cloned code, some of their investigated subject systems do not agree with their conclusions. In order

to investigate why the results differ, recently we [78] have performed a comprehensive empirical study that aims to analyze the comparative stability of cloned and non-cloned code using three methods associated with respective set of stability measurement metrics. From our four-dimensional investigation on 12 diverse subject systems written in three programming languages considering three clone types, it reveals that clones in Java and C systems are not as stable as clones in C# systems. Furthermore, a systems development strategy might play a key role in defining its comparative code stability scenarios.

In order to investigate the relationship between code clones and maintenance effort, Lozano et al. [71] compared measures of the maintenance effort on methods having clones against those have not. Their initial results suggest that functions having clones change more often than the functions which do not have clones. Later, by conducting a more comprehensive case study with four open source Java projects, Lozano and Wermelinger [69] found that having a clone may increase the maintenance effort of changing a method. They also found that some methods seem to significantly increase their maintenance effort when a clone was present. However, they concluded that there seems to be no systematic relation between cloning and such maintenance effort increase.

Change Anomalies

Aversano et al. [5] investigated the change patterns of code clones when bug fixing takes place. They found 17 bug fixes that involved changes to code clones. Among them four changes were consistent, whereas, six changes were classified as independent evolution because the bug was only corrected in some of the clones. The rest seven changes were classified as late propagation.

In order to examine the characteristics of late propagation in more detail, recently Barbour et al. [9] have conducted an empirical study on two open source Java systems, where they considered only Type-1 and Type-2 clones. Their studies conclude that in general late propagation is more risky than other clone genealogies. Then they classified late propagation into eight categories based on the sequence of changes, and identified which changes contributed most to faults. Their study reveals that when a clone experiences inconsistent changes and then a resynchronizing change is applied without any modification to the other clone in a clone pair, and when two clones undergo an inconsistent modification followed by a re-synchronizing change that modifies both the clones in a clone pair are more risky than others.

Bakota et al. [7] analyzed suspicious changes to code clones, called clone smells, to identify potential problems. They defined four distinct clone smells: *Vanished Clone Instance (VCI), Occurring Clone Instance (OCI), Moving Clone Instance (MCI)*, and *Migrating Clone Instance (MGCI)*. While the *VCI* and *OCI* are same as the *Delete* and *Add* change pattern respectively as described in Section 2.2.1, the *MCI* happens when a fragment that moves to another clone class in a later version, and the *MCGI* happens when the moved clone fragment in *MCI* pattern comes back to the previous clone class in a later version. Then they searched for these four patterns in 12 versions of Mozilla Firefox and manually investigated whether these change patterns were related to bug or not. Among the 60 clone smells they identified, six changes were

actually related to bug. Finally, they concluded that in some cases, clones are harmful since they facilitate incomplete bug removal.

Bettenburg et al. [13] performed an empirical study on the effect of inconsistent changes on software quality at the release level. Based on a case study on two open source software systems, they observed that only 1% to 3% of inconsistent changes to clones introduce software defects.

Chapter 3

Evaluating Code Clone Genealogies at Release level: An Empirical Study

3.1 Motivation

Previous studies were highly influenced by the idea that clones are harmful and can be removed through refactoring [56]. This conventional wisdom was first challenged by Kim et al. [56]. They provided a clone genealogy model and analyzed clone genealogies of two open source software systems. Their empirical study on code clone genealogies reveals that clones are not always harmful. Programmers intentionally practice code cloning to achieve certain benefits [52, 55]. During the development of a software system, many clones are short lived. Refactoring them aggressively can overburden the developers. Their study also shows that many long-lived consistently changing clones are not locally refactorable. Such clones cannot be removed from the system through refactoring [56].

We are motivated by the work of Kim et al. [56]. They were the first to analyze clone genealogies. However, they only analyzed two small Java systems. They also speculated that the selected systems might not have captured the characteristics of larger systems and thus, further empirical evaluations need to be carried out for larger systems of different languages. Later several other researchers also investigated the maintenance implications of clones. Kapser and Godfrey [52] conducted several studies in the area and showed that clones might not always be harmful and even could be useful in a number of ways. Krinke [62, 61] studied change types and the stability of code clones based on the changes between the revisions of several open source systems. Although he analyzed several systems written in C, C++ and Java, he did not focus on evaluating clone genealogies. Bettenburg et al. [13] analyzed inconsistent changes of code clones to determine their contribution to software defects. They also noted the importance of a release level empirical study compared to that at the revision level.

To the best of our knowledge, no further extensive empirical evaluations have been carried out to examine the code clone genealogies with different languages or variable program sizes. In this study, we followed the footsteps of Kim et al. [56] by conducting an in-depth empirical study on the evaluation of clone genealogies in 17 open source systems covering four popular programming languages, C, Java, C++ and C#. However, unlike Kim et al. [56], we did not work at the revision level; rather, we analyzed the evolution of clones at

the release level since they are less affected by short term experimentations of the developers in the software development process [13]. The systems are selected from different areas and have rich development histories. In particular, we focus on the following two research questions:

1. How do the clone genealogies look like in open source software written in different languages and of different sizes with variable release histories?

2. Do clone genealogies at the release level share any common quantitative characteristics, and do any particular type of genealogies exhibit higher longevity than the others?

With an extensive study of 17 open source systems written in four different languages, we have reached the following conclusions:

1. Most of the clone classes are propagated through subsequent releases either without any changes or with changes only in identifier renaming. Many of them reach to the final releases of the subject systems and contribute to the number of alive genealogies. We have found that, on average about 67% of the genealogies among all systems do not have any addition or deletion of lines or any syntactic changes. Moreover, an average of roughly 69% of these syntactically similar genealogies reach to the final releases.

2. We have observed that from about 11% to 38% of the genealogies were changed consistently over the entire course of the evolution.

3. Among the dead genealogies, many of them were removed within a few releases.

4. Clone evolution is not highly affected by development languages or project sizes.

The rest of the chapter is organized as follows. Section 3.2 outlines the study approach. In Section 3.3, we describe the experimental setup and then present the results of the case study in Section 3.4. Section 3.5 describes the threats to validity of our study and in Section 3.6 we discuss some other studies related to ours. Finally, Section 3.7 concludes the chapter with our next steps.

3.2 Study Approach

Our primary objective is to study how code clones evolve over different releases during system evolution in terms of genealogy. In addition to this, we also want to investigate whether the findings by Kim et al. [14, 15] based on two small Java systems also hold for other systems of diverse varieties, varying system sizes and systems written in different programming languages. Our objective is not to validate the findings of Kim et al. by replicating the same experiment with exactly the same settings, rather we wanted to examine how code clones evolve in software systems of varying sizes written in different programming languages using their clone genealogy model. Thus, we develop a clone genealogy extractor similar to theirs except that the location overlapping function is replaced by a snippet matching algorithm. Kim et al. developed a *diff* based

location tracker that maps the line numbers of a snippet to its old line numbers in the previous release. They also discussed that the location overlapping function did not work well when lines are modified or reordered in a file because *diff* cannot capture such changes. The purpose of the location overlapping function was to find out the exact mapping of a clone class from the previous release to the next. To fulfill the same objective we have developed a location independent approach, snippet matching function that maps a clone class from the previous release to its next based on identifier matching. The following paragraph discusses how our modified Clone Genealogy Extractor (CGE) works.

3.2.1 Clone Genealogy Extractor

Our clone genealogy extractor automatically extracts clone genealogies across the releases of a program. The steps are summarized as follows: (1) first, we collect multiple releases of a program and then sort them in chronological order; (2) second, we run CCFinderX[1] on all these releases with a batch processor; (3) third, we collect the clone class information on each release produced by CCFinderX, and (4) finally, the output and the intermediate files generated by CCFinderX are then used as input for the CGE.

In order to map clone classes of successive releases, the CGE uses both *TextSimilarity* and *SnippetMatching* functions as described below. The CGE maps clone classes based on the highest text similarity and snippet matching scores. If the highest text similarity score is different from the highest snippet matching score, the heuristic selects both of them to avoid ambiguity. That is why a genealogy may have more than one lineage. Figure 3.1 represents such an example clone genealogy that consists of three clone lineages marked with different line styles. All the three lineages evolve from the same clone class that consists of three code snippets (A, B, C) and is called the source of the lineages. Each clone lineage describes how a sink node evolves from the source node. For example, the sink of one of the clone lineages that consists of two code snippets (E, G) evolves from the source node with addition and inconsistent changes, subtraction and inconsistent change, addition and consistent changes evolutions patterns through the release history. Thus, a clone genealogy captures the evolution of a clone class through the release history, and all the lineages that belong to a clone genealogy originated from that clone class. The following subsections describe how the *TextSimilarity* and *SnippetMatching* techniques work.

3.2.2 Text Similarity

The text similarity between two code snippets C_1 and C_2 is determined by calculating the common token sequence with respect to their token sizes. By considering tokens generated by CCFinderX, we count the textual matches across releases. Equation 3.1 below describes the TextSimilarity function. Here $|C_1|$ and $|C_2|$ are the number of tokens in C_1 and C_2 respectively. $|C_1 \cap C_2|$ is the number of common ordered tokens between C_1 and C_2 , calculated using the longest common subsequence (LCS) algorithm. In order to have

[1] http://www.ccfinder.net

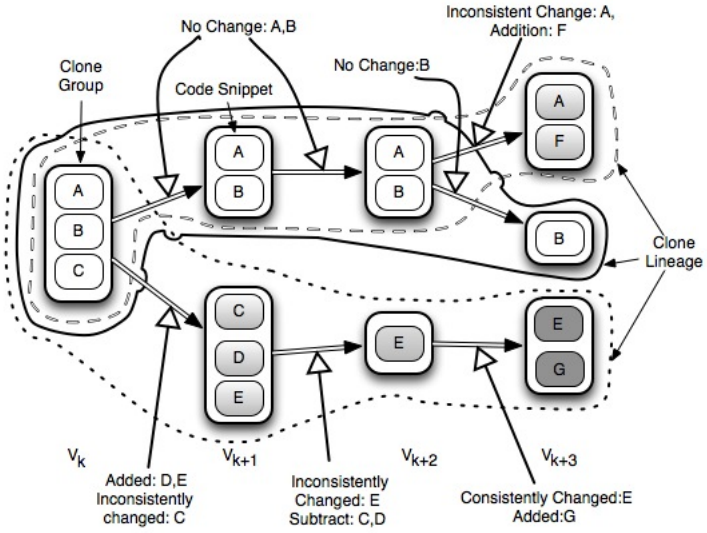


Figure 3.1: A clone genealogy

consistency with Kim et al. [56], we used a text similarity heuristic of 0.3. With this similarity threshold, the length and size of the genealogies are neither overestimated nor underestimated [56].

$$Text\ Similarity(C_1, C_2) = \frac{2 \times |C_1 \cap C_2|}{|C_1| + |C_2|} \tag{3.1}$$

3.2.3 Snippet Matching

By applying the text similarity heuristic, we can eliminate many uninteresting mappings that are not syntactically similar. However, the text similarity score itself is not always enough to get a better result. In snippet matching, on the other hand, we match the snippets based on the similarity of identifiers. The text similarity function produces a higher value than the given threshold for all of the mappings that are syntactically similar. However, in such cases, it is highly probable that they have different identifier names. The snippet matching algorithm is applied on all the mappings produced by the text similarity function above. The algorithm takes two code fragments and produces a value between 0 and 1 to reveal how much these snippets are identical by their identifier names. We first extract the identifiers from each of the snippets and then apply LCS algorithm on them to find the matching score using equation 3.2.

$$SnippetMatch(S_i, S_j) = \{\frac{LCS(IS_i, IS_j)}{len(IS_i)} + \frac{LCS(IS_i, IS_j)}{len(IS_j)}\}/2 \tag{3.2}$$

where, $IS_i = \{$set of identifiers in snippet, $S_i\}$, $IS_j = \{$set of identifiers of snippet, $S_j\}$, and $LCS(IS_i, IS_j)$ = {Longest common subsequence for the identifiers of the snippets S_i and S_j}.

It is possible that some of the identifiers might be common between two code snippets of two different clone classes of two successive releases, but it is unlikely that they maintain the same sequence and produce

a higher similarity value. Again, it is possible that some identifiers might be renamed in the next release. In such cases, the same snippets in two releases might produce very low snippet matching similarity value. To overcome such situations, we calculate the snippet matching values for all possible pairs between two clone classes of two successive releases and take the one with the maximum similarity value. There is a threat to this approach in the cases where all the identifier names of all snippets in the same clone class are changed/renamed in the next release. However, in our experience, such a situation is very unlikely to occur.

Now for each system, we have collected the total number of genealogies including the number of alive and dead genealogies. We then study what proportion of the genealogies are changed consistently and what proportion of them remain syntactically same.

3.3 Experimental Setup

In this section we provide a brief overview of the systems we have studied, and the clone detection tool we used for the experiment.

3.3.1 Subject Systems

We studied 17 open source software systems[2] covering four different programming languages, C, C++, Java and C# as shown in Table 3.1. The sizes of these systems range from approximately 9K to 204K source lines of code (SLOC), excluding comments and blank lines. The systems are selected from different domains such as text editor, email client, graphics library, test framework and so on.

3.3.2 Clone Detection

We used the AIST CCFinderX[3] to detect code clones in each release. CCFinderX is a major revision of CCFinder [51]. CCFinderX is instructed to detect clones with TKS (minimum number of distinct types of tokens) set to 12 (default setting). In order to detect clones of large enough for practical significance, we set the minimum token length to 30. The same value for the minimum token length was also used in many other research projects in the past [56].

3.4 Study Results

This section presents the results of our study. Since the subject of this study is code clone evolution in terms of clone genealogy, at first we characterize different types of genealogies and then discuss our findings pertaining to them.

[2]http://sourceforge.net
[3]http://www.ccfinder.net

Table 3.1: Subject Systems

Language	Subject System	SLOC	Duration	No. of Releases
Java	JUnit	2,179 - 8,785	2003-05-12 to 2009-12-08	20
	CAROL	2,812- 11,694	2002-11-12 to 2005-04-13	10
	dnsjava	11,025- 23,334	2001-03-29 to 2009-11-21	22
	JabRef	11,352- 74,104	2003-11-30 to 2010-04-14	33
	iText	51,860- 82,164	2002-03-07 to 2008-01-25	49
C++	KeePass	14,789- 43,644	2003-11-17 to 2006-10-14	35
	Notepad++	26,937- 81,980	2003-11-25 to 2007-02-04	30
	7-Zip	71,638- 100,823	2003-12-11 to 2009-02-03	45
	eMule	6,803- 203,780	2002-07-07 to 2010-04-07	73
C	Wget	14,209- 40,021	1998-09-23 to 2009-09-22	17
	Conky	7,029-42,060	2005-07-20 to 2010-03-30	70
	ZABBIX	12,468- 70,890	2004-03-23 to 2010-01-27	28
	Claws Mail	126,247- 203,783	2005-03-19 to 2010-01-31	47
C#	NAnt	686-52,533	2001-07-19 to 2007-12-08	22
	iTextSharp	33,545-163,890	2003-02-04 to 2007-03-08	26
	Process Hacker	10,349-123,878	2008-10-17 to 2010-01-23	38
	ZedGraph	2,439-26,433	2004-08-02 to 2008-12-12	28

3.4.1 Clone Genealogy

This subsection characterizes the evolution of clone classes in terms of genealogies. We will focus on four types of genealogies, (1) alive genealogy, (2) dead genealogy, (3) syntactically similar genealogy, and (4) consistently changed genealogy in order to discuss the evolution characteristics. We have already given the definitions of dead, alive, consistently changed genealogies in Section 2.2.1. Here the term *syntactically similar genealogy* (SSG) refers to those genealogies in which the clone classes are propagated through subsequent releases either without any changes or with changes only in formatting and identifiers (e.g., renaming of identifiers) in their code snippets. No lines are added or deleted in the snippets. However, cloned snippets could be moved from one location to another in same file of the subsequent releases. Table 3.2 presents the total number of genealogies and the proportions of the four types of genealogies mentioned above.

From Table 3.2 we see that the proportions of alive and dead genealogies are not largely affected by programming languages or program sizes. For Java, C and C++ systems, the values are very close. The proportions of alive genealogies of these systems vary from 69% to 72% whereas C# systems contain almost 76% of alive genealogies, the highest among the four languages. On the other hand, when we examined the subject systems in terms of program size (Table 3.3) we see that in general, the average proportions of alive genealogies increased with the increase of program size. It means more genealogies disappeared from the small systems compared to that of larger ones, which suggests that perhaps clones are more manageable in systems with a smaller size compared to a larger one. Thus, a clone tracking and maintenance tool might be more effective for larger systems. In the following subsections we will have a closer look at the four types of genealogies.

3.4.2 Consistently Changed Genealogy

From Table 3.2 we see that the number of consistently changed genealogies varies from 10.43% to 38.30% for the subject systems. The average number of consistently changed genealogies varies in terms of program size (17.71% to 24.6%) or implementation language (16.84% to 26.18%). As we see the variations are not too drastic and do not reveal any systematic change pattern. However, from our study we see that the number of consistently changed genealogies is not very high (on average 24.28%).

Among the subject systems, `Carol` and `dnsjava` were analyzed by Kim et al. [56]. Even though they studied at the revision level and we studied at the release level, we observed a similar proportion of consistently changed genealogies in Carol. However, there is a bit difference in the number of genealogies detected. They found 122 genealogies from which 13 were eliminated due to template based programming, whereas, we found 141 genealogies. It should be noted that we did not consider template based programming because we believe that such clones are nevertheless clones. Moreover, we have considered release level candidates and applied a combination of snippet matching and text similarity algorithms (discussed earlier). For `dnsjava`, on the other hand, we experienced a significant difference from them. Possible reasons could be that Kim et al. [56]

Table 3.2: Clone Genealogies

Subject System	**# Gen**	**AG** [%]	**DG** [%]	**SSG** [%]	**CCG**[%]
JUnit	127	78.74	21.26	81.89	15.75
Carol	141	44.68	55.32	56.73	38.30
dnsjava	417	82.97	17.03	85.37	12.23
JabRef	1132	73.41	26.59	66.25	26.06
iText	1568	68.75	31.25	74.62	20.22
Avg. of Java Systems		71.43	28.57	72.67	21.77
KeePass	790	70.76	29.24	73.54	20.63
Notepad++	977	81.99	18.01	73.69	19.86
7-ZIP	1427	65.38	34.62	64.62	24.46
Emule	3547	66.08	33.92	59.57	29.86
Avg. of C++ Systems		68.79	31.21	64.32	26.18
Wget	206	57.77	42.23	60.68	28.64
Conky	1328	53.69	46.31	82.45	11.97
ZABBIX	1026	65.79	34.21	49.31	28.65
Claws Mail	2363	85.57	14.43	63.26	27.08
Avg. of C Systems		71.68	28.32	65.43	23.40
NAnt	625	76.96	23.04	55.20	34.08
iTextSharp	2666	77.16	22.84	86.38	10.43
Process Hacker	950	71.79	28.21	73.26	20.32
ZedGraph	374	76.74	23.26	62.83	24.87
Avg. of C# Systems		76.00	24.00	77.55	16.84
Avg. of all Systems		70.33	29.67	66.56	24.28

Table 3.3: Distribution of Genealogies by Program Size

Subject System	**AG** [%]	**DG** [%]	**SSG** [%]	**CCG**[%]
<50K	64.65	33.35	76.15	17.71
50K-100K	71.04	28.96	65.32	24.60
>100K	74.58	25.42	69.36	22.78

considered revisions until November 2004 whereas we studied releases until November 2009, and some major changes took place in the code-base of `dnsjava` in May 2005. This might have caused many new clones, and most of the new clone classes were propagated to the final release contributing to the higher proportion of alive genealogies.

3.4.3 Alive Genealogies

In this study, we have found that a substantial proportion of genealogies of all systems are alive, which is 70.33% of total genealogies on average (Table 3.2). For example, out of 3547 genealogies in `Emule`, 2344 have at least one clone class in the final release, thus about 66% of total genealogies in `Emule` are counted as alive. For `dnsjava`, `Notepad++`, and `Claws Mail` the proportions of alive genealogies are even more than 80%. The only exception is `Carol`, in which nearly 45% of all genealogies are found alive. The `Carol` project is now closed and a lot of refactoring was done in the final release [6], which is probably a reason for this relatively low number of alive genealogies compared to others.

One possible reason behind this large number of alive genealogies is that a significant number of clone classes were created in just a couple of releases prior to the final release, and they are counted as alive since it is unknown when they will be removed in the future releases. Table 3.4 presents the total number of alive genealogies, genealogies that are created within final five releases and the alive genealogies that survive more than half of the release histories for each system. From the table we get a fairly complete picture of alive genealogies including their lifetimes.

3.4.4 Syntactically Similar Genealogies

We further investigate what proportion of clone genealogies remains syntactically the same throughout the evolution. It is important to study such SSGs because clone classes of these genealogies seem stable during the evolution, and thus one may not need any extra care for them (because where there is probability of change, there is a fear of inconsistent changes). Thus, aggressively refactoring them might not be worthwhile. We have noticed that an enormous proportion of clone genealogies are syntactically similar, and on average 66.56% of all the subject systems (Table 3.2). The highest proportion of syntactically similar genealogies is found in `iTextSharp`, roughly 86%, whereas the lowest is nearly 50% for `ZABBIX`. If we look at them by language (Table 3.2) we see that the numbers of such genealogies in C and C++ systems are lower than the systems of other two languages. About 64.32% and 65.43% of genealogies are syntactically similar for C++ and C systems respectively whereas for the systems of the other two languages the value varies from 72.67% to 77.55%. We also noticed variations in terms of program sizes (Table 3.3). In particular, systems with sizes ranging from 50K to 100K LOC show fewer syntactically similar genealogies compared to the systems of the other two size ranges.

We further examine whether there are any relationships between these syntactically similar genealogies and alive genealogies. From Table 3.5, we notice that on average 69.04% of syntactically similar genealogies

Table 3.4: Alive Genealogies

Subject System	**AG**	**AG created within last five releases**	[%]	**AG that survived more than half of release histories**	[%]
JUnit	100	31	31	68	68
dnsjava	346	15	4.34	82	23
Carol	63	47	74.60	17	26.98
JabRef	831	72	8.66	400	48.13
iText	1078	490	45.45	765	70.96
KeePass	559	163	29.16	241	43.11
Notepad	801	59	7.36	587	73.3
7Zip	933	34	3.64	678	72.66
Emule	2344	385	16.42	365	15.57
Wget	119	16	13.44	74	62.18
Conky	713	490	68.74	136	19.07
ZABBIX	675	11	1.62	467	69.18
Claws Mail	2022	46	2.29	1335	66.02
NAnt	481	275	57.17	62	12.89
iTextSharp	2057	864	42	1094	53.18
Process Hacker	682	236	34.60	158	33.17
ZedGraph	287	23	8.01	174	60.63
Avg. of all systems	-	-	23.11	-	47.57

Table 3.5: Syntactically similar clone genealogies

Subject System	**Alive SSG**	**% of alive SSG of total SSG**	**% of alive SSG of total AG**
JUnit	85	81.73	85.00
Carol	34	42.50	53.97
dnsjava	310	87.08	89.60
JabRef	530	70.67	63.78
iText	811	69.32	75.23
Avg. of Java Systems		71.95	73.20
KeePass	402	69.19	71.91
Notepad++	568	78.89	70.91
7ZIP	586	63.56	62.8
Emule	1262	59.73	53.83
Avg. of C++ Systems		64.99	60.77
Wget	60	48.00	50.42
Conky	590	53.88	82.74
ZABBIX	279	55.13	41.33
Claws Mail	1220	81.60	60.33
Avg. of C Systems		66.72	60.90
NAnt	235	68.12	48.86
iTextSharp	1734	75.29	84.30
Process Hacker	505	72.56	74.04
ZedGraph	175	74.47	60.98
Avg. of C# Systems		74.02	75.53
Overall		69.04	66.61

Table 3.6: Syntactically similar clone genealogies by program size

Program Size	**% of alive SSG of total SSG**	**% of alive SSG of total AG**
<50K	64.29	75.72
50K-100K	68.18	62.70
>100K	71.45	66.44

reached to the final releases of the subject systems. On the other hand, on average about 66.61% of alive genealogies did not change syntactically throughout their entire lifetimes. These indicate that most of the clone classes that do not change syntactically are unlikely to be removed during the evolution of the software systems. SSGs are cost-effective in the sense that they require little or no maintenance effort. Instead of aggressively refactoring them, we may track the evolution of such clones so that we can differentiate them from other types of genealogies, those may require more care. In terms of program size, the proportion of syntactically similar alive genealogies over SSG increases with the increase of program size (Table 3.6). It means more SSGs were propagated to the final releases in larger systems than those of smaller ones. This implies that possibly for smaller systems developers can handle clones more effectively than that for larger ones. However, no strong change relationship was observed for the proportions of alive SSGs over the total number of alive genealogies.

3.4.5 Dead Genealogies and Volatile Clones

We were also interested to see how long dead genealogies survive in the systems in terms of the number of releases. For this purpose, we used the term *k-volatile genealogy*, which refers to a dead genealogy that disappears within k versions.

To visualize this scenario, we used the same approach defined by Kim et al. [15] as follows:

Let, $f(k)$ denote the number of genealogies with age k , $f_{dead}(k)$ denote the number of dead genealogies with age k, $CDF_{dead}(k)$ denote the cumulative distribution function of $f_{dead}(k)$ and it is the ratio of *k-volatile* genealogies among all dead genealogies. $R_{volatile}(k)$ denotes the ratio of *k-volatile* genealogies among all genealogies in a system.

Figure 2(a-d) represents $CDF_{dead}(k)$ and $R_{volatile}(k)$ for the largest and smallest subject systems for each of the language categories. Here, the horizontal axes represent the ages of the genealogies in terms of releases and vertical axes represents the values of $CDF_{dead}(k)$ or $R_{volatile}(k)$.

Figures 2(a) and 2(c) represent the $CDF_{dead}(k)$ and $R_{volatile}(k)$ for the largest systems of each of the language categories respectively. The largest Java system is `iText`. We see from the graph that for this system, 16% of all dead genealogies (5% of all genealogies) disappeared within six releases. In `Claws Mail` (largest C system), 28% of all dead genealogies (5% of all genealogies) disappeared within five releases, and within 10 releases roughly 50% of all dead genealogies (7% of all genealogies) disappeared. For `eMule` (largest C++ system), 33% of all dead genealogies disappeared within only five releases. For the largest C# system, `iTextSharp` we found that the initial value for $CDF_{dead}(k)$ and thus also $R_{volatile}(k)$ to be smaller compared to the other systems. The possible reason behind this difference is that a higher number of dead genealogies (in total 382) span over 19 releases, which is more than 50% of all dead genealogies.

The same attributes for the smallest systems of each language categories are provided in Figs. 2(b) and 2(d). The smallest Java system in our study is `JUnit`. We found that all the dead genealogies (about 21% of all genealogies) of this system disappeared within six releases from when they were created.

`KeePass Password Safe` is the smallest C++ system with 43K LOC in its final version. Among the dead genealogies for this system, 12% disappeared within five releases. The smallest C system, `Wget` also shows a similar trend but with a much higher ratio. In this particular scenario, 60% of all dead genealogies (25% of all genealogies) disappeared within only six releases and about 97% of all dead genealogies (40% of all genealogies) disappeared within 10 releases. When we plot the same attribute for `ZedGraph` (smallest C# system), we found that this system maintains a similar trend (12% of all genealogies and approximately 52% of all dead genealogies disappeared within five releases).

The above data did not reveal any systematic relationship between $CDF_{dead}(k)$ and $R_{volatile}(k)$ for the language categories. However, we have found that even at the release level, the number of volatile clones was not negligible. Moreover, many of them propagate through subsequent releases without any changes. These findings indicate that aggressive refactoring is possibly not a cost-effective solution for such clones and may call for alternative measures such as tracking and managing them in their evolution.

3.5 Threats to Validity

One of the major possible confounds to this study is that the clone detector we used might have missed certain clones in the systems (false negatives) or detected clones that are not clones in practice (false positives). We used CCFinderX with settings (minimum token length of 30 and minimum token set size of 12) that allow it to detect clones of reasonable size. Although with this setting, some clones might have been missed or some false positive clones might have been considered, we have chosen to use CCFinderX in our study to be consistent with the study of Kim et al. [56] since one of our research objectives was to investigate whether software systems of different languages and of different sizes and varieties show similar trends at the clone genealogy level to that observed by Kim et al. Moreover, CCFinder is recognized as a state of the art clone detector having high recall, although its precision is lower than some other tools [12].

A major part of this study is to map the clone classes from one release of a system to the next for extracting clone genealogies. While we have manually verified all the clone genealogies of some small systems, it was very difficult to manually verify the genealogies for all the systems. In our experience, although we did not find any false positive mappings (at least within our given settings and heuristics) except a few due to CCFinder finding false positive clones, we cannot guarantee that there are no false positive mappings in the results.

Another possible confound to this study is the limited number of samples. However, to our knowledge this is the first study on the maintenance implications of clones, and in particular on evaluating clone genealogies that considers 17 open source systems of different languages of diverse varieties. Since, all the systems in our study are open source, one may argue that a similar study on industrial systems may produce different results.

3.6 Related Work

In recent years, studying the maintenance implications of clones, which is also one of the objectives of this study, has become an active research topic. Kapser and Godfrey [54] conducted large-scale empirical studies and concluded that clones are not necessarily harmful and found several patterns of clones that could be useful in many cases. Juergens et al. [49], on the other hand, argued that unintentionally created inconsistent clones always leads to faults, and concluded that clones could be harmful in software maintenance. While we also studied the maintenance implications of clones, our study significantly differs from theirs in the sense that they did not study the evolution of clones.

Krinke [61] analyzed many revisions of five open source software systems and found that half of the changes to code clone classes are inconsistent and that corrective changes following inconsistent changes are rare. In another study, he found that cloned codes are more stable than non-cloned codes and thus require less maintenance effort compared to non-cloned code [62]. Our study differs from his in that we work on releases instead of revisions, and that we particularly focus on evaluating clone genealogies.

Bettenburg et al. [13] studied the inconsistent changes of clones at the release level. They noted that the number of defects through inconsistent changes is possibly substantially lower at the release level than at the revision level. They reported that many clones are created during the software development process due to the experimentation of developers, which the developers can manage well. Thus they worked at the release level instead of the revision level. In order to avoid the affect of such short-term clones, we also choose to work at the release level. However, while they focus on finding the relation of inconsistent changes to software defects for two open source systems, we particularly focus on evaluating clone genealogies using 17 open source systems written in four different languages.

Lozano et al. [69, 71] conducted several studies on the maintenance implications of clones. While they could not find any systematic relation between cloning and maintenance efforts, they concluded that change efforts might increase for a method when it has clones. Although the underlying clone detection tool is the same as ours, the approach is different from ours in many aspects. In particular, they work on the revision level, whereas we work on the release level and that they focus on the changes at the function level, whereas we focus on the clone level itself. Moreover, they studied only Java systems, which might have also affected the findings.

Göde [32] proposed a computationally efficient approach that models Type-1 clone evolution based on the source code changes made between consecutive program versions of several open source systems. While he concluded that the ratio of clones decreased in the majority of the systems and clone fragments survived more than a year on average, no general conclusion on the consistent or inconsistent changes to clone classes was proposed. Our work differs from his in several ways. In particular, he used an incremental clone evolution model and only considered Type-1 clones whereas we considered both Type-1 and Type-2 clones, and that he worked at the revision level whereas we worked at the release level. Bakota et al. [7] proposed a machine

learning approach for detecting inconsistent clone evolution situations and found different bad smells using twelve versions of Mozilla Firefox. However, they studied the evolution patterns of clone fragments whereas we studied clone classes, and they worked at the revision level (and only 12 monthly revisions of Mozilla Firefox) whereas we studied release versions of many systems written in different languages.

Thummalapenta et al. [96] performed an empirical evaluation on four open source C and Java systems for investigating to what extent clones are consistently propagated or independently evolved. While they focused on identifying evolution of cloned codes over time and relating the evolution pattern with other parameters (clone granularity, clone radius and cloned code fault-proneness), we focus on evaluating clone genealogies with 17 open source software systems covering four popular programming languages.

The most closely related work to ours is the study of Kim et al. [56], which is also one of the motivations of our study. However, they studied only two small Java systems and at the revision level. On the other hand, we studied at the release level and with 17 diverse varieties of open source systems written in four different programming languages. Furthermore, instead of location mapping, we have used snippet matching together with text similarity for mapping the clone classes from one version to the next. This allows us to map clone classes even when lines are modified or reordered in the next version. Aversano et al. [5] extended the clone evolution model of Kim et al. [56] by grouping inconsistent changes to independent and late evolution classes. Again, they studied only two open source Java systems namely `ArgoUML` and `dnsjava` and reported contradictory findings for the consistently changed clone classes.

3.7 Summary

In this chapter, we have presented an empirical study for evaluating code clone genealogies using 17 diverse categories of open source software systems written in four different programming languages. We have set up our experiment based on the genealogy model of Kim et al. [56] and extended their empirical study in different dimensions. While Kim et al. concentrated on the consistently changed genealogies, and the nature of volatile clones by analyzing two small Java systems, we attempted to draw a more detailed picture of clone genealogies by analyzing a larger number of systems, and systems written in different development languages, systems of varying size, and systems with varying development histories. Kim et al. found that (at the revision level) from 36% to 38% of genealogies were changed consistently, whereas we have found that (at the release level) from 11% to 38% of genealogies were changed consistently, which does not seem contradictory. Again, they reported that volatile clones were disappearing within a short time from the systems and noted that almost from 48% to 72% of volatile clones were disappearing within eight check-ins. We also found that even at the release level many volatile clones disappear within a few releases. In addition, our study reveals some other interesting characteristics of code clone genealogies. We have found that for all subject systems, many genealogies are alive and long-lived, which implies that more clone classes are created than those that are removed. In most of the genealogies for the subject systems, clone classes are propagated

through releases either without any change or with changes just in identifier renaming. Hence, it is possible that these types of genealogies do not need any extra care during maintenance. Also, they are less likely to be removed from the systems, and on average almost 69% of them reached to the final release. Moreover, on average nearly 67% of total alive genealogies did not contain any line additions or deletions or identifier renaming. Since we have studied a variety of systems, the results also indicate that it is possible that such a trend holds even when the systems are implemented in different languages, are from different areas and are of different sizes. We have noticed that clones are perhaps more manageable in smaller systems compared to larger ones.

In summary, this study confirms some of the the previous findings of Kim et al.[56] and extends the existing knowledge of clone evolution in several ways for Type-1 and Type-2 clones. However, we know a very little about how Type-3 clones evolve in systems. In the next chapter we propose an automatic framework, gCad, for extracting and classifying all the three types (Type-1, Type-2, and Type-3) of clone genealogies to analyze their evolution.

Chapter 4

An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies

4.1 Motivation

Accurately mapping clones between versions of a program, and classifying their change patterns are the fundamental tasks for studying clone evolution. Researchers have used three different approaches to map clones across multiple versions of a program. In the first approach [7, 56, 91], clones are detected in all the versions of interest and then clones are mapped between consecutive versions based on heuristics. In the second approach [5], clones are detected in the first version of interest, and then they are mapped to consecutive versions based on change logs provided by source code repositories such as *svn*. In the third approach [36, 81], clones are mapped during clone detection based on source code changes between revisions. A combination of the first and second approaches has also been used in some studies [13].

Although intuitive, each of these approaches has some limitations. In the first approach, a number of the similarity metrics used to map clones have quadratic time complexities [40]. In addition, if a clone fragment changes significantly in the next version and goes beyond the given similarity threshold of the clone genealogy extractor, a mapping may not be identified. In the second approach, only clones identified in the first version are mapped. Therefore, we do not know what happens to clones introduced in later versions. The third approach ("incremental approach") avoids some of the limitations of the previous two approaches by combining detection and mapping, and works well for mapping clones in many versions. By integrating clone detection and clone mapping this approach can be faster than the approaches that require clone detection to be conducted separately for each version. Although this incremental approach is fast enough both for detection and mapping for a given set of revisions, it might not be as beneficial at the release level [36] because there might be a significant difference between the releases. Furthermore, in the sole available incremental tool, *iClones* [36] (available for academic purposes), when a new revision or release is added for mapping, the whole detection and mapping process should be repeated since clones are both detected and mapped simultaneously. Clone management is likely being conducted on a changing system, and it is a disadvantage for an approach to require detecting clones for all revision/versions each time a new revision/version is added. Another issue with the incremental mapping is that it cannot utilize the results

obtained with a classical non-incremental clone detection tool as the detection of clones and their mapping is done simultaneously. Most of the existing clone detection tools are non-incremental. There is also no representative tool available. Depending on the task at hand and the availability of tools, one might want to study cloning evolution with several clone detection tools. It is thus important to have a clone evolution analysis tool in place independent of the clone detection tools. Scalability of the incremental approaches is another great challenge because of huge memory requirements.

Again, while most of these approaches [5, 7, 13, 56, 91] are based on the state of the art detection and mapping techniques, they only focused on Type-1 and Type-2 clones. Roy and Cordy [89] show that there are a significant number of Type-3 clones in software systems and thus extracting the genealogies of such clones and understanding their change patterns is equally important.

In this chapter, we propose a framework for extracting both exact (Type-1) and near-miss (Type-2 and Type-3) clone genealogies across multiple versions of a program, and identifying their change patterns automatically. The framework works with any existing clone detection tool that represents a clone fragment by its file name and begin-end line numbers. Genealogies are constructed incrementally by merging current mapping results with previously stored genealogies to give a complete result. To validate the effectiveness of our proposed framework, we developed a prototype clone genealogy extractor (*CGE*), extracted both exact and near-miss clone genealogies across many releases of three open source systems including the Linux Kernel, adapted the *CGE* with other clone detection tools, and evaluated the correctness of the mappings reported by the prototype in terms of precision and recall. We also compared our result qualitatively and quantitatively with a result of an incremental clone detection tool, *iClones* [36]. The experimental results suggest that the proposed framework is scalable and can identify the evolutionary patterns automatically by constructing genealogies for both exact and near-miss clones. We name our prototype as gCad.

The rest of this chapter is structured as follows: Section 4.2 briefly describes the model of clone genealogy. In Section 4.3 we describe the proposed framework, whereas Section 4.4 outlines the details of our evaluation procedure. In Section 4.5 we compare our method with others. Section 4.6 discusses the threats to validity of our work. In Section 4.7 we discuss the related work to ours, and finally, Section 4.8 concludes the chapter.

4.2 Model of Clone Genealogy

Although we have already discussed the basic model of clone evolution in Section 2.2.1, in this section we discuss how we have adapted some of the parts of the model to fit it in dealing with the evolution of Type-3 clones. We have used the same definitions for *Add, Delete*, and *Same* patterns as described in Section 2.2.1 since they are based on appearances and disappearances of fragments only. However, the definitions of *Consistent Change* and *Inconsistent Change* patterns for Type-3 clones are not that straight forward because in a Type-3 clone class, there may be additions and deletions of lines between clone fragments. Therefore, it is an open question what would be the change pattern of a clone class if a change takes place in that part of

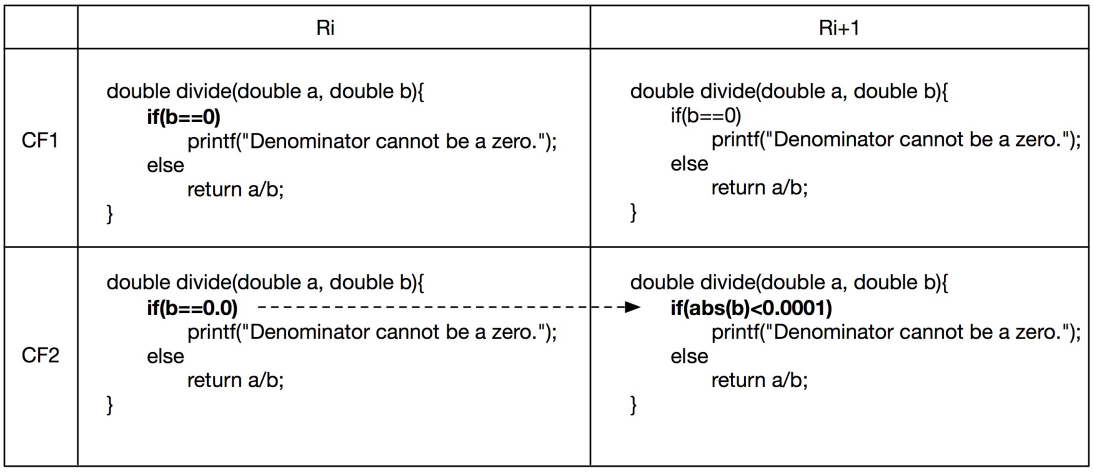


Figure 4.1: A Type-3 clone class having two clone fragments

a clone fragment which is not common with its siblings. The classification would be more complicated when a clone class involves more than two fragments. Let us explain the situation with two hypothetical examples.

First, let there are be fragments in a Type-3 clone class in release R_i as shown in Figure 4.1 that actually do a simple division operation. Although these two fragments are semantically the same, one line between these fragments is not textually the same. During the evolution of the program, a developer suddenly noticed that the *if condition* of one of these fragments is too precise which could result in an unexpected behaviour of the program when the value of b is very near to zero, and fixed the condition in release R_{i+1}. Although the same change is also required to the other fragment, the developer did not notice that. Now one may think that this is a consistent change since the change took place in the gap (where source lines are not common between fragments). On the other hand, one may argue that this is an inconsistent change since one clone fragment of that clone class changed whereas another did not. We might need to see the other fragment as well to verify whether the same change is needed to that fragment or not. And for the given example, the applied change does not eliminate the bug completely.

Second, if there are more than two fragments in a Type-3 clone class, the non-identical lines may not be the same for all clone pairs. Figure 4.2 represents such a clone class. Let the first, second, and third clone fragments of this clone class in release R_i be CF_1, CF_2, and CF_3 respectively. Now we see that although the modified line (shown using right arrow) of CF_2 is common with CF_3 but not common with CF_1. Therefore, we can conclude this change is a *consistent change* only considering that the change took place in a gap of clone pair CF_1 and CF_2. One of the solutions to overcome this problem is to determine the change patterns for all possible pairs in a clone class without considering the changes in gap. If all pairs change consistently, then the change pattern will be classified as consistent change. However, the threat explained in the previous example will be still present. Therefore, both of these approaches have some advantages and disadvantages. Thus, in our framework, we have considered both of these approaches as two different modes. We leave the

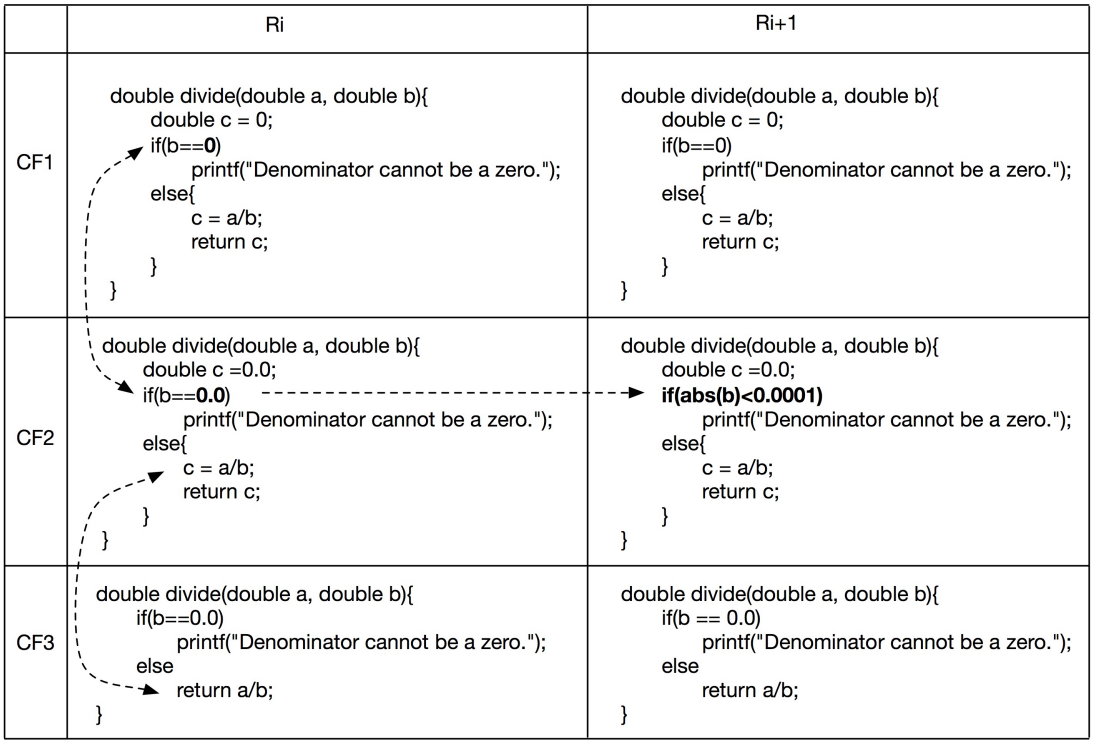


Figure 4.2: A Type-3 clone class having three clone fragments

choice to the maintenance engineers/researchers, as to which mode they would like to use according to their context of use. The two modes are as follows:

4.2.1 Liberal Mode

In this mode, we have excluded the change(s) in gap of all possible clone pairs in the same clone class to determine the change patterns of that clone class. If the similar lines of each clone pair in the same clone class change consistently with respect to one another, then we will consider this change as a consistent change. Therefore, the change shown in the Figure 4.1 will be classified as a consistent change.

Advantage: It reports actual inconsistent changes to the developers/maintenance engineer which can save a lot of time.

Disadvantage: It can ignore some potential threats as shown in Figure 4.1.

4.2.2 Strict Mode

In this mode, we do not consider the gap as any special case. If the changes to the clone fragments in the same class are different, no matter that they are in the gaps or common parts, it will be considered as an inconsistent change.

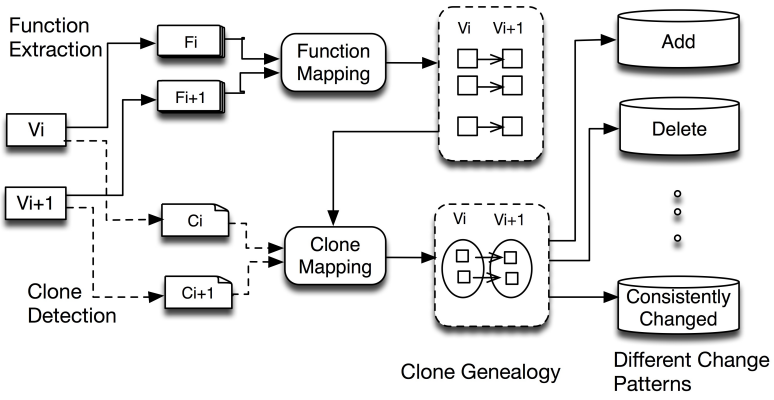


Figure 4.3: The proposed framework for two versions of a program

Advantage: It can report all inconsistent changes to the developers and thus reduce the threats as shown in Figure 4.1.

Disadvantages: Developers may waste some time by investigating many uninteresting inconsistent changes.

Here we should note that the aforementioned modes are applicable for Type-3 clones only where the addition, deletion, and replacement of lines are permitted in addition to modifications of identifiers. We should also note that since Type-3 clones allow addition and deletion of lines, all the clone fragments of a particular clone class could still form the same clone class in the next version even if one or more fragments of that class have been changed inconsistently. The dissimilarity between the fragments of a clone class usually depends on the heuristics/similarity threshold of the associated clone detection tools.

4.3 The Proposed Framework

In this section we discuss our proposed framework to build a near-miss clone genealogy extractor (*CGE*). Usually a *CGE* accepts n versions of a program, maps clone classes between the consecutive versions, and extracts how each clone class changes throughout an observation period. A version may be a release or a revision. The approach has four phases: (1) Preprocessing (2) Function Mapping (3) Clone Mapping, and (4) Automatic Identification of Change Patterns. Figure 4.3 shows how these steps work together to construct clone genealogies of a software system. At first we describe a naïve version of our *CGE* that maps clones between two versions of a program. Then we will describe a complex *CGE* that can deal with n versions of a program, and can work with various clone detection tools.

4.3.1 Preprocessing

For two given consecutive versions, v_i and v_{i+1} of a software system, first we extract all the function signatures from both the versions. For extracting functions we use TXL [19], a special-purpose programming language that supports lightweight agile parsing techniques. We exploit TXL's extract function, denoted by [ˆ], to enumerate all the functions. For each function we store the function signature, class name, file name, the start and end line number of the function in the file, and its complete directory location in an XML file. Certainly, one can use a tool other than TXL for extracting this required information. Once the preprocessing is completed for a particular version V_i, there is no need to preprocess it again.

4.3.2 Function Mapping

Once the preprocessing is done, we map the functions of v_i to those of v_{i+1} instead of mapping their clones. A function is the smallest unique element of a software project if we consider the signature of a function along with its class name and complete file path. Therefore, we can use these attributes as a composite key to map functions between two versions. There are already a few great studies [100, 105] where program elements are mapped based on the function and file names. However, in practice some functions could be renamed, or could move to different files or directories during the evolution of the system. Table 4.1 summarizes all possible change-scenarios of a function during evolution. We have two advantages of mapping the functions before mapping the clones. First, a very few functions are renamed in the next version compared to the total number of functions in a version [39, 57]. Therefore, most of the functions could be mapped by only using their signatures which is computationally very fast. Second, if a function name has not been changed in the next version we can accurately map them even if the body of the function has been changed significantly. This overcomes the disadvantage of threshold based mappings that are dependent on the text similarity of two functions. In the following subsections, we discuss how we adapted and optimized some established function mapping techniques [100, 105] to match functions that are common between two versions, v_i and v_{i+1} of a program.

When function names remain the same

This part of the algorithm maps those functions for which the function names have not been changed in the next version (change scenarios 1-4 in Table 4.1). For each function, if its name occurs once in v_i and once in v_{i+1}, it is considered the same function without considering any further information. Thus even if the function has been moved to another file or directory, our algorithm would map it correctly. One might argue that a function *foo* might be deleted from v_i and a new function having the same name could be inserted in the next version. However, our experience suggests that such circumstances seem to occur rarely. On the other hand, if two or more functions exist having the same name in either one or both versions of the program, our algorithm maps them applying the following rules. For each of the functions we check the locations of

Table 4.1: Change Scenarios of Functions

Scenarios	Signature	Body	Location
1	not changed	not changed	not changed
2	not changed	not changed	changed
3	not changed	changed	not changed
4	not changed	changed	changed
5	changed	not changed	not changed
6	changed	not changed	changed
7	changed	changed	not changed
8	changed	changed	changed

the functions. If they are from the same location, then their signatures must be different, and they will be matched by their signatures. In contrast, if they are from different locations, then their locations will be used to resolve the mapping. For example, sometimes it might happen that the method signatures and file names both are the same. It indicates that they are from different packages/directories. In this situation we will consider the hierarchy level of their directory until we find different names to map them properly.

When function names have changed

This part maps those functions, which have been renamed in the next version (change scenarios 5-8 in Table 4.1). From the previous step, the functions (D) of v_i that have not been mapped into any function in v_{i+1} are the possible candidates for being renamed. That is, if O is the set of function names of v_i, and N is the set of function names of v_{i+1}, $D = O - N$. Similarly, the set of either new or renamed functions (A) in v_{i+1} is $N - O$. Therefore, we need to find an appropriate mapping between D and A. To find an appropriate mapping between D and A, we use Kim et al.s' [57] automatic detection of origin analysis method. Kim et al. introduced eight similarity factors to determine whether a function has been renamed or not. They used all possible combinations of those factors to find the renamed functions, and report that adding more factors does not necessarily improve the accuracy of the origin detection. They also noted that 90.2% of renamed functions could be accurately mapped using only function name and body, whereas the best accuracy was only 91% using more factors. By further investigating their result we noticed that the average number of renamed functions per version is very few; less than 2%. Therefore, we use only the function name and body to find the origin of functions in the set A because we can achieve a considerable performance gain by sacrificing only 0.8% accuracy for the renamed functions, which is negligible (0.016%) compared to the total number of functions.

We use the longest common subsequence count ($LCSC$) similarity metrics to find the origin of a function

```
boolean isContained(Block CF, Function F) {
    return ((CF.FileName == F.FileName)
              AND (CF.BeginLine >= F.BeginLine)
              AND (CF.EndLine <= F.EndLine))
}
```

Figure 4.4: Algorithm for mapping block to function

because only checking common parts of two fragments (function body) or two names (function names) is not sufficient. We also have to check that the common part also maintains the same sequence as its origin. We use equation 4.1 to calculate *LCSC* similarity metrics of two fragments, A and B where $|A|$ and $|B|$ are the number of elements in A and B respectively. It returns a value between 0 and 1 where 0 means no similarity whereas 1 means exactly the same. LCSC calculates name similarity at the character level and text similarity at the line level. We use comment-free pretty-printed lines to ignore formatting or commenting differences. Now for two functions $A \in v_i$ and $B \in v_{i+1}$, we identify A as the origin of B if the average of ($LCSC\ Similarity(name_A, name_B)$ and $LCSC\ Similarity(body_A, body_B)$) is greater than 0.6 which was found as one of the best thresholds by Kim et al [57]. If we find more than one possible origins, we take the one that gives the highest average. However, we did not consider any split or merging of functions during function mapping.

$$LCSC\ Similarity = \{\frac{LCSC_{AB}}{|A|} + \frac{LCSC_{AB}}{|B|}\}/2 \tag{4.1}$$

4.3.3 Clone Mapping

At this point we have the function mappings between versions v_i and v_{i+1}. We also assume that the clones have already been detected from these two versions using a clone detection tool. Typically a clone detection tool reports results as a collection of clone classes where each clone class has two or more clone fragments. A clone fragment could be of any granularities such as function, structural block, arbitrary block and so on. Let $CC^i = \{cc_1^i, cc_2^i, ..., cc_n^i\}$ be the reported clone classes in v_i where $cc_j^i = \{CF_{j1}^i, CF_{j2}^i, ..., CF_{jm}^i\}$. Here CF_{jk}^i refers to the clone fragments of the clone class cc_j^i where $1 \leq k \leq m$. Now the algorithm attempts to map each clone fragment CF^i to its contained (parent) function, F^i in v_i. We say that a clone fragment (CF) is contained by a function (F) if both are in the same file, and the range of line numbers of (CF) is within the range of line numbers of F. In algorithmic form,

We should note that we already have the required information (i.e., file names and begin-end line numbers) for functions from the *Preprocessing* phase (Section 4.3.1) and for the clone fragments from the subject clone detection tool.

Now let us assume a clone fragment CF^i is mapped with a function F^i in version v_i, and another clone fragment CF^{i+1} is mapped with a function F^{i+1} in version v_{i+1}. Let us also assume that F^i is in fact mapped

with F^{i+1} obtained from the Function Mapping phase (Section 4.3.2). If F^i contains only one clone fragment, CF^i we can easily map that clone fragment in v_{i+1} since we already know the corresponding mapped function F^{i+1} in v_{i+1}, and the mapped clone fragment CF^{i+i} should be found in v_{i+1} if it has not been disappeared. However, if F^i has more than one clone fragments, we attempt to map each of the clone fragments between functions F^i and F^{i+1}. For example, if F^i has m clone fragments and F^{i+1} is the corresponding mapped function, we map the clone fragments in two ways. First, we calculate the *LCSC Similarity* (Section 4.3.2) for each of the clone fragments of function F^i with the clone fragments of function F^{i+1}. Second, based on the similarity scores and the relative locations of the clone fragments in the corresponding functions, we take the final decision for mapping. There might be still some clones that have not been mapped yet. They might be file clones, clones that span more than one functions, clones in declarations, or clones in C preprocessor code. We map such clones by following the method discussed in Section 4.3.2 with the exception that instead of using function names as one of the attributes, we use file names of the clone fragments. Therefore, we now have the mapping for each of the clone fragments CF^i_{jk} of version v_i to a clone fragment in version v_{i+1} if it has not been deleted in version v_{i+1}. Because the functions of versions v_i and v_{i+1} are already mapped in the *Function Mapping* phase (Section 4.3.2), the mapping of function clones is straightforward, and of course there is no question of mapping blocks to their corresponding functions.

The next step is to map the clone classes of the two versions. For each of the clone fragments CF^i_{jk} of the clone class cc^i_j in version v_i we find the corresponding clone fragment $CF^{i+1}_{j'k'}$ in version v_{i+1}. In principle, if cc^i_j of version v_i remains the same or changed consistently in version v_{i+1}, all the mapped fragments of cc^i_j in v_i should be in the same clone class in v_{i+1}. However, if the consistent changes are made to the extent that the size of the fragments is under the minimum clone size threshold of the subject clone detector, the detector will not detect this clone class. When we deal with Type-3 clones, all these mapped clone fragments of CF^i_{jk} could be found in the same clone class in v_{i+i} even though they have been changed inconsistently but still their dissimilarity is under the given threshold set by the subject clone detection tool. Of course, a clone class may split due to the extensive inconsistent changes between its clone fragments. Therefore, in order to find out the mapping of the clone class cc^i_j in v_{i+1}, we find all the clone class(es) $\{cc^{i+1}_x, cc^{i+1}_y, ...\}$ from v_{i+1} that form clone classes with any of the clone fragments of cc^i_j of v_i. If all the clone fragments are mapped to the same class, cc^{i+1}_x, we map $cc^i_j \rightarrow cc^{i+1}_x$. On the other hand if they are mapped to multiple classes, $\{cc^{i+1}_x, cc^{i+1}_y, ...\}$ we map $cc^i_j \rightarrow \{cc^{i+1}_x, cc^{i+1}_y, ...\}$.

4.3.4 Automatic Identification of Change Patterns

Automatic and accurate identification of change patterns is one of the important features of a clone genealogy extractor. There is a marked lack of in-depth study in the literature on this issue [40], especially for near-miss clones. In this study, we also attempt to deal with this important issue. Among the change patterns (discussed in Section 2), the identification of *Same*, *Add*, and *Delete* change patterns are fairly straightforward because these change patterns are classified based on the appearances or disappearances of the clone fragments of a

clone class of version v_i in version v_{i+1}. The clone classes of v_i those are not spilt in v_{i+1} are the candidate set for these sort of change patterns. Let us assume that from the *Clone Mapping* phase we obtain a mapping, $cc_j^i \rightarrow cc_{j'}^{i+1}$, where cc_j^i and $cc_{j'}^{i+1}$ have p and q number of clone fragments respectively. Now if $cc_{j'}^{i+1}$ has m clone fragments that have their origins in cc_j^i, we can identify the aforementioned patterns automatically as follows:

- if $m = p = q$ then it is the *Same* change pattern.
- if $q > m$ then it is the *Add* change pattern.
- if $p > m$ then it is the *Delete* change pattern.

Here it should be noted that the *Add* and *Delete* patterns are not mutually exclusive since additions and deletions of clone fragments in a clone class could take place simultaneously. Unlike the above patterns, the identification of consistent and inconsistent change patterns is challenging. It becomes more complicated when we deal with Type-3 clones since for Type-3 clone classes, all the clone fragments of a particular clone class could form the same clone class in the next version even if one or more clone fragments of that class have been changed inconsistently. Therefore, we cannot conclude whether a particular change pattern is consistent or inconsistent even if all the clone fragments of a clone class in v_i form a clone class in v_{i+1}.

In order to deal with this issue, we use a multi-pass approach that makes decisions in each pass, and identifies consistent and inconsistent change patterns gradually. First, the program identifies the clone classes that have not been changed in the next version at all (*Static*), and those clone classes that have been split. The program identifies the split clone classes as inconsistent change because certainly their fragments have been changed inconsistently, and thus they are part of two or more clone classes in the next version. Identification of these change patterns is straightforward and accurate because there is no ambiguity in selecting them. Fortunately, our program can make a decision for most of the change patterns during the first pass because a large number of clone classes do not change at all. We have shown this phenomena for Type-1 and Type-2 clones at the release level in our previous study [91]. This seems to hold for Type-3 clones as well (see Table 4.4).

In the second pass, the program will make a decision for each of the Type-1 clone classes. If $cc_j^i \rightarrow cc_{j'}^{i+1}$ is such a mapping, the program computes the differences between each of the clone fragments of cc_j^i with the corresponding clone fragments of $cc_{j'}^{i+1}$ using *diff*. If the differences for each of the fragment pairs (CF_{jk}^i, $CF_{j'k'}^{i+1}$) are the same, then the clone class is classified as consistent change, otherwise as inconsistent change.

In the third pass, the program will make a decision for each of the Type-2 clone classes. Since the clone fragments of these clone classes have variations in their identifiers, we do not exploit *diff* directly because the differences will not be the same even if the fragments have been changed consistently. In order to deal with this issue we consistently rename the identifiers of the clone fragments using TXL. For example, if the first identifier and all its occurrences in a fragment is replaced by x_1, the second identifier and all of its

occurrences will be replaced by x_2 and so on. We then compute the differences. As before if the differences are the same we classify the change pattern as consistent change, otherwise as inconsistent change.

Now in the fourth pass, the program will apply either *strict* or *liberal* mode depending upon user's choice to determine the change patterns of Type-3 clones.

Strict Mode: The program in strict mode will make a decision for those Type-3 clone classes where modifications of different fragments of the same clone class are only limited to line additions or deletions but do not have any variable renaming by applying the same algorithm as we described in the second pass. For the rest of the clone classes, the program will detect the change patterns as it detected in the third pass.

Liberal Mode: In this mode, the program will detect the change patterns for all possible pairs. If a clone class has only two fragments, it has only one clone pair. There will be $\binom{n}{2}$ clone pairs for a class having n fragments. Let $CP^i_{j1}(CF^i_{j1}, CF^i_{j2})$ be a clone pair where CF^i_{j1} and CF^i_{j2} are two clone fragments. For each such a clone pair, our program first computes the common ordered lines between the clone fragments CF^i_{j1} and CF^i_{j2} by applying *Longest Common Subsequence (LCS)* algorithm. This eliminates the gap between two clone fragments in the clone pair. Now we eliminate the corresponding gap from $CP^{i+1}_{j1}(CF^{i+1}_{j1}, CF^{i+1}_{j2})$ as well. Therefore, now there will be no effect of the gap in the decision making process of determining change patterns of $CP^i_{j1} \rightarrow CP^{i+1}_{j1}$. In order to determine the change pattern of $CP^i_{j1} \rightarrow CP^{i+1}_{j1}$, we apply the same algorithm that we used for *strict mode*. If the change patterns of all of the clone pairs of a given class are consistent, the program will mark the clone class as a consistently changed clone class, otherwise as inconsistently changed clone class. However, in order to reduce the time complexity, instead of determining the change patterns of all clone pairs, we have considered only those clone pairs in which at least one clone fragment was changed.

It should be noted that inconsistent reordering of statements are also considered as inconsistent changes and thus the weakness of *diff* that it cannot detect reordering of statements does not have any impact on our results. All of the identified mappings as well as their change patterns are stored in an XML file for future use.

4.3.5 Construction of Genealogies

In order to keep the discussion simple, we have described the algorithm only for two versions of a program. Now we extend it for n versions to construct the clone genealogies. Let us assume that $CGE(v_i, v_{i+1})$ is the algorithm for only two versions, whereas $CGE(v_1, v_2, ..., v_n)$ is the algorithm for n versions. Practically the n-version algorithm is the combined result of $CGE(v_1, v_2)$, $CGE(v_2, v_3)$, ..., $CGE(v_{n-1}, v_n)$. Our mapping is thus incremental in nature where we can easily integrate the mappings of the previous versions with a new version. If a genealogy of a clone class propagates through *(p+1)* versions, we call it a p-length genealogy. Now if a p-length genealogy has any inconsistent change pattern(s) during the propagation, it will be classified as *Inconsistently Changed Genealogy (ICG)*. Similarly, if a genealogy has any consistent change pattern(s) but does not any inconsistent change pattern(s) during the propagation, it will be classified as *Consistently*

Changed Genealogy (CCG). If a genealogy has been never changed during the observation period, it will be classified as a *Static genealogy*. Similarly, if a genealogy has any *Add* or *Delete* change pattern(s), it will be classified accordingly. We should note that most of the genealogies have two types of change patterns. One pattern is based on how it has been changed, such as *Static, CCG, ICG*, and another is based on the addition or deletion of clone fragments in a clone class such as *Add, Delete, Same*.

4.3.6 Time Complexity

In this section we provide the time complexity of the proposed mapping algorithm for two versions v_1 and v_2 of a program. We represent a version v_i by $(l_i, nf_i, ncf_i, ncc_i)$ where l_i, nf_i, ncf_i, ncc_i denote the LOC, number of functions, number of clone fragments, and number of clone classes respectively in version v_i. For function mapping, at first the algorithm matches name signatures of the corresponding functions of the two versions, v_1 and v_2. This step takes only nf_1 units of time because these mappings are done using a hash map (Section 4.3.2). Let us assume that α number of mappings are found after the execution of the previous step. Certainly, $min(nf_1, nf_2) \geq \alpha$. Now the origin detection of the remaining functions (Section 4.3.2) takes approximately $t = (nf_1 - \alpha) \times (nf_2 - \alpha) \times (l_1/nf_1) \times (l_2/nf_2)$ units of time, where l_i/nf_i is the average length of a function in v_i in terms of lines of code. Therefore, the required time for function mapping is $nf_1 + t$ units of time. Once all the functions are mapped, the time complexity for mapping each clone fragment to its contained function is also linear, which could be achieved by constructing a multi-valued hash map where file name is the key and the functions of the file are values.

For mapping the clone classes (Section 4.3.3), the required time is ncf_1 units because we already have the mappings for each of the clone fragments of the two versions. Therefore, the speed of the whole process is inversely proportional to the number of functions that do not change their names (here the value of α) in the next version, and the number of clone fragments that do not belong to any function. Therefore, for the best case where all of the function names remained the same in the next version ($\alpha = nf_1$) and each clone fragment belongs to a function, the total time complexity is linear. However, it is quadratic in the worst case where all of the function names are changed in the next version ($\alpha = 0$) and all the clone fragments are outside of functions boundaries. Fortunately, on average 98% of the functions do not change their names [57], and from our experiment we found that only a few number of clone fragments are on the outside of functions, which indicates that the time complexity is almost linear with respect to the number of functions.

4.4 Evaluation of the Framework

In order to validate the efficacy of the proposed framework, we have developed a prototype clone genealogy extractor, gCad, as discussed in Section 4.3. Currently, gCad can automatically extract both exact and near-miss clone genealogies at function or block level across multiple versions of a system, and classifies them automatically into meaningful change patterns as discussed in Section 2. Although it can be easily adaptable

Table 4.2: Features Supported

Features	Currently Suupported
Types of Clones	Type-1, Type-2 and Type-3
Granularity of clones	Block (Arbitrary + Structural), Function,
Clone Relation	Clone Pairs, Clone Class, RCF [41]
Adaptability to Tools	NiCad, CCFinderX, iClones
Adapt. to Languages	C, C#, Java
Software Versions	Revision level, Release level
Types of Genealogies	Same, Add, Delete, Static, CCG, ICG
Mode of Operation	Strict, Liberal
Scalability	Large systems (e.g., Linux releases)

to other languages for which a TXL grammar is available, we tested it for three languages, Java, C, and C#. gCad can deal with large systems such as the Linux Kernel which ensures its scalability. Currently, gCad can work with three clone detection tools, NiCad [87], CCFinderX[1] and iClones [36]. However, since it relies on relatively little information (e.g. file names and begin-end line numbers of the clone fragments), it is adaptable to any clone detection tools that provides such information. Table 4.2 summarizes the features that are currently available in gCad.

4.4.1 Experiment

In this section, we discuss the details of our experimentation that we performed to evaluate gCad. We designed our case study in such a way that it validates each characteristic of gCad as discussed above. We chose three open source systems, `ArgoUML`[2], the Linux Kernel[3], and iTextSharp[4]. We have selected these systems because they are developed in three different popular languages (Java, C, and C#), their size varies from medium to very large-scale in terms of lines of code. Furthermore, the interval length between releases of each system varies from a few days to several months. Table 4.3 summarizes the key attributes of the last releases of these systems that we considered.

Since currently gCad supports three clone detection tools, we used all of them to extract genealogies and to validate the result provided by gCad from different perspectives. Among them NiCad provides us the facility to detect both exact and near-miss clones at the function or block level. Because one of our primary objectives is to construct and classify near-miss clone genealogies, we used NiCad to detect code

[1] http://www.ccfinder.net/
[2] http://argouml-downloads.tigris.org/
[3] http://www.kernel.org/
[4] http://sourceforge.net/projects/itextsharp/

Table 4.3: Subject Systems

Attributes	ArgoUML	Linux Kernel	iTextSharp
Prog. Lang.	Java	C	C#
Start Date	Oct 4, 2008	Dec 14, 2009	Dec 8, 2009
End Date	Jan 24, 2011	Jan 5, 2011	Feb 18, 2011
Start Release	0.27.1	2.6.32	5.0.0
End Release	0.32.beta2	2.6.37	5.0.6
# Releases	47	45	7
# Functions	14798	244311	8162
Cloned Functions	3238	39266	1226
Type-1 clone class	169	659	53
Type-2 clone class	226	4483	53
Type-3 clone class	422	9076	282

Table 4.4: Test Results in Strict Mode

Change Patterns	ArgoUML			Linux Kernel		iTextSharp		
	Function	Block		Function	Block	Function	Block	
	NiCad	NiCad	CCFinderX	NiCad	NiCad	NiCad	NiCad	CCFinderX
Exe. time/ver	0.77s	2.02s	25s	1m18s	2m35s	0.50s	1.33s	3m38s
Same	893	1376	2266	16836	61362	479	821	959
Add	267	435	309	2255	9373	50	100	20
Delete	40	72	95	334	1603	17	18	14
Static	1350	2451	2019	18642	83295	500	840	897
CCG	33	41	307	1277	1796	20	52	83
ICG	463	719	510	4509	13535	102	136	20
Total Genealogies	1846	3211	2838	24428	98626	622	1028	1000

clones in all of the subject systems with NiCad setting of minimum clone size 5 LOC, consistent renaming of identifiers and 30% dissimilarity threshold. With this setting we detect a fairly good number of near-miss clones because we allow consistent renaming of identifiers as well as a dissimilarity threshold of 30% which allows 30% dissimilarity of the clone fragments in their pretty-printed normalized format. We also used CCFinderX to detect code clones in two systems, `ArgoUML` and `iTextSharp`, with its default settings to see whether gCad works well with other clone detection tools or not. Finally, we used iClones to compare the accuracy of our mappings as discussed in Section 4.5.

Result

First, we ran gCad in *strict* mode on a *Mac Pro* that has a 2.93 GHz Quad-core Intel Xeon processor and 8 GB of memory (though our program uses a single processing unit). As we mentioned earlier, in *strict* mode

Table 4.5: Test Results in Liberal Mode using NiCad for Block Clones

	ArgoUML	**Linux Kernel**	**iTextSharp**
Execution time/version	2.36s	4m 1s	1.66s
ICG due to inconsistent changes in gap	55	1445	20

gCad considers the changes in gap to determine change patterns. Table 4.4 shows the results of our study for the different types of genealogies including the execution time of gCad. Here the execution time is the average time that is taken by gCad for two consecutive versions of a given subject system. From the first row we see that gCad takes less than a second to a couple of minutes for mapping the clones across two versions and for finding different change patterns, depending on the size of the given system and the level of granularity. However, the number of functions that have not changed their names in the next version plays a key role in reducing the execution time. As we assumed, the number of renamed functions is very few for all of the systems. Approximately 2% of total functions have been renamed on average for the Linux Kernel, whereas for `ArgoUML` and `iTextSharp` it is less than 0.5%. We also found that functions are not renamed that much between minor releases. Most of the related studies did not report the execution time of their clone genealogy extractor except Bakota et al. [7]. As they reported, their preparation step took approximately 2 hours on an IBM BladeCenter LS20 machine equipped with 10 modules, each of them operating with two AMD Dual Core Opteron 275 processors, running on 2.2 GHz containing 4 GB of memory each, and the other step, construction of the evolution mapping, took approximately three hours to complete on one processor core for Mozilla Firefox having 12 versions. Since gCad is not exactly the same as of theirs, we could not directly compare our execution time with theirs. However, the execution time taken by gCad is far less than theirs and looks reasonable enough to be used for software maintenance purposes.

Identifying the evolving change patterns for Type-3 clones is always challenging and time consuming. From the results we see that most of the genealogies of all the subject systems at any level of granularities (approximately 76%-80% for function and 76%-81% for block) did not change at all during the observation period, which is consistent with our previous results [91] where only Type-1 and Type-2 clones were considered. However, there is a large amount of *ICG* compared to *CCG*. In order to see how many inconsistent changes were reported due to the inconsistent changes in gap, we ran gCad in *liberal* mode on the same machine. In this phase we have only considered the clones detected by NiCad clones because changes in mode only affect the change patterns of Type-3 clones. In addition, we only considered block clones because they also include function clones. Table 4.5 presents the new data for the genealogies in the *liberal* mode. As expected, gCad took some extra time for excluding the changes in gap and to detect the change patterns with respect to all possible pairs. However, the time still looks reasonable to be used in the real world software maintenance purpose. Also from Table 4.5, we see that there are 55, 1445, and 20 genealogies for

`ArgoUML`, `Linux Kernel`, and `iTextSharp` respectively that were detected as *ICG* in *strict* mode due to the inconsistent changes in gap. Therefore, as we explained in Section 2.2.1, developers can save a lot of time of manually investigating inconsistently changed genealogies by using the *liberal* mode of gCad. However, some real semantic inconsistencies could be still ignored. Therefore, it is up to developers which mode they want to consider.

4.4.2 Correctness of Mapping

Evaluating the correctness of clone mapping is important for any clone genealogy extractor because all other calculations (e.g., identifying evolution patterns) depends on it. To evaluate the mapping (M) of clones between two versions of a program reported by a tool, the correct set of mappings (E) for the given versions is needed. The requirement of E makes the whole evaluation process challenging because it is very difficult to determine. Our quantitative evaluation is based on two criteria: precision and recall. While precision expresses the percentage of mapping that are correct, recall gives the percentage of correct mappings that the tool finds. We can compute precision and recall with the following two equations:

$$Precision = |M \cap E|/|M| \tag{4.2}$$

$$Recall = |M \cap E|/|E| \tag{4.3}$$

While manual verification of each mapping gives us an idea about the precision, it does not give any hints about recall. Therefore, besides manually validating the genealogies, we also performed a couple of automatic tests to measure both the precision and recall artificially.

Identity test

In this step we have assumed that there are two versions of a program that are identical, in other words, there has been no change between these two versions. Practically, the same program has been used as the inputs of gCad. The major advantage of this test is that the oracle mapping set (E) between these two versions of program is known and it is one to one self-mapping of each clone fragment. Therefore, the precision and recall for this step could easily be computed. We conducted the test for each of the versions we analyzed. For each of the tests, the value of both precision and recall were 1.0. Thus this test ensures that the gCad works well for Scenario 1 in Table 4.1.

Mutation/Injection Framework

We have adapted an existing mutation/injection based framework for evaluating clone detection tools [88] to evaluate the correctness of the mapping detected by gCad. However, instead of completely automating the framework we followed a semi-automated approach and only for function clones to avoid errors. As in the original framework we also have two phases as follows:

Table 4.6: Recall and Precision of the Prototype

Subject Systems	Block Clones	Function Clones		
	Manual	Automated		Manual
	Precision	Precision	Recall	Precision
ArgoUML	0.98	1.0	0.97	0.99
Linux Kernel	0.99	0.96	0.94	0.99
iTextSharp	0.99	1.0	0.98	0.99

Generation Phase In the generation phase, we create a number of mutated versions of the original code base. To get a mutated code base, first we keep a copy of the original code base as the starting version of the subject system. We then make a second copy of the original code base, pick a clone class randomly from it, and remove the selected clone class from it. Once a clone class is selected, we change the fragments of that class in such a way that mimics a developers possible change scenarios following Roy and Cordy's editing taxonomy [88]. A TXL-based mutation process was used to apply the changes (single or multiple) to mutate each of the fragments to create a mutated clone class. Then we inject the mutated clone class into copied code base randomly but manually in such a way that the resulting code base is still syntactically correct. Now we can consider this mutated code base as the next evolving version of the original code base where a clone class has been changed. In this way, we created 70 mutated code bases for each of the systems (last version) for scenarios 2-8 of Table 4.1 where each of the scenarios applied to 10 mutated code bases. For each of the scenarios where the function body was changed (scenarios 3,4,7, and 8), we applied consistent changes on half of the clone classes and applied inconsistent changes to the rest. All of these changes are stored in a database, which works as the oracle (E) for evaluating the correctness of the mapping. We then detected the mappings (M) and identified the change patterns between the original code base and the mutated code bases for each of the systems. Now that we have the oracle (E) and the detected genealogies (M), we can compute the precision and recall using equations 4.2 and 4.3 above.

Tool Evaluation Phase Table 4.6 summarizes the results of our test cases. Among the 70 clone classes in the 70 mutants of `ArgoUML`, 69 of them were detected by NiCad because one of the clone classes was mutated beyond the similarity threshold of NiCad. On the other hand, gCad reported 67 genealogies. When we manually investigated why the two genealogies were missing, we found that the name and body of two clone classes were changed significantly. Therefore gCad could not find the origins of their fragments. However, all the detected genealogies were correct. In the Linux Kernel, among the 70 test cases, gCad reported 69, among which 66 were the correct mappings. We experienced incorrect mappings for three of the test cases because their corresponding clone classes were mapped incorrectly due to extensive change in the function bodies and function names. Similarly we found 69 correct mappings for iTextSharp. Although this approach

computes the precision and recall artificially, it provides sufficient hints whether gCad works well for those editing scenarios during clone evolution.

Manual verification

Besides the automatic testing discussed above, we also manually validated our result. Since it is almost impossible to check manually all the mappings for all the systems, we systematically analyzed all the mappings across the last five versions for each system. At first, we analyzed the *Static* genealogies. If the clone fragments of such genealogies have not been moved, we did not need to investigate their source code because they have not changed at all. We manually checked all the *Static* genealogies for which some of the clone fragments have been moved. However, we did not find any false mapping for such genealogies. For all the changed genealogies, at first we investigated the correctness of the mapping by investigating their source code, function names, and file paths. After confirming the correctness of the mapping, we investigated the change patterns using a visual *diff*. We were not quite sure about some of the mappings, which were excluded from the calculations. During the manual verification we noticed that gCad detected many genealogies where clone fragments changed in such a way that it would have been difficult to detect them using any heuristic or text-based similarity approach. For all of the systems we found that at least 98% of both the reported mappings and change patterns were correct. However, we found a few incorrect mappings when a function was renamed or deleted but a similar type of function was in the system. Table 4.6 summarizes the result of our manual verification. However, measuring the recall manually was out of scope.

4.5 Comparing gCad with Other Methods

Besides the extensive manual verification and automatic testing, we also compared the quality of the mappings and change patterns from gCad with the mappings provided by iClones, and our previous approach [91], which was actually adapted from Kim et al. [56]. We have chosen the `GNU Wget`[5] as the subject system for this purpose because both the clones and their mappings across seven versions of this system detected by iClones are available online. Furthermore, the size of this system is reasonable enough to manually investigate and compare each of the genealogies reported by the two tools under consideration. Therefore, to compare the mappings and the change patterns with iClones, at first we downloaded the *wget.rcf* file from the Software Clones website[6] that contains both the clones and their mapping information for seven versions of the subject system. CYCLONE [41] is another tool that can interpret the mapping information from iClones and construct genealogies from it. We ran CYCLONE on the *wget.rcf* to extract the clone genealogies. On the other hand, we extracted only the clone classes from *wget.rcf* for all the seven versions of the subject system, and applied gCad on them to extract the clone genealogies. It means that CYCLONE and gCad used the

[5] http://www.gnu.org/software/wget/
[6] http://softwareclones.org/

same clone classes to construct genealogies but CYCLONE used the mappings from iClones, whereas, gCad mapped the clone classes using our proposed method. We found that CYCLONE reported 370 genealogies, whereas, gCad reported 374 genealogies along with their change patterns. We then manually investigated each of the genealogies from these tools. We found 364 genealogies which were common between the two results. We investigated the 6 genealogies that gCad missed but was detected by iClones. We noticed that iClones returns some overlapping clone classes (which could be considered as false positives and should not have been detected), making two clone classes using the same clone fragments with only minor differences in the token sequence numbers of the fragments. While iClones maps such clone classes gCad does not in the cases when minor variations in token sequence numbers are within the same line because of its line-based comparison. This is in fact expected as such mappings can be considered false positives.

gCad also detected 4 more valid mappings than iClones. Since iClones exploits the difference information of the modified files in mapping clones, it cannot map those clones accurately in the cases where the functions/methods of the associated clone fragments are re-ordered w.r.t. their source coordinates within the same files in the next version. Our manual investigation confirms that this is exactly the reason why iClones missed these four mappings. Because gCad first maps the functions/methods and then locates clone fragments within the corresponding functions/methods, we have successfully detected these mappings

Our comparison results with iClones (considering its output as benchmark) confirm that gCad is no less than iClones/CYCLONE, but rather is better both in terms of not reporting the mappings of some false positive clones and accurately mapping Type-3 clones. An independent comparison of gCad with iClones by using a third party clone detection tool for the detection of clones might have given a better picture in comparing the accuracy of iClones and gCad. However, such an experiment was not possible since iClones maps the clones during the detection and thus adapting a third party clone detection tool was not applicable.

Since iClones maps the clone classes incrementally during the detection of clones, it is obviously faster than gCad. In order to compare the scalability, we have run iClones with the Linux Kernel versions, and experienced "Out of Memory" error messages. We then attempted to run iClones only with two versions of Linux Kernel and experienced the same error. For this experiment we used a Mac Pro machine that has a 2.93 GHz Quad-core Intel Xeon processor with 8 GB of memory. However, when we talked to the author of iClones, he suggested that he successfully worked with Linux Kernel in a server machine of 64 GB memory. Unfortunately, we do not have such a high configured server within our reach.

In order to compare gCad with our previous method [91] we used CCFinderX for clone detection since it was based on CCFinderX. We ran both gCad and our previous method separately on the detected clone classes to construct genealogies, and found 81 and 75 genealogies respectively. All of the 75 clone genealogies detected by our previous method were also detected by gCad. When we investigated the six additional genealogies reported by gCad, we noticed that they are in fact valid genealogies. Our previous method missed them because the clone fragments were significantly changed to the extent that their text similarities were less than the given threshold. However, gCad was successful in accurately identifying those fragments

Table 4.7: Comparison Results

Clone Detector	Clone Mapping and Construction of Genealogies	# Genealogies	Common
iClones	(iClones + CYCLONE)/gCad	370/374	364
CCFinderX	Saha et al.[91]/gCad	75/81	75

by first mapping the functions and then locating the clone fragments within the functions. These comparison results confirm that gCad not only can detect all the genealogies reported by our previous approach but also can accurately handle the mapping of Type-3 clones. The comparison results are summarized in Table 4.7.

CloneTracker is a great tool for tracking clones in the IDE [27]. However, our work is fundamentally different from that of CloneTracker both in terms of objectives and the representation of clone fragments. We attempted to identify clone genealogies and classify their change patterns. On the other hand they attempted to assist developers in managing clones in evolving software in the IDEs. CloneTracker can neither construct clone genealogies nor it can classify them. However, we were still interested to see how well the CRDs of CloneTracker can map clone classes of different versions of a software systems. We experienced that it is not straightforward at all. The authors also noted similar concerns and used several heuristics for checking the robustness of CRDs for tracking clones over versions. Using a dummy project we experienced similar results as [27] that it can track about 95% of Type-1 and Type-2 clones. However, in case of Type-3 clones we experienced difficulty with CloneTracker. They also noted that CloneTracker cannot handle Type-3 clones because of the possible changes in the anchors of CRDs [27]. In contrast, our approach is specifically designed for tracking Type-3 clones.

4.6 Threats to Validity

There are two potential threats to validity of our result.

First, in our function mapping algorithm, we did not consider the merging and splitting of functions. Therefore, if a function splits in the next version, our algorithm first attempts to map it to the part in the next version for which the function name remains the same as of previous version. If the name of any of the parts does not match, our algorithm maps it to the part in the next version for which source code was more similar to its origin. However, the effect of this situation should be minimal on the results for clone genealogies and change patterns because if any clone fragments of a clone class splits in the next version then all fragments of that clone class should be split accordingly for maintaining consistent change, and gCad should map similar fragments in the next version and thus will detect the change as consistent change, otherwise will detect as inconsistent change.

Second, there might have been some unintentional errors during the manual verification due to the lack

of the domain knowledge or human errors.

4.7 Related Work

Kim et al. [56] were the first who mapped code clones across multiple revisions. They used two metrics, *text similarity* and *location overlapping* to map clones between two revisions. They implemented a *CGE* that is able to extract and classify genealogies automatically. However, they only considered Type-1 and Type-2 clones, detected by CCFinder. Aversano et al. [5] extended Kim et al.'s study, and investigated how clones are maintained when an evolution activity takes place. However, they manually investigated all the genealogies for finding change patterns and mostly focused on Type-1 clones. We overcome the limitations of these two studies by proposing a fast and scalable approach that can extract not only Type-1 and Type-2 but also Type-3 clone genealogies, works with different clone detection tools and can automatically identify the change patterns at both function and block levels. To understand the stability of cloned code Krinke conducted two separate but related studies [61, 62] for Type-1 clones. Like Aversano et al. he also considered the clones of the first revision in the observation period and examined the changes of clones by extracting the changes from the source code repository. Thummalapenta et al. [96] conducted a similar study to understand the maintenance implications of clones.

Bakota et al. [7] proposed an AST based machine learning approach for mapping clones across consecutive versions. They used a number of similarity metrics such as file name, position and lexical structure of the clone instances, and so on, to find the appropriate mapping between two clone instances. However, using a large number of similarity features makes the mapping process computationally expensive. In contrast, we used simple similarity metrics which ensured fast computation while maintaining high accuracy.

As of Duala-Ekoko and Robillard [27] (Section 4.5), Bettenburg et al. [13] also used a *CRD* to map clones between two versions. However, they did not identify the change patterns of the genealogies automatically.

As of Göde and Koschke [36] (Section 4.5), Nguyen et al. [81] also introduced an incremental clone detection tool, ClemanX. The advantages of both of the approaches mostly relies on the tiny changes between revisions. Furthermore, though they mapped clones across multiple versions, they did not classify genealogies based on the change patterns as we did.

Lozano and Wermelinger [70] mapped clones' imprint across multiple versions. Like us they also mapped all the functions/methods before mapping clone classes. However, they did not consider Type-3 clones, whereas we applied a sophisticated technique both for fast and efficient mapping of Type-3 clones and identify the change patterns of such clones with high precision and recall.

4.8 Summary

As for Type-1 and Type-2 clones, extracting the genealogies of Type-3 clones and identifying the change patterns are equally important, especially because there are a significant number of such clones in software

systems. However, mapping Type-3 clones across multiple versions of a program and automatically identifying their change patterns is challenging. In this chapter, we proposed a scalable and adaptable framework that can extract both exact and near-miss (Type-2 and Type-3) clone genealogies and can identify their change patterns automatically. To validate the efficacy of the proposed framework, we developed a prototype and experimented both with multiple versions of three open source systems including the Linux Kernel and a mutation/injection-based framework. We also manually analyzed many detected genealogies including their change patterns, and compared the mapping reported by our prototype with that of an incremental clone detection tool. Our experience suggests that the proposed method is adaptable to other clone detection tools, and reasonably fast while maintaining high precision and recall. Furthermore, it can even tolerate significant changes between two versions and thus could be effectively used to map clones at the release level as well as at the revision level. We believe that this approach would be useful for researchers and developers when studying the evolution of both exact and near-miss clones. In future we would like to conduct large scale empirical studies in the evolution of near-miss clones both at the release and revision levels and investigate the stability of inconsistent changes in Type-3 clones compared to that of Type-1 and Type-2 clones along with the intentionality of such inconsistent changes.

Chapter 5

Understanding the Evolution of Type-3 Clones: A Genealogy-based Empirical Study

5.1 Motivation

Understanding the evolution of clones is important to manage clones properly. There has been quite a bit of research on studying code clone evolution. Most of these studies investigate retrospectively how clones are modified by constructing a clone genealogy. Retrospection helps us to understand and maintain clones in a number of ways such as understanding the behaviour of the clones, distinguishing good clones from bad clones, developing new tools to manage clones and so on. The more we capture the diverse behaviour of clones, the better we will able to manage it in cost effective ways. However, the existing knowledge about the evolution of code clones is limited since the conclusion drawn from most of the previous studies [5, 56, 61, 91, 96] are based on only the Type-1 and Type-2 clones. Although there exist a few studies [3, 104] where Type-3 clones were also considered to examine and predict the overall cloning status in evolving software, none of them attempted to understand how each of the clone classes actually evolve by extracting their genealogies. Recently, Göde and Koschke [37] conducted a study where they analyzed a subset of Type-3 clones to determine the frequency and risks of changes to clones. However, their Type-3 clones only allow parallel gaps between clone fragments rather than allowing additions or deletions of lines in arbitrary positions, and they mainly focused on inconsistent changes w.r.t. risks associated with them. Therefore, we believe that existing knowledge about the evolution of code clones is not sufficient to manage Type-3 clones properly.

There are mainly three reasons for which one cannot get the complete picture of code clones without studying the evolution of Type-3 clones. First, Roy and Cordy [89] show that there are a substantial number of Type-3 clones in most systems. Second, the exclusion of Type-3 clones do not only eliminates Type-3 clone pairs but also classifies some clone classes wrongly as Type-1 and Type-2 which are actually Type-3 by ignoring their siblings that have additions or deletions of lines. Finally, if any Type-1 or Type-2 clone class converts into Type-3 during evolution, the clone genealogy extractor that works with only Type-1 and Type-2 clone classes will misinterpret the change as removal of clone class, and thus the overall result will be incorrect. Thus, to gain a broader understanding, to capture a more detailed picture, and to manage all

types of clones properly we have to understand the changeability of Type-3 clones. However, extracting the history of Type-3 clones and classifying their change patterns automatically are always challenging due to their diverse variety among clone fragments in the same clone class. In the previous chapter we overcame this problem by introducing a prototype, gCad [92] that can extract and classifies both exact and near-miss clone genealogies with high precision and recall. In this chapter we provide an in-depth empirical study of both exact (Type-1) and near-miss (Type-2 and Type-3) clone genealogies in six open source systems using gCad.

Our hypothesis is that the evolution of Type-3 clones should be significantly different compared to that of Type-1 and Type-2 clones due to more variations among the clone fragments in a clone class.

In particular, we focus on five specific research questions to test our hypothesis:

1. Do Type-3 clone classes exhibit similar change patterns as of Type-1 and Type-2 clones?
2. Do Type-3 clone classes change more often than Type-1 and Type-2 clones?
3. Do any Type-1 or Type-2 clone converts to the Type-3 clone class during their evolution or vice versa?
4. Is there any difference between the survival/life time of Type-3 clone classes and that of the other two?
5. What are the frequent syntactic changes to Type-3 clone classes during the evolution? Do those changes also apply to Type-1 and Type-2 clone classes?

In summary, we found the following while answering the research questions:

For most of the systems, inconsistent changes to the Type-3 clones are more frequent compared to consistent changes, whereas, the scenario is completely opposite for the Type-1 and Type-2 clones. Since, the number of Type-3 clone classes are higher than that of other two types, the overall proportion of consistent changes has been found to be low, as opposed to the higher percentages reported by the previous studies [56, 91].

In the six systems we studied, we found that the more textual dissimilarity among clone fragments, the more frequently they are likely to change. On average, 42% of Type-3 genealogies changed throughout their evolution whereas only 30% of Type-1 and 35% of Type-2 genealogies changed during the same epoch. Furthermore, the average change frequency of Type-3 clone classes was 1.7 whereas it was approximately 1.40 for both of Type-1 and Type-2 clone classes.

When clone classes evolve, a considerable number change their types due to inconsistent changes. In our study, we found 87 conversions between Type-1 and Type-3, and 114 conversions between Type-2 and Type-3 in total, whereas, we found only six conversions between Type-1 and Type-2.

Although we have found that the type of a clone class affects its change pattern and the frequency of changes, no systematic relationship has been found, at least statistically, between clone removal and its types.

While for some systems Type-1 clones disappeared more than Type-3 clones; the opposite picture is found for others. The results suggest understanding the characteristics of how clones actually evolve may call for new approaches to clone management.

Among many changes to clones, additions/deletions of *if statement* have been found to be frequent. Other major changes are changes in *data types*, additions of various statements, changes in the name or parameters of a called function. We have also identified some hotspots, where inconsistent change may lead to error.

The rest of this chapter is organized as follows. Section 5.2 describes our study procedure. Section 5.3 presents the data we collected and answers the five research questions. We discuss our results in section 5.4. Section 5.5 discusses the possible threats to validity of our results. In Section 5.6 we discuss the related work to ours, and finally Section 5.7 concludes the chpater with our directions for future research.

5.2 Study Setup

In this section we will outline our study setup to collect the relevant data to answer the research questions. It includes choice of our subject systems, description of parameters for detecting clones and extraction of genealogies, overall procedure for collecting and investigating data, and a brief description of the statistical method that we have used in our analysis.

5.2.1 Subject Systems

We studied six open source software systems for our case study. In order to select subject systems, we have given preference to those which have been used in some previous studies including ours, have a long development history, and are of reasonable size. We also ensured that multiple developers have contributed to these systems so that we can assume that none is an expert of the whole system. Based on these criteria, we have chosen the following -

- `dnsjava`[1] is an implementation of DNS in Java.
- `JabRef`[2] is an open source bibliography reference manager that works on Windows, Linux and Mac OS X.
- `ArgoUML`[3] is an interactive, graphical software design environment that supports the design, development and documentation of object-oriented software applications.
- `ZABBIX`[4] provides monitoring and tracking facility for network servers, devices and other resources.
- `Conky`[5] is a free, light-weight system monitor.

[1] http://www.xbill.org/dnsjava
[2] http://jabref.sourceforge.net
[3] http://argouml.tigris.org
[4] http://www.zabbix.com
[5] http://conky.sourceforge.net

Table 5.1: Subject Systems

Prog. Lang.	System	No. of Releases	Start Release	End Release	Start Date	End Date	Duration	SLOC
Java	dnsjava	50	0.9.2	2.1.1	April 19, 1999	Feb 10, 2011	131 months	6,290-15,018
	JabRef	27	1.5	2.4.2	Aug 15, 2004	Nov 1, 2008	50 months	22,041-69,170
	ArgoUML	48	0.27.1	0.32.BETA2	Oct 4, 2008	Jan 24, 2011	26 months	176,618-202,555
C	ZABBIX	31	1.0	1.8.4	Mar 23, 2004	Jun 1, 2011	86 months	9,252-62,845
	Conky	28	1.1	1.8.1	July 20, 2005	Oct 5, 2010	62 months	6,555-39,810
	Claws Mail	44	2.0.0	3.7.9	Jan 30, 2006	April 9, 2011	63 months	133,642-189,786

- `Claws Mail`[6] is an email client and news reader that support POP3, IMAP, SMTP and many other protocols.

The sizes of these systems (last release) vary from approximately 15K to 203K source lines of code (SLOC), excluding comments and blank lines. Because we wanted to conduct manual analysis both for the detected clones (e.g., in removing false positive and uninteresting clones) and for answering the research questions, we intentionally did not choose very large systems. Of course several of the systems are reasonably large and we chose these projects from a variety of application domains to avoid any kind of biasing towards any specific kind of software system. A more detailed overview of the subject systems is presented in Table 5.1.

In any study of clone evolution, the choice of interval length between two consecutive versions plays a key role in the result. A previous study [56] shows that there are many clones that are created for experimental purposes by the developers. However, when a version of a software system is officially released, the source code is expected to be in a stable form, and thus any inconsistent changes to clone fragments between two releases should be either intentional or accidental, which we consider important for our study. Therefore, we have chosen release level instead of revision level for our study.

5.2.2 Clone Detection

In this section we discuss the procedure for detecting clones in each system. This includes preparing source code for clone detection, the choice of clone detection tool, the parameters that we used to detect clones, and the procedure for filtering out uninteresting clones.

Preparing source code

In the source code directory of any system, there may exist some files which are not exactly related to this study. The presence of these files may affect the study result. Therefore we excluded those files manually before detecting clones. For example, since we are interested to investigate system logic, we ignored all the header (*.h) files. Similarly we excluded auto generated code by the IDE since they are not maintained by

[6] http://www.claws-mail.org

Table 5.2: NiCad Setting for Clone Detection

Setting	Value
Granularity of Clones	Block
Minimum Clone Length	5 LOC
Filtering of Statements	None
Renaming of Identifiers	Blind
Dissimilarity threshold	30%

any developers. We also removed all the test cases from each system by identifying them using their file names, function name and comments.

Clone Detection Tool

Since the main objective of our study is to evaluate both exact and near-miss clone genealogies, we used the NiCad-2.6.3 [20] clone detection tool for detecting clones. NiCad has already been found to be effective in detecting near-miss clones while maintaining high precision and recall [88, 87, 89]. NiCad can also be instructed to extract clones at either function or block level.

NiCad Settings for Clone Detection

We have carefully chosen the following NiCad settings given in Table 5.2 to detect clones. One of the most important parameter for detecting clones is minimum clone size in terms of lines. The precision of the result will be increased with the increase of minimum LOC, while recall will be increased with the decrease of that. We have chosen five LOC as the minimum clone size because it is enough to eliminate getter and setter methods while maintaining good recall. On the other hand manual verification of data allowed us to filter out uninteresting clones. The next important parameter for detecting Type-3 clones is the UPI threshold which actually means how much dissimilar fragments could be in the same clone class. From the table we see that the value of the dissimilarity threshold is 30% which allows 30% dissimilarity among the clone fragments in a clone class in their pretty-printed normalized format.

5.2.3 Extraction of Genealogies

We have used our recently introduced near-miss clone genealogy extractor, gCad [92], to automatically extract and classify clone genealogies for our study. However, we have extended gCad significantly for this study so that we can collect all the relevant data automatically from gCad to answer the research questions. In order to get a more accurate result, we divided the whole process of extraction and classification of genealogies done by gCad in two steps. In the first step, gCad classifies all the clone classes into different types (Type-1 or

Type-2 or Type-3) according to the definitions described in Section 2.1, maps the clone classes and classifies their change patterns (Static or *CC* or *IC*) between each two consecutive versions on a single run. All of these mapping information and change patterns of each clone class along with their types for each consecutive versions are stored in an XML file. In the second step, gCad constructs genealogies by merging the results of each consecutive version stored in the previous step, computes other relevant data such as frequency of changes, age of genealogies, and again stores the results in another XML file. gCad also provides a set of GUIs to navigate the mapping and change patterns of clone classes between two consecutive versions, and to investigate the genealogies. These GUIs allow users to manually verify each of the mappings, to correct the inconsistencies if there are any, or to filter out any uninteresting clone genealogies from the study. For example, if a user finds any change patterns that should be an inconsistent change but gCad classified it as a consistent change, he can change it easily through the GUI by just clicking a drop-down menu. Now if the users run the second step of gCad again, the corrected information will be used while constructing genealogies. However, in our previous study [92] we found that the accuracy of gCad is reasonably high in terms of both precision and recall. In this study, we have classified the Type-3 clone genealogies using both *strict* and *liberal* mode of gCad. In the *strict* mode gCad considers the changes in gap to determine change pattern, whereas, in *liberal* mode it does not.

5.2.4 Procedure

This section describes the procedure we used in our study.

1. For each project, we capture all minor and major releases during the time period provided in Table 5.1.
2. We run NiCad to detect clones in each of the releases.
3. We run gCad to extract and classify clone genealogies.
4. We investigate each clone genealogies manually to see if there are any genealogies that are made of false positives or uninteresting clones. If yes, we remove them from the study. All the changes were properly updated and stored.
5. Finally, we again run the second step (described in the previous section) of gCad to get the accurate result.

5.2.5 Statistical Analysis

In order to test the presence of a significant difference among proportions of different clone evolution patterns for different types of clones (RQ1), we have used the Chi-Square test. It tests differences in proportions for dichotomic data in contingency tables, with number of rows or columns greater than two. However, the Chi-Square test has two limitations. It cannot estimate the *p-value* well if 1) any expected frequency is less

than one, and 2) more than 20% of the frequencies have values less than five. We performed the Chi-Square test assuming a significance level of 95%.

5.3 Results

This section reports the results of our empirical study with respect to the five research questions defined in Section 1. We answered all the research questions based on what we observed. Since we are neither the system developers nor the application domain experts, we do not answer the questions from the *why* or *how* perspective.

RQ-1: Do Type-3 clone classes exhibit similar change patterns as of Type-1 and Type-2 clones?

In order to investigate whether the change patterns of Type-3 clones are similar to those of Type-1 or Type-2 clones, first we investigated if there are any relationships between the change patterns of clone fragments and their types. Although determining the change patterns for Type-1 and Type-2 clone classes is fairly straight forward, for Type-3 clones it is complicated as changes may take place in the gaps between two clone fragments in a clone pair. We have already discussed these change scenarios in Section 2.2.1. In order to deal with this issue, as we mentioned earlier, we have run gCad both in *strict* and *liberal* modes. In both modes, gCad automatically groups all the genealogies for each system by their associated clone types and change patterns as presented in Table 5.3. To illustrate, the first row of the table represents that for `dnsjava` we have found four genealogies where Type-1 clone classes have not changed, one clone genealogy where Type-1 clone class has changed consistently, and one genealogy where Type-1 clone class has changed inconsistently. Then, in order to investigate the desired relationships, we performed the Chi-Square test on a contingency table, where columns represent the evolution patterns of the genealogy (*Static*, *Consistent Change*, and *Inconsistent Change*) and rows represent the types of clones. In this test, our null hypothesis is that different types of clones do not have any impact on the different change patterns. It should be noted that we did not perform the test for `dnsjava`, `JabRef`, and `Conky` since they have more than 20% of cells having frequencies less than five. However, the p-value obtained from the rest of the systems (2.07E-6 for `ArgoUML`, 5.46E-4 for `ZABBIX`, and 3.59E-10 for `Claws Mail`) in *strict* mode strongly suggests that there is a significant difference among the proportion of change patterns for different types of clones. Even when we excluded the changes in the gaps using the *liberal* mode of gCad, the p-value (8.83E-5 for `ArgoUML`, 0.06 for `ZABBIX`, and 0.004 for `Claws Mail`) suggests some differences, although not as strong as the previous mode, among the proportion of change patterns for different types of clones. Therefore, we can reject the null hypothesis for these subject systems.

From Table 5.3 we see that, for most of the systems, the proportion of inconsistent changes to the Type-3 clones is substantially higher than that of Type-1 and Type-2 clones when we considered the changes in the gaps. On average of all clone genealogies that we observed, approximately 36% of the Type-3 clones have changed inconsistently, whereas, this measure is roughly 21% for Type-1 and 15% for Type-2 clones. When

Table 5.3: Distribution of Genealogies by Clone Types and Change Patterns

System	Mode	Types	Static	Consistent Change	Inconsistent Change
dnsjava	-	Type-1	4	1	1
	-	Type-2	1	1	2
	Strict	Type-3	57	6	69
	Liberal	Type-3	57	16	59
JabRef	-	Type-1	17	2	9
	-	Type-2	20	10	4
	Strict	Type-3	206	43	192
	Liberal	Type-3	206	73	162
ArgoUML	-	Type-1	125	4	33
	-	Type-2	132	6	12
	Strict	Type-3	1239	29	473
	Liberal	Type-3	1239	87	415
ZABBIX	-	Type-1	12	7	6
	-	Type-2	16	11	6
	Strict	Type-3	144	45	165
	Liberal	Type-3	144	70	140
Conky	-	Type-1	4	2	3
	-	Type-2	10	6	4
	Strict	Type-3	108	11	43
	Liberal	Type-3	108	27	27
Claws Mail	-	Type-1	37	13	8
	-	Type-2	56	40	25
	Strict	Type-3	369	120	353
	Liberal	Type-3	369	218	255
Total	-	Type-1	199	29	60
	-	Type-2	235	74	53
	Strict	Type-3	2123	254	1295
	Liberal	Type-3	2123	491	1058

we investigated each of the systems separately, we observed that all of them follow a similar trend with a little bit of an exception in `Conky`. From Figure 5.1 we see that the proportion of inconsistent changes to Type-3 clones in `Conky` is lower than that of Type-1 clones. We believe that the reason behind this exception was there were fewer Type-1 clones compared to the large number of Type-3 clones. In contrast to inconsistent changes, the proportion of consistent changes to Type-3 clones are lower than that of other types. We have found that only about 7% of genealogies have been classified as *CCG* for Type-3 clones, whereas, they are 11% for Type-1 clones and 21% for Type-2 clones. From the last row of Table 5.3, the calculated relative rate of *CC* w.r.t. *IC* for all the Type-1, Type-2, and Type-3 clones are 52%, 140%, and 20% respectively, which indicate that the proportion of consistent changes to Type-2 clones is substantially higher than that of other two types. From Figure 5.1 we also see this trend for each of the systems separately. One may argue that in *strict* mode we considered the changes in gap, and thus the rate of inconsistent change is high. However, when we excluded the changes in the gaps using *liberal* mode, although the rate of inconsistent change decreases, the overall pattern remained the same.

Finally, we observed that Type-3 clones are significantly less stable than Type-1 and Type-2 clones. On average, 58% of all Type-3 genealogies are static, whereas, it is 70% for Type-1 and 65% for Type-2 clone genealogies. Here, we have got two exceptions in the systems `Conky` and `ArgoUML`. In `Conky`, the reason should be the same as discussed before, whereas, in `ArgoUML` most of the clones (above 70%) were static in nature regardless of their types. In conclusion, we can say that majority of the clones are static in nature regardless of their types. However, if they are changed, Type-2 clones are more inclined to be changed consistently, whereas, inconsistent changes are more frequent to Type-3 clones.

RQ-2: Do Type-3 clone classes change more often than Type-1 and Type-2 clones?

The more a clone class changes, the more additional cost it may incur since changing one clone fragment may cause it to propagate the same change to all of its siblings. Furthermore, the probability of inconsistent changes will increase with the increase of the number of changes. Therefore, in order to investigate the change frequency of Type-3 clones and to compare it to that of Type-1 and Type-2 clones, we counted the number of changes (frequency) in each clone genealogy. Table 5.4 presents the change frequencies of the genealogies for each system grouped by their clone types.

From Table 5.3 and Figure 5.1 we have already seen that the number of modified Type-3 clones is higher than that of Type-1 and Type-2 clones. From Table 5.4 and Figure 5.2 we see that the change frequencies of Type-3 clones are also significantly high compared to that of both Type-1 and Type-2 clones. From the results we observe that most of the Type-1 clones either did not change at all or changed once. Only exception is `Claws Mail` where five clones were modified twice and another five were modified three times. However, we have not found any Type-1 clones in any system that changed more than three times over the evolution period, whereas, for each system, there are a considerable number of Type-3 clones that changed more than five times.

One might argue that the number of changes to Type-3 clones is higher than that of Type-1 and Type-2

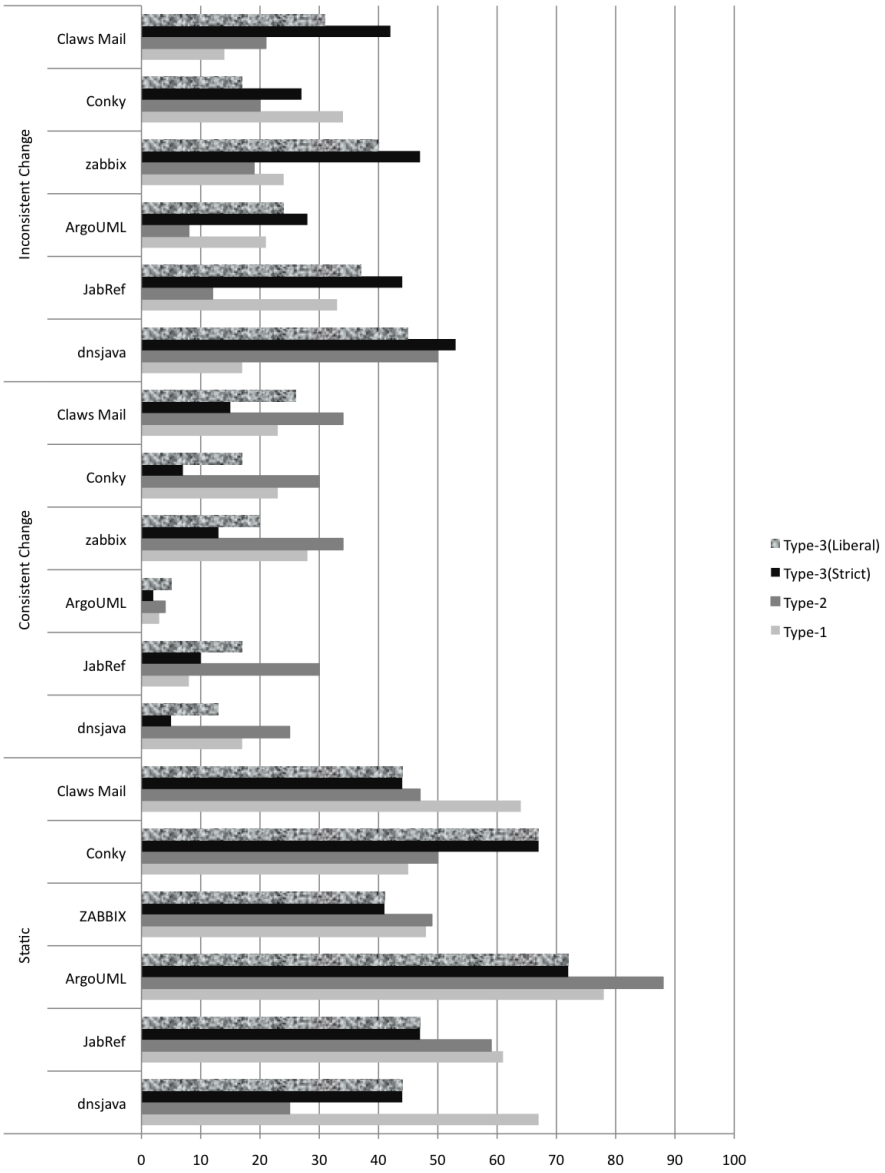


Figure 5.1: Percentages of various change patterns

Table 5.4: Change Frequencies of Clone Genealogies

Subject	**Clone**	**Frequency of Changes**						
System	**Types**	**0**	**1**	**2**	**3**	**4**	**5**	**5+**
dnsjava	Type-1	4	1	1	0	0	0	0
	Type-2	1	3	0	0	0	0	0
	Type-3	57	43	16	6	7	2	1
JabRef	Type-1	17	7	3	1	0	0	0
	Type-2	20	13	1	0	0	0	0
	Type-3	206	141	42	27	11	7	7
ArgoUML	Type-1	125	27	7	3	0	0	0
	Type-2	132	13	3	2	0	0	0
	Type-3	1239	332	101	42	20	4	3
ZABBIX	Type-1	12	8	3	2	0	0	0
	Type-2	16	10	5	1	1	0	0
	Type-3	144	122	55	14	10	6	3
Conky	Type-1	4	3	0	0	2	0	0
	Type-2	10	8	1	1	0	0	0
	Type-3	108	30	13	6	2	1	2
Claws Mail	Type-1	37	17	1	3	0	0	0
	Type-2	56	43	12	6	2	2	0
	Type-3	369	271	108	47	28	11	8
Total	Type-1	199	63	15	9	2	0	0
	Type-2	235	90	22	10	3	2	0
	Type-3	2123	939	335	142	78	31	24

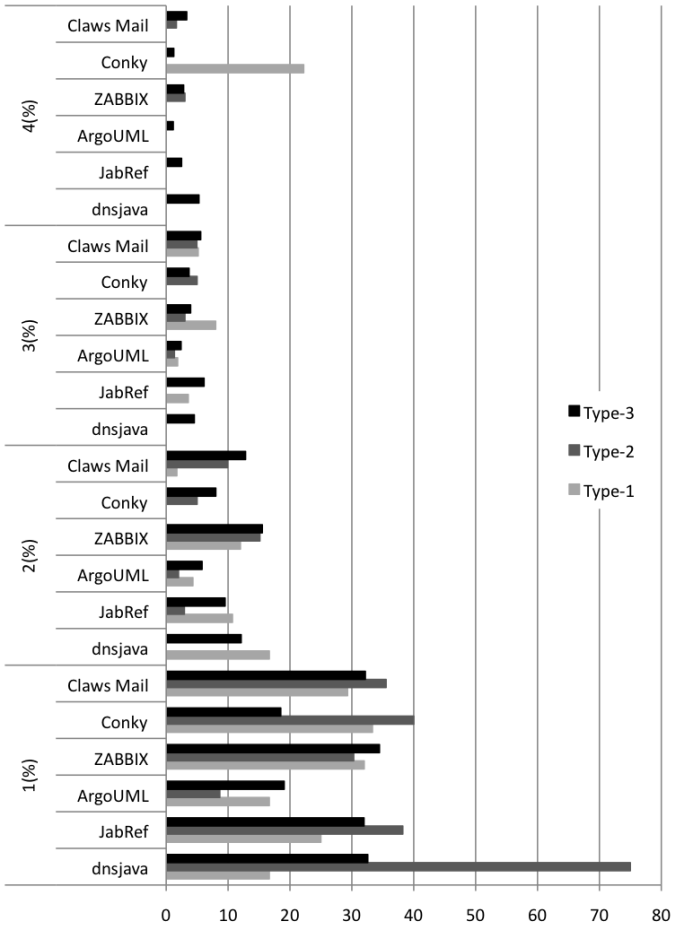


Figure 5.2: Percentages of frequency of changes

clones because they are much more in each system than the other types. Since most of the entries in the Table 5.4 for Type-1 and Type-2 clones are zero, we did not perform the Chi-Square test to understand the difference between each types in terms of change frequency. However, we have calculated the weighted average of the frequency of changes for all the three types of clones separately. The results show that Type-3 clones change 1.8 times on average, whereas the rate is 1.4 for both Type-1 and Type-2 clones. From Figure 5.2, we also see that 15% to 42% of clones changed once regardless of their types. However, significantly higher percentages of Type-3 clones changed more than two times compared to that of Type-1 and Type-2 clones for most of the systems. We have not included the clone classes that changed more than four times in the figure because there are no Type-1 or Type-2 clones that exist in these groups.

RQ-3: Do any Type-1 or Type-2 clone converts to the Type-3 clone class during their evolution or vice versa?

This research question asks whether the type of a clone class changes during the evolution or not. For example, throughout the evolution, a Type-1 clone class could be converted into the Type-2 due to identifier

Table 5.5: Conversion of Clone Types

System	Type-1↔Type-2	Type-1↔Type-3	Type-2↔Type-3
dnsjava	0	2	4
JabRef	1	9	6
ArgoUML	3	48	47
ZABBIX	0	12	12
Conky	0	2	4
Claws Mail	2	14	41

renaming in the next version, or a Type-1/Type-2 clone class could be converted into a Type-3 clone class in the next version due to inconsistent changes. An answer to this question is important because if a Type-1 or Type-2 clone class converts into a Type-3 clone class in the next version, a clone detector will not be able to find the clone class if it is restricted to find only Type-1 and Type-2 clones. Therefore, in this case, a clone genealogy extractor that works with only Type-1 and Type-2 clones will report that the clone class has been removed from the system although actually it is present as a Type-3 clone.

Table 5.5 shows that a number of clone classes in each system converts into another type during their evolution. Conversions may occur in both directions. For example, a Type-1 clone class could convert into Type-3 in the next version. Oppositely a Type-3 clone class may convert into Type-1 clone class. From the table we see that a considerable number of conversions between Type-1 and Type-3 clones, and between Type-2 and Type-3 clones. However, we have observed very few conversions between Type-1 and Type-2 clones. We manually investigated many of these conversions to answer *RQ-5*. We found that most of the Type-1 and Type-2 clones converted into Type-3, whereas, very few Type-3 clones converted into Type-1 or Type-2. We have also found some interesting conversions, where a Type-3 clone class converted into Type-1 clone class, but in the next version it again converted back to Type-3. Some of the conversions were semantic preserving. Figure 5.3(a) shows a clone fragment of a Type-1 clone class in `dnsjava` that converted into Type-3 clone class in the next version for just the omission of a *this* keyword (Figure 5.3(b)). This type of change indicates that either the programmers were not aware of its other siblings or they did not care about consistency.

RQ-4: Is there any difference between the survival/life time of Type-3 clone classes and that of the other two?

Since different types of clones have different levels of textual similarity, the removal of code clones from a software system may depend on the types of clones. For example, since the clone fragments in a Type-1 clone class are exactly similar, refactoring of those fragments may be easier than for Type-2 or Type-3 clone classes. In order to investigate whether the removal of clones depends on their types, first we classified each

```
public Object sendAsync(final Message query, final ResolverListener listener) {
         final Object id;
         synchronized (this) {
                 id = new Integer(uniqueID++);
        }
        String name = this.getClass() + ": " + query.getQuestion().getName();
        WorkerThread.assignThread(new ResolveThread(this, query, id, listener), name);
        return id;
}
```

(a) dnsjava-1.1.4/org/xbill/DNS/ExtendedResolver.java

```
public Object sendAsync(final Message query, final ResolverListener listener) {
        final Object id;
        synchronized (this) {
                id = new Integer(uniqueID++);
        }
        String name = getClass() + ": " + query.getQuestion().getName();
        WorkerThread.assignThread(new ResolveThread(this, query, id, listener), name);
        return id;
}
```

(b) dnsjava-1.1.5/org/xbill/DNS/ExtendedResolver.java

Figure 5.3: Conversion of a clone class from Type-1 to Type-3

of the clone genealogies as either dead or alive based on its existence in the final release that we considered. The bar chart in Figure 5.4 shows the proportion of the DG and AG for each subject system grouped by the types of clones.

From Figure 5.4 we see that although most of the systems have a higher proportion of alive genealogies than dead genealogies, no systematic relationship has been found between the removal of code clones and their types. For example, in `Claws Mail` and `ArgoUML`, only 23% and 27% of Type-1 clones have disappeared throughout the evolution period respectively. On the other hand, more than half of the Type-1 clones disappeared from `Conky`, `JabRef`, and `dnsjava`. We have found a similar scenario for the other types as well.

In order to investigate whether there are any relationships between the survival/life time of clones and their types, we measured the life time of each genealogy in terms of number of releases. Since the alive genealogies are still evolving and we cannot predict when they will disappear, we collected the data separately for alive and dead genealogies. We use the terms *survival time* and *life time* for dead and alive genealogies respectively. For example, if a clone class appears in ith release, stays in the system until jth release, and finally disappears in *(j+1)*th release, we say that the clone class survived *(j-i+1)* releases. On the other hand, if the clone class still exists in the final (fth) release that we considered, we say that the life time of the clone class is *(f-i+1)* releases.

Table 5.6 presents the average *survival time* for the dead genealogies and *life time* of alive genealogies for each system grouped by the type of clones. Although we can see from the table that the *survival time* of Type-2 clones is higher than that of four systems, the difference is not that significant. We have found exactly similar characteristics for *life time* of alive genealogies as well. Therefore, although the complexity of removing clones may depend on the level of dissimilarity among clone fragments, i.e, clone types, statistically

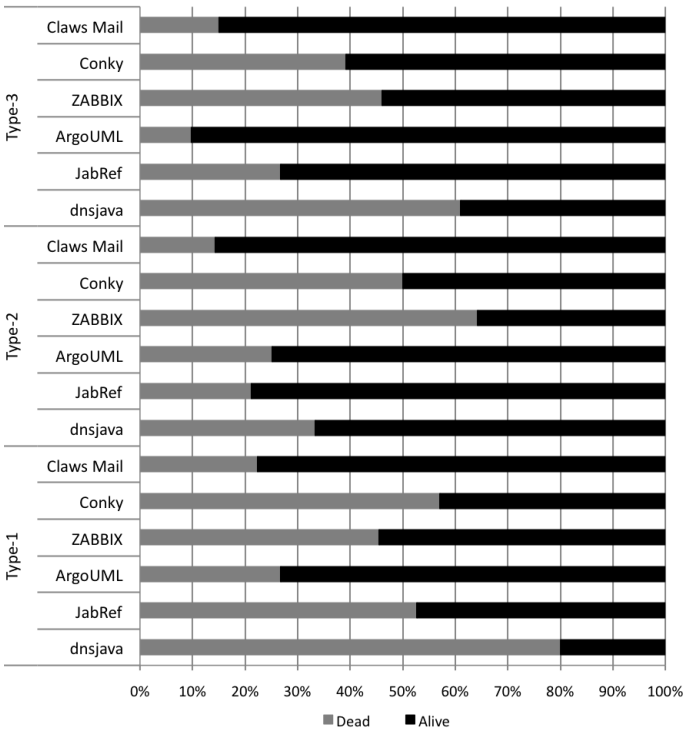


Figure 5.4: Proportion of dead and alive genealogies

we have not found any strong relationships between them.

RQ-5: What are the frequent syntactic changes to Type-3 clones during the evolution? Are those changes also applied to Type-1 and Type-2 clone classes?

While numeric data about different characteristics are useful to understand the high level behaviour of code clones, understanding the characteristics of how clones actually evolve may open new approaches to clone management [56]. In order to answer how Type-3 clones actually evolve and if those changes are similar to the changes of Type-1 or Type-2 clones, we have manually investigated 150 changes to clones, in which there are 50 clones of each type. We have also verified if those changes were consistent or inconsistent. In order to make the result comparable we have considered equal number of consistent and inconsistent changes for each type of clone classes. The candidate clone classes for manual verification were chosen randomly. However, to get an unbiased result and for the convenience of manual verification, we discarded the clone classes that have more than 20 LOC. Table 5.7 presents all the syntactic changes that we observed grouped by clone types (before their change) and change patterns.

From the table we see that although the majority of changes to Type-3 clones are either changes in *data types* or additions of lines, for all types of clones, changes related to *if statement* are more frequent. Interestingly, all changes to *data type* for Type-3 clones were inconsistent, whereas most of them were consistent for Type-1 clones. No relationship has been found between the changes related to *if statement* and the change pattern. However, we have noticed that most of the additions of lines in Type-1 and Type-2 clones were

Table 5.6: Average Survival/Life Time in terms of Number of Releases

Subject Systems	Average Survival Time of *DG*			Average Life Time of *AG*		
	Type-1	**Type-2**	**Type-3**	**Type-1**	**Type-2**	**Type-3**
dnsjava	11	14	11	5	25	21
JabRef	8	5	8	15	17	15
ArgoUML	14	14	15	39	41	38
ZABBIX	9	11	10	17	15	13
Conky	8	10	9	12	6	7
Claws Mail	13	21	20	29	33	32

function calling, whereas, they were variable assignments in Type-2 clones.

The most important finding of our manual analysis is that we have found some hotspots, where most of the changes were consistent regardless of their types. They are *loop condition*, *called function name*, *variable renaming*, and *API*. For example, from the table we see that there are three inconsistent changes in the *called function name* for Type-2 clones. Among them, we have found two consecutive inconsistent changes in `ArgoUML` where the first inconsistent change (release 29.1→29.2) was fixed by the second inconsistent change (release 29.2→29.3). Therefore, any inconsistent changes to these hotspots could be reported as suspicious changes that may call attention from the developers for quick manual verification. We have also noticed an interesting characteristic in the change of *called function parameters*. Whenever the number of parameters were changed, i.e, a parameter was added/deleted, the change was consistent. However, when there were changes in passing the value of parameters, the changes were inconsistent in most cases.

From the result, we have seen that although some hotspots are equally important regardless of types to manage clones, some changes (*data type*) resulted differently to different types of clones. Therefore, we believe that a robust clone management system should consider the types of clones as one of the important parameters.

5.4 Discussion

In this study, we attempted to understand the evolution of Type-3 clones in open source systems with respect to five research questions. Among them, three research questions (RQ-1, RQ-2, and RQ-4) were attempted to answer by researchers in the literature [56, 61, 91], although in most cases they did not consider Type-3 clones. The other two research questions (RQ-3 and RQ-5) are new to the best of our knowledge. In our study we have observed a number of findings that are not analogous to that reported by the previous studies. Now we discuss *why* and *how*, we believe, our results differ from theirs.

In the first research question, we calculated the number of genealogies that have changed consistently or

Table 5.7: Various Syntactic Changes to Clones During the Evolution

Syntactic Change	**Type-1**		**Type-2**		**Type-3**	
	CC	**IC**	**CC**	**IC**	**CC**	**IC**
Insertion of *if* statement	5	3	3	7	4	3
Deletion of *if* statement	2	3	3	2	-	-
Changes in *if* condition	1	-	-	5	-	1
Changes in *loop* condition	-	-	-	-	2	-
Changes in *CF* name	3	1	7	3	5	-
Changes in *CF* parameters	1	5	2	2	5	-
Addition of lines	3	7	6	4	3	10
Deletion of lines	2	2	-	-	-	-
Variable renaming	-	-	2	-	2	-
Changes in data type	4	1	1	1	-	9
Changes in *API*	1	-	-	-	1	-
Assignment operation	1	1	1	1	3	-
Changes in Logic	2	1	-	-	-	2
Total	25	25	25	25	25	25

CF = Called Functions

inconsistently. We have found that most of the clone classes (60%) do not change at all throughout their life time. However, if they changed, most of the changes are inconsistent in nature. We found that only 8% of all clone genealogies were classified as *CCG* in *strict* mode, whereas it is 14% in *liberal* mode. This finding is not consistent with the the findings of previous studies [56, 91] where only Type-1 and Type-2 clones were considered and it is reported that up to 38% of all clones are maintained consistently. However, our results completely agree with that of a study [37] conducted by Göde and Koschke where they also considered a subset of Type-3 clones (gapped clones). But they did not analyze the clones by types. When we delved into deeper and analyzed the genealogies separately in terms of clone types, we observed that inconsistent changes are very frequent to Type-3 clones. We believe that since the previous studies did not take Type-3 clones into account, they found fewer inconsistent changes.

In the second research question, we counted the number changes to each genealogy. Although this question was also analyzed by Göde and Koschke [37], our objective was different from theirs. Their intention was to find out those clone classes that contribute additional costs in the maintenance phase. Therefore, they analyzed all the clones together. However, we tried to understand the type specific characteristics of clones, and thus we analyzed all types of clones separately. In that sense, our result is new and it reveals that the change frequency of Type-3 clones are higher than that of Type-1 and Type-2 clones.

In the fourth research question, we investigated average survival/life time of code clones. Although we found some significant differences among different types of clones in terms of change patterns and frequency of changes, we did not observe any systematic relationships in this case. In a recent study, Cai and Kim [14] discovered that the number of developers who modified clones and the time since the last addition or removal of a clone to its class are highly correlated with the survival time of clones. Transitively, we anticipate that perhaps these factors are not related to clone types. Therefore, we did not find any relationship between clone types and survival time. However, there may be a lot of situations that can affect the result. For example, Type-3 clones change a lot (from this study) but difficult to refactor [97]. Therefore, there are more Type-3 clones in the systems. On the other hand, developers may not be interested to refactor Type-1 clones since they are more likely to be static. Testing these hypothesis could be the further research of interest.

5.5 Threats to Validity

In this section we discuss the possible threats to validity of our results.

5.5.1 Clone Detection

The results of any study on clones are contingent on the raw clone data, which again depends on the selection of clone detection tool and the parameters used to detect clones. In order to mitigate this threat we have carefully chosen the clone detection tool NiCad which has been found to be effective detecting both exact and near-miss clones [88, 87, 89]. The setting we used for clone detection were shown to have high precision

and recall[88, 89]. On the other hand, by manual verification, we ensured that there are no false positives in the result. However, certainly NiCad might miss several clones that was out of scope of our study.

5.5.2 Extraction and Classification of Genealogies

The evolutionary data of our study vastly depends on the result of gCad. If gCad does not find any mapping of a clone class between two consecutive versions, a single genealogy could be divided into two parts. Moreover, a clone class can wrongly map to a clone class in the next version due to ambiguous situation. However, our experience from previous study [92] and extensive manual analysis in this study suggest that these situations are very rare.

5.5.3 Manual Assessment

There might have been some unintentional errors during the manual verification due to the lack of domain knowledge or human errors. However, we address this threat by discussing various ambiguous situations between us and with other lab mates who are either expert in this area or have prior experiences to work with code clones.

5.5.4 Generaliziblity

As case study subjects, we picked six open source projects. Although we have carefully chosen these subject systems from different application domains, our findings may not be generalizable to other open source projects or industrial projects. This threat can be mitigated by adding more subject systems (both open source and industrial), which are part of our future work.

5.6 Related Works

Kim et al. [56] are the first who tracked code clones across multiple revisions and observed how clones evolve in a software system. Based on a case study of two small Java systems, they observed that 36% to 38% of clone genealogies consist of clones that have changed consistently. They also observed that many clones are volatile in nature, and aggressively refactoring them may not be worthwhile. In another study, Cai and Kim [14] investigated the characteristics of long live clones and predicted the survival time of clones. In our previous study [91], we extended the study of Kim et al. [56] by studying 17 open source systems of diverse variety and found no surprising results. However, in all these studies, CCFinder was used as the clone detection tool which mainly considers Type-1 and Type-2 clone classes. In contrast, we included Type-3 clone classes, and found that the evolution of Type-3 clones are quite different than that of Type-1 and Type-2 clones.

Aversano et al. [5] performed an empirical study to investigate how clones are maintained when an evolution activity or a bug fix takes place. Based on a case study of two Java systems, they reported that

either for bug fixing or for evolution purposes, most of the cloned code is consistently maintained during the same cochange or during temporally close cochanges. However, they considered only exact clones and a few gapped clones. Unlike their study, we studied both exact and near-miss clone genealogies to understand the evolution of Type-3 clones compared to that of Type-1 and Type-2 clones.

Krinke [61] analyzed five open source software systems and found that half of the changes to code clone classes are inconsistent and that corrective changes following inconsistent changes are rare. In another study [62], he found that cloned codes are more stable than non-cloned codes and thus require less maintenance effort compared to non-cloned code. However, conclusions drawn from these studies were only analyzing exact clones. Later Lozano and Wermelinger [70] conducted an experiment for measuring the maintenance effort on methods consisting of cloned code and on that have not. Although they found that having a clone may increase the maintenance effort of changing a method, the characteristics analyzed in these methods did not reveal any systematic relations between cloning and such maintenance effort increase. Our study differs from these in that instead of comparing between cloned code and non-cloned code, we evaluated and compared different types of clone genealogies to answer several research questions.

Bakota et al. [7] proposed a machine learning approach for detecting inconsistent clone evolution situations and found different bad smells using twelve versions of Mozilla Firefox. However, they studied the evolution patterns of clone fragments whereas we studied that of clone class level.

In order to investigate the relationship between inconsistent changes of clones and bug, Battenburg et al. [13] performed an empirical study on three Java systems. They reported that many clones are created during the software development process due to the experimentation of developers, which the developers can manage well. Thus they worked at the release level instead of the revision level. In order to avoid the effect of such short-term clones, we also chose to work at the release level. However, while they focus on finding the relation of inconsistent changes to software defects for three open source systems, we focused on both consistent and inconsistent changes to clones to understand their change behaviour. Furthermore, we analyzed Type-3 clones, which they did not.

Duala-Ekoko et al. [27] developed a clone tracker that works in IDE to assist developers to manage clones efficiently. They tracked clones using clone region descriptor (CRD) which is location independent.However, like our study, their objective was not to study the change patterns of clones, but rather to support clone management.

Thummalapenta et al. [96] performed an empirical evaluation on four open source C and Java systems for investigating to what extent clones are consistently propagated or independently evolved. While they focused on identifying the evolution patterns of clones over time and relating those patterns with other parameters (clone granularity, clone radius and cloned code fault-proneness), we focused on understanding the difference of the evolution of various types of clones in terms of change patterns, change frequencies, and survival time.

Göde and his colleagues [32, 36, 37] conducted several studies to understand various characteristics of code clones. First, Göde [32] studied the evolution of only Type-1 clones in nine open source systems, and

found that the ratio of clones decreased in the majority of the systems. However, no general conclusion on the consistent or inconsistent changes to clone classes was reported. Later Göde and Koschke [36] extended the previous study by including Type-2 clones, and found that clones are changed rarely during their lifetime. If they are changed, they tend to be changed inconsistently. In a recent study [37], they conducted a study to assess the intentionality of the inconsistent change where they included a subset of Type-3 clones. Based on a case study of three projects, they reported that the rate of unintentional inconsistent change is very small. However, their Type-3 clones are conceptually gapped clones which could not allow arbitrary additions and/or deletions of lines among clone fragments. Unlike their approach, we have used a more robust definition of Type-3 clones which allows addition/deletions of lines in arbitrary positions in the clone fragments, and analyzed all the genealogies with respect to five research questions.

5.7 Summary

A recent trend in clone research focuses on the management of clones in a cost-effective way, rather than just removing them because of the trade-offs among the associated cost, risks and benefits of removing clones. However, it is not possible to manage clones properly and cost-effectively without understanding the diversified behaviour of clones during their evolution, in particular for Type-3 clones. In this study, we unveiled some interesting characteristics of the evolution of both exact and near-miss clones that either contradict some existing findings or are new to the literature. We believe that these findings could be helpful to design better tools and techniques for maintaining clones. In future we also would like to use these findings to develop a robust clone management tool.

Chapter 6

Finding Relevant Attributes for Clone Removal from Developers Perspective: An Empirical Study

6.1 Motivation

Previous research shows that, in general, code clones are neither good nor bad. It is also not possible or practical to eliminate all the clone classes from a software system [56]. However, unnecessary duplication of codes can impose a requirement of additional effort during software maintenance since they may need synchronized changes. Considering this importance, developers remove clones to minimize the negative effects. However, for many systems, clone removal is still not a part of daily maintenance activities [33]. We have identified two potential reasons for this. First, clone detectors return a huge set of clones, and aggressively refactoring all the clones is not worthwhile since many clones are either volatile [56], or do not change in their entire life time [91, 92]. Therefore, their unnecessary removal is certainly not a cost effective solution. In response, although there are a few tools for clone refactoring, they do not assist developers to filter out those clones that are irrelevant to clone management. Second, although it is important to understand the refactoring of clones from a maintainer's point of view, there is a marked lack of research regarding this. While reasons for the creation of code clones and their effects have been extensively investigated [52, 55], and methods do exist that provide support to eliminate duplication [8, 29, 42, 50, 58], only one study exists that investigated clone removal from developer's perspective [33]. Besides, it is still unknown which attributes developers consider most to refactor a clone class.

The goal of this study is to systematically investigate the potential attributes related to clone refactoring during software evolution by examining the attributes that the developers prefer to refactor in most of the clone classes. Earlier, Göde [33] examined three simple metrics namely LOC, text similarity and distance in the source tree to understand how developers remove clones. However, he did not find any significant relationship between clone removal and those metrics. Finally, he concluded that more complex metrics such as change frequency of clones should be examined to better understand the removal process. The main limitation of his study was that he did not consider the change history of clones, which is one of the most important factors to assess the impact of clones on software maintenance and thus may affect clone removal. In this study, we have considered a wide range of metrics and attributes of clone classes as well as their

change histories to get a complete picture of clone removal in real world software development. In summary, we have found that developers are inclined to refactor clone classes having two or three fragments that are closely located to each other in the same file, and changed at least once. On the other hand, it seems that the size, text similarity, granularity, and change patterns of a clone class do not affect the removal process significantly.

The rest of this chapter is organized as follows. Section 6.2 introduces important terminology and metrics. Section 6.3 describes our study procedure. Section 6.4 presents our results. We discuss our findings in section 6.5. Section 6.6 discusses the possible threats to validity of our results. In Section 6.7 we discuss the related work to ours, and finally Section 6.8 concludes the chapter with our directions for future research.

6.2 Metrics

In this section we introduce the metrics that we will use throughout this chapter. In order to identify the key attributes for clone removal, we have considered a wide range of attributes which are expected to have direct or indirect relationships with the required effort to refactor a clone class. We have compiled this set of metrics by surveying current literature relevant to this study [14, 37, 33, 56].

Text Similarity: The text similarity between two code snippets C_1 and C_2 is determined by calculating their common lines with respect to their sizes. Equation 6.1 below describes the *Text Similarity* function. Here $|C_1|$ and $|C_2|$ are the pretty-printed LOC of C_1 and C_2 respectively. $|C_1 \cap C_2|$ is the number of common ordered lines between C_1 and C_2, calculated using the longest common subsequence (LCS) algorithm.

$$Text\ Similarity(C_1, C_2) = \frac{2 \times |C_1 \cap C_2|}{|C_1| + |C_2|} \tag{6.1}$$

Finally, we calculated the text similarity of a clone class by calculating the average *Text Similarity* of all clone pairs of that class.

Entropy of Dispersion: We have used an entropy measure to characterize the physical distribution of clones at the file level by using equation 6.2. Here p_i denotes the probability of clones located in file i. For example, if all the clone fragments reside in the same file, the dispersion entropy will be 0. If the entropy is low, clones are concentrated in only a few files. If the entropy is high, clones are equally dispersed across different files.

$$entropy = \sum_{i=1}^{n} -p_i log(p_i) \tag{6.2}$$

6.3 Study Setup

In this study, we have used the same subject systems that we used for the empirical study described in Chapter 5. We have also used the same tools and settings as described in Section 5.2 for detecting clones

and extracting genealogies. Now we describe the procedure that we followed throughout the study to find the relevant attributes of clone removal.

1. For each project, we capture all minor and major releases during the time period provided in Table 5.1.
2. We run NiCad to detect clones in each of the releases.
3. We run gCad in *strict* mode to extract and classify clone genealogies. Here we want to capture all possible inconsistent changes including the inconsistent changes in gap. Therefore, we use the *strict* mode of gCad.
4. We investigate each clone genealogies manually to see if there are any genealogies that are made of false positives or uninteresting clones. If yes, we remove them from the study. All the changes are properly updated and stored.
5. We investigate all the dead genealogies to see how the clones were removed. A clone class may disappear for several reasons. First, it may be removed through refactoring. Second, the clone fragments may be deleted since they are no longer needed. Third, the clone fragments may have been changed inconsistently in such a way that their dissimilarity is greater than the threshold set by a clone detector. Finally clone fragments may have been changed in such a way that the resulting length of fragments are less than the given minimum length set by the clone detector. In our study, we considered only those clones which were removed through refactoring, and exclude the ones which disappeared due to other reasons.
6. Then we calculate all the target metrics both for the removed clone and alive clone classes. Finally we find the key attributes which are more related to the clone refactoring than others.

6.4 Results

This section reports the results of our empirical study to find the attributes which make a clone class attractive for removal. We will explore each of the attributes and investigate whether a given attribute is related to intentional clone removal or not.

6.4.1 Text Similarity

The degree of text similarity among the clone fragments in a clone class is important information as it corresponds to the differences and thus may be proportional to the refactoring effort. For example, every Type-1 clone class should be refactored with minimum effort through the *extract method* or through other appropriate refactoring methods. Surprisingly, at least for the observed subject systems, we did not find any significant relationship between text similarity and clone removal. Table 6.1 presents the average text similarity of removed clones and alive clones for the subject systems. From the table we can see that although

Table 6.1: Actual and Normalized Textual Similarity of Removed and Alive Clones

Subject System	Actual Text Similarity				Normalized Text Similarity			
	Removed Clones		Alive Clones		Removed Clones		Alive Clones	
	Average	SD	Average	SD	Average	SD	Average	SD
dnsjava	0.6	0.2	0.67	0.18	0.8	0.12	0.81	0.1
JabRef	0.76	0.18	0.68	0.18	0.85	0.13	0.82	0.11
ArgoUML	0.76	0.2	0.66	0.17	0.85	0.14	0.8	0.14
ZABBIX	0.72	0.19	0.73	0.17	0.83	0.16	0.83	0.11
Conky	0.76	0.16	0.69	0.15	0.88	0.09	0.84	0.09
Claws Mail	0.73	0.2	0.65	0.22	0.87	0.12	0.82	0.15

SD = Standard Deviation

the average text similarity of removed clones in four systems are higher than that of alive clones, the difference is not that significant. Furthermore, two subject systems show opposite trends. The standard deviation also shows that developers remove clones having any degree of similarity.

Sometimes the actual text similarity does not estimate the actual effort for refactoring. For example, in case of different identifier names in different clone fragments, text similarity of a clone class may be very low although they are easily refactorable. That is why we also investigated the normalized text similarity by removing the identifier differences. However, from Table 6.1 we see that although the overall value of normalized text similarity increased, still they do not show any promising relationships between this similarity and clone removal. This finding also indicates that developers do not search for any specific type of clones (e.g. Type-1, Type-2, or Type-3) for refactoring.

Developers remove clone classes having varying degree of similarity if it is refactorable at all.

6.4.2 Number of Fragments/Clone Class

In order to investigate the relationship between the number of fragments in a clone class and clone removal, we calculated the average number of fragments in the removed clone class and that of alive clone classes for each subject system. From the result presented in Table 6.2, it seems that developers are more interested to refactor the clone classes having two or three clone fragments. For each subject system, the average clone fragments for removed clone classes was less than that of alive clone classes, and for most cases the difference was significant. One may argue that most of the clone classes may have two or three clone fragments in a system. However, results in Table 6.2 show that both `ArgoUML` and `Conky` have on average nine clone fragments per class, but developers refactored the clone classes which have only two or three clone fragments.

Table 6.2: Average Number of Fragments/Clone Class

System	Removed Clones	Alive Clones
dnsjava	2.25	2.75
JabRef	2.31	4.17
ArgoUML	2.12	9.12
ZABBIX	2.31	4.53
Conky	2.37	9.31
Claws Mail	2.88	2.95

We saw a similar trend in `JabRef` as well. Although there were 74 clone classes having more than three fragments, only four clone classes were refactored.

> Developers are more interested to refactor the clone classes which have fewer fragments.

6.4.3 LOC/Clone Fragment

The size of the clone fragments are expected to have a relationship with the refactoring effort especially when the candidate clone class is not Type-1. For any given text similarity less than 1, the number of dissimilar lines will increase with the increase of fragment size. In order to investigate the relationship between lines of code per clone fragment in a clone class and clone removal, we have calculated the average number of pretty-printed lines of code per fragment for removed clones and alive clones separately. We also have calculated the standard deviation for each of the measurements to understand how much data are spread out over a large range of values. From Table 6.3 we see that for four subject systems the average size of clone fragments for removed clone class are greater than that of alive clone classes, whereas for other two they are slightly less. Therefore, from the high level view it seems true that developers are interested to refactor larger clones, which is analogous to Kim et al.'s [55] anticipation. However, the high standard deviation of the sizes of removed clones indicates that developers remove varying sizes of clone fragments. Our manual investigation also confirms this finding. We have found many instances where developers removed clones less than seven lines as well as more than 100 LOC per fragment. The same is true for alive clones as well which made it difficult to derive any relationship with size of the fragments to clone removal.

> Developers remove clone fragments of diverse sizes.

Table 6.3: Lines of Code per Clone Fragments

Subject System	Removed		Alive	
	Average	SD	Average	SD
dnsjava	10	3	11	5
JabRef	17	15	13	9
ArgoUML	16	13	15	19
ZABBIX	26	25	21	21
Conky	20	23	15	7
Claws Mail	15	9	16	18

6.4.4 Level of Granularity

Since a number of studies show that the most frequently used refactoring technique to remove a clone class is the *extract method*, we expected that developers would be more comfortable to remove function/method clones, and we will get higher proportion of function/method clones removal. In order to investigate the relationship between granularity and clone removal, we examined two different levels of granularity - function/method and block, i.e, lines of code encompassed by two curly brackets. However, the data presented in Table 6.4 shows that developers remove both function and block clones as per their needs. From the table we see that for most of the systems developers remove almost the same proportion of function and block clones. For `ZABBIX` and `Conky`, the proportion of block clones removal was even higher. However, it should be noted that the refactoring rate of the two big systems `ArgoUML` and Claws Mail was far less than others. On the other hand, as the result shows, the developers of the comparatively small systems such as `dnsjava`, `ZABBIX`, `conky` were more aware of creating clones and were active in refactoring clones. For example, in `dnsjava` and `Conky`, we have found only one and two function clones respectively. And, in `ZABBIX`, although we found eight function clones, seven of them were refactored by the developers. This indicates that although from the high level view, we cannot draw any conclusion, the low level details show that perhaps developers remove exact function clones as much as possible. Here one thing also should be noted: since NiCad only reports structural blocks, i.e., line sequences encompassed by curly brackets, it may be easier for the developers to refactor these types of block clones. The result may vary if one uses other clone detection tools, which report arbitrary sequences of lines as block clones.

> Developers remove both function and block clones as per their need. However, it seems that they are inclined to refactor exact function clones.

Table 6.4: Level of Granularity Vs Clone Removal

Subject System	Function			Block		
	Total	**Rem.**	[%]	**Total**	**Rem.**	[%]
dnsjava	69	37	53.62	25	15	60.0
JabRef	204	41	20.09	110	21	19.09
ArgoUML	1183	97	8.19	305	20	6.55
ZABBIX	201	78	38.80	134	62	46.26
Conky	115	35	30.43	59	30	50.84
Claws Mail	510	40	7.84	337	29	8.60

Table 6.5: Comparison of Entropy of Dispersion

System	Removed Clones	Alive Clones
dnsjava	0.71	0.9
JabRef	0.53	0.98
ArgoUML	0.82	1.3
ZABBIX	0.35	0.53
Conky	0.18	0.24
Claws Mail	0.3	0.7

6.4.5 Entropy of Dispersion

As expected, we have found a relationship between the entropy of dispersion and clone removal. From Table 6.5 we see that for each subject system, the average entropy of dispersion for removed clones is significantly less than that of alive clones. This indicates that the more closely located the clone fragments of a given clone class are, the more attractive it is for refactoring. When we delved deeper, we found a very interesting relationship between entropy and number of fragments related to clone removal. Most of the refactored clone classes had two fragments if their entropy was greater than zero, i.e., they were not located in the same file. For example, in `JabRef` and `Zabbix`, developers refactored 37 and 43 clone classes respectively for which entropy were more than zero. Among them only two clone classes in `JabRef` and 10 clone classes in `Zabbix` had three fragments. The rest had only two fragments.

> Developers are inclined to refactor the clone classes for which the clone fragments are closely located.

Table 6.6: Removal of Clones Classified by Change Patterns

Subject System	Static Clone Classes			Consistently Changed CC			Inconsistently Changed CC		
	Total	Removed	[%]	Total	Removed	[%]	Total	Removed	[%]
dnsjava	60	27	45.00	8	3	37.50	49	27	55.10
JabRef	217	52	23.96	53	3	5.66	132	15	11.36
ArgoUML	1435	109	7.60	39	4	10.26	440	19	4.31
ZABBIX	166	88	53.01	61	18	29.51	109	35	32.11
Conky	121	44	36.36	19	7	36.84	37	16	43.24
Claws Mail	445	58	13.03	172	7	4.07	304	7	2.30

6.4.6 Change Patterns

Compiling potential candidates for clone refactoring should be closely dependent on how they have been changed throughout their life-time. For example, some researchers argue that refactoring a static clone class is worthless as they do not impose any additional cost during maintenance. On the other hand, it can save a lot of effort if developers refactor a clone class which changes consistently. Therefore, we expected that we will get many instances of removing consistently changed clone classes in the real world software systems. But surprisingly the result shows that the developers do not search for any particular change pattern to remove clones. From Table 6.6 we see that for three systems static clones were removed most, for two systems inconsistently changed clones were removed most, and for another system the consistently changed clone classes were removed. When we investigated the reasons, we found that the existing definitions of various change patterns of clones in the current literature are based on only textual change. During our manual investigation, we found some static clones that were refactored before they changed. On the other hand, some clone classes were refactored that were changed inconsistently in terms of text but changed consistently in terms of semantics. Furthermore, some clone classes were refactored since they were mistakenly changed inconsistently. Therefore, text-based change patterns are not enough to capture the relationship between change patterns and clone removal. Unfortunately there is still no tool available that can classify the change patterns of a clone class semantically.

> Developers do not search for any particular change patterns to remove clones.

6.4.7 Age

Although the age of a clone class is not directly related to the effort required to refactor it, the information about the age of dead genealogies can be helpful to get an overview on how fast developers respond to refactor clones. In order to investigate this, we counted the age of each clone class that was removed through refactoring, and categorized them by age. Figure 6.1 and 6.2 show the number of clone classes removed with respect to the number of releases for the dead clones of Java and C systems respectively. From the figures we see that almost 50% of dead clones in four subject systems were refactored within only five releases.

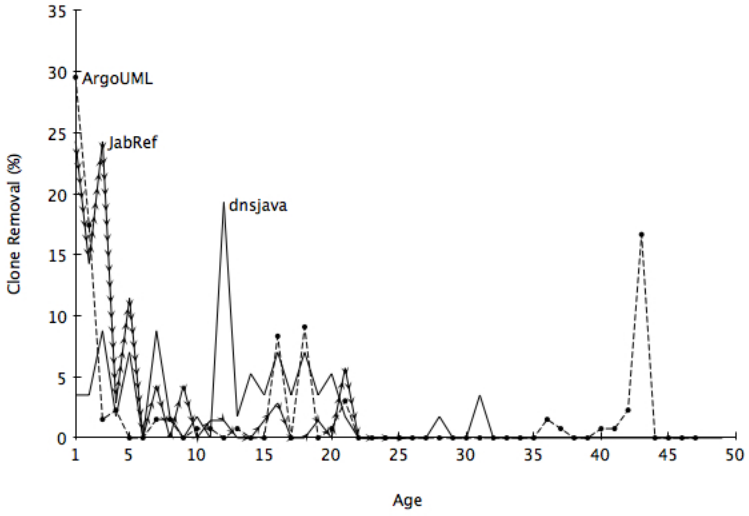


Figure 6.1: Clone removal categorized by age for Java systems

Although a considerable number of clones stayed for a long time in all systems, they were refactored when a major restructuring took place. For example, 17% of the refactored clone classes in `ArgoUML` stayed through 43 releases and 35% of clone classes in Claws Mail stayed through 27 releases, and finally they were removed in a specific release. We have seen the same trend for other systems also.

> Generally developers try to refactor a clone class within a few releases after they are introduced. However, some long-lived clones are removed when major restructuring takes place.

6.4.8 Frequency of Changes

Frequency of changes to clone classes is one of the most important metrics in terms of clone management, since applying the same changes to all clone fragments is costly. Generally any consistent change to clone classes multiply the change effort by the number of fragments in that class. On the other hand, omission of a single change to any clone fragment may introduce bugs or may result in unexpected behaviour. So technically it may be worthwhile to refactor a clone class when it is likely to change consistently. In order to investigate how developers deal with the issue, we counted the change frequencies of each removed clone class. Table 6.7 shows that most of the removed clones were changed only once. The average change frequency of clone classes that changed for each system is less than two. We have found very few clone classes that were refactored after more than two changes. However, we have found many clone classes that were alive although they changed. Many of them were not attractive because they have either three or more fragments or higher entropy of dispersion. However, as per our observations, still there were opportunities to refactor a considerable number of clones that could save the additional cost of co-change.

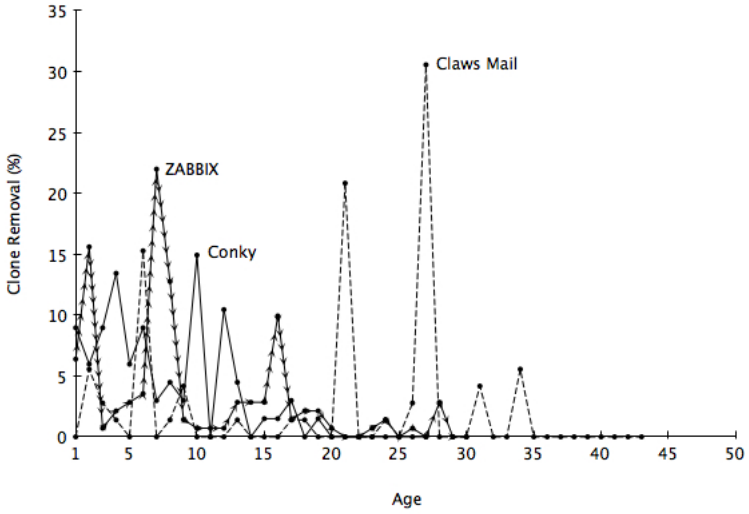


Figure 6.2: Clone removal categorized by age for C systems

Table 6.7: Change Frequency of Removed Clones

Subject	**Change Frequency**			
System	**1**	**2**	**>2**	**Average**
dnsjava	16	9	5	1.80
JabRef	11	4	3	1.72
ArgoUML	17	4	2	1.48
ZABBIX	30	16	7	1.74
Conky	10	8	5	1.95
Claws Mail	9	2	5	1.57

Developers consider to refactor a clone class after it has been changed once if they are interested to refactor.

6.5 Discussion

Although there is an ongoing debate about whether clones are harmful or not, most of the researchers agree that aggressively refactoring all the clones is not worthwhile. Then the first question that arises is which clones are to be regarded as relevant and which are to be regarded as irrelevant to software maintenance? As Göde [33], we also believe that an important contribution to this answer is to analyze which clones

developers select for removing. It would have been great if researchers could have determined a threshold for each attribute that can separate refactored clones from alive clones. Unfortunately this is not possible since refactoring of clones is still not a part of daily software maintenance activities. Therefore, it is expected that there are still many clones in the alive category, which are worth refactoring. These clones will adversely affect the various metric values for alive clones. That is why, instead of comparing the metrics value of refactored clones with that of alive clones, we have concentrated on only refactored clones to understand the important attributes for clone refactoring. However, we have looked at the same attributes for alive clones as well, and identified those attributes as relevant for which values were distinct from the overall values of all clones. Following the procedure, we have got some important attributes such as number of clone fragments in a clone class, entropy of dispersion, age, and change frequency which could be combined to select clones for refactoring.

In our study, we have found many instances where developers removed a clone class with high effort. However, developers removed very few clone classes that had a large number of fragments and are located in different files. It is still unclear whether it happened due to lack of tool support to select clones for refactoring or lack of intention. However, based on our observations from this study, we believe that three important search criteria would be useful for developers to select clones for refactoring depending on various situations. Of course the clone sets in the search result must be refactorable and have to be attractive for refactoring according to the findings described in Section 6.4. The proposed search criteria are -

- **Recently introduced clone classes:** This search could be helpful when removal of clones is a part of daily software maintenance activities. Then the developer can decide whether they should refactor the clone classes or not since they are very similar to the frequently refactored clone classes.

- **Recently modified clone classes:** This search could be helpful when a clone class was previously ignored but now it needs attention. Since it has been recently changed it may change in future again, and refactoring this type of clones can save additional change cost. In this study, we have also found many clones that were refactored after they have been changed once.

- **Frequently modified clone classes:** This search could be helpful to make the hotlist since developers have ignored these clone classes earlier although they changed a lot. The more likely a clone class to be changed, the more additional cost it may incur. Therefore, either the developers should use any alternate way to remove this type of clones or they can tag the clone classes so that they should not appear again in the next search result.

6.6 Threats to Validity

In this section we discuss the possible threats to the validity of our results.

Clone Detection: The results of any study on clones are contingent on the raw clone data, which again

depend on the selection of clone detection tool and the parameters used to detect clones. In order to mitigate this threat we have carefully chosen the clone detection tool NiCad which has been found to be effective detecting both exact and near-miss clones [88, 87, 89]. The setting we used for clone detection were shown to have high precision and recall[88, 89]. Furthermore, by manual verification, we ensured that there are no false positives in the result. However, certainly NiCad may miss a few clones which might have affected our findings.

Extraction and Classification of Genealogies: The evolutionary data of our study vastly depends on the result of gCad. gCad may map a clone class wrongly to a similar but different clone class in the next version due to ambiguous situation or may miss a mapping. However, the result from previous study [92] and extensive manual analysis in this study suggest that these situations are very rare.

Manual Assesment: There might have been some unintentional errors during the manual verification due to the lack of the domain knowledge or human errors. However, we address this threat by discussing various ambiguous situations between us and with other lab mates who are either expert in this area or have prior experiences to work with code clones.

Generaliziblity: As case study subjects, we picked six open source projects. Although we have carefully chosen these subject systems from different application domains, our findings may not be generalizable to other open source projects or industrial projects. This threat can be mitigated by adding more subject systems (both open source and industrial), which will be part of our future work.

6.7 Related Works

Removing duplication through refactorings is an essential part of code clone research. There has been quite a bit of research on various tools and methods for clone rafactoring till now. While they differ significantly in many aspects, they are also related to this study.

The simplest way of refactoring clones is the *extract method* [29, 42, 50, 58] that replaces the cloned code by a function call to a newly created function created from the shared code of the clone fragments. Fanta and Rajlich [29] remove function and class clones from industrial object-oriented systems with the aid of automated restructuring tools. Higo et al. [42] also use metrics calculated from clone information and architectural data, and remove clones with *extract method* and *pull-up method* refactorings. After detecting clones with an AST-based approach, Juillerat and Hirsbrunner [50] also use the *extract method* refactoring for Java language. Komondoor and Horwitz developed a semantics preserving procedure extraction algorithm that works on PDG-based clones [58]. On the other hand, Balazinska et al. [8] used design patterns to remove duplication from Java code. Although diverse approaches exist to support developers in removing duplication, none of these studies investigated clone removal from the developer's perspective. Only, in their ethnographic study of copy and paste programming practices, Kim et al. [55] anticipated that developers may be inclined to refactor larger and frequently copied fragments.

Tairas and Gray [95] investigated the refactoring of code clones retrospectively in an open source system to find whether developers refactor sub-clones or not. Based on two separate studies, they found a considerable number of instances of sub-clone refactoring where only part of the clone ranges are actually refactored. Finally, they concluded that facilities of sub-clone refactoring should be also incorporated to the clone management system. While their objective was finding sub-clone refactoring in open source system, we were interested to find out relevant attributes regarding clone removal in general.

Göde [33] is the first and so far the only one who investigated clone removal retrospectively from a maintainer's point of view. The objective of the study was to understand how developers see and deal with clones in real world software systems. Based on a case study of four subject systems, he found many instances of deliberate clone removal. He noticed that most of the clones were refactored through *method extraction*. He also initially investigated the attributes which developers prefer to refacor clones. However, the study was limited to only three attributes - LOC, text similarity, and distance between fragments in source code tree. After a preliminary investigation, he did not find any clear relationship except that developers refactored closely located clones. Finally, he concluded that more complex metrics are needed to identify which clones are attractive for removal. In this study we have considered a wide range of metrics derived from existing literature, and found some promising results to identify attractive clones for removal.

Choi et al. [17] used various combinations of metrics to extract clones for refactoring and asked a developer to determine the precision of the candidate clone set. Based on the feedback of the developer, they selected a combination of metrics that provided best precision for the given clone set. However, there are two potential threats to validity of their results. First, the result was derived from only one system and is dependent on the opinion of one developer. Second, their selection procedure was dependent on only three specific metrics. Furthermore, they did not use any metrics from the change history of clones, which is one of the most important resources to assess the vulnerability of clones. Unlike theirs, we investigated how the developers deal with clones in the real world software systems to get unbiased results, and have found more precise results than theirs using eight attributes and six open source systems.

In a recent study, Zibran and Roy [101] presented a refactoring effort model, and proposed a constraint programming approach for conflict-aware optimal scheduling of code clone refactoring. On the other hand, the objective of our study is to identify the attributes which developers prefer most when they actually remove clones in real world.

6.8 Summary

We have already seen that most of the software systems have a significant amount of cloned code. Previous study also showed that aggressive refactoring of all clone classes is not always beneficial. These actually call for selective refactoring. We believe investigating the attributes developers prefer to select clones for refactoring would be helpful to find out the appropriate clone classes for removal. In this chapter, we have

found several attributes such as fewer clone fragments in a clone class, low entropy of dispersion, recently introduced or modified clone classes that draw developers' attention in the decision making process of clone removal. We have also found that many such clones are still alive that are worth refactoring. We believe that the practical findings from this study along with the existing theory about clone refactoring will be helpful to develop a comprehensive refactoring engine in future, which will be able to take advantage of cloning by minimizing its negative effects.

Chapter 7

Conclusion

7.1 Concluding Remarks

Whether clones are useful or harmful, it is believed that they have an impact on software maintenance. It is also not possible or worthwhile to refactor all the clones from a software system because of the trade-offs among the associated cost, risks and benefits of removing clones. Thus, the recent trend in clone research focuses on the management of clones in a cost-effective way, rather than removing them aggressively. However, without understanding how clones actually evolve in systems, it would be hard to manage them. Therefore, both the clone detector and the genealogy extractor are required to build a clone management system. While a clone detector allows developers to discover clones in the source code, a clone genealogy extractor allows them to apply proper treatment by giving relevant information about the past history of the clone classes. However, current knowledge about the evolution of code clones is mostly limited to Type-1 and Type-2 clones. In this thesis, we have advanced the state-of-the-art in the evolution of clone research in the context of both exact (Type-1), and near-miss (Type-2 and Type-3) clones.

Earlier, it was widely believed that clones are inherently undesirable and should be removed through refactoring. This conventional wisdom was first challenged by Kim et al. [56]. They showed that approximately between 34% and 36% of entire clones (Type-1 and Type-2) changed consistently at the revision level. Furthermore, many clones are volatile. Therefore, aggressively refactoring them would not be worthwhile. However, their study was performed on two small Java systems. Therefore, many other researchers including the authors themselves speculated that the selected systems might not have captured the characteristics of larger systems and thus, further empirical evaluations need to be carried out for larger systems of different programming languages. In order to see whether their conclusions hold for other systems as well (or not), in the first phase of this thesis (Chapter 3), we developed a clone genealogy extractor as of Kim et al. [56] and performed a large-scale empirical study at release level for evaluating code clone genealogies using 17 diverse categories of open source software systems written in four different programming languages. From our study, we got the similar results to Kim et al. [56]. In addition, our study reveals some other interesting characteristics of code clone genealogies. We have found that many clone classes are propagated through releases either without any changes or with changes just with identifiers renaming. Hence, it is possible that these types of genealogies do not need any extra care during software maintenance. We have also noticed

that clones are perhaps more manageable in smaller systems compared to larger ones. However, till now researchers have only focused on the evolution of Type-1 and Type-2 clones.

In the second phase of this thesis (Chapter 4), we have thus proposed a framework that can automatically extract (Type-1) and classify near-miss (Type-2 andType-3) clone genealogies. In order to validate our proposed framework, we have developed a prototype clone genealogy extractor gCad, and experimented both with multiple versions of three open source systems including the Linux Kernel and a mutation/injection-based framework. We also manually analyzed many detected genealogies including their change patterns, and compared the mapping reported by our prototype with that of an incremental clone detection tool. Our experience suggests that gCad is reasonably fast while maintaining high precision and recall. It is also adaptable to other clone detection tools. Furthermore, it can even tolerate significant changes between two versions and thus could be effectively used to map clones at the release level as well as at the revision level. We believe that this approach would be useful for researchers and practitioners when studying the evolution of both exact and near-miss clones.

Third, we have performed an empirical study on Type-1, Type-2 and Type-3 clones in six open source systems (Chapter 5) using an extended version of gCad. In this study, we unveiled some interesting characteristics about the evolution of both exact and near-miss clones that either contradict some existing findings or are new to the literature. Some of the important findings are: the rate of inconsistent change in Type-3 clones is higher compared to other types, there is a higher frequency of changes to Type-3 clones, and transition from one type of clones to another type during evolution. We also identified some hotspots, where inconsistent changes may lead to error.

Finally, in the last phase of this thesis (Chapter 6), we have investigated the removal of clones from the developer's point of view in order to recognize which attributes they consider to refactor most of the clones. We have found that several attributes such as fewer number of clone fragments in a clone class, low entropy of dispersion, recently introduced or modified clone classes draw developer's attention more than the other attributes. We have also found that many such clones are still alive which are worth refactoring. We believe that the practical findings from this study along with the existing theory about clone refactoring would be helpful to develop a comprehensive refactoring engine in the future, which will be able to take advantage of cloning by minimizing its negative effects.

7.2 Future Research Directions

The work presented in this thesis provides opportunities for more research into the area of clone evolution and help to build a clone management system. In this section we discuss some of the opportunities.

7.2.1 Reordering of Lines

In order to track clones across versions we used the *longest common subsequence* algorithm. However, the limitation of LCS is that it cannot capture the reordering of lines. Although we have overcome this limitation by first mapping functions using their signatures, it may affect the result when function names have been changed in the next version. Therefore, the quality of mapping could be improved by using a more sophisticated text similarity algorithm which is able to capture the reordering of lines.

7.2.2 Late Propagation

It is a specific change pattern of clone evolution where one or more clone fragments in a clone class are changed inconsistently in one version but in a later version the same change is propagated to the rest of the clone fragments of that class and all the clone fragments become again synchronized [5]. Although there are a couple of studies [5, 9] where researchers detected *late propagation* for Type-1 and Type-2 clones, there is no tool that can detect late propagation for Type-3 clones. In the future we plan to incorporate this change pattern to gCad so that we can effectively deal with Type-3 clones.

7.2.3 Unintentional Inconsistent Changes

Unintentional inconsistent changes to clone classes have always been a concern to developers because they may often result in unexpected behaviour of programs. However, a couple of studies [37] including ours show that inconsistent changes to code clones are frequent. Therefore, it would be great if it were possible to somehow identify which inconsistent changes were intentional and which were not.

7.2.4 Study of Clone Evolution in IDE:

Although studying the evolution of clones is important to manage code clones properly, the available tool support in IDEs is still inadequate. Enabling developers to analyze the evolution of clones within an IDE could be a good future research project.

7.2.5 Visualizing the Evolution of Clones

A genealogy extractor provides relevant information about evolving clones to comprehend and manage them properly and cost-effectively. However, investigating and observing facts in a huge set of text-based data provided by a clone genealogy extractor is a time consuming and cumbersome task without the support of a visualization tool. Although visualizing the evolution of code clones is out of the scope of this thesis, from the experience while analyzing the evolutionary data of code clones, we have proposed an idea for effectively visualizing clone evolution. To get a more comprehensive description about the idea, readers are referred to the paper [93]. In future, we would like to implement the idea on top of gCad to visualize the evolution of code

clones. Finally, we believe all these findings will help build a comprehensive clone analysis and management system [102, 103].

References

[1] Raihan Al-Ekram, Cory Kapser, Richard Holt, and Michael Godfrey. Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems. In *Proceedings of the 2005 International Symposium on Empirical Software Engineering*, ISESE '05, pages 376–385, Noosa Heads, Australia, 2005. IEEE Computer Society.

[2] Ghazi Alkhatib. The Maintenance Problem of Application Software: An Empirical Analysis. *Journal of Software Maintenance*, 4:83–104, 1992.

[3] Giuliano Antoniol, Massimiliano Di Penta, Gerardo Casazza, and Ettore Merlo. Modeling Clones Evolution Through Time Series. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '01, pages 273–280, Florence, Italy, 2001. IEEE Computer Society.

[4] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. Analyzing Cloning Evolution in the Linux Kernel. *Information & Software Technology*, 44(13):755–765, 2002.

[5] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How Clones are Maintained: An Empirical Study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, CSMR '07, pages 81–90, Amsterdam, The Netherlands, 2007. IEEE Computer Society.

[6] Brenda S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, pages 86–95, Toronto, Canada, 1995. IEEE Computer Society.

[7] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. Clone Smells in Software Evolution. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, ICSM '07, pages 24 –33, Paris, France, 2007. IEEE Computer Society.

[8] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Advanced Clone-Analysis to Support Object-Oriented System Refactoring. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, WCRE '00, pages 98–107. IEEE Computer Society, 2000.

[9] Liliane Barbour, Foutse Khomh, and Ying Zou. Late Propagation in Software Clones. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 273–282, Williamsburg, VA, USA, 2011. IEEE Computer Society.

[10] Hamid A. Basit, Damith C. Rajapakse, and Stan Jarzabek. Beyond Templates: A Study of Clones in the STL and Some General Implications. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 451–459, St. Louis, MO, USA, 2005. ACM.

[11] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377, Bethesda, Maryland, USA, 1998. IEEE Computer Society.

[12] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.*, 33:577–591, 2007.

[13] Nicolas Bettenburg, Weyi Shang, Walid Ibrahim, Bram Adams, Ying Zou, and Ahmed E. Hassan. An Empirical Study on Inconsistent Changes to Code Clones at Release Level. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 85–94, Lille, France, 2009. IEEE Computer Society.

[14] Dongxiang Cai and Miryung Kim. An Empirical Study of Long-Lived Code Clones. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: part of the joint European Conferences on Theory and Practice of Software*, FASE '11/ETAPS '11, pages 432–446, Saarbrücken, Germany, 2011.

[15] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. Function Clone Detection in Web Applications: A Semiautomated Approach. *J. Web Eng.*, 3:3–21, 2004.

[16] Moses S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, STOC '02, pages 380–388, Montreal, Quebec, Canada, 2002. ACM.

[17] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, and Tateki Sano. Extracting Code Clones for Refactoring using Combinations of Clone Metrics. In *Proceeding of the 5th international workshop on Software clones*, IWSC '11, pages 7–13, Waikiki, Honolulu, HI, USA, 2011. ACM.

[18] James R. Cordy. Comprehending Reality: Practical Challenges to Industrial Adoption of Software Maintenance Automation. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 196–206, Portland, Oregon, USA, 2003. IEEE Computer Society.

[19] James R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61:190–210, 2006.

[20] James R. Cordy and Chanchal K. Roy. The NiCad Clone Detector. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*, ICPC '11, pages 219–220, Kingston, ON, Canada, 2011. IEEE Computer Society.

[21] Neil Davey, Paul Barson, Simon Field, Ray Frank, and Stewart Tansley. The Development of a Software Clone Detector. *International Journal of Applied Software Technology*, 1:219–236, 1995.

[22] Ian J. Davis and Michael W. Godfrey. Clone Detection by Exploiting Assembler. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 77–78, Cape Town, South Africa, 2010. ACM.

[23] Ian J. Davis and Michael W. Godfrey. From Whence It Came: Detecting Source Code Clones by Analyzing Assembler. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 242–246, Beverly, MA, USA, 2010. IEEE Computer Society.

[24] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Michael Pfaehler, and Bernhard Schaetz. Model Clone Detection in Practice. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 57–64, Cape Town, South Africa, 2010. ACM.

[25] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 603–612, Leipzig, Germany, 2008. ACM.

[26] Christoph Domann, Elmar Juergens, and Jonathan Streit. The Curse of Copy&Paste Cloning in Requirements Specifications. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 443–446, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking Code Clones in Evolving Software. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 158–167, Minneapolis, USA, 2007. IEEE Computer Society.

[28] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 109–118, Oxford, England, UK, 1999. IEEE Computer Society.

[29] Richard Fanta and Václav Rajlich. Removing Clones from the Code. *Journal of Software Maintenance*, 11:223–243, 1999.

[30] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, 1987.

[31] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1st edition, 1999.

[32] Nils Göde. Evolution of Type-1 Clones. In *Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 77–86, Edmonton, Canada, 2009. IEEE Computer Society.

[33] Nils Göde. Clone Removal: Fact or Fiction? In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 33–40, Cape Town, South Africa, 2010. ACM.

[34] Nils Göde and Jan Harder. Clone Stability. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 65–74, Oldenburg, Germany, 2011. IEEE Computer Society.

[35] Nils Göde and Jan Harder. Oops! . . . I Changed It Again. In *Proceeding of the 5th international workshop on Software clones*, IWSC '11, pages 14–20, Waikiki, Honolulu, HI, USA, 2011. ACM.

[36] Nils Göde and Rainer Koschke. Studying Clone Evolution Using Incremental Clone Detection. *Journal of Software Maintenance and Evolution: Research and Practice*, 2010.

[37] Nils Göde and Rainer Koschke. Frequency and Risks of Changes to Clones. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 311–320, Waikiki, Honolulu, HI, USA, 2011. ACM.

[38] Michael Godfrey and Qiang Tu. Growth, Evolution, and Structural Change in Open Source Software. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 103–106, Vienna, Austria, 2001. ACM.

[39] Michael W. Godfrey and Lijie Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Trans. Softw. Eng.*, 31:166–181, 2005.

[40] Jan Harder and Nils Göde. Modeling Clone Evolution. In *Proceedings of the 3rd International Workshop on Software Clones*, IWSC '09, pages 17–21, Kaiserslautern, Germany, 2009.

[41] Jan Harder and Nils Göde. Efficiently Handling Clone Data: RCF and Cyclone. In *Proceeding of the 5th international workshop on Software clones*, IWSC '11, pages 81–82, Waikiki, Honolulu, HI, USA, 2011. ACM.

[42] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Refactoring Support Based on Code Clone Analysis. *Lecture Notes in Computer Science*, 3009:220–233, 2004.

[43] 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 *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 73–82, Antwerp, Belgium, 2010. ACM.

[44] Pankaj Jalote. *An Integrated Approach to Software Engineering*. Springer-Verlag New York, Inc., 1991.

[45] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105, Minneapolis, USA, 2007. IEEE Computer Society.

[46] John H. Johnson. Identifying Redundancy in Source Code Using Fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*, CASCON '93, pages 171–183, Toronto, Ontario, Canada, 1993. IBM Press.

[47] John H. Johnson. Substring Matching for Clone Detection and Change Tracking. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 120–126, Victoria, BC, Canada, 1994. IEEE Computer Society.

[48] Elmar Juergens, Florian Deissenboeck, Martin Feilkas, Benjamin Hummel, Bernhard Schaetz, Stefan Wagner, Christoph Domann, and Jonathan Streit. Can Clone Detection Support Quality Assessments of Requirements Specifications? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 79–88, Cape Town, South Africa, 2010. ACM.

[49] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do Code Clones Matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Vancouver, Canada, 2009. IEEE Computer Society.

[50] Nicolas Juillerat and Beat Hirsbrunner. An Algorithm for Detecting and Removing Clones in Java Code. In *Proceedings of the 3rd International Workshop on Software Evolution through Transformations*, SeTra '06, pages 63–74, 2006.

[51] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.*, 28:654–670, 2002.

[52] Cory Kapser and Michael W. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 19–28, Benevento, Italy, 2006. IEEE Computer Society.

[53] Cory J. Kapser and Michael W. Godfrey. Supporting the Analysis of Clones in Software Systems: Research Articles. *J. Softw. Maint. Evol.*, 18:61–82, 2006.

[54] Cory J. Kapser and Michael W. Godfrey. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software. *Empirical Softw. Engg.*, 13:645–692, 2008.

[55] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ISESE '04, pages 83–92, Redondo Beach, CA, USA, 2004. IEEE Computer Society.

[56] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE '05, pages 187–196, Lisbon, Portugal, 2005. ACM.

[57] Sunghun Kim, Kai Pan, and E. James Whitehead, Jr. When Functions Change Their Names: Automatic Detection of Origin Relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 143–152, Pittsburgh, PA, USA, 2005. IEEE Computer Society.

[58] Raghavan Komondoor and Susan Horwitz. Effective, Automatic Procedure Extraction. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 33–42, Portland, Oregon, USA, 2003. IEEE Computer Society.

[59] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Benevento, Italy, 2006. IEEE Computer Society.

[60] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–309, Stuttgart, Germany, 2001. IEEE Computer Society.

[61] Jens Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 170–178, Vancouver, BC, Canada, 2007. IEEE Computer Society.

[62] Jens Krinke. Is Cloned Code More Stable than Non-cloned Code? In *Proceeding of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '08, pages 57 –66, Beijing, China, 2008. IEEE Computer Society.

[63] Jens Krinke. Is Cloned Code older than Non-Cloned Code? In *Proceeding of the 5th international workshop on Software clones*, IWSC '11, pages 28–33, Waikiki, Honolulu, HI, USA, 2011. ACM.

[64] Bruno Lagüe, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proceedings of the International Conference on Software Maintenance*, ICSM '97, pages 314–321, Bari, Italy, 1997. IEEE Computer Society.

[65] Seunghak Lee and Iryoung Jeong. SDD: High Performance Code Clone Detection System for Large Scale Source Code. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 140–141, San Diego, CA, USA, 2005. ACM.

[66] António M. Leitão. Detection of Redundant Code Using R^2D^2. *Software Quality Control*, 12:361–382, 2004.

[67] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-Paste and Related Bugs in Operating System Code. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI '04, pages 20–20, San Francisco, CA, 2004. USENIX Association.

[68] Hui Liu, Zhiyi Ma, Lu Zhang, and Weizhong Shao. Detecting Duplications in Sequence Diagrams Based on Suffix Trees. In *Proceedings of the XIII Asia Pacific Software Engineering Conference*, APSEC '06, pages 269–276, Bangalore, India, 2006. IEEE Computer Society.

[69] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. In *Proceedings of the 24th International Conference on Software Maintenance*, ICSM '08, pages 227–236, Beijing, China, 2008. IEEE Computer Society.

[70] Angela Lozano and Michel Wermelinger. Tracking Clones' Imprint. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 65–72, Cape Town, South Africa, 2010. ACM.

[71] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the Harmfulness of Cloning: A Change Based Experiment. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 18–21, Minneapolis, USA, 2007. IEEE Computer Society.

[72] Giuseppe A. Di Lucca, Massimiliano Di Penta, and Anna Rita Fasolino. An Approach to Identify Duplicated Web Pages. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, COMPSAC '02, pages 481–486, Oxford, England, 2002. IEEE Computer Society.

[73] Udi Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, USENIX '94, pages 1–10, San Francisco, California, 1994.

[74] Andrian Marcus and Jonathan I. Maletic. Identification of High-Level Concept Clones in Source Code. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 107–114, Lawrence, Kansas, USA, 2001. IEEE Computer Society.

[75] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of the 12th International Conference on Software Maintenance*, ICSM '96, pages 244–. IEEE Computer Society, 1996.

[76] Robert C. Miller and Brad A. Myers. Interactive Simultaneous Editing of Multiple Text Regions. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, USENIX '01, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association.

[77] Manishankar Mondal, M. Saidur Rahman, Ripon K. Saha, Chanchal K. Roy, Jens Krinke, and Kevin A. Schneider. An Empirical Study of the Impacts of Clones in Software Maintenance. In *Proceedings of the Student Research Symposium Track of the 19th IEEE International Conference on Program Comprehension*, pages 242–245, Kingston, ON, Canada, 2011. IEEE Computer Society.

[78] Manishankar Mondal, Chanchal K. Roy, M. Saidur Rahman, Ripon K. Saha, Jens Krinke, and Kevin A. Schneider. Comparative Stability of Cloned and Non-cloned Code: An Empirical Study. In *Proceedings of the Software Engineering Track of the 27th ACM Symposium on Applied Computing*, SAC '12, 8 pages, Riva del Garda, Trento, Italy, 2012. ACM. (to appear).

[79] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 87–94, Ottawa, Canada, 2002. IEEE Computer Society.

[80] Hoan A. Nguyen, Tung T. Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 440–455, York, UK, 2009. Springer-Verlag.

[81] Tung T. Nguyen, Hoan A. Nguyen, Jafar M. Al-Kofahi, Nam H. Pham, and Tien N. Nguyen. Scalable and incremental clone detection for evolving software. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, ICSM '09, pages 491–494, Edmonton, Alberta, Canada, 2009. IEEE Computer Society.

[82] Jean-Francois Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. Extending Software Quality Assessment Techniques to Java Systems. In *Proceedings of the 7th International Workshop on Program Comprehension*, IWPC '99, pages 49–56, Pittsburgh, Pennsylvania, USA, 1999. IEEE Computer Society.

[83] Nam H. Pham, Hoan A. Nguyen, Tung T. Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 276–286, Vancouver, Canada, 2009. IEEE Computer Society.

[84] Foyzur Rahman, Christian Bird, and Premkumar T. Devanbu. Clones: What is that Smell? In *Proceedings of the 7th International Working Conference on Mining Software Repositories*, MSR '10, pages 72–81, Cape Town, South Africa, 2010. IEEE Computer Society.

[85] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into System-Wide Code Duplication. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 100–109. IEEE Computer Society, 2004.

[86] Chanchal K. Roy and James R. Cordy. A Survey on Software Clone Detection Research. Tech. Report 541, Queens School of Computing, Kingston, ON, Canada, 2007.

[87] Chanchal K. Roy and James R. Cordy. NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, Amsterdam, The Netherlands, 2008. IEEE Computer Society.

[88] Chanchal K. Roy and James R. Cordy. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '09, pages 157–166, Denver, Colorado, USA, 2009. IEEE Computer Society.

[89] Chanchal K. Roy and James R. Cordy. Near-miss Function Clones in Open Source Software: An Empirical Study. *J. Softw. Maint. Evol.*, 22:165–189, 2010.

[90] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting Code Clones in Binary Executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 117–128, Chicago, IL, USA, 2009. ACM.

[91] Ripon K. Saha, Muhammad Asaduzzaman, Minhaz F. Zibran, Chanchal K. Roy, and Kevin A. Schneider. Evaluating Code Clone Genealogies at Release Level: An Empirical Study. In *Proceedings of the Tenth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '10, pages 87–96, Timisoara, Romania, 2010. IEEE Computer Society.

[92] Ripon K. Saha, Chanchal K. Roy, and Kevin A. Schneider. An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 293–302, Williamsburg, VA, USA, 2011. IEEE Computer Society.

[93] Ripon K. Saha, Chanchal K. Roy, and Kevin A. Schneider. Visualizing the Evolution of Code Clones. In *Proceeding of the 5th international workshop on Software clones*, IWSC '11, pages 71–72, Waikiki, Honolulu, HI, USA, 2011. ACM.

[94] Harald Störrle. Towards clone detection in UML domain models. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 285–293, Copenhagen, Denmark, 2010. ACM.

[95] Robert Tairas and Jeff Gray. Sub-clones: Considering the Part Rather than the Whole. In *Proceedings of the 2010 International Conference on Software Engineering Research & Practice*, SERP '10, pages 284–290, Las Vegas, Nevada, USA, 2010.

[96] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An Empirical Study on the Maintenance of Source Code Clones. *Empirical Software Engineering*, 15:1–34, 2010.

[97] Rebecca Tiarks, Rainer Koschke, and Raimar Falke. An extended assessment of type-3 clones as detected by state-of-the-art tools. *Software Quality Journal*, 19(2):295–331, 2011-06-01.

[98] 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 *Proceedings of the 18th IEEE Working Conference on Reverse Engineering*, WCRE '11, pages 13–22, Limerick, Ireland, 2011. IEEE Computer Society.

[99] Richard Wettel and Radu Marinescu. Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments. In *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC '05, pages 63–70, Timisoara, Romania, 2005. IEEE Computer Society.

[100] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE Trans. Softw. Eng.*, 30:574–586, 2004.

[101] Minhaz F. Zibran and Chanchal K. Roy. A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '11, pages 105–114, Williamsburg, VA, USA, 2011. IEEE Computer Society.

[102] Minhaz F. Zibran and Chanchal K. Roy. Towards Flexible Code Clone Detection, Management, and Refactoring in IDE. In *Proceeding of the 5th international workshop on Software clones*, IWSC '11, pages 75–76, Waikiki, Honolulu, HI, USA, 2011. ACM.

[103] Minhaz F. Zibran and Chanchal K. Roy. IDE-based Real-time Focused Search for Near-miss Clones. In *Proceedings of the Software Engineering Track of the 27th ACM Symposium on Applied Computing*, SAC '12, 8 pages, Riva del Garda, Trento, Italy, 2012. ACM. (to appear).

[104] Minhaz F. Zibran, Ripon K. Saha, Muhammad Asaduzzaman, and Chanchal K. Roy. Analyzing and Forecasting Near-Miss Clones in Evolving Software: An Empirical Study. In *Proceeding of the 16th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '11, pages 295–304, Las Vegas, Nevada, USA, 2011. IEEE Computer Society.

[105] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Scotland, UK, 2004. IEEE Computer Society.