MINING AND ANALYSIS OF CONTROL STRUCTURE VARIANT CLONES

Guo Qiao

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

April 2015

© Guo Qiao, 2015

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Guo Qiao**
Entitled:	**Mining and Analysis of Control Structure Variant Clones**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. Adam Krzyzak ______	Chair
Dr. Juergen Rilling ______	Examiner
Dr. Emad Shihab ______	Examiner
Dr. Nikolaos Tsantalis ______	Supervisor

Approved by ______

Chair of Department or Graduate Program Director

______

Dean of Faculty

Date ______

Abstract

Mining and Analysis of Control Structure Variant Clones

Guo Qiao

Code duplication (software clones) is a very common phenomenon in existing software systems, and is also considered to be an indication of poor software maintainability. In recent years, the detection of clones has drawn considerable attention. The majority of existing clone detection techniques focus on the syntactic similarity of code fragments, and more specifically, they support the detection of Type-1 clones (i.e., identical code fragments except for variations in whitespace, layout, and comments), Type-2 clones (i.e., structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout, and comments), and Type-3 clones (i.e., copied fragments with statements changed, added, or removed in addition to variations in identifiers, literals, types, layout and comments).

However, recent studies have shown that when developers implement the same functionalities, their code solutions may differ substantially in terms of their syntactical structure. This is because developers follow different programming styles or language features when implementing, for instance, control structures, such as loops and conditionals. From the perspective of clone management, different strategies are required to detect and refactor these control structure variant clones. Thus, there is a clear need for functionality-aware clone mining approaches, which are capable of distinguishing functional clones from syntactical clones.

In this thesis, we are proposing a method for mining control structure variant clones. More specifically, the proposed approach can mine clones which use different, but functionally equivalent control structures to implement functionally similar iterations and conditionals. Our method is evaluated on six open-source systems by manually inspecting the mined clones and computing the precision and recall of our technique. Moreover, we create a publicly available benchmark of control structure variant clones. Based on the clones we found, we also propose some improvements to tackle the limitations of JDeodorant in the refactoring of control structure variant clones.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Nikolaos Tsantalis, for his continuous support towards my research work and the writing of my thesis, as well as for his patience, motivation, and immense knowledge. Without his guidance and support, I could not have completed this thesis.

Apart from my advisor, I would also like to thank my thesis examiners, Dr. Juergen Rilling and Dr. Emad Shihab, for taking the time to read my thesis and for their valuable suggestions.

I would like to thank Dr. Iman Keivanloo for his help in my research, as well as all other faculty members of the Department of Computer Science and Software Engineering, for providing the necessary guidance.

I thank my fellow lab mates: Davood Mazinanian for his continuous help in my research, Zackary Valenta, for his great contribution to this project, Giri Panamoottil Krishnan for helping me to start my work, and my other friends at Concordia University for all their support towards the completion of this thesis.

I would also like to thank NSERC, the Faculty of Engineering and Computer Science, and the Financial Aid and Awards Office at Concordia University for their generous financial support for my studies.

Last but not least, I would like to thank my family, especially my parents, for their love, selfless support, and encouragement to finish my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter discusses the motivation of this thesis and challenges involved, and also briefly introduces the contributions and describe the overall approach.

1.1 Software Maintenance and code duplication

Software maintenance is defined as the process of modifying software systems or components after the delivery to the customer. Maintenance activities involve correcting faults, improving attributes, or adapting to a changed environment. There are four types of maintenance according to Lientz and Swanson [LS80]: corrective, adaptive, perfective, and preventive.

1. Corrective maintenance deals with the repair of faults found.

2. Adaptive maintenance deals with adapting the software to changes in the environment, such as new hardware or the next release of an operating system. Adaptive maintenance does not lead to changes in the system's functionality.

3. Perfective maintenance mainly deals with accommodating new or changed user requirements. It concerns functional enhancements to the system.

4. Preventive maintenance concerns activities aimed at increasing the system's maintainability, such as updating documentation, adding comments, and improving design quality.

In the past decades, a great deal of software systems have been created in the industry, and these systems are important to the enterprises. Meanwhile, software maintenance is gaining more importance, because maintenance activities have become a major part of the software development process. At present, among the global software population, more than 77% of the people are engaged in software maintenance compared to 23% developing new applications [Jon06]. Recent studies have also shown that most of the effort and cost is spent during the maintenance phase of the entire software life cycle. The total cost of system maintenance is estimated to comprise at least 50% of total life cycle costs [Vli08]. Particularly for large systems which are modified frequently, the maintenance cost could be more than the sum of all the other development phases. Thus, in recent years, there has been an increasing interest in the field of software maintenance research.

If the quality of software did not decline as time goes on, software maintenance would be much easier. However, software is just like people, and thus aging is inevitable. Software aging is a phenomenon plaguing many long-running complex software systems, which exhibit performance degradation or an increasing failure rate [CNPR14]. The aging process may be caused by many reasons, but one of the most important reasons is known as "Ignorant surgery" [Par94]. This refers to the maintenance activities performed by new programmers, who may not understand the original design concept of the system. In the entire life cycle of large-scale systems, it is rare for the people who maintain the system to be the same people who initially built it. "Ignorant surgery" may not affect the execution of the system, so people could easily ignore the negative effects to the system design quality.

Code clones are very common results of this kind of "Ignorant surgery" and also considered an indication of poor software maintainability and system aging. Due to subjective or objective reasons, a programmer may choose to copy and paste an existing code fragment instead of understanding the design and seeking the opportunity for code reuse. As more and more clones are introduced, the system may not crash immediately but the system design quality will decline rapidly. The original design structure would be modified beyond recognition, because of random clone patches. If this continues, the system will gradually become unmaintainable. Besides the disastrous consequences that could be caused by code clones, these clones could also increase maintenance cost significantly. An example of this would be, if there is a bug in one clone fragment and this piece of code is copied and pasted in ten different places in the system. When a programmer wants to fix this bug, he needs to find all ten clones and fix them one by one, which would take ten times the effort normally required. In a large system, without well-recorded documentation, this task would be very difficult, time-consuming, and error-prone.

After all this discussion, it is intriguing to know how many code clones could actually exist in industrial systems. Currently, software code clone management is a major area of interest within the field of software maintenance research. In a previous study about the existence of code clones, Chen et al. [CWT14] analyzed 43 open-source Java projects. They found that the percentage of clones in all those systems varied from 6.5% to 59.5%. The average proportion of clones is 14.6% in general. In a recent case study conducted by Kapser and Godfrey, 13% of 4407 functions from the Linux File System are considered clones [KG03].

1.2 Software Code Clones

Software code clones are code fragments which are considered identical or similar to each other. Code fragments *CF1* and *CF2* are clones of each other, if they are consider similar according to some given definition of similarity. If f is the similarity function, while *CF1* and *CF2* comply with *f(CF1) = f(CF2)*, then they can form a clone pair *(CF1,CF2)*. If there are more than two code fragments which are considered similar, they then form a clone class or clone group *(CF1, CF2, CF3 etc.)*.

1.2.1 Clone Types

The most widely accepted definition of code clone types is from Roy et al. [RCK09]. On the basis of variant differences, which exist between code clones, Roy et al. categorized clones into four types.

Type-1: Identical code fragments except for variations in whitespace, layout and comments.

Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type-3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Type-4: Two or more code fragments that perform the same computation but are implemented by different syntax text.

According to the above definitions, it is clear that Type-1 and Type-2 code clone definitions are more straightforward, since they are supposed to have the same number of code statements. The definition of Type-3 code clone is more abstract, because it is difficult to find out if a code fragment is copied or not. Aside from that, there is no specific explanation of how many additional statements between two Type-3 clones are acceptable. Thus, Type-3 clone detection tools normally provide a parameter to configure the acceptable clone variance ratio in order to control the range of the results. In general, Type-1, Type-2, and Type-3 clones are syntactically similar, thus, they are also classified as *Syntactical clones*. Type-4 clones are the most challenging clones to detect, because they may differ substantially in terms of their syntactical context.

1.2.2 Type-4 clones and Semantic clones

Due to the broad definition of Type-4 clones, in recent years, many works introduced new terms about Type-4 clones, such as semantic clones, functionality equivalent clones, or simions. However, to date, there is no certain consensus on the definition of Type-4 clones. There are two main opinions, one supports that Type-4 clones are the same as semantic clones, the other opinion supports that semantic clones should have identical Program Dependence Graphs [FOW87] (PDGs) while Type-4 clones may not. In order to elaborate more on the problem, we list some different definitions below.

Elva et al. [ELE+12]: Semantic clones are functionally identical code fragments.

Gabel et al. [GJS08]: Two disjoint, possibly noncontiguous sequences of program syntax S1 and S2 are semantic code clones if and only if S1 and S2 are syntactic code clones or PDG of S1 is isomorphic to PDG of S2.

Kamiya [Kam13]: Semantic clones are code fragments equivalent in terms of method invocations.

Yoshioka et al. [YYFI11]: If a pair of code fragments have similar control flow and have many overlapping statements, we regard them as "Semantic similar clones".

Shafieian and Zou [SZ12] support that Type-4 clones are also called semantic clones and this type of clone is undecidable in general.

Kim et al. [KJKY11] support that Type-4 clones are semantic clones and proposed four subcategories of Type-4 clones.

Jürgens et al. [JDH10] introduce *Simions*, which are behaviorally similar code fragments, where behavioral similarity is defined with respect to input/output behavior. Meanwhile, they support that *Simions* are comparable to Type-4 clones.

Based on the aforementioned definitions, it is clear that the definition of Type-4 clone remains inconclusive. The only thing for sure is that syntax variation is an important feature of Type-4 clones. However, syntactical differences could be present in many forms and some of them could be trivial differences which do not affect code functionality (e.g. Rename Variable, Different Literal Value, etc.). On the contrary, some differences are related to the computation performed in the code (e.g. Different Control Structure, Different Data Structure, etc.). Thus, it is difficult to clarify all possible scenarios with one definition.

Wang et al. [WWSM14] proposed eight types of code variations, one of those variations is control structure variation. Jürgens et al. [JDH10] manually found functionally equivalent code fragments which use different control structures. Kim et al. [KJKY11] proposed a subcategory of Type-4 clones, having control replacement with semantically equivalent control structure. It refers to clones that use different control structures to implement the same loop scenario. It is clear that among all kinds of syntax variations, control structure variation has drawn much attention. Aside from that, control structure is directly related to the computation performed in the code. In order to avoid confusion, we introduce the term *Control Structure Variant Clones*, which are clones that use different control structures to implement the same functionality. Furthermore, we support that control structure variant clones is a subcategory of Type-4 clones. The aim of this thesis is to investigate control structure variant clones, which may include syntactical differences. Thus, it does not engage with Type-1, Type-2 and Type-3 clones.

1.2.3 Control Structure Variant Clones

All the above-mentioned types of clones could be produced in any stage of the software life cycle. However, control structure variant clones and syntactical clones are normally produced by completely different activities, and the causes of the creation of code clones can be divided into two categories. Most syntactical clones, which share textually similarities are results of copy-paste activities. In the coding process, programmers are normally inclined to choose low-cost copy-paste techniques instead of implementing everything from scratch. This makes programmers believe that they can save time and effort, but it may cause potential problems for system maintenance in the future. On the other hand, control structure variant clones normally result from different implementations of the same functionality. Programmers work independently without knowing that they are implementing the functionality which already exists. Thus, control structure variant clones are more likely to be created unintentionally and differ substantially. In the industry, when developers face deadline pressures and cost restrictions, they are more likely to perform cloning activities, but not to investigate the system design and legacy code. Thus, we have the hypothesis that control structure variant clones occur much less frequently than syntactical clones.

Control structure variant clones are created because of two main reasons. *New control constructs*: The programming languages evolve over time by adding new features and constructs to make the development of code easier and more flexible, and thus increase the productivity of the developers. A typical example is the introduction of the *enhanced for* or *for each* loop in Java 5. Along with the programming languages, software systems evolve by adapting existing code to the new features and constructs [DRNN13]. However, developers might update only a subset of the clones in an existing clone group to make use of the new language constructs (i.e., inconsistent clone evolution), mainly because they are not aware of the existence of the rest of the clones (e.g., they are responsible for maintaining a specific part of the system containing only a subset of the clones). Due to these inconsistent updates, the original clones become *divergent* [BKZ11], and it therefore becomes harder to keep track of them and co-evolve them consistently.

Diverse programming styles: Developers tend to follow different programming styles when implementing iterations and conditionals [BSK11]. For instance, some developers prefer the use of *enhanced for* over *iterator-based* loops, or in some cases, find the use of the conditional (or ternary) operator `?:` more convenient instead of using the traditional `if-else` control structure. Due to these diverse programming styles, developers working in team projects introduce clones with radically different control structures when implementing similar functionalities.

1.3 Software Clone Management

For the purpose of efficient clone management, it is necessary to distinguish control structure variant clones from simple syntax clones. Clone detection is an important part of clone research, however, the management of clones is equally important. Due to the negative effects caused by software clones, researchers and practitioners have agreed that code clones should be managed efficiently. "Clone management summarizes all process activities which are targeted at detecting, avoiding, or removing clones" [Gie06]. Thus, clone management involves a wide range of activities: detecting, refactoring, visualizing, and tracking of clones. In all these activities, clone refactoring is the best way to reduce or eliminate the negative effects of clones.

Fowler [Fow99] summarized some approaches for clone refactoring, such as *Extract method*, *Pull up method*, *Form template method*, etc. The core idea of all these approaches is to extract the common code and put it in a new method in order to eliminate the clones. For Type-1 clones, the common code extraction is much easier because they have exactly the same code statements. For Type-3 clones, one challenge is the presence of gapped statements, which are additional or missing statements. Some precondition checking is required to make sure the gapped statements can be moved outside the common code without changing the original functionality [KT14]. However, for control structure variant clones, the strategy required is more complex. Because the syntax difference is semantically equivalent, thus, instead of moving the differences outside the common code, we need to further analyze the functionality of the clones. More specifically, we need to first unify the different code syntax without changing the functionality, and then replace the original code with the unified code. Thus, from the perspective of clone refactoring, we support that it is necessary to distinguish functionally equivalent clones from syntactically equivalent clones.

In previous studies, syntactical clones have already been handled very well by existing clone management tools. However, few tools are able to deal with Type-4 clones which are syntactically dissimilar but functionally equivalent. More specifically, few tools are able to deal with Type-4 clones which are syntactically dissimilar, but functionally equivalent. More specifically, few tools can detect Type-4 clones with high precision, and no existing tool can interpret and analyze the syntax differences existing in Type-4 clones. In order to extend the benefits of clone management to a wider range of clones, an approach, which can filter out functional clones with high accuracy and also being able to interpret cloning information is required. This thesis will address these open problems and also investigate real open-source projects to find functional clones in order to create a clone benchmark.

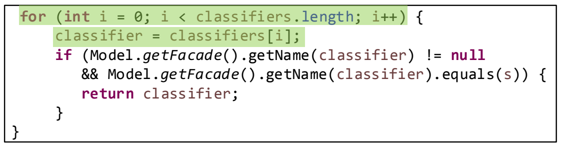
1.4 Research Challenges in the unification of Type-4 clones

Many clone detection tools can support syntactical clone detection very well, e.g.CCFinder [KKI02], Deckard [JMSG07], CloneDR [BYM+98], Sebyte [KRR12a] and Cedar [TJG11]. However, Jürgens et al. [JDH10] carried out an experiment which shows that even the state-of-the-art token-based and AST-based clone detectors are not able to recognize semantically similar clones. Thus, the challenge is more about how to detect advanced clones based on their functionality instead of their syntax. In order to perform functionality analysis, we attempt to evaluate two main aspects of code - *control structure*, which represents the execution flow, as well as, *data structures, and method calls used*, which represent the code functionality. However, the detection process is not straightforward, since we want to recognize functionality equivalent clones rather than clones which are just the same. On one hand, among all the control structures which are frequently used in Java, we try to match functionally equivalent structures. On the other hand, based on all the data structures defined in Java, we try to detect compatible data structure usages.

1.4.1 Control Structure Variations

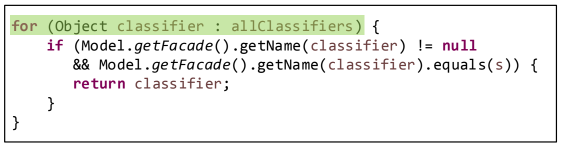
Control structures are used to organize the execution flow of code and are crucial for the implementation of functionality. Syntax-based techniques can only match control structures of the same type, but it is highly likely that different control structures are used for the same purpose. There are seven frequently-used control structures in Java - `For loop`, `Enhanced for loop`, `While loop`, `Do-While loop`, `If-else statement`, `Switch statement`, and `Conditional statement`. Apart from this, all the loop structures can be implemented as index-based (e.g., `i<10`) or iterator-based (e.g., `iterator.hasnext()`). In other words, Java supports various functionally equivalent control structures.

Normally, different control structures differ substantially in terms of code syntax. Thus, the approaches depending on the analysis of code syntax will not work for the detection of syntax-dissimilar clones. As illustrated in Figure 1, two clones implement the same functionality, they traverse the data structures `classfiers` and `allClassfiers`, and then pass the element `classifier` as an argument to the same method `Model.getFacade().getName(classifier)`. The for loop in clone (a) includes three parts in the conditional expression (Initialization, termination and increment), while the enhanced for loop statement only has two parts (variable declaration and name of collection/array). Aside from that, for the construction of general for loop, all those three expressions are optional. Consequently, significant syntactical differences could exist between the general for loop and enhanced for loop, which would introduce difficulties for the detection of this type of clone.



```
for (int i = 0; i < classifiers.length; i++) {
    classifier = classifiers[i];
    if (Model.getFacade().getName(classifier) != null
        && Model.getFacade().getName(classifier).equals(s)) {
        return classifier;
    }
}
```

(a) For loop implementation

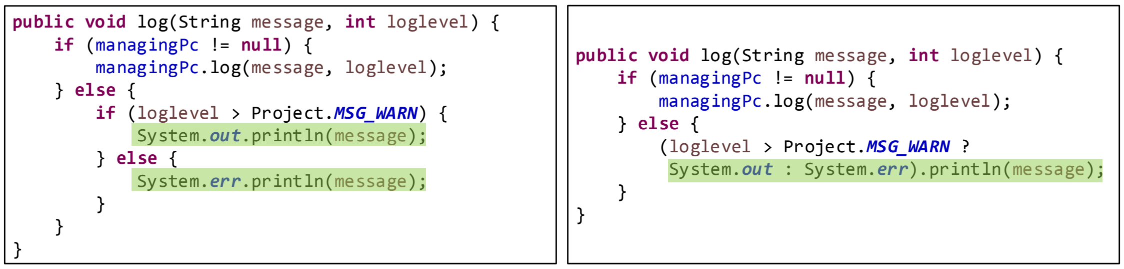


```
for (Object classifier : allClassifiers) {
    if (Model.getFacade().getName(classifier) != null
        && Model.getFacade().getName(classifier).equals(s)) {
        return classifier;
    }
}
```

(b) Enhanced for loop implementation

Figure 1: Clones using for loop and enhanced for loop to implement the same loop scenario.

Figure 2 illustrates two clones using `if-else` statement and `ternary operator` to implement the same functionality, which invokes the `.println()` method from `System` class. The code syntax of these two clones are significantly different, so, they cannot be detected with the same strategy which can work for syntactical clones. However, if an approach could analyze the functionality of these two clones, their functionalities would be found to be just the same. Furthermore, these two clones can be unified into the same syntax, making them become Type-1 clones which can be easily dealt with.



```java
public void log(String message, int loglevel) {
    if (managingPc != null) {
        managingPc.log(message, loglevel);
    } else {
        if (loglevel > Project.MSG_WARN) {
            System.out.println(message);
        } else {
            System.err.println(message);
        }
    }
}
```

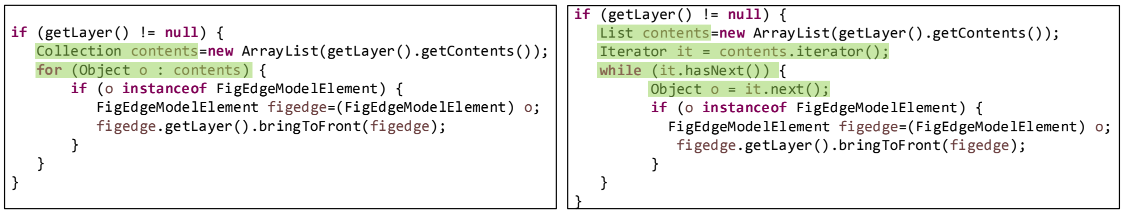
```java
public void log(String message, int loglevel) {
    if (managingPc != null) {
        managingPc.log(message, loglevel);
    } else {
        (loglevel > Project.MSG_WARN ?
         System.out : System.err).println(message);
    }
}
```

(a) For loop implementation (b) Ternary operator implementation

Figure 2: Clones using if-else and ternary operator for the same functionality

1.4.2 Gap statements

Gap statements refer to those unmatched statements between two clones. Along with the usage of different control structures, gap statements are also introduced into clones, which significantly increases the syntactical difference between two code fragments. Figure 3 presents two clones- one uses `enhanced for loop`, and the other uses `while` loop to implement the same functionality. Aside from this, clone (b) has two gap statements which cannot be matched with any statement in clone (a). Due to these two additional statements, and the different control statement, the overlap between two clones is less than 60 %, which is far beyond the detection capability of syntax-based clone detection tools. In order to recognize them as clones, a functionality-aware clone detection approach is required, which should be able to find that the two loops are the same and those gap statements are part of the loop.



```java
if (getLayer() != null) {
   Collection contents=new ArrayList(getLayer().getContents());
   for (Object o : contents) {
       if (o instanceof FigEdgeModelElement) {
          FigEdgeModelElement figedge=(FigEdgeModelElement) o;
          figedge.getLayer().bringToFront(figedge);
       }
   }
}
```

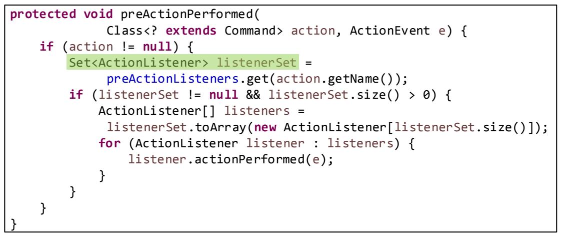
```java
if (getLayer() != null) {
   List contents=new ArrayList(getLayer().getContents());
   Iterator it = contents.iterator();
   while (it.hasNext()) {
          Object o = it.next();
          if (o instanceof FigEdgeModelElement) {
            FigEdgeModelElement figedge=(FigEdgeModelElement) o;
             figedge.getLayer().bringToFront(figedge);
          }
   }
}
```

(a) Enhanced for loop implementation (b) While loop implementation

Figure 3: Clones using different control structures and containing gap statements

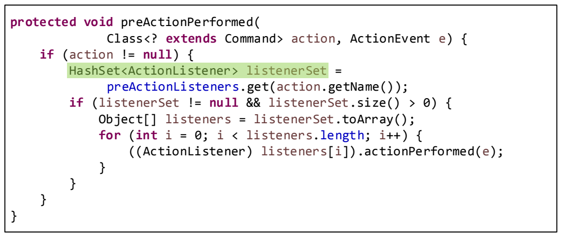
1.4.3 Interchangeable data structures

Apart from the control structures used in a code fragment, another important aspect of coding is the data structures used inside the code fragment. Two code fragments which have the same control structure are not necessarily clones of each other. Most syntax-based detection approaches are sensitive to syntactical differences such as different data structures (e.g., `List` and `Collection` or Set⟨String⟩ and HashSet⟨String⟩), but these different data structures can be used for the same functionality. In figures 3 and 4, aside from the different loops, we can also find different data structures being used inside the loops for the same purpose. In figure 4, clone (a) declares a Set to keep all the elements, while clone (b) uses a HashSet which actually is a concrete implementation of interface Set. These two data structures are actually interchangeable.



```java
protected void preActionPerformed(
             Class<? extends Command> action, ActionEvent e) {
    if (action != null) {
        Set<ActionListener> listenerSet =
             preActionListeners.get(action.getName());
        if (listenerSet != null && listenerSet.size() > 0) {
            ActionListener[] listeners =
             listenerSet.toArray(new ActionListener[listenerSet.size()]);
            for (ActionListener listener : listeners) {
                listener.actionPerformed(e);
            }
        }
    }
}
```

(a) Clone using HashSet



```java
protected void preActionPerformed(
             Class<? extends Command> action, ActionEvent e) {
    if (action != null) {
        HashSet<ActionListener> listenerSet =
             preActionListeners.get(action.getName());
        if (listenerSet != null && listenerSet.size() > 0) {
            Object[] listeners = listenerSet.toArray();
            for (int i = 0; i < listeners.length; i++) {
                ((ActionListener) listeners[i]).actionPerformed(e);
            }
        }
    }
}
```

(b) Clone using Set

Figure 4: Clones using different data structures for the same functionality.

1.4.4 Motivation

As mentioned in Section 1.2.3, semantic clones do exist in the real world because programmers have diverse programming styles or miss the migration of some clones to new language constructs. Meanwhile, existing clone detection tools are unable to detect semantic clones accurately and understand the cloning information. Jürgens et al. [JDH10] asked 400 students to implement the same functionality independently and received 156 implementations of the specification, of which 109 were compiled and passed their test suit. However, the selected state-of-the-art token-based and AST-based clone detectors(i.e., ConQAT and DECKARD) did not achieve a recall of more than 10%, even though they were executed with a very unrestrictive configuration that would yield far too many false positives in practice. From the results of this experiment, we can draw two conclusions. First, the clones which are not created by copy-paste are likely to differ in terms of code syntax, which is why most of the study objects could not be recognized as clones by syntax-based tools. Second, existing clone detection tools do not work on syntactically different clones. Jürgens et al. also manually investigated the source code of the open-source project Jabref to find syntactically different, but behaviorally similar clones. In class Util of Jabref, 52 methods were checked, of which 32 were at least partly behaviorally similar to other methods within Jabref or to methods from the Apache Common library.

Semantic clone research is becoming a major area of interest within the field of software clone research. Lavoie and Merlo have pointed out the need to find a way of detecting clones of a higher level than Type-3, also mentioning that their approach [LM11] is unable to detect semantic clones. Here the semantic clones are comparable to functional clones according to their proposed definition. JPlag is a state-of-the-art plagiarism detection tool [PMP00], but Prechelt et al. have listed some attacks which can confuse JPlag and cause the detection to fail. One of these attacks is from clones which use variant control structures, but retain the same semantics. Likewise, the attack from clones which use different data structures to implement the same functionality can also cause the detection to fail. There is a consensus in the literature stating that finding functionally equivalent clones is a challenging and interesting research problem.

Functionality-aware clone mining techniques are an important complement to syntax-based clone detection techniques. Jiang and Su [JS09] have carried out an experiment to detect functionally equivalent clones and compared their results with those from a syntax-based detection tool DECKARD. They found out that 58% of functionally-equivalent code clones are syntactically different and not reported by DECKARD. In other words, these clones cannot be detected by existing syntax-based detection tools. Furthermore, the detection of clones is just the beginning. In order to manage the clone codes properly or refactor them, it is necessary to understand and analyze the cloning information. It is clear that we need functionality-aware clone mining approaches, in addition to syntactic clone detection approaches, in order to extend the benefits of clone management to a wider range of use.

1.5 Contribution

In summary, the main contributions of this thesis are listed as follows:

1. We propose a lightweight approach for mining control structure variant clones at the source code level. Currently, our approach supports six types of control structure variants.

2. By applying our approach on six open-source systems, we carry out an empirical study about control structure variant clone diversity and distribution in open-source projects. As expected, control structure variant clones occur much less frequently than syntactical clones, but the syntax could vary significantly.

3. We manually examine the clone results reported by our approach in order to create a control structure variant clone benchmark. This dataset is publicly available[1] and can be used to evaluate

[1]http://goo.gl/ye0ucR

and compare clone detection/refactoring techniques on advanced clones.

4. We analyze the clone cases from the refactoring perspective, and propose improvements that could be made to tackle the limitations of existing clone refactoring techniques.

1.6 Overview of the approach

This thesis presents an approach for the mining and analyzing of control structure variant clones. The proposed approach complements the existing syntax-based clone detection techniques. Our approach can take the preliminary detection results from any clone detection tool and filter out control structure variant clones at both the method level and code fragment level. Furthermore, it also presents the cloning information in a comprehensive HTML report.

The mining process consists of two main steps: *Control structure matching* and *Function similarity evaluation*. In the first step, the approach detects similar control structure subtrees within the body of two methods by applying a relaxed control structure matching algorithm, which can match *functionally equivalent* control structures. We have improved the traditional tree matching algorithm in order to tolerate syntactically different control structures. In the second step, the approach computes the functional similarity of the code statements inside the detected equivalent control structures, based on the commonly-used method calls and data structures. Likewise, instead of a normal comparison, our approach performs a post-processing in order to handle interchangeable data structures and method expressions. We use the *Jaccard Similarity Coefficient* to quantify the functional similarity. If two clone candidates contain equivalent control structures and their similarity score is above a specific threshold ϕ, we then consider them to be true positives. For the evaluation of our approach, since there were no other tools performing the same task, we manually examined the reported clones and calculated the recall and precision of the results. In the end, the approach reported 285 true positive control structure variant clone pairs, which were categorized according to their similarity scores. For each clone pair, the approach generated a report presenting the syntax differences and refactorability analysis.

Currently, our approach is capable of handling the most-often used control structures, such as: *For loop*, *Enhanced for loop*, *While loop*, *Do-While loop*, *If-else statement*, and *Conditional operator*. Apart from that, the approach can also capture clones which apply different collection implementations, such as, *List*$\langle$*String*$\rangle$ and *ArrayList*$\langle$*String*$\rangle$, *HashSet* and *LinkedHashSet*, etc. After investigating the cloning information of the clones found, we have proposed some improvements meant to tackle the limitations of existing clone refactoring approaches.

1.7 Thesis Outline

The remainder of this thesis is organized as follows: in Chapter 2, we provide related works for this research. In Chapter 3, we discuss our proposal and how to implement our approach. In Chapter 4, we present our experiment results and evaluation of our approach. In Chapter 5, we discuss the case study from refactoring perspective. Finally, in Chapter 6, we discuss the conclusions and future work.

Chapter 2

Literature Review

In this chapter, the related work is organized into three parts: Semantic Clone Detection, Clone Management and Clone Benchmarks. The first part (Semantic Clone Detection) is further divided into source-code-based and bytecode-based techniques.

2.1 Semantic Clone Detection

2.1.1 Bytecode Based

Keivanloo et al. [KRR12b] introduced a tool named Sebyte to detect clones at the Java bytecode level. Java bytecode is an intermediate-level code, which realizes the cross-platform portability of Java language. There are two advantages of performing clone detection at the bytecode level. First, all kinds of loops and condition statements are converted into the same machine instructions at this level. Therefore, some code clones which look dissimilar at the source code level may become more similar at the bytecode level. It enables the detection of code clones with advanced syntactical differences that normally cannot be tolerated in source code clone detection techniques. The other advantage is method inlining in Java bytecode. When converting source code to byte code, the method calls in the source code to be replaced by the body of the invoked methods, so the gap statements at the source code level may be eliminated because of the code fragment reform.

Sebyte uses a metric-based approach mixed with a pattern matching technique. In order to reduce the size of the data to be processed, Sebyte extracts abstract information from the original file instead of processing the entire code fragment. It extracts Java type names and method names from the corresponding bytecode.

Based on the collected information, Sebyte evaluates the pattern similarity and content similarity of each clone pair candidate. In order to evaluate the pattern similarity, seven types of clone matching patterns are proposed in this paper [KRR12b]. These patterns represent seven types of syntactical dissimilarities resulting from three main types of operations: repetition, sliding, and gaps. Repetition refers to the duplicate statements, sliding means the shift of code block between a clone pairing, and gaps are the unmatched statements between two clones. In the first phase, Sebyte detects clone pair candidates which conform to these patterns. They support that this approach can find clone pairs where extreme gaps exist. However, the detection results may also be limited within these seven patterns. For example, none of these seven patterns include reordered statement clones, which are very common in practice.

After the detection based on pattern similarity comparison, the preliminary results are reported. Nevertheless, these results include many false positives. Thus, in the third step, Sebyte evaluates the content similarity of each pairing to improve the precision. The content similarity is evaluated based on the Jaccard Coefficient of the collected information. The Jaccard Coefficient is popularly used to measure the similarity of sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets. In their work, the Java type name sets and method name sets were used as sample sets. Two metrics were defined for content similarity evaluation, ω was for the similarity of method calls and φ was for the similarity of Java types. For each pair of code clone candidates, if both of these two metric values are higher than the threshold values, this pairing is reported in the final result. The results are recorded in plain text, and the reported clone groups consist of many clones. In each group, the first line lists the Query method, which is the reference method. Following the reference method, ten or less clone methods are listed, which are considered similar to the reference method. Each method has a similarity score indicating the similarity between the corresponding clone method and the reference method. In addition, the classpath, method signature, start, and end line number of each method are also recorded in the final results.

In the evaluation phase, Sebyte was applied to four Java projects. The detection results of Sebyte were compared with results reported by the other five detection tools. The general agreement between the different result sets was not high. The highest agreement was between Sebyte and Nicad when these two tools were applied to project EIRC - Sebyte reported 24 clone classes while NiCad reported 17, and the agreement was about 70%. Meanwhile, the agreements between Sebyte and all the other tools were less than 20%. In order to evaluate the performance of Sebyte, Iman et al. created an oracle through manual examination. A total number of 700 pairs of clone candidates were examined, including both true and false positives. Based on this reference set, the recall and precision value were 92% and 79%, respectively.

Kamiya [Kam13] proposed an approach to detect semantic clones from Java byte code and implemented a prototype tool named Agec. The semantic clones are viewed as code fragments equivalent in terms of method invocations. Each code clone candidate is represented with a sequential fragment, all direct and indirect method calls inside the clone candidate are considered as units in the sequential fragment, and the same sequential fragments are considered as clones. This approach is able to find semantic clones which cannot be detected by static source code analysis, because all the indirect method calls are taken into account. First, method execution traces are created for each method. The method execution trace is equivalent to the aforementioned sequential fragment, and each method is extracted and placed in order. When an indirect method call appears, the trace splits and the indirect method call is added into the sequence. Based on the method trace of the entire method, n-grams are extracted from all the traces. An n-gram is a sub-string of the method execution trace, and the user can customize the n value to control the size of the clone reported. In the final step, the common n-grams which started from distinct points are reported as clones.

This approach can eliminate the syntax dissimilarity caused by the method calls, which can be replaced by inlining methods. However, the code structure difference was not discussed in this paper. Obviously, it may affect the execution of the method. This approach was applied to project ArgoUML, and in total, 4,634 clone n-grams were reported from 1,232,292 n-grams.

2.1.2 Source Code Based

Gabel et al. [GJS08] have extended the original tree matching technique proposed by Jiang et al., known as DECKARD [JMSG07], to enable the detection of semantic clones. They consider semantic clones to be code fragments with isomorphic PDGs, and therefore, the first step of their approach is to extract the PDGs of all clone candidates. A program dependence graph is a static representation of the flow of data in a procedure, and consists of nodes and edges which are extracted from the code. Nodes represent code statements and are categorized into two types. The first is a normal node, which represents a non-control statement, while the other is a control node, which represents a control statement. Control statements are either loops or conditionals. PDG edges connect the nodes and represent data dependencies or control dependencies between those nodes. Despite all the advantages of the PDG representation, the traditional PDG mapping calculation is particularly expensive. Therefore, the clone detection approaches which take advantage of PDG may not be able to scale up to industrial-size projects. Thus, based on the PDGs selected, the approach generates the Abstract Syntax Tree (AST) for each corresponding PDG in order to convert the PDG mapping problem into a tree matching problem and reduce the computation cost. The authors also mention that most state-of-art clone detection tools are resilient to minor code modifications, but sensitive even to minor structural differences. They presented three types of differences: interleaved statements, reordered statements, and semantically equivalent control structure. The clones which include these three types of differences are normally textually dissimilar, and it is difficult to detect them through syntax comparison. Therefore, the authors proposed a technique to slice PDG and identify syntactically different clone candidates. Each independent PDG subgraph is considered as a clone candidate and the corresponding AST was generated for each subgraph. Furthermore, in order to simplify the calculation, the tree similarity calculation problem was converted into a vector similarity calculation. Each AST was transformed into a characteristic vector consisting of node types specified by the corresponding language grammar. For Java, it includes expressions, method calls, variable declarations, etc. Finally, the clone candidates comparison is performed on millions of vectors with the Locality Sensitive Hashing algorithm [GIM99]. Their technique has been evaluated on several million-line open-source projects, including the Linux kernel. The clone examples presented in this paper are all clones with interleaved statements. However, these clones are considered Type-3 clones, according to the definition supported by Roy et al. [RCK09]. Aside from that, no control structure variant clone examples are presented in this paper.

Elva et al. [ELE+12] developed a tool named JSCTracker for the detection of method-level semantic clones in Java source code. They consider semantic clones to be functionally identical code fragments, and define functional similarity in terms of input and output behaviour. Input is determined by the value of parameters passed to the method and the state of the heap when the method is invoked, while output is defined in terms of the return values and effects of the method (i.e. persistent changes to the heap as a result of the method execution). In the first phase, they extract the *methodtype* (i.e., signature, parameter types, and the return type) and the *effects* (i.e., modified static and instance fields) of each method. In the second phase, they filter the methods which have the same *methodtype* and *effects*, and treat them as clone candidates. In the final phase, they automatically generate test cases to evaluate the actual dynamic behaviour of the clone candidates by using method calls to run each member of an equivalence class on the same input, and then compare the corresponding outputs. The authors also did a case study covering 22 classes. A total number of 13 semantic clones were injected into these classes, and in the end, all the clones were successfully found. Aside from that, JSCTracker was also applied to the open-source projects DSpace and JabRef, and four clones were found inside JabRef.

A recent paper from Jürgens et al. [JDH10] discussed about the clones which are textually different. They introduced the term *Simions* to name the textually different clones in order to avoid confusion with existing clone definitions, and supported that simions are not produced from a common origin. In other words, the simions are not created by copy&paste activities, but because of different implementations of the same functionality. The authors conducted a case study in order to find out how well the current state-of-art clone detection tools can detect simions. They asked 400 students to implement the same functionality independently and developed a test suite to evaluate the correctness of each implementation. In the end, 109 copies passed the tests and were accepted in the reference set. In the evaluation phrase, ConQAT and DECKARD were applied to detect clones from the reference set. As mentioned in this paper, both tools were configured with much less restrictive parameters in order to get the highest recall value. Theoretically, since all the copies in the reference set implemented the same functionality, any pair of copies should be reported as code clones. However, the recall values of ConQAT and DECKARD were both under 10%. This result implies that even with a looser configuration, the existing detection tools cannot detect behaviourally similar, but textually different clones with high confidence. These authors also investigated the existence of behaviourally similar code clones in the real world. They manually examined 6000 LOC from the open-source project JabRef and found that in the Util class, 32 out of 52 methods were partly or completely behaviourally similar to another method in JabRef or the existing Java libraries. This proved that simions do exist in real systems, however, most of them cannot be detected automatically with a tool. According to the variant syntax differences found in the real simions cases, the authors reported six types of syntax variations.

Syntactic variation: Different concrete syntax used to express equivalent abstract syntax. For example, the different statements to declare a variable.

Organization variation: Different hierarchies of method calls to implement the same calculation. For, example the inlining method and temporary variable.

Generalization: Comprises differences in the level of generalization. For example List$\langle String\rangle$ and ArrayList$\langle String\rangle$.

Delocalization: Code relocation because of the reordering of statements. For example, the position of declaration statements in the code fragment could be flexible.

Unnecessary code: The irrelevant statements which are inside the code fragment, but do not affect the functionality of the code. For example, the log statements.

Different data structure or algorithm: Methods using different data structures or algorithms to solve the same problem.

They named the currently existing clone detection approaches as representation-based detection, and supported that these approaches are fundamentally unsuited to detecting behaviourally similar clones. The control structure variant differences which we try to address belong to the category "Different data structure or algorithm". Meanwhile, our approach can also handle the differences caused by Generalization.

Jiang and Su [JS09] presented a tool named EqMiner to mine functionally equivalent, but syntactically different clones (i.e., Clones not created by copying and pasting activities). As the authors mention in the paper, it is a common intuition that the semantically similar, but syntactically different clones exist in industry systems. However, there is no empirical study about this. Thus, they propose the approach which evaluates the functional equivalence based on the input and output behaviour of the code fragments. If two clone candidates take the same input and always give the same output, these two candidates are then considered functionally equivalent clones. The first step of their approach is code chopping. The code fragments are divided in statements, and their approach extracts all possible consecutive subsequences which have more than n lines of statements from the statement sequence. All the subsequences are considered candidates for detecting functionally equivalent clones and are passed to the next step for testing evaluation. The test case design is inspired by Schwartzs randomized polynomial identity testing. The Schwartz-Zippel lemma states that a few random tests are sufficient to decide, with high probability, whether two polynomials are equivalent. All the test cases are generated automatically and randomly. In the beginning, all the candidates taking the same input are placed in the same cluster, and later on, they are divided into different clusters if they have different output. In the last step, the candidates that have the same input and output, but share too many overlapping statements are filtered out in order to avoid repetition. The rest of the candidates are reported as functionally equivalent clones.

The authors applied the proposed approach to Linux Kernel and finally got 42,830,319 code fragments. Through the comparison between the results reported by EqMiner and DECARD, which is a syntactically similar code detection tool, they found that more than 58% of the functionally equivalent clones are syntactically different. Their approach could successfully detect functionally equivalent clones. However, the results also include syntactical clones, while our approach only detects control structure variant clones. In addition, their approach does not investigate the implementation details inside the code fragments, and it cannot compare the functionality of the clones. This is one of the problems that we are attempting to address in our research.

Wang et al. [WWSM14] proposed a metrics-based and graph-based combined approach to detect semantic clones. They divided the traditional detection approaches into five categories: text-based, token-based, tree-based, metrics-based, and graph-based. They considered that the first three categories of approaches cannot handle significant textual differences, the metrics-based approach has a low precision, while the graph-based approach has a high computational complexity. Based on these facts, they proposed a combined approach named CMGA (Metrics-based and graph-based combined approach), which works at the method level. They listed eight types of differences that may exist between code clones, which they named code variations, including: 1) Code formats - a variation caused by different comments or blank lines etc, 2) Different forms of code implementation, (e.g the variable declaration and value assignment can be implemented with two lines of code or a single statement), 3) Equivalent expressions, which are textually different because of redundant parentheses or the use of different operators, 4) Redundancies, caused by useless code or temporary variables, 5) Control structure variations (e.g. implementation of the same functionality with different control structures), 6) Renamed variables, 7) Reordered statements, and 8) Different program module structures. In these eight types of differences, variations 1, 6, and 7 can already be handled well with existing detection tools, meanwhile, handling variations 2, 3, 4, and 5 is still an open problem.

The detection process of their approach was divided into two stages - metrics-based and graph-based. The main task of the first stage was to prune the clone search space with a low computation complexity, so they proposed a metrics vector for each clone candidate and set a threshold to filter out results. The low precision results generated in the first stage needed to be refined, so in the second stage, the approach performed an analysis of the PDGs of each clone candidate. Since many candidates were filtered out in the first stage, the graph comparison should only be performed on a few cases. This design enabled this approach to be scalable to large-size industry systems.

One important part of their approach is code normalization. The normalization process transfers the syntactically different source code into unified formats. Likewise, in order to reduce the computation complexity, they designed two types of normalization: basic code and advanced code normalization. Basic normalization is applied in the first stage, while advanced normalization is performed in the second stage. Basic normalization includes three operations: 1) Separating compound statements into statement sequences, for example, separating the initialization and declaration of variables, 2) Unifying expressions (e.g. reorganizing logical expressions), 3) Unifying control structures (e.g removing empty or unreachable branches, transforming nested selections into multi-branch selections). In the first stage, normalization is performed on Control Dependence Trees(CDTs). They defined the CDT as the tree representation of a Program dependence graph without goto statements. After the PDG is converted to CDT, metrics are calculated based on the comparison of CDTs in order to reduce the calculation complexity. Advanced normalization is performed in the second stage, and includes five types of normalizations: 1) Advanced control structure transformation (e.g. eliminate `if-break` statements), 2) Eliminating useless statements, 3) Renaming variables with unified format, 4) Statement reordering, 5) Function call inlining.

Both their approach and ours divide the detection process into two stages. However, the difference is that in the first stage, we compare the structure of the clone fragments, while they evaluate the semantic similarity of the clone fragments. With their approach, after the normalization, all the syntactically different clones are converted into the same syntax, and finally they detect the clones based on the syntax of the code. Thus, their results include all types of clones, while our approach analyzes the functionality of the code and detects only control structure variant clones.

2.2 Clone Refactoring and visualization

A common limitation of clone detection tools is that they do not provide much information about the differences between two clones. However, this is critically important for the clone maintenance task. Thus, clone detection is the initial step, and an approach for clone analysis and visualization is required for eliminating the negative effects of clones.

Xing et al. [XXJ11] presented a tool named CloneDifferentiator, which is used to analyze the existing differences between two clones. Furthermore, this tool can display clones with highlighted types of differences. Their approach is based on PDG analysis and is able to categorize the clones in a task-oriented manner. In other words, they analyze clones from the perspective of clone maintenance. They define seven types of differences (e.g. Differential property, Unmatched parameters, Unmatched block pair, Partially matched branch, Additional branch, Additional operation, and Additional block) existing in clones, and CloneDifferentiator can highlight them with different colours. However, they do not support the recognition of functionally equivalent syntax differences. In our approach, we introduce advanced matching for the display of functional clones which have syntactical differences. Thus, our approach is better when it comes to understanding and analyzing advanced clones.

Krishnan and Tsantalis [KT14] have proposed a technique to refactor clones which have non-trivial syntactical differences. Their approach can optimize matching of clone statements, which maximizes the number of matching statements and minimizes the number of differences. The number of differences is important, because it will directly affect the number of parameters that need to be introduced in order to refactor the clones. Their approach can support the matching of more than 20 types of expressions as long as they return the same class/primitive type.(e.g. Method Invocation `String name = name.trim()` and string Literal `String name = "name"` etc.). Furthermore, they support the parameterization of seven types of differences (e.g different variable identifier, different literal value, etc.). However, not all types of differences are able to be parameterized. As mentioned in the paper, there is some precondition checking that needs to be done prior to the parameterization of the differences. One of these preconditions is that the unmatched statements should be movable before or after the matched statements without breaking existing data-, anti-, and output-dependences. The unmatched statements refer to the gapped statements in Type-3 clones, and this precondition checking is necessary for the refactoring of Type-3 clones. However, it is not perfect for functionally equivalent clones. Most of the gapped statements in functionally equivalent clones are not movable, but we can still somehow unify the syntactical representation of the differences in order to refactor them. We will discuss this in the section 5.

2.3 Clone Benchmarks

In the evaluation of clone detection tools, precision and recall are two important factors that are often measured in experiments. To enable the evaluation of those two factors, an accurate clone benchmark is required. Through the comparison between the reported clone set and the reference clone set, people can investigate the recall and precision of the detection results. Thus, the accuracy of the evaluation depends on the quality of the reference set. The more accurate clones available in the reference set, the more reliable the evaluation of the accuracy of different detection tools.

Bellon et al. [BKA$^+$07] conducted a study about the comparison of six different clone detection tools. In the study, the main task was to evaluate the degree of accuracy of different clone detection tools. Besides Bellon, there are six other authors providing six clone detection tools (Dup [Bak95], CloneDR [BYM^{+}98], CCFinder [KKI02], Duplix [Kri01], CLAN [KDM^{+}96], Duploc [DNR06]) as candidates for this study. These six tools are all state-of-the-art clone detection tools, but they are all capable of detecting different types of clones and are based on different techniques, which work on different information from the code (e.g. text, lexical, syntactic information, software metrics, program dependency graphs). The authors applied their tools on the same set of systems respectively, which included 4 C systems and 4 Java systems. All the clones that contained more than six lines of code were reported in their experiment results. The authors carefully chose systems with different programming languages and different sizes to make sure there was as little bias as possible in the experiment.

At the end, the authors of those six tools submitted a total number of 325,935 clones. In order to evaluate the results automatically and accurately, Bellon spent 77 hours building the benchmark manually. He viewed 2% of all 325,935 submitted clone candidates, which was 6,528 clones. These clones were automatically selected and equally distributed in all projects to make sure that there was no personal preference to any specific tool. At the end, 4,319 clones passed the manual examination and were accepted in the reference set. Apart from those ones, 50 extra artificial clones were randomly injected into the eight projects in order to get a more accurate recall value. Each clone reference in the reference set consisted of two clones, each being specified by the following information: filename, start line number, end line number, and clone type. In the similarity evaluation phase, they calculated the ratio of statement overlap between the reported clone pair and the referenced clone pair. If the overlap ratio was over a specific threshold, the clone candidate would be considered a true positive.

Bellon benchmark is a landmark clone reference set that is still widely used today. In our research we have also created a benchmark of control structure variant clones. Unlike the Bellon Benchmark, which consists of Type-1, Type-2, and Type-3 clones, our benchmark excludes the syntactical clones and only contains control structure variant clones, which have more semantic interest.

In the real world, most industry projects are too large for manual examination, so Lavoie and Merlo [LM11] have proposed an approach for constructing a reference clone dataset automatically. Their approach uses the Levenshtein metric and the M-tree data structure, and can support the oracle of Type-3 clones. However, they consider the Type-3 clones as a superset of Type-1 and Type-2 clones. The definition of Type-3 clones they support in this paper includes two subcategories. One category represents structure-substituted clones, which are copied fragments wherein partial program structure has been substituted. The other category represents modified clones, which are copied fragments where code has been deleted, added, or both. Based on this definition, the reference set created in fact includes all Type-1,2,3 clones.

The Levenshtein distance is an edit distance which indicates the minimal number of single character edits required to change one sequence to the other. Therefore, it must be calculated over two code fragments, which are treated as two string sequences. As long as the Levenshtein distance between two clone fragments is smaller than a specific threshold value, these two fragments would be reported as clones. Obviously, the pairwise comparison between all possible fragment pairs is impossible, so the M-tree(metric tree) data structure is used to prune the search space [CPZ97]. All the nodes in the M-tree data structure are categorized into two types - non-leaf-nodes and leaf-nodes. Each non-leaf-node has three parts - an object to identify itself, a pointer to the subtree where its children reside, and radius R which defines a circle search space. Each leaf-node may have several data objects. Based on this kind of structure, when they try to query a clone candidate within a specific distance, the comparison would only be performed among the most relevant nodes. Thus, it is able to prune the search space and reduce the number of distance computations between the code fragment pairs.

In order to evaluate their technique, the approach was applied on two projects - Tomcat 5.5 and Eclipse 3.3. In both projects, when the threshold value for the Levenshtein distance was set to 0.3, 2933 clones were reported from Tomcat and 316,728 clones were reported from Eclipse.

Their approach and ours use different techniques considering clone detection as text comparison. In Lavoie's approach, as long as the Levenshtein distance between two code fragments is below the threshold value, they are reported as clones. Thus, their detection results have a high recall value, but also contain some false positive cases. Our approach takes advantage of both tree matching and similarity metrics filtering. The comparison is based on the code control structure and Java type bindings. Since we want to create a benchmark with accurate clone cases, we chose an approach which ensures high precision. However, due to the limited types of functionally equivalent syntactical differences we can recognize, we may miss some cases which have syntactical differences beyond those covered. Apart from the different techniques used, we also target detecting different types of clones. As mentioned in their paper, their reference set only contains Type 1,2,3 clones, while we focus on a subcategory of Type-4 clones, namely control structure variant clones (CSVC).

Based on the Bellon benchmark, Murakami et al. [MHK14] made an improvement by adding the line number of gapped statements of Type-3 clone pairs. In addition to the location information provided in the Bellon benchmark, they also provide the location of unmatched statements. Thus, in the comparison of Type-3 clone candidates and clone references, the gapped statements can be excluded. In this way, the overlap ratio between the clone candidate and clone reference can be more accurately computed. They conducted an experiment on four Java projects, namely Netbeans, Ant, JDT Core, and Swing, all of which were also used in Bellon's experiment. In this experiment they evaluated three detection tools, namely NiCad, Scorpio, and CDSW, with their own benchmark and the Bellon benchmark, respectively. The results were different. Scorpio detected more clones in the Netbeans project, but produced fewer in the other three. CDSW was the clone detector proposed by Murakami, which reported more clones in half of the projects. Thus, we can see that an accurate benchmark is of crucial importance for the evaluation of clone detection tools.

Chapter 3

Mining Control Structure Variant Clones

In this chapter, we will first give an overview of the approach, then present our functionally equivalent control structure matching algorithm and our approach for quantifying the functional similarity.

3.1 Overall Approach

The proposed approach for mining control structure variant clones (CSVC) is specifically designed for Java projects, and it consists of two main steps:

1. **Control Structure Relaxed Matching:** In order to compare the control structures of a pair of methods, we extract the Control Dependence Tree (CDT) from each method and try to find functionally equivalent common sub-trees between the CDTs.

2. **Functional similarity evaluation:** Based on the common sub-trees found in the first step, we extract the type bindings from all used variables, literals, and method calls inside the code fragments representing the functionality of corresponding codes. Then, we use the Jaccard Similarity Coefficient to quantify the functional similarity of the clones.

Figure 5 illustrates the complete clone mining process of the proposed approach. In the first step, our approach takes groups of clone candidates as input and extracts all possible clone pairs. The second step generates the Control Dependence Trees (CDTs) for each pair of clones from the same group. In the third step, we apply a relaxed version of a tree matching algorithm proposed

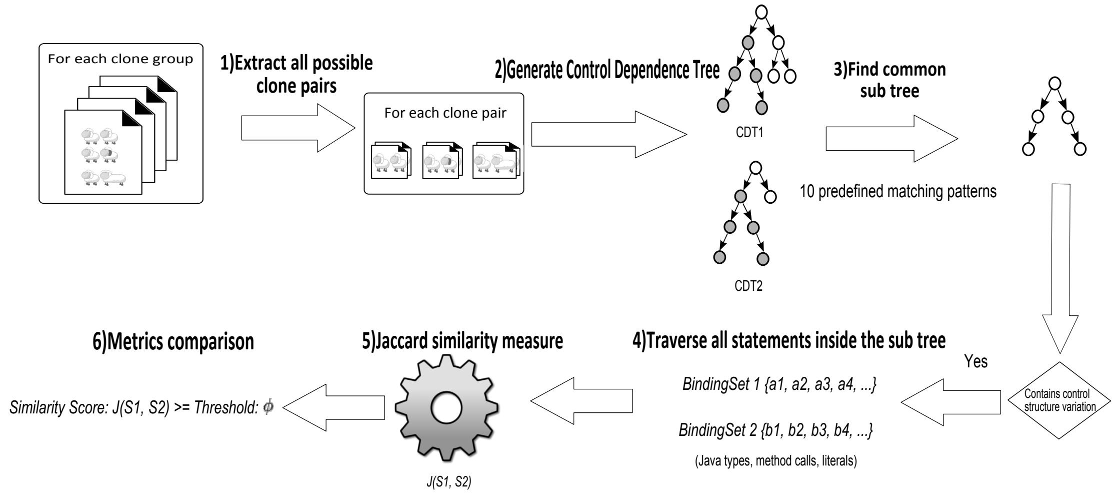
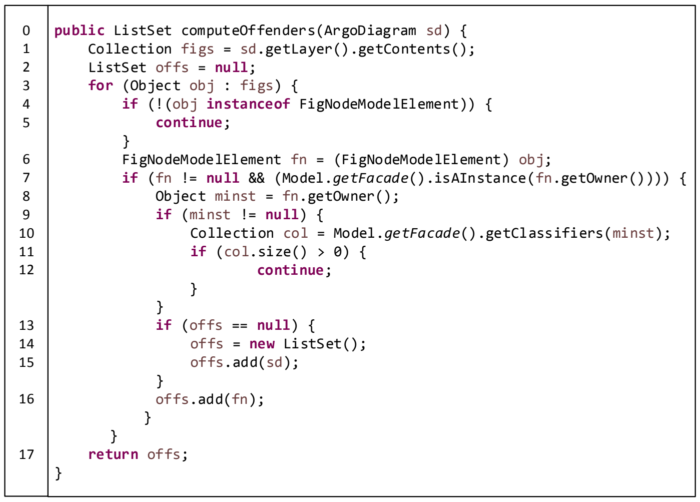

For each clone group
1)Extract all possible clone pairs
For each clone pair
2)Generate Control Dependence Tree
CDT1
CDT2
3)Find common sub tree
10 predefined matching patterns
Contains control structure variation
Yes
4)Traverse all statements inside the sub tree
BindingSet 1 {a1, a2, a3, a4, ...}
BindingSet 2 {b1, b2, b3, b4, ...}
(Java types, method calls, literals)
5)Jaccard similarity measure
J(S1, S2)
6)Metrics comparison
Similarity Score: J(S1, S2) >= Threshold: ϕ

Figure 5: Overview of the proposed approach

by Krishnan and Tsantalis [KT14], which returns the largest common subtrees between the input CDTs. The original algorithm was improved to tolerate functionally equivalent control structure variations. All the clone candidates that include matched subtrees and also contain control structure variations pass to the next step for the computation of functional similarity. In the fourth step, we implement a visitor to traverse all the statements located inside the common subtrees in order to collect code fingerprints, which represent the functionality of the code. The code fingerprints include three categories of type bindings, namely variable types, method call return types, and literal types. In the fifth step, based on the fingerprints collected in previous step, we use the Jaccard Similarity Coefficient to quantify the functional similarity of the code fragments. Finally, all the clone candidates having similarity scores higher than a fixed threshold are reported as control structure variant clones.

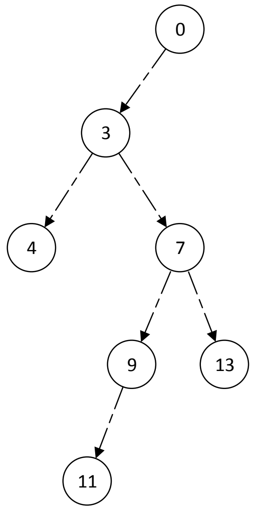
3.2 Control Dependence Tree Relaxed Matching

The *Control dependence tree* (CDT) represents the control structure of a code fragment. The CDT has exactly the same structure with the *Control Dependence Graph* (CDG) with the only difference that it contains only the predicate nodes of the CDG. As illustrated in Figure 6, it presents the code example we found in ArgoUML project and the corresponding CDT. Each CDT node represents a control statement inside the code fragment (e.g. *If, While, For, Switch* etc.) and the edges represent control dependences between the control nodes.



```
0     public ListSet computeOffenders(ArgoDiagram sd) {
1         Collection figs = sd.getLayer().getContents();
2         ListSet offs = null;
3         for (Object obj : figs) {
4             if (!(obj instanceof FigNodeModelElement)) {
5                 continue;
              }
6             FigNodeModelElement fn = (FigNodeModelElement) obj;
7             if (fn != null && (Model.getFacade().isAInstance(fn.getOwner()))) {
8                 Object minst = fn.getOwner();
9                 if (minst != null) {
10                    Collection col = Model.getFacade().getClassifiers(minst);
11                    if (col.size() > 0) {
12                            continue;
                      }
                  }
13                if (offs == null) {
14                    offs = new ListSet();
15                    offs.add(sd);
                  }
16                offs.add(fn);
                 }
            }
17        return offs;
      }
```

(a) Code Example



$$\begin{tikzcd}
 & & 0 \arrow[dl, dashed] & \\
 & 3 \arrow[dl, dashed] \arrow[dr, dashed] & & \\
4 & & 7 \arrow[dl, dashed] \arrow[dr, dashed] & \\
 & 9 \arrow[dl, dashed] & & 13 \\
11 & & &
\end{tikzcd}$$

(b) Control Dependence Tree

Figure 6: Code example and corresponding Control Dependence Tree

Our goal is to find clones with variations in the control structures they use to perform iterations and conditionals. Thus, CDT is the most appropriate structure for the control structure comparison.

In order to enable the matching of control structures which are functionally equivalent but contain syntactical variations, we have developed two advanced matching functions, one for matching loop variants, and the other one for conditional variants. In the following section, we will discuss them respectively.

3.2.1 Loop variant matching

Table 1 shows some examples of loop variants that are syntactically different but functionally equivalent (i.e., they perform the same iteration). In general, developers use three different variations when implementing loops, which can be summarized as follows:

- *Iterator-based loop*: `for` or `while` loop that uses an iterator to traverse the elements of a collection.
- *Index-based loop*: `for` or `while` loop that uses an index variable to keep track of the number of performed iterations.
- *Enhanced for loop*: special `for` loop designed to iterate through the elements of a collection or an array. It was introduced in Java 5, and uses internally an iterator.

Table 1: Loop variants

Variant	Code snippet	start index	end index	step
iterator-based (while loop)	`Iterator<String> it = list.iterator(); while(it.hasNext()) { System.out.println(it.next()); }`	`iterator()` 0	`it.hasNext()` size	`it.next()` 1
iterator-based (for loop)	`for(Iterator it = list.iterator(); it.hasNext();) { System.out.println(it.next()); }`	`iterator()` 0	`it.hasNext()` size	`it.next()` 1
index-based (while loop)	`int m = 0; while(m < list.size()) { System.out.println(list.get(m)); m++; }`	`m=0` 0	`list.size()` size	`m++` 1
index-based (for loop)	`for(int n=0; n < list.size(); n++) { System.out.println(list.get(n)); }`	`n=0` 0	`list.size()` size	`n++` 1
enhanced for loop	`for(String output : list) { System.out.println(output); }`	0	size	1

Every loop structure has a set of variables that control the termination of the loop. In *index-based* loops (Table 1) the loop variables are usually compared in the loop's termination condition with an expression indicating the final iteration value (e.g., `list.size()` in the infix expression `m < list.size()` using the "less than" comparison operator). In *iterator-based* loops (Table 1) the loop variables are iterator objects that provide special functions to access the next element in the traversed collection (e.g., `it.next()`) and check whether the collection contains more elements to be traversed (e.g., `it.hasNext()`). Finally, in *enhanced for* loops the loop variable is hidden from the developer, as it is implemented internally as an iterator object by the language. By analyzing the source code and finding the way that a loop variable is initialized, used, and updated, it is possible to determine the *start index*, *end index*, and *increment/decrement step* of the loop, as shown in the examples of Table 1.

We consider two loops L_i and L_j as functionally equivalent, if they have the same number of *loop variables*, and there is a one-to-one correspondence between their loop variables. As illustrated in Table 1, there are five different implementations of the same functionality (i.e., traversing all elements of a list and printing them one by one). These clones use different control structures, however, they are functionally equivalent. The first two clones use iterator-based loops, they have an additional statement which converts the list to an iterator. For all the iterator-based loops, if they only have one loop variable, we set a default value for the loop variable (start index=0, end index=size, step=1). In clone 3 and 4 which use index-based loops, the start index is indicated by *m=0* and *n=0*, and the increment step is determined by *m++* and *n++*. They both start from 0 and increment step equals to 1. Apart from that, both these two loops traverse all the elements in the list, thus, the end index is the size of the list. By default, in every *enhanced for* loop we assign a pseudo-variable with start index equal to 0, end index equal to the size of the iterated collection or array, and increment step equal to 1.

Two loop variables are considered as *equivalent* if they have the same values for start index, end index, and increment/decrement step. Based on this definition, each variant in Table 1 has a single loop variable, and all loop variables have the same values (start index=0, end index=size, step=1), thus these five loops are considered as functionally equivalent with each other.

Therefore, the problem of determining the functional equivalence of two loop structures can be abstracted to finding and comparing the start index, end index, and increment/decrement step of their corresponding loop variables. This process is implemented in four steps that will be explained in detail in the next paragraphs.

Finding the loop variables: We initially consider as candidate loop variables all variable identifiers that exist in the termination condition of the loop. Next, for each candidate loop variable, we check if it is updated in any of the ways shown in Table 4, either within the body of the loop, or within the updater expression of a `for` loop. We consider as *loop variables* only those candidate variables that are updated during the execution of the loop.

Finding the start index of a loop variable: The start index is determined by finding the last modification of the loop variable before the execution (i.e., first iteration) of the loop. There are two main cases regarding the last modification of the loop variable:

(a) The loop variable is declared only once either before the loop, or within the initializer expression of a `for` loop. In that case, we proceed by analyzing the initializer expression of the loop variable within its declaration.

(b) The loop variable is modified in multiple assignments before the loop. In that case, we are primarily interested in the last assignment, since it kills all previous definitions of the loop variable. If the last assignment is conditional (i.e., it is nested under an `if` statement), then we cannot safely determine the start index value, and thus we assign a “variable” value indicating that the start index cannot be determined at compile-time. If the last assignment is unconditional, then we proceed by analyzing the right operand expression of the assignment.

If the extracted expression matches one of the expression types shown in Table 2, then we can determine a value for the start index of the examined loop variable. The start index can take two possible values, namely "number" indicating that the index of the first iteration is known at compile-time, and "variable" indicating that the index of the first iteration cannot be determined at compile-time (i.e., the expression is a variable or a method call that can be evaluated only at runtime).

Table 2: Supported expressions for the start index

Expression Type	**Example**	**Value**
Method Invocation (iterator-based)	`Collection.iterator()`	0
	`Vector.elements()`	0
	`List.listIterator()`	0
	`List.listIterator(10)`	10
	`List.listIterator(x)`	Unknown
Class Instantiation (iterator-based)	`new StringTokenizer(str)`	0
Number Literal (index-based)	`10`	10
Other Expression	`x` or `foo()`	Unknown

Finding the end index of a loop variable: The end index is determined by analyzing the parent expression in which the loop variable identifier exists within the termination condition of the loop. There are two main cases with respect to the parent expression of the loop variable identifier:

(a) The parent expression is an infix expression with a comparison operator, and the loop variable is the left or right operand of the infix expression. In that case, we proceed by analyzing the other operand of the infix expression.

(b) The parent expression is a method invocation, and the loop variable is the reference through which the method is invoked. In that case, we proceed by analyzing the entire parent expression.

If the extracted expression matches one of the expression types shown in Table 3, then we can determine a value for the end index of the examined loop variable. The end index can take three possible values, namely "size" indicating that all the elements of the involved collection or array will be traversed, "number" indicating that the number of total iterations is known at compile-time, and "variable" indicating that the exact number of iterations cannot be determined at compile-time (i.e., the expression is a variable or a method call that can be evaluated only at runtime).

Finding the step of a loop variable: The step value is determined by analyzing the expressions that update the loop variable (in the ways shown in Table 4) either within the body of the loop, or

Table 3: Supported expressions for the end index

Expression Type	Example	Value
Method Invocation (iterator-based)	`Iterator.hasNext()`	size
	`ListIterator.hasPrevious()`	0
	`Enumeration.hasMoreElements()`	size
	`StringTokenizer.hasMoreTokens()`	size
Method Invocation (index-based)	`Collection.size()`	size
	`String.length()`	size
Qualified Name (index-based)	`array.length`	size
Number Literal (index-based)	`10`	10
Other Expression	`x` or `foo()`	Unknown

within the updater expression of a `for` loop. We are primarily interested in the update expressions that are directly nested within the body of the loop. If there are conditional updates (i.e., variable updates nested under `if` statements), then we cannot safely determine the step value, and thus we assign a "variable" value indicating that the increment/decrement step cannot be determined at compile-time. If all variable updates are directly nested within the body of the loop, then the step value is computed as the sum of the values corresponding to each individual update expression according to Table 4. If at least one of the values in the sum computation is "variable", then the result of the summation is "variable", otherwise it is a number.

Table 4: Supported expressions for the step value

Expression Type	Example	Value
Method Invocation (iterator-based)	`Iterator.next()`	1
	`ListIterator.previous()`	-1
	`Enumeration.nextElement()`	1
	`StringTokenizer.nextElement()`	1
	`StringTokenizer.nextToken()`	1
Postfix Expression (index-based)	`i++`	1
	`i--`	-1
Prefix Expression (index-based)	`++i`	1
	`--i`	-1
Assignment with Number Literal (index-based)	`i += 10` or `i = i+10`	10
	`i -= 10` or `i = i-10`	-10
Other Expression	`i = x` or `foo()`	Unknown

3.2.2 Conditional variant equivalence

Table 5 shows different types of conditional variants that are functionally equivalent. In general, developers use three different variations when implementing conditionals, which can be summarized as follows:

- *If-else statement*: it is the most commonly used control structure for implementing conditional logic.

- *Conditional expression or ternary operator*: it is mostly used within conditional assignment or return statements, where the expression to be assigned or returned is determined based on a condition.

- *Switch statement*: it is used to check the equality of an evaluated expression over a set of fixed values (i.e., literals, or constants). In contrast to the *if-else* statement, it cannot be used to check the evaluated expression based on ranges of values or conditions other than equality, and it does not allow the comparison of the expression with variables that will be evaluated at runtime.

Table 5: Conditional variants

Variant	Code snippet	if-else equivalent
variable assignment with ternary operator	`var = a > b ? a : b;`	`if (a > b) var = a; else var = b;`
invocation expression with ternary operator	`(a > b ? System.out : System.err) .print(msg);`	`if (a > b) System.out.print(msg); else System.err.print(msg);`
invocation argument with ternary operator	`System.out.print( a > b ? a : b);`	`if (a > b) System.out.print(a); else System.out.print(b);`
return statement with ternary operator	`return a > b ? a : b;`	`if (a > b) return a; else return b;`
switch case statement	`switch (var) { case a: ... break; case b: ... break; default: ... }`	`if (var == a) { ... } else if (var == b) { ... } else { ... }`

The conditional expression and `switch` statement are more restrictive control structures compared to the `if-else` statement, and can be only used in special cases. This means that every conditional expression and `switch` statement can be converted to an equivalent `if-else` statement, but not every `if-else` statement can be converted to the other two control structures. It should be noted that in order to support the matching of an `if-else` statement with a conditional expression, we represent the assignment, method invocation, and return statements containing conditional expressions as nodes in the generated CDTs.

An `if-else` statement is equivalent with a conditional expression under the following conditions:

(a) The condition of the `if-else` statement matches with the condition of the conditional expression.

(b) The `if-else` statement has a `then` case and an `else` case, and each case contains a single statement. The case statements, and the statement containing the conditional expression, hereafter denoted as S_{ce}, have the same AST type (i.e., assignment, invocation, or return statements).

(c) Based on the AST type of the statements the following sub-conditions should hold:

- *Assignment statement*: Both assignments in the `if-else` statement have the same variable in their left-hand side. The right-hand sides of the assignments in the `if-else` statement match with the corresponding `then` and `else` cases of the conditional expression.
- *Return statement*: The returned expressions in the `if-else` statement match with the corresponding `then` and `else` cases of the conditional expression.
- *Method invocation statement*: The invocations in the `if-else` statement and S_{ce} refer to the same method. - If the conditional expression is in the i^{th} argument of the method invocation, then the i^{th} arguments of the method invocations in the `if-else` statement should match with the corresponding `then` and `else` cases of the conditional expression. All the remaining arguments of the method invocations in the `if-else` statement should match with the corresponding arguments in S_{ce}.
  - If the conditional expression is in the invoker expression of the method invocation, then the invoker expressions of the method invocations in the `if-else` statement should match with the corresponding `then` and `else` cases of the conditional expression. All arguments of the method invocations in the `if-else` statement should match with the corresponding arguments in S_{ce}.

An `if-else-if` statement is equivalent with a `switch` statement (assuming there is no fallthrough between different cases) under the following conditions:

(a) The number of `if` cases in the `if-else-if` chain is equal to the number of `switch` cases.

(b) If the `if-else-if` chain ends with a final `else` case, then the `switch` statement should have a `default` case.

(c) All conditions in the `if` cases follow the form `e1 == e2` or `e1.equals(e2)`.

(d) All `e1` expressions in the conditions of the `if` cases match with the expression evaluated in the `switch` statement.

(e) There is a one-to-one match between the set of `e2` expressions in the conditions of the `if` cases and the set of expressions in the `switch` cases.

3.3 Functional similarity evaluation

In the previous step, described in Section 3.2, we extracted pairs of functionally equivalent CDT subtrees within different methods, which had the same tree-structure with variations in some of the matched control structures. The reported results of the first step had a high recall, but also included many false positive cases (low precision), because the code nested under the variant control structures may differ. In order to reduce the false positive results reported in the previous step, it is necessary to evaluate and quantify the functional similarity of each clone candidate pair.

For the similarity evaluation, the comparison of LOC or method invocation sequences of two clone candidates would be too simple. In order to assess the functional similarity of the code nested under two control structures in an efficient manner, Java type bindings were extracted from the corresponding code fragment and the similarity was evaluated through bindings comparison. For the purpose of binding collection, we developed an AST visitor which collects all the type binding keys of the used variables, literals, and all the method calls inside the code fragment. Using the Jaccard similarity on two extracted binding sets, our approach could evaluate the functional similarity of two clone candidates.

3.3.1 Java IBinding

A binding can be considered an unique string that represents a variable, object type, or a method invocation in the Java language. The interface IBinding has six sub-interfaces, of which three are

Table 6: Java type binding keys

AST node	**Code**	**Binding key**
IMethodBinding	properties.getProperty(name)	Ljava/lang/System;.getProperties() java.util.Properties;
ITypeBinding	String	Ljava/lang/String;

related to each method, namely IMethodBinding, ITypeBinding, and IVariableBinding. However, IVariableBinding uniquely represents the identifier of variables. In order to enable our approach to tolerate syntactical differences caused by renamed variables, variable bindings were not taken into account in the similarity evaluation. ITypeBinding represents the Java types of all the objects and variables, and IMethodBinding represents the method signatures. Thus, we have chosen these two types of bindings to represent all the method calls, Java types, and literal types inside the code fragment. These bindings are considered to be the fingerprints of the corresponding code, and as illustrated in Table 6, each binding indicates the qualified name of the object. Furthermore, we can analyze the inheritance hierarchy through the binding of corresponding objects. Compared to the information we could get from the source code, we could acquire more information about the corresponding source code entity through type bindings. Based on the analysis and post-processing of the bindings, we could enable the approach to recognize semantically similar data structures and methods.

There are two advantages to using the fingerprints of code instead of analyzing the source code. First, each binding is unique, which helps to avoid confusion with different methods that have the same name. For example, if we have two methods both named `list()`, but which are from different classes and perform different functionalities, they should not be considered to be functionally equivalent. In this situation, the functional difference cannot be recognized from the code syntax, but it can be detected from the different bindings. Also, through the binding of each entity inside the code, we were able to analyze the inheritance structure of the object in order to know the super type and generic type of the entity. This was helpful in enabling our approach to tolerate different data structures and different generic types existing in clones.

3.3.2 Implementation of Binding Visitor

In order to collect the bindings of each statement, we have implemented the class *BindingVisitor* by extending the class *ASTVisitor*, which was provide by Eclipse JDT framework. In the class

BindingVisitor, we have defined eight *visit(ASTNode)* methods for three types of AST nodes: *SimpleName*, *Method Invocation*, and *Literals* (*StringLiteral, CharacterLiteral, TypeLiteral, NumberLiteral, NullLiteral, BooleanLiteral*). Each statement nested inside the advanced matching subtree was passed to the visitor to collect all the bindings inside the statement. Apart from that, in order to make our approach tolerant to functionally equivalent syntax differences, we have performed two post-processing operations on the collected bindings:

1. In order to deal with cases of methods with the same signature being called from different Collection implementations (e.g., method `vector.add()` called through a `Vector` instance in the first fragment, and method `arrayList.add()` called through an `ArrayList` instance in the second fragment), we generalize all Collection subtypes to `java.util.Collection` (i.e., the root type in the Collection hierarchy) in the binding keys. Through this process, all the comparable collection implementations listed in Table 7 would not reduce the similarity score. Furthermore, different generic type differences could also be handled. This improvement has made our approach more flexible to syntax differences.

Table 7: Comparable Collection Implementations

SuperType	**Class**	**Binding**
List	ArrayList	Ljava/util/ArrayList<>;
List	LinkedList	Ljava/util/LinkedList<>;
List	Stack	Ljava/util/Stack<>;
List	Vector	Ljava/util/Vector<>;
Set	HashSet	Ljava/util/HashSet<>;
Set	LinkedHashSet	Ljava/util/LinkedHashSet<>;
Set	TreeSet	Ljava/util/TreeSet<>;

2. In order to deal with loop variants where one of them contains an additional statement that is responsible for getting the current iteration element (e.g., `list.get(i)`, `vector.elementAt(i)`, or `iterator.next()`), while the other one does not contain such a statement (e.g., `enhanced for` loop), we ignore the binding keys that are related to the iteration process. All methods listed in Table 8 are for the same purpose, which is accessing the next element inside the data structure. As a result, they are considered to be functionally equivalent method calls. Our approach is capable of tolerating these syntactically different, but functionally equivalent method calls.

Table 8: Comparable Additional Statement

Class	Method	Binding
ArrayList	.get(int)	Ljava/util/ArrayList;.get(I)TE;
LinkedList	.get(int)	Ljava/util/LinkedList;.get(I)TE;
Vector	.get(int)	Ljava/util/Vector;.get(I)TE;
AbstractList	.get(int)	Ljava/util/AbstractList;.get(I)TE;
List	.get(int)	Ljava/util/List;.get(I)TE;
AbstractSequentialList	.get(int)	Ljava/util/AbstractSequentialList;.get(I)TE;
Stack	.get(int)	Ljava/util/Vector;.elementAt(I)TE;
Vector	.elementAt(int)	Ljava/util/Stack;.elementAt(I)TE;
Iterator	.next()	Ljava/util/ListIterator;.next()TE;
ListIterator	.next()	Ljava/util/LinkedList;.get(I)TE;
Enumeration	.nextElement()	Ljava/util/Enumeration;.nextElement()TE;
StringTokenizer	.nextElement()	Ljava/util/StringTokenizer;.nextElement()LObject;
StringTokenizer	.nextToken()	Ljava/util/StringTokenizer;.nextToken()LString;

3.3.3 Jaccard Similarity Coefficient

Jaccard Similarity coefficient is popularly used to compare the proximity of two data sets. Based on all the bindings collected in the visitor, we have created two binding key sets. Assuming that B_i is the set of binding keys extracted from the first code fragment, and B_j is the set of binding keys extracted from the second code fragment, we have defined the *functional similarity* of the two fragments as the *Jaccard similarity coefficient* of B_i and B_j, which is computed as:

$$J(B_i, B_j) = \frac{B_i \cap B_j}{B_i \cup B_j}$$

The Jaccard similarity coefficient ranges within $[0, 1]$. In our case, a coefficient value equal to zero would indicate that the code fragments do not have any common bindings, while a value equal to one would indicate that the code fragments have only common bindings (i.e., no uncommon bindings). Our assumption is that the more bindings two code fragments have in common (and the less uncommon bindings they have), the more functionally similar they are, since they use the same types and methods to implement their functionality.

We consider *control structure variant clones* to be the pairs of CDT subtrees having a functional similarity that is larger than or equal to a threshold value ϕ.

$J(B_i, B_j) \geq \phi$, where $0 \leq \phi \leq 1$

Obviously, a large threshold value would result in less false positives (i.e., reported cases not having the same functionality) and higher precision, but would also result in more false negatives (i.e., unreported cases having the same functionality) and lower recall. On the other side, a small threshold value would result in more false positives (lower precision) and less false negatives (higher recall). Therefore, in order to find a reasonable trade-off between precision and recall, we perform an experiment in Section 4, where we detect candidate control structure variant clones using different threshold values and determine the true positives, false positives, and false negatives through manual inspection.

Chapter 4

Evaluation

In order to find the best threshold value that produces results with a reasonable trade-off between precision and recall, we applied our method to six open-source projects using a range of threshold values. Through manual examination of the reported clones, we identify the true positives (TP), false positives (FP), and false negatives (FN) to calculate the precision and recall. Finally, we measured the execution time of the proposed method for the analysis of each project.

4.1 Study Setup

For the purpose of evaluation, we applied our approach on open-source projects. In order to avoid bias in the selection of projects, we adopted 6 open-source Java systems that have been already used as experimental subjects in other studies related to semantic clones. As shown in Table 9, the selected projects come from different application domains, have a different development history, ranging from 2 to 13 years, and vary in size, ranging from 93 to 196 KLoC. These variation points certainly affect the characteristics of the detected clones with respect to their domain-specificity, and the maturity/size of the involved code, thus allowing for more generalizable results. Our approach can support any clone detection tool, as long as it provides the location information of the clone candidates. However, each different tool has its own advantages and limitations.

We set the following criteria for the selection of an appropriate clone detector for our experiment:

1. The tool should be able to detect clones with control structure variations.

2. The tool should be available for download.

3. The tool should take a reasonable time to detect clones.

We found many different tools which support the detection of semantic clones, but not all of them are available for download. After searching, we were able to download and try five different detection tools: CCFinder, JSCtracker, NiCad, Deckard, and Sebyte. According to the aforementioned criteria, CCFinder was only able to find Type-1 and Type-2 clones, so it could not provide interesting control structure variant clones. Also JSCtracker was not able to finish the detection process within 48 hours, while NiCad and Deckard could return only a very limited number of control structure variant clones with a restrictive configuration. When the configuration was set with unrestrictive parameters, NiCad returned abnormal clone groups in terms of their size (e.g., 4,000 clones in one group), and Deckard was not able to finish the detection process within 24 hours. SeByte [KRR14] satisfied all criteria, because by design it supports the detection of control structure variations. At the bytecode level, the loops and conditionals have the same implementation, so the syntactical differences are eliminated. Moreover, Sebyte was made available to us by its authors and was the fastest semantic clone detection tool that we had tried. Thus, we selected SeByte, a Java bytecode clone detection tool, to extract the initial set of clone pairs.

SeByte accepts two parameters controlling the size of the clones being reported. The first parameter is related to the minimum number of method calls that should be present in the clone fragments (we have set this parameter value to 2). The second parameter is related to the minimum number of lines of source code in the clone fragments (we have set this parameter value to 4). This configuration can ensure that as many clone candidates as possible will be reported. We have included all reported clone pairs in our evaluation, regardless of the similarity score assigned to them by SeByte.

Table 9: Examined projects

Project	**Domain**	**Age**[†]	**KLoC**[*]	**Classes**[*]
Apache Ant 1.7.0	Java application build tool	6.5	116	1,481
Apache JMeter 2.11	server performance testing tool	13	97	1,072
Columba 1.4	email client	2	101	1,711
Hibernate 3.3.2	Java persistence framework	7.5	173	2,397
JabRef 2.10	bibliography reference manager	10	93	1,002
ArgoUML 0.34	UML modeling tool	9	196	2,298

[†] years of development from the initial release to the examined release
[*] computed with SonarQube

4.1.1 Determining Threshold Value ϕ

In order to find an appropriate threshold value (i.e., a value that yields a reasonable trade-off between precision and recall) for the proposed functional similarity measure, we applied our method using different threshold values, ranging from 0.3 to 0.9 (with a 0.1 step), and determined the True Positive (TP) and False Positive (FP) clone pairs through manual inspection. A True Positive is a clone pair returned by the proposed method that has a control structure variation and both clone fragments implement the same functionality. A False Positive is a clone pair returned by the proposed method that has a control structure variation, but the clone fragments implement a different functionality. We set three criteria to label a candidate clone pair as a true positive (i.e., a real case of control structure variant clone).

1. The clone fragments should perform a similar computation. If the outcome of the computation is a variable, the computed variables in both clone fragments should have the same type.
2. The clone fragments should implement the same sequence of algorithmic steps.
3. The clone fragments should operate on the same or interchangeable data structures (e.g., different implementations of a `Collection`, `StringTokenizer` and `String[]`).

Table 10 shows the number of True Positives and False Positives that were obtained using different threshold values ($0.3 \leq \phi \leq 0.9$) in the examined projects.

Table 10: True Positives (TP) and False Positives (FP) for different threshold values ($0.3 \leq \phi \leq 0.9$)

ϕ	**Ant**		**JMeter**		**Columba**		**Hibernate**		**JabRef**		**ArgoUML**		**Total**	
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
0.9	8	0	1	0	3	0	3	0	5	0	50	1	70	1
0.8	9	0	2	0	3	0	6	0	9	0	73	6	102	6
0.7	11	0	8	0	3	0	6	0	14	0	87	20	129	20
0.6	19	5	13	0	3	0	7	0	21	8	116	39	179	52
0.5	26	12	26	0	7	1	11	2	26	12	163	117	259	144
0.4	27	21	29	7	7	6	11	26	27	24	172	188	273	272
0.3	28	45	30	23	7	15	11	61	29	49	180	282	285	475

In order to compute the precision (P), recall (R), and F-measure (F) obtained with different threshold values, we used the following formulas:

$$P = \frac{TP}{TP + FP} \qquad R = \frac{TP}{TP + FN} \qquad F = 2 \cdot \frac{P \cdot R}{P + R}$$

The computation of recall requires to determine the False Negative (FN) clone pairs. A False Negative is a clone pair not returned by the proposed method that has a control structure variation and both clone fragments implement the same functionality. To extract the False Negatives, we considered the results obtained with a 0.3 threshold as the baseline for computing recall (i.e., the True Positives obtained with a 0.3 threshold are all possible true occurrences). Therefore, the number of False Negatives for $0.3 \leq \phi \leq 0.9$ is computed as $FN_\phi = TP_{0.3} - TP_\phi$. The reason we selected this particular threshold value as the baseline is that as it can be observed in Table 10, when $\phi = 0.3$, in all projects the number of True Positives slightly increases or remains the same, while the increase in the number of False Positives is significantly larger.

Table 11 shows the precision, recall, and F-measure obtained with different threshold values ($0.3 \leq \phi \leq 0.9$). The best obtained F-measure values are highlighted in bold. Based on these results, we can conclude that a reasonable trade-off between precision and recall can be achieved, when the value of ϕ is 0.5. Taking into account all instances mined from the examined projects (last column in Table 11), setting the threshold value to 0.5 achieved a performance score of 0.64 (precision), and 0.91 (recall).

Table 11: Precision (P), Recall (R), and F-measure (F) for different threshold values ($0.3 \leq \phi \leq 0.9$)

ϕ	**Ant**			**JMeter**			**Columba**			**Hibernate**			**JabRef**			**ArgoUML**			**All projects**		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
0.9	1	0.29	0.44	1	0.03	0.06	1	0.43	0.60	1	0.27	0.43	1	0.17	0.29	0.98	0.28	0.43	0.99	0.25	0.39
0.8	1	0.32	0.49	1	0.07	0.13	1	0.43	0.60	1	0.55	0.71	1	0.31	0.47	0.92	0.40	0.56	0.94	0.36	0.52
0.7	1	0.39	0.56	1	0.27	0.42	1	0.43	0.6	1	0.55	0.71	1	0.48	0.65	0.81	0.48	0.61	0.87	0.45	0.59
0.6	0.79	0.68	0.73	1	0.43	0.60	1	0.43	0.6	1	0.64	0.78	0.72	0.72	0.72	0.75	0.64	0.69	0.77	0.63	0.69
0.5	0.68	0.93	**0.79**	1	0.87	**0.93**	0.88	1	**0.93**	0.85	1	**0.92**	0.68	0.90	**0.78**	0.58	0.91	**0.71**	0.64	0.91	**0.75**
0.4	0.56	0.96	0.71	0.81	0.97	0.88	0.54	1	0.7	0.30	1	0.46	0.53	0.93	0.68	0.48	0.96	0.64	0.50	0.96	0.66
0.3	0.38	1	0.55	0.57	1	0.72	0.32	1	0.48	0.15	1	0.27	0.37	1	0.54	0.39	1	0.56	0.38	1	0.55

* The recall is computed relatively to the true positives obtained with a threshold value equal to 0.3

4.1.2 Execution Time

Table 12 shows the execution time of the proposed technique on the clone pairs detected by SeByte. On average, the analysis of a single clone pair takes 8.8 milliseconds. It is negligible in practical usage, so the proposed approach can scale up to large scale industrial systems. Another interesting finding is that out of the 390,976 method-level clone pairs reported by SeByte, only 285 were found to contain control structure variant clone fragments. This means that the number of control structure variant clones is significantly less than the number of syntactical clones. However, these control structure variant clones are far more difficult to detect and track their evolution. This is mainly because of two reasons. First, it is more difficult to be aware of the presence of control structure variant clones, because in most of the cases they are introduced independently of each other in contrast to syntactical clones resulting from copy&paste activities. Second, control structure variant clones have very complex syntactical differences, which make very difficult their detection by standard clone detectors. These two observations both highlight the importance of an automated clone mining method, because it is impossible to manually filter control structure variant clones from a large dataset containing all types of clones.

Table 12: Execution time

Project	**SeByte clone pairs**	**Total time (sec)**	**Average time (ms)**
Ant	54,143	444	8.20
JMeter	64,537	384	5.95
Columba	61,769	293	4.74
Hibernate	75,776	617	8.14
JabRef	26,296	302	11.48
ArgoUML	108,455	1,406	12.96
All projects	390,976	3,446	8.81

* Measurements performed on Intel Core i7-3770 3.4 GHz with 8GB DDR3.

Chapter 5

Clone Refactoring Case Study

As discussed in Section 1.3, finding these control structure variant clones makes possible to examine the different strategies required to refactor them. Thus, based on the 285 true positive control structure variant clones that we found from the six examined open-source projects, we performed some analysis from the perspective of clone refactoring.

In this section, we will first discuss the categorization of the control structure variant clones based on their control structure differences, and then present the challenges we faced when trying to refactor them with JDeodorant[1], a code smell detection and refactoring tool. At the end, we propose some improvements that can be made to extend the refactoring capabilities of JDeodorant.

5.1 Categorization of Control Structure Variations

The clone pairs reported by our approach include different control structure variations. However, with these real cases from open source projects, we wanted to investigate more about how often different control structure variant clones occur as well as the main reason for the generation of control structure variant clones. Thus, we set two research questions for their investigation :

Q1 : Which variations are most frequently occurring in control structure variant clones?

Q2 : How does the evolution of a programming language affect the introduction of control structure variant clones?

Therefore, we manually examined all true positive clone pairs and categorized them based on the different control structures they used.

[1]http://jdeodorant.com/

After examining all 285 true positive cases, we found that there were six different types of control structures involved in the detected control structure variant clones, namely *Enhanced For*, *For*, *While*, *Do-While*, *If-Else Statement*, and *Ternary Operator*. Furthermore, *For Loop* and *While Loop* are both divided into two subcategories, namely *Iterator-based* (e.g., `iterator.hasNext();`) and *Index-Based* (e.g., `i<list.size();`).

For the conditional control structures, we found 18 cases where an *if-else statement* was replaced with the *ternary operator*, or vice versa.

For the loop control structures, as illustrated in Table 13, the six types of loops, namely Enhanced For, For (iterator-based), For (index-based), While (iterator-based), While (index-based), and Do-While form 15 different combinations. Currently, our approach supports 13 of these variation combinations. We do not support the variations marked as N/A in Table 13. Among the 13 supported variation combinations, we found instances in the examined open-source projects for seven of them, while we did not find any instance for the remaining six variation combinations.

Table 13: Loop control structure variations

	For (iterator)	**For** (index)	**While** (iterator)	**While** (index)	**Do-While**
Enhanced for	42	58	109	0	0
For (iterator-based)		N/A	20	0	0
For (index-based)			28	8	0
While (iterator-based)				N/A	0
While (index-based)					2

The results shown in Table 13 indicate that various syntactically different control structures are actually used to implement the same functionality in open-source projects. However, the number of clone pairs in each variation category varies significantly.

The top three largest categories (*Enhanced For* VS *While (iterator)*, *Enhanced for* VS *For (index)* and *Enhanced for* VS *For (iterator)*), have 209 clone pairs, accounting for 73% of all cases. This statistic answers research question **Q1**, which is that these four types of control structures (*Enhanced for*, *Iterator-based while*, *Iterator-based for*, and *Index-based for*) are more prone to appear in clone variations. We can especially see that *Enhanced for loop* is involved in all of these three largest groups. This verifies our assumption made in Section 1.2.3 that control structure variant clones are introduced from migrating a subset of the clones in an existing clone group to new language constructs.

The largest category, *Enhanced for loop VS While loop (Iterator-based)*, has 109 cases, indicating that the enhanced for loop and the iterator-based while loop were most often used as alternatives to each other. If we analyze these two control structures from the functionality perspective, we can find that the enhanced for loop is internally implemented with an iterator, which has exactly the same functionality as the iterator-based while loop. However, due to the significant syntax differences between them, most syntax-based clone detection tools would omit them.

The second-largest category, *Enhanced for loop VS For loop (Index-based)*, has 56 cases, while the third-largest category, *Enhanced for loop VS For loop (Iterator-based)* has 42 cases. All together, 98 regular for loop implementations are shown to be clones of enhanced for loop implementations. The interesting finding is that, even in the same clone group, the code implementation is not consistent, some clones are converted to enhanced for loop, while others remain as regular for loop.

This further strengthens our aforementioned assumption that when the enhanced for loop was introduced in Java 5, some traditional for loop implementations were replaced with the enhanced for loop. However, due to the existence of clones in a large system, the developers failed to find and update all duplications. Thus, some of the clones were omitted from being updated to use the enhanced for loop construct. Even worse, the incomplete updating caused the syntactical clones to become control structure variant clones, which are much more difficult to track and manage.

These results clearly show that the answer to the **Q2** is that, the evolution of a programming language is one of the major reasons affecting the consistency of code implementation and leads to the generation of functionally equivalent clones, which results in more challenging problems for software maintenance.

5.2 Clone Refactoring evaluation with JDeodorant

JDeodorant is a clone refactoring tool implemented by Krishnan and Tsantalis [KT14], which is capable of refactoring various types of syntactical clones. However, due to the lack of real functional clones, it has not been applied on clones which include significant syntactical differences. In order to find out how well JDeodorant can deal with syntactically different clones, we applied it to the control structure variant clones that we found. Clone refactoring is essentially the process of removing clones by unifying the common code. According to the refactoring techniques defined by Fowler [Fow99], one of the most commonly used refactoring strategies for clones is the *Extract Method Refactoring*. It extracts common code fragments and parameterizes any syntactical differences. After that, the clones are replaced with an invocation of the newly extracted method. However, the refactoring process should be accompanied by some preconditions, which ensure that the behavior of the program will be preserved after refactoring. Krishnan and Tsantalis [KT14] defined four preconditions:

1. The parameterization of differences between the matched statements should not break existing data-, anti-, and output-dependences.
2. The unmatched statements should be movable before or after the matched statements without breaking existing data-, anti-, and output-dependences.
3. The duplicated code fragments should return at most one variable of the same type.
4. Matched branching (break, continue) statements should be accompanied with corresponding matched loop statements.

5.2.1 Refactorability of control structure variant clones

Based on the precondition violations detected by JDeodorant, we categorize the control structure variant clones into five categories:

1. *Refactorable* includes the clone pairs that have no precondition violations and can be directly refactored.
2. *Collection type mismatch* includes the clone pairs which cannot be refactored, because they contain loops operating on different types of collections.
3. *Unmatched statement* includes the clone pairs which are not refactorable, because of unmatched statements.
4. *Collection type mismatch & Unmatched statement* includes the clone pairs which are not refactorable, because they contain loops operating on different types of collections and unmatched statements.
5. *Others* includes the clone pairs which are not refactorable, because of other reasons.

Figure 7 shows the percentages of the aforementioned categories separately for each examined project, as well as for all projects together. In total, 42% of the control structure variant clones are

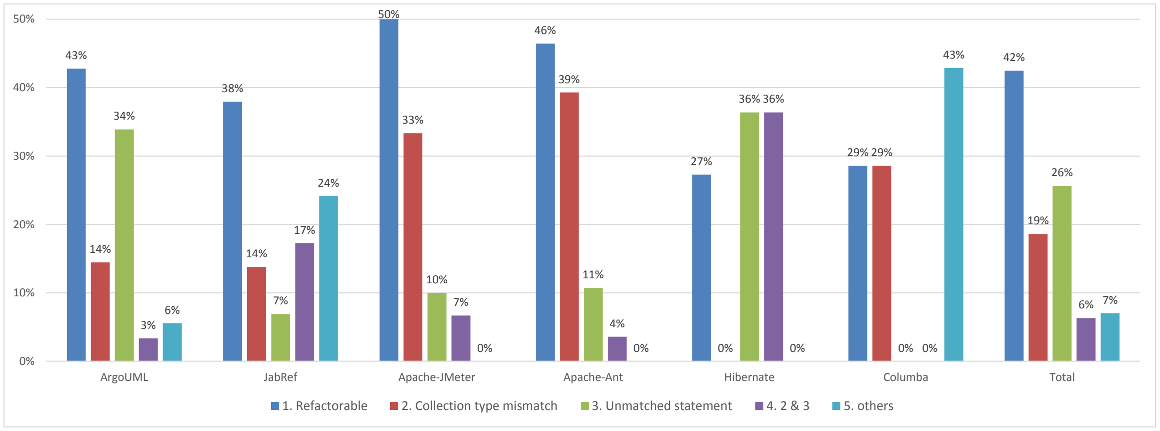


Figure 7: Refactorability analysis of the detected control structure variant clones

directly refactorable. 19% of the clones belong to category #2, which means that in order to make their refactoring feasible we should first find a way to convert the collection used in the first clone, to the one used in the second clone. Finally, 26% of the clones are not refactorable, because they contain statements inside the bodies of the control structure variant loops that cannot be matched due to extreme syntactical differences. These unmatched statements depend on the iteration, and thus cannot be moved outside the loop, which is a violation of precondition #2. We will discuss some of these cases in the next sub-section.

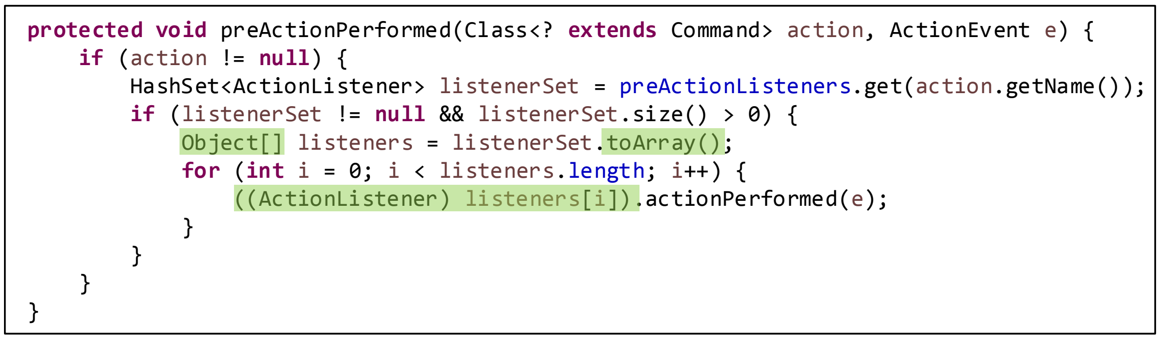
5.2.2 Variations hindering clone refactoring

During the examination of the control structure variant clones, we found that apart from variations in the control structure, there exist other functionally equivalent variations that hinder the refactoring of the clones. Thus, in this subsection we discuss five types of common variations.

Initialization of arrays from collections

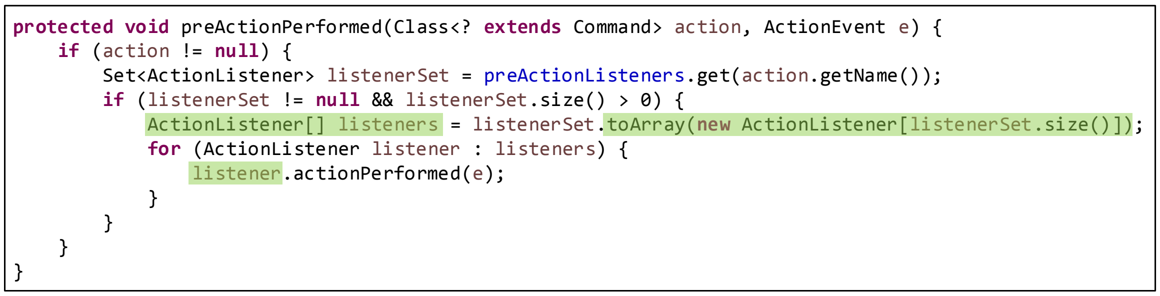
In the clone pair presented in Figure 8, we can find three types of variations, 1) Control structure variation (*for loop* vs *enhanced for loop*), 2) Collection type variation (`HashSet<ActionListener>` and `Set<ActionListener>`), and 3) Array initialization variation (`.toArray()` without arguments and `.toArray(new ActionListener[size])`).

The first two types of differences can be already unified by JDeodorant, but the different initializations of arrays cannot be matched. The first method `.toArray()` returns an array of `Objects`, while the second method `.toArray(T[])` returns an array of `ActionListener` objects. Thus, JDeodorant



```
protected void preActionPerformed(Class<? extends Command> action, ActionEvent e) {
    if (action != null) {
        HashSet<ActionListener> listenerSet = preActionListeners.get(action.getName());
        if (listenerSet != null && listenerSet.size() > 0) {
            Object[] listeners = listenerSet.toArray();
            for (int i = 0; i < listeners.length; i++) {
                ((ActionListener) listeners[i]).actionPerformed(e);
            }
        }
    }
}
```

(a) Clone 1



```
protected void preActionPerformed(Class<? extends Command> action, ActionEvent e) {
    if (action != null) {
        Set<ActionListener> listenerSet = preActionListeners.get(action.getName());
        if (listenerSet != null && listenerSet.size() > 0) {
            ActionListener[] listeners = listenerSet.toArray(new ActionListener[listenerSet.size()]);
            for (ActionListener listener : listeners) {
                listener.actionPerformed(e);
            }
        }
    }
}
```

(b) Clone 2

Figure 8: Functionally equivalent clones-1 in project ArgoUML

considers these two statements to be unmapped and suggests to move the unmapped statements outside the clone in order to refactor them. These two statements have direct data dependencies on the loop statements, and according to precondition #2, they are not movable, otherwise the existing data dependencies would break. Therefore, this pair of clones is considered as not refactorable by JDeodorant. However, through manual examination, we can see that in clone 1, the element of the array is finally casted into an `ActionListener` object, which is the same as in clone 2. Therefore, the syntactical differences can be actually eliminated if we perform some normalization on the code.

Temporary variables

As illustrated in Figure 9, despite the control structure variation (*for loop* vs *while loop*) in these two clones, there is another difference which introduces a gap statement. In clone 1, the argument `ce` passed to the method `.stateChanged(ce)` is a local variable declared in statement `ChangeEvent ce = new ChangeEvent(this);`, but in clone 2, the temporary variable is inlined as an argument in the method call `.stateChanged(new ChangeEvent(this))`.

JDeodorant can currently refactor these two clones by introducing two parameters. The first parameter will be for the two Collections being different fields (`mChangeListeners` and `listeners`),

```
private void notifyChangeListeners() {
    ChangeEvent ce = new ChangeEvent(this);
    for (int index = 0; index < mChangeListeners.size(); index++) {
        mChangeListeners.get(index).stateChanged(ce);
    }
}
```

(a) Clone 1

```
private void fireFileChanged() {
    Iterator<ChangeListener> iter = listeners.iterator();
    while (iter.hasNext()) {
        iter.next().stateChanged(new ChangeEvent(this));
    }
}
```

(b) Clone 2

Figure 9: Functionally equivalent clones-1 in project Jabref

and the second parameter will be for the difference in the arguments of the `.stateChanged()` method calls. However, there is a better refactoring solution which requires one less parameter. It is obvious that the inlined argument can be extracted and placed into an independent statement. In this way, it will be possible to unify the code into the same syntax and eliminate the differences. This problem occurs very often in the clone cases we examined. Thus, it will be helpful if we can perform the normalization by extracting all inlined expressions used as arguments and create local variables for each one of them.

Exchange of method invocation expression with argument

In figure 10, we can see two control structure variant clones, and there is an `if statement` inside each clone loop where the conditional expression is an `equals()` method invocation.

```
Collection dataTypes = Model.getCoreHelper().getAllDataTypes(model);
for (Object dataType : dataTypes) {
    if (typeName.equals(Model.getFacade().getName(dataType))) {
       return dataType;
    }
}
```

(a) Clone 1

```
Collection classes = getCoreHelper().getAllClasses(ns);
Iterator it = classes.iterator();
while (it.hasNext()) {
    Object candidateClass = it.next();
    if (Model.getFacade().getName(candidateClass).equals(identifier)) {
        return candidateClass;
    }
}
```

(b) Clone 2

Figure 10: Functionally equivalent clones-2 in project Jabref

Although the two loops can be recognized as functionally equivalent, these two clones still cannot be refactored, because in clone 1 the conditional expression is in the form `objectA.equals(objectB)`, while in clone 2 it is reversed and implemented in the form `objectB.equals(objectA)`.

Thus, in the clone matching process, variable `typeName` is considered as matching with expression `Model.getFacade().getName(candidateClass)`, and variable `identifier` is considered as matching with expression `Model.getFacade().getName(dataType)`. Thus, we need to introduce two parameters in order to parameterize these two differences. However, the parameterization of these two differences violates precondition #1, because the expressions to be parameterized have direct data dependencies on the loop statements. Therefore, the clones cannot be refactored with the current refactoring strategy supported by JDeodorant. However, these two method invocations are essentially the same. If we normalize the invocation of method `.equals()`, these two clones will become refactorable.

Alternative branching statements

The clones shown in Figure 11 use different control structures. In addition to the control structure difference, clone 1 uses a `break;` statement and returns variable `stereotype` and the end of the method, while clone 2 directly returns variable `tagDefinition` without using a `break;` statement.

According to precondition #4, the clones cannot be refactored, because the `break;` statement of clone 1 cannot be matched with a corresponding `break;` statement in clone 2. However, if we analyze

```
private Object getStereotype(Object modelElement, String stereotypeName) {
    Object stereotype = null;
    Collection stereotypes = StereotypeUtility.getAvailableStereotypes(
            modelElement);
    for (Iterator it = stereotypes.iterator(); it.hasNext();) {
        Object candidateStereotype = it.next();
        if (getFacade().getName(candidateStereotype).equals(
            stereotypeName)) {
            stereotype = candidateStereotype;
            break;
        }
    }
    return stereotype;
}
```

(a) Clone 1

```
protected Object getTagDefinition(String stereoName, String tdName) {
    Object stereo = getCppStereotypeInModel(stereoName);
    assertModelElementContainedInModels(stereo);
    Collection tagDefinitions = getFacade().getTagDefinitions(stereo);
    for (Object tagDefinition : tagDefinitions) {
         if (tdName.equals(getFacade().getName(tagDefinition))) {
             return tagDefinition;
         }
    }
    return null;
}
```

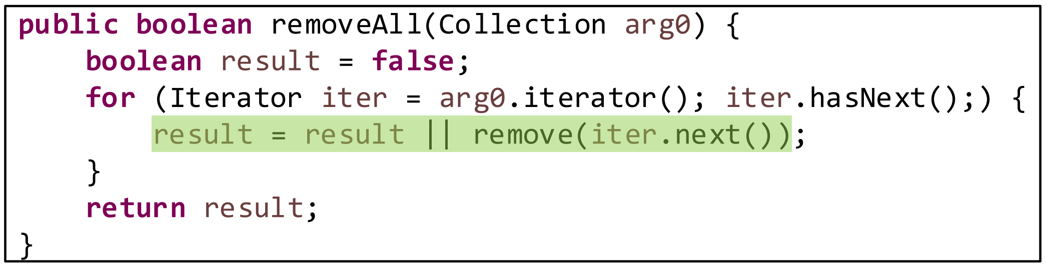
(b) Clone 2

Figure 11: Functionally equivalent clones-2 in Project ArgoUML

the execution of the code, we can find that these two different execution flows are essentially the same. Both implementation access the elements of a collection, find the correct element, and then return it. The only difference is, in clone 1, the found element `candidateStereotype` is assigned to a new variable `stereotype`, and then the loop breaks and `stereotype` is returned, while in clone 2, the found element `tagDefinition` is not assigned to a new variable and is returned directly without a `break;` statement.

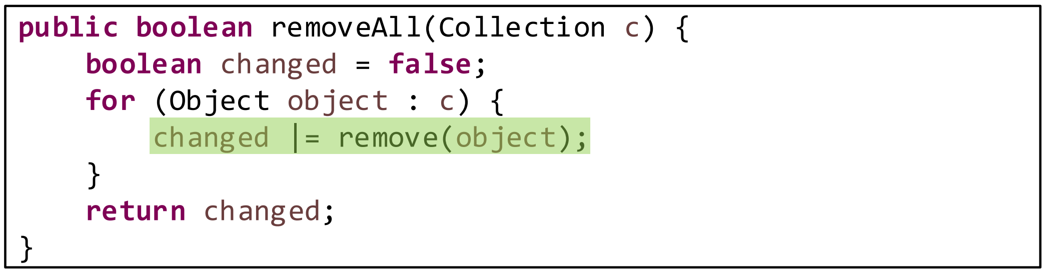
Replacement of compound assignment operator with regular assignment

Java supports a large number of operators, and most of them can be represented in different forms of syntax. As shown in the example presented in Figure 12, clone 1 uses the logical-OR operator `||` on the operands `result` and `remove(iter.next())` and then assigns the result to variable `result`, while clone 2 uses the compound assignment operator `|=` to directly assign the result of the bitwise-OR operation `|` between operands `changed` and `remove(object)` to variable `changed`.



```java
public boolean removeAll(Collection arg0) {
    boolean result = false;
    for (Iterator iter = arg0.iterator(); iter.hasNext();) {
        result = result || remove(iter.next());
    }
    return result;
}
```

(a) Clone 1



```java
public boolean removeAll(Collection c) {
    boolean changed = false;
    for (Object object : c) {
        changed |= remove(object);
    }
    return changed;
}
```

(b) Clone 2

Figure 12: Functionally equivalent clones-3 in Project ArgoUML

Because of these extreme syntactical differences, JDeodorant considers these two statements to be unmapped and suggests to move the unmapped statements outside the clone in order to refactor them. However, these two statements have direct data dependencies on the loop statements, and according to precondition #2, they are not movable, otherwise the existing data dependencies would break. A normalization of the compound assignment operator to the corresponding regular assignment operator would make this case refactorable.

Chapter 6

Conclusion and Future Work

The management of syntactically different functional clones is a challenging and interesting research problem, and has a great impact on software maintainability. This work complements existing research towards the detection and management of clones with significant variations in the control structures being used to implement loops and conditionals.

With the proposed approach, we can mine control structure variant clones by supporting the detection of 13 types of functionally equivalent control structure variations, regardless of the Collection types being iterated. Our method can accurately find clones with significant syntax differences with a precision of 0.64 and a recall of 0.91, using a threshold value for the functional similarity of the clones equal to 0.5. Finally, the proposed approach can also extract clone difference information, which can be used to assess the refactorability of the clones.

By analyzing the control structure variant clones found in six open-source projects, we have categorized the clones based on their control structure variations. The results indicate that various control structures are involved in control structure variant clones, and the evolution of a programming language is one of the major reasons causing the introduction of these clones. For example, the *Enhanced for* loop, which was introduced in Java 5, appears in the vast majority of the detected clones (73%). All detected control structure variant clones have been made publicly available and can serve as a benchmark for clone detection and refactoring tools.

With the control structure variant clones that we have found, we conducted an evaluation about the refactoring capability of JDeodorant, a state-of-the-art clone refactoring tool. Out of the 285 control structure variant clones found, 42% of them could be directly refactored by JDeodorant. For the cases which could not be refactored, we have listed the variations hindering their refactoring and discusses the improvements that could be made to extend the refactoring capability of JDeodorant.

As future work, we first plan to improve the refactoring technique in order to refactor these control structure variant clones. To achieve this goal, we need to develop a function which can unify the different code implementations, such as the ones shown in Section 5.2.2, into the same syntax. After that, both clones will be replaced with the unified implementation and become syntactical clones, which can be more easily handled by JDeodorant. In addition, we also want to improve the technique to handle variations in the Collection types being iterated (e.g., *Array* vs `Collection`, *String array* vs `StringTokenizer`). With this improvement, JDeodorant will be able to refactor an additional 19% of the control structure variant clones.

Bibliography

[Bak95] B. 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, Washington, DC, USA, 1995. IEEE Computer Society.

[BKA+07] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions.*, 33(9):577–591, Sept 2007.

[BKZ11] Liliane Barbour, Foutse Khomh, and Ying Zou. Late propagation in software clones. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pages 273–282, 2011.

[BSK11] David J. Barnes and Dermot Shinners-Kennedy. A study of loop style and abstraction in pedagogic practice. In *Proceedings of the Thirteenth Australasian Computing Education Conference*, pages 29–36, 2011.

[BYM+98] 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, Washington, DC, USA, 1998. IEEE Computer Society.

[CNPR14] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. A survey of software aging and rejuvenation studies. *ACM Journal on Emerging Technologies in Computing Systems*, 10(1):8:1–8:34, January 2014.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference* *on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[CWT14] Xiliang Chen, Alice Yuchen Wang, and Ewan Tempero. A replication and reproduction of code clone detection studies. In *Proceedings of the Thirty-Seventh Australasian Computer Science Conference - Volume 147*, ACSC '14, pages 105–114, Darlinghurst, Australia, Australia, 2014. Australian Computer Society, Inc.

[DNR06] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):37–58, January 2006.

[DRNN13] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. A large-scale empirical study of java language feature usage. Technical Report 13-02, Iowa State University, Department of Computer Science, 2013.

[ELE$^+$12] Rochelle Elva, Gary T. Leavens, Rochelle Elva, Dr. Gary, and T. Leavens. Jsctracker: A semantic clone detection tool for java code, 2012.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[Gie06] Simon Giesecke. Generic modelling of code clones. In *Proceedings of Duplication, Redundancy, and Similarity in Software, ISSN 16824405, Dagstuhl*, 2006.

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[GJS08] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[JDH10] Elmar Jürgens, Florian Deissenboeck, and Benjamin Hummel. Code similarities beyond copy paste. In *CSMR'10*, pages 78–87, 2010.

[JMSG07] 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, Washington, DC, USA, 2007. IEEE Computer Society.

[Jon06] Capers Jones. Abstract the economics of software maintenance in the twenty first century, 2006.

[JS09] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 81–92, New York, NY, USA, 2009. ACM.

[Kam13] Toshihiro Kamiya. Agec: An execution-semantic clone detection tool. In *ICPC*, pages 227–229, 2013.

[KDM$^+$96] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Reverse engineering. chapter Pattern Matching for Clone and Concept Detection, pages 77–108. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[KG03] Cory Kapser and Michael W. Godfrey. Toward a taxonomy of clones in source code: A case study, 2003.

[KJKY11] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. Mecc: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 301–310, New York, NY, USA, 2011. ACM.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[Kri01] Jens Krinke. Identifying similar code with program dependence graphs, 2001.

[KRR12a] I Keivanloo, C.K. Roy, and J. Rilling. Sebyte: A semantic clone detection tool for intermediate languages. In *2012 IEEE 20th International Conference on Program Comprehension (ICPC)*, pages 247–249, June 2012.

[KRR12b] Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *IWSC'12*, pages 36–42, 2012.

[KRR14] Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. Sebyte: Scalable clone and similarity search for bytecode. *Science of Computer Programming*, 95:426–444, 2014.

[KT14] G.P. Krishnan and N. Tsantalis. Unification and refactoring of clones. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, pages 104–113, 2014.

[LM11] Thierry Lavoie and Ettore Merlo. Automated type-3 clone oracle using levenshtein metric. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 34–40, New York, NY, USA, 2011. ACM.

[LS80] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[Par94] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[PMP00] Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. Jplag: Finding plagiarisms among a set of programs. Technical report, 2000.

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, May 2009.

[SZ12] Saeed Shafieian and Ying Zou. Comparison of clone detection techniques. Technical report, Queen's University, August 2012.

[TJG11] Robert Tairas, Ferosh Jacob, and Jeff Gray. Representing clones in a localized manner. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 54–60, New York, NY, USA, 2011. ACM.

[Vli08] Hans van Vliet. *Software Engineering: Principles and Practice.* Wiley Publishing, 3rd edition, 2008.

[WWSM14] Tiantian Wang, Kechao Wang, Xiaohong Su, and Peijun Ma. Detection of semantically similar code. *Frontiers of Computer Science*, pages 1–16, 2014.

[XXJ11] Zhenchang Xing, Yinxing Xue, and S. Jarzabek. Clonedifferentiator: Analyzing clones by differentiation. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 576–579, Nov 2011.

[YYFI11] Shunsuke Yoshioka, Norihiro Yoshida, Kyohei Fushida, and Hajimu Iida. Scalable detection of semantic clones based on two-stage clustering. International Symposium on Software Reliability Engineering (ISSRE 2011), 11 2011.