Dealing with Clones in Software : A Practical Approach from Detection towards Management

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

By
Md. Sharif Uddin

Permission to Use

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

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

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

Abstract

Despite the fact that duplicated fragments of code also called code clones are considered one of the prominent code smells that may exist in software, cloning is widely practiced in industrial development. The larger the system, the more people involved in its development and the more parts developed by different teams result in an increased possibility of having cloned code in the system. While there are particular benefits of code cloning in software development, research shows that it might be a source of various troubles in evolving software. Therefore, investigating and understanding clones in a software system is important to manage the clones efficiently. However, when the system is fairly large, it is challenging to identify and manage those clones properly. Among the various types of clones that may exist in software, research shows detection of near-miss clones where there might be minor to significant differences (e.g., renaming of identifiers and additions/deletions/modifications of statements) among the cloned fragments is costly in terms of time and memory. Thus, there is a great demand of state-of-the-art technologies in dealing with clones in software.

Over the years, several tools have been developed to detect and visualize exact and similar clones. However, usually the tools are standalone and do not integrate well with a software developer's workflow. In this thesis, first, a study is presented on the effectiveness of a fingerprint based data similarity measurement technique named 'simhash' in detecting clones in large scale code-base. Based on the positive outcome of the study, a time efficient detection approach is proposed to find exact and near-miss clones in software, especially in large scale software systems. The novel detection approach has been made available as a highly configurable and fully fledged standalone clone detection tool named 'SimCad', which can be configured for detection of clones in both source code and non-source code based data. Second, we show a robust use of the clone detection approach studied earlier by assembling its detection service as a portable library named 'SimLib'. This library can provide tightly coupled (integrated) clone detection functionality to other applications as opposed to loosely coupled service provided by a typical standalone tool. Because of being highly configurable and easily extensible, this library allows the user to customize its clone detection process for detecting clones in data having diverse characteristics. We performed a user study to get some feedback on installation and use of the 'SimLib' API (Application Programming Interface) and to uncover its potential use as a third-party clone detection library. Third, we investigated on what tools and techniques are currently in use to detect and manage clones and understand their evolution. The goal was to find how those tools and techniques can be made available to a developer's own software development platform for convenient identification, tracking and management of clones in the software. Based on that, we developed a clone-aware software development platform named 'SimEclipse' to promote the practical use of code clone research and to provide better support for clone management in software. Finally, we performed an evaluation on 'SimEclipse' by conducting a user study on its effectiveness, usability and information management. We believe that both researchers and developers would enjoy and utilize the benefit of using these tools in different aspect of code clone research and manage cloned code in software systems.

Acknowledgements

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

I would like to thank Dr. Carl Gutwin, Dr. Nadeem Jamali and Dr. Aryan Saadt-Mehr for their willingness to take part in the advisement and evaluation of my thesis work.

Thanks to all of the members of the Software Research Lab with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Minhaz Fahim Zibran, Muhammad Asaduzzaman, Ripon Saha, Mohammad Asif Ashraf Khan, Md. Saidur Rahman, Masud Rahman, Shamima Yeasmin, Manishankar Mondal and Jeff Svajlenko.

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

I thank Dr. Abram Hindle for being involved in one of my study and the anonymous reviewers for their valuable comments and suggestions in improving the papers produced from this thesis.

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

Finally, I would like to express my deepest affection and gratefulness to my my mother Jahanara Begum and my father Md Kutub Uddin for the unconditional love and support throughout this period.

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

Contents

List of Tables

List of Figures

List of Abbreviations

AST	Abstract Syntax Tree
API	Application Program Interface
DPM	Dynamic Pattern Matching
IDE	Integrated Development
PDG	Program Dependency Graphs
IR	Information Retrival
LSH	Locality Sensitive Hashing

Chapter 1

Introduction

1.1 Motivation

Software that is useful and successful in practice is almost always needs to be maintained through continual change and improvements. This maintenance phase is one of the most important parts of the Software Development Life Cycle as it consumes a large part of the overall life-cycle costs. Studies show that up to 80% of total effort on software is spent on its maintenance [4]. Besides, potential business opportunities are lost if the changes in the software are not done quickly and reliably. There are several reasons that might cause changes in software including fixing errors, adding or enhancing features, or improving performance. Throughout this changing life-cycle, multiple copies of similar code fragments or artifacts also known as software clones or code clones in a software system can be modified by various development activities performed by the developer. These activities collectively control the evolution of code clones in the life-cycle of a software system.

Software systems can contain cloned code in amounts ranging from 5-15% of the code-base [146] to as high as 50% [144]. Thus, the effects of clones in software maintenance and evolution have become one of the prime concerns of the software engineering community [57, 71, 77, 86, 118, 128, 140, 154, 159, 168]. While there are some notable benefits of code cloning including faster development and the reduction of maintenance effort and costs as revealed in a number of studies [9, 26, 71, 77, 95, 106, 140, 154], it comes with some bad side effects as well. For example, studies [130, 13, 17, 70, 118, 119, 129] show concrete empirical evidence of a clone's harmful impacts (e.g.: bug propagation, inconsistent code change, increased instability of code) on software maintenance. Therefore, arguably it is inconclusive whether clones are harmful or not. However, researchers commonly agreed that in order to make best use of the good aspects of code cloning as well as reduce the possible harmful effects, an efficient and cost-effective way is needed to manage the clones.

1.2 Problem Statement

Cloning can be a substantial problem during development and maintenance unless special care is taken to find and track existing clones and their evolution [6]. Source code clone detection has been greatly studied over the past decade. A number of state of the art clone detection tools [147, 78, 88, 91, 30, 20, 25] are in use in software clone research. In recent times, clone detection research focuses on faster, more robust,

incremental and semantic clone detection approaches. The research trend in the area also covers the study in determining what to do with the clones when they are detected. A significant portion of the research includes but is not limited to finding ways to manage clones, gaining more control of their generation and studying clone evolution and their effects on the evolution of software.

This thesis covers some areas of the current software code clone based research, which can be defined as the following research problems:

1. *Fast and scalable detection of near-miss clones:* One major problem for clone detection on large corpora is the performance of querying and retrieving possible clones. Existing popular techniques [23, 88, 113] have several deficiencies, such as not supporting the detection of Type-3 near-miss clones where lines could be modified, added and/or deleted in the copied fragments, and not scaling adequately to handle clone detection in large systems [78]. Therefore, a fast and highly scalable near-miss clone detection technique would be a useful addition to the clone detection based study.

2. *Highly customizable and portable clone detection service:* Most of the popular clone detection tools currently being used in practice are standalone. One major usability issue is that the detection service of those tools are not very portable to other applications (clone visualization, analysis, etc.), where clone detection is a prerequisite. Manual feeding of pre-detected clones into those applications adds an external tool dependency on those applications, which sometimes raise various compatibility issues. Besides, limited tailorability restricts the use of existing standalone clone detection tools on data having diverse characteristics. Therefore, there is demand to have a highly customizable and portable clone detection service/API. Such an API can easily be plugged into a clone processing application to have the clones in a system detected on the fly and processed in an organized and efficient way.

3. *Managing clones in a software development environment:* Convenient access to clone information from within the development environment is a key factor in managing clones in software. Usually, it is the developer's activities that directly (e.g., by copying, pasting or modifying code) or indirectly (e.g., by using code generation programs or tools) introduce cloned code in a system. Manual tracking of those activities whether they unknowingly cause clones in the system might be inconvenient for developers, especially in large systems. Besides, without an effective use of clone information, it is also hard for a developer to manage clones efficiently in the system (e.g., refactoring and removing cloned code) or to reduce their negative effects during software evolution. Even if the knowledge of clones in a system is available (for example, by using a standalone clone detection tool), in most cases, a developer needs to apply that knowledge by manual investigation and identification inside the working code-base to locate the clone and possibly take some actions on them. The bottom line, however, is that the lack of availability state of the art clone detection and management features in a developer's working platform could make clone management in software a challenging and potentially error prone task.

1.3 Contributions of the Thesis

In this thesis, a fast and scalable near-miss structural software clone detection process is presented. The process has been made available for practical use in the form of a standalone clone detection tool as well as a portable clone detection library/API. A clone-aware software development platform is also proposed here. The idea is to make the stat-of-the-art clone based technologies available to the developer on top of a single platform for convenient clone management in software. The overall contributions of the thesis are three-fold, as follows:

- *First*, a study is done on the effectiveness of *simhash* [33], a state of the art fingerprint based data similarity measurement technique for detecting the syntactic code clones (textually similar code) in large scale software systems. '*simhash*' is indeed found effective in the experiment in identifying various types of clones in a software system and enables faster near-miss clone detection in large corpora. In the experiment, a novel approach is developed for structural clone detection based on fingerprinting of source code using *simhash*. A multi-level indexing scheme on *simhash* values is also proposed to organize the pre-processed codebase on which the clone detection algorithm is applied. The indexing scheme speeds up the potential clone search and allows the approach to be scalable by maintaining the indices in persistent storage. Finally, *SimCad* is developed to make the structural clone detection approach usable in practice. As a highly configurable and scalable standalone clone detection tool, *SimCad* can be configured and used for detecting clones in both source and non-source code data.

- *Second*, the challenges in using standalone clone detection tools in other clone based research is studied. Based on the study, a clone detection API *SimLib* is developed to serve as an off-the-shelf library to make the *SimCad*'s clone detection service more portable and thus overcome some of the usability issues of a typical standalone clone detection tool. This library can be integrated easily into other compatible applications to provide on-demand clone detection service from within the host application; a practical example of which has been shown later in the final part of the thesis. Besides, the modular architecture makes *SimLib* a highly configurable and extensible API that can be tailored with minimal effort to build a fully customized clone detection tool for target data. At the end, a user study has been performed to get some user feedback on installation and use of the API towards clone detection needs in various forms.

- *Finally*, we conducted a user study in order to answer three research questions on finding a viable platform for managing clones by the developers. The developer centric study is performed to find some efficient way to provide tool support for dealing with clones in the software within a platform where clone management activities can easily be integrated into software developers' development workflow. Based on that study, *SimEclipse* is developed as an IDE (Integrated development Environment) plugin for detection, visualization and managing clones in software. This plugin stands as a practical example

for a successful use of *SimLib* API for providing clone detection service to another tool. It provides a clone detection friendly and clone-aware software development environment that provides a basis for developing IDE based tools for managing clones. By using the clone tracking feature of *SimEclipse* a developer can gain more control of the generation and evolution of clones in project. Since clones are evolved during various development activity performed by the developer, a clone-aware development environment could be the place where evolution of clones can be managed more easily and conveniently.

1.4 Related Publications

A couple of parts of this thesis have been previously published. This section lists these publications. I was the primary author and conducted the research under the supervision of Chanchal K. Roy and Kevin A. Schneider.

- Chapter 3 has been published in Proceedings of the 18th IEEE Working Conference on Reverse Engineering (WCRE 2011) co-authored with Chanchal K. Roy, Kevin A. Schneider and Abram Hindle [172].
- Chapter 4 has been published in Proceedings of the Tool Demonstration Track of the 21st IEEE International Conference on Program Comprehension (ICPC 2013) co-authored with Chanchal K. Roy, and Kevin A. Schneider [171].

1.5 Outline of the Thesis

In this chapter, motivation toward the thesis is discussed in terms of software clone detection and clone management and described the contributions of the thesis. The rest of the thesis is organized as follows:

Chapter 2 presents relevant terminology and outlines the related research which will form the foundation we build upon in this thesis.

Chapter 3 presents a study on finding the effectiveness of *simhash* based fingerprinting in software code clone detection.

In Chapter 4, *SimCad* is introduced as a highly configurable and scalable clone detection tool developed based on an earlier study.

Chapter 5 presents an overview of usability issues of typical standalone clone detection tools in clone based research and clone management. It demonstrates *SimLib*, a portable clone detection library available to be used as a clone detection service provider to a host application along with a small scale user study on the installation and target use of the API.

Chapter 6 presents a user study in order to answer three research questions on finding a way to provide efficient clone management support to the developers. It covers the integration of state of the art clone detection and analysis techniques into software developer's development workflow so that clones in software can be managed more conveniently from the environment were they evolve. It demonstrates *SimEclipse*, an

IDE plug-in that provides a clone-aware software development environment along with a number of clone based technologies that would help developers in managing clones in software.

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

Chapter 2

Background and Related Work

This chapter presents background information related to the research area of this thesis along with previous studies done in those areas. Since this thesis primarily focuses on dealing with clones in software, from detection to manage, it will cover the description of code clones, reasons for cloning, effects of clones in software, approaches for clone detection, clone evolution and clone management.

2.1 Software Code Clones

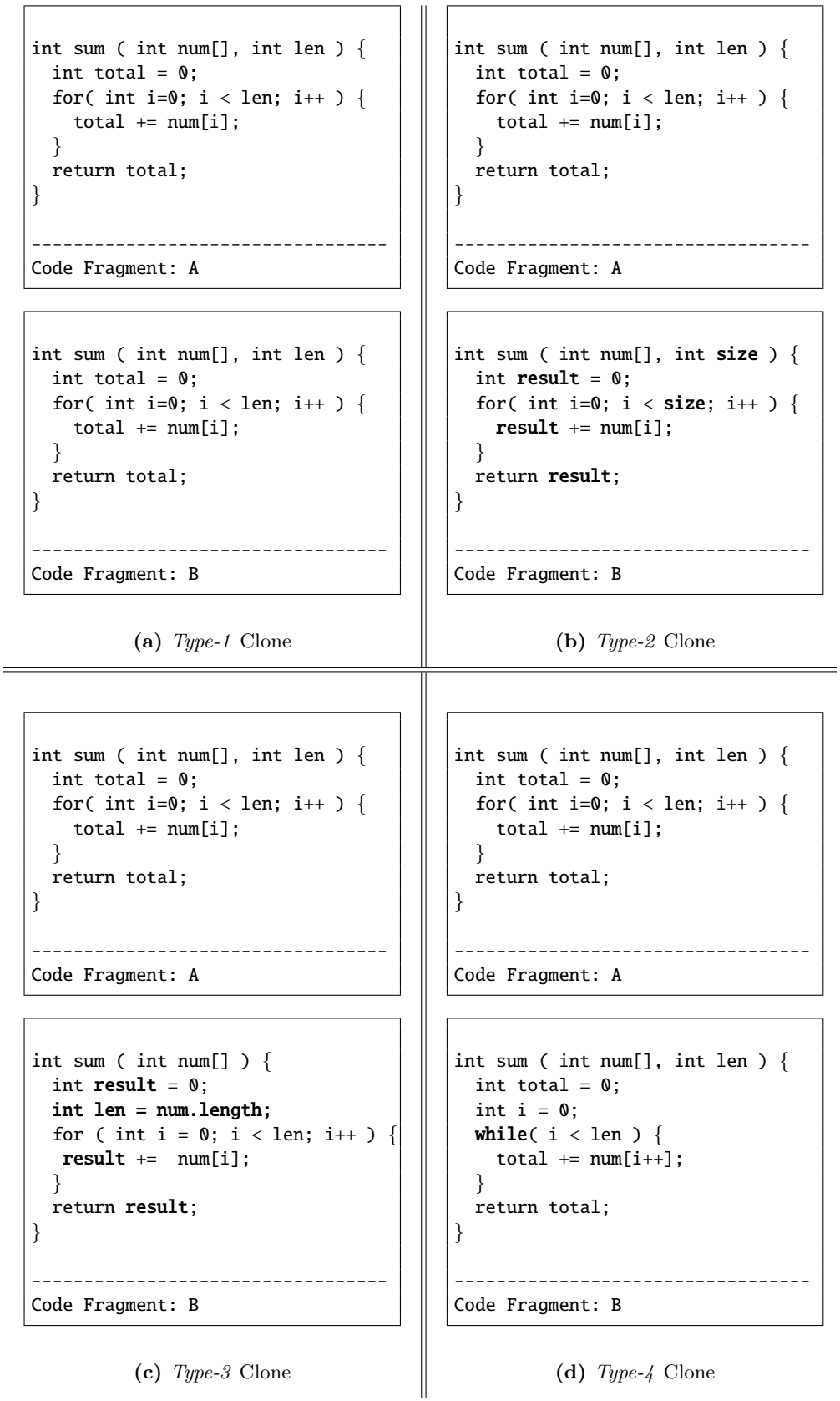
The definition of a code clone is still more or less vague. Usually, it is described as portions of source code or code fragments at different locations in a software project/program that are identical or very similar. The *similarity* here may also be defined in various ways and can refer to textual, structural or semantic aspects of the source code. Selim et al. [160] defined code clones as sets of syntactically or semantically similar code segments residing at different locations in the source code. In the words of Baxter et al. [23], "Clones are segments of code that are similar according to some definition of similarity". Symbolically, a clone fragment can be represented as a tuple with three variables (f, s, l), where f denotes the file location, s denotes start line of the clone fragment and l denotes the length of the fragment starting from s.

There is also no precise minimum size for a code clone. Clone studies define this size differently in terms of number of lines, tokens or AST (Abstract Syntax Tree)/PDG (Program Dependency Graph) nodes with respect to their experimental context [151]. Besides, a pair of clone fragments can either partially overlap to each other or one of them can be located completely within another. Let, $CF_1 = (f, s_1, l_1)$ and $CF_2 = (f, s_2, l_2)$ be two clone fragments. Then the length of the shared region can be measured as: $min(s_1 + l_1, s_2 + l_2) - max(s_1, s_2)$. Given that a negative length means no overlap.

2.2 Clone Types

Types of clones are defined based on the degree of similarity among the clone fragments. The following categorization of clone types has widely been accepted in the literature [101, 143, 146].

Type-1 (Exact Clones): Identical code fragments without considering the variations in white-space and comments (Figure 2.1a).



(a) *Type-1* Clone

(b) *Type-2* Clone

(c) *Type-3* Clone

(d) *Type-4* Clone

Figure 2.1: Types of Code Clone

Type-2 (Renamed/Parameterized Clones) : Code fragments that are structurally/syntactically similar but may contain variations in identifiers, literals, types, layouts and comments (Figure 2.1b).

Type-3 (Gapped Clones): Code fragments with modifications in addition to those defined for *Type-2* clones, such as insertion, deletion or modification of statements (Figure 2.1c). *Type-2* and *Type-3* clones are collectively termed as *near-miss* clones in literature [146].

Type-4 (Semantic Clones): Code fragments with the same functionality with or without being textually similar(Figure 2.1d).

Although, the definition of *Type-1* and *Type-2* clones are somewhat specific, the definition of *Type-3* clones still remains vague [101]. This is because the definition does not precisely indicate how much differences in terms of addition, modification, or deletion of statements are allowed in code segments to be regarded as a *Type-3* clone. For *Type-3* clones, it may require setting up a boundary of acceptable differences (between two *Type-3* clone fragments) that occur for such modifications in terms of line numbers, token numbers, amount of text and so on. Researchers commonly consider code segments as *Type-3* clones when the difference in the code components (lines, tokens etc.) remains below a self-defined dissimilarity threshold [12, 113, 147, 184]. However, a consensus on an appropriate threshold value for the definition of a *Type-3* clone has yet to be established [101].

2.3 Clone Granularity

The clone type definitions stated above are based on the notion of an arbitrary code segment. They do not define how much of contiguous code can be considered a clone. Contiguous portions of source code at different levels of granularity are used in the literature. The following are the most commonly used granularities, which yield the notion of source code clones:

File clone: Two source files containing some good amount of similar source code.

Class clone: Two classes of source code written in an object-oriented language when the classes have identical or near-identical code.

Function clone: Two functions are considered as clones when their bodies contain similar code to each other.

Block clone: When the contents of two blocks of code (usually a collection of statements performing a unit of work, marked bounded by some boundary marking character e.g., opening and closing braces, brackets or indentation, or the like) are similar enough.

Arbitrary statements clone: When two groups of statements at arbitrary regions of the source file are found to be similar enough, they are also regarded as clones (CCFinder [88] detects such clones).

Structural clone: Structural clones denote the design level similarities among the patterns of interrelated classes [19] emerging from design and analysis space at the architecture level.

Model based clone: Applications in some domain (e.g.: embedded systems, automobile/aviation design) are developed from a model designed with a domain specific modeling language. Unexpected overlaps and duplications in such models [44] are termed as model based clones.

2.4 Cloning Relations and Grouping

A clone relationship exists between two code fragments that are clones of each other according to some definition of similarity as stated earlier. Such a relationship is *reflexive* (i.e., if code segment X is a clone of Y, then the reverse is also true). The *transitive* relationship also exists for *Type-1* and *Type-2* clones (i.e., if code segment X is a clone of Y, and Y is a clone of Z, then X is also a clone of Z). However, such a *transitive* property may not exist for *Type-3* clones [24]. The definition also imply that there exists a subset relationship among the *Type-1*, *Type-2*, and *Type-3* clones. Mathematically,

$$Type - i \subseteq Type - j \quad (for\ i \in \{1, 2\}\ and\ j = i + 1)\ [96]. \tag{2.1}$$

Researchers present clones in some groups that represent cloning relations among the related fragments. This grouping leads to a better understanding of the cloning status of the system. The following clone grouping is predominant in code clone literature.

Clone Pair: Two code fragments similar to each other form a *Clone Pair*.

Clone Class: A *Clone Class* is a group of clone fragments which are similar to each other. Therefore, a *Clone Class* may have two or more code fragments where each pair of code fragments forms a *Clone Pair*.

Super Clone: The set of *Clone Class*es that belong to the same source code location form a *Super Clone*, also known as *Clone Class Family*. Alternately, *Super Clone* is the aggregation of the *Clone Class*es that cross-cut in the same source code region, e.g.: file, directory, function, class or package.

2.5 Reasons for clones in Software

Most software systems usually contain a significant amount of cloned code and the amount of cloning varies depending on the domain and origin of the software system [146]. However, clones do not occur in software systems by themselves, rather they evolve from various development activities performed by the developer during the software's development and maintenance phase. There are a number of factors that might force or influence the developers and/or maintenance engineers making cloned code in the system [12, 23, 93, 99, 143]. Kapser et al. [90] identified a set of eight cloning patterns that explain the motivations of cloning with

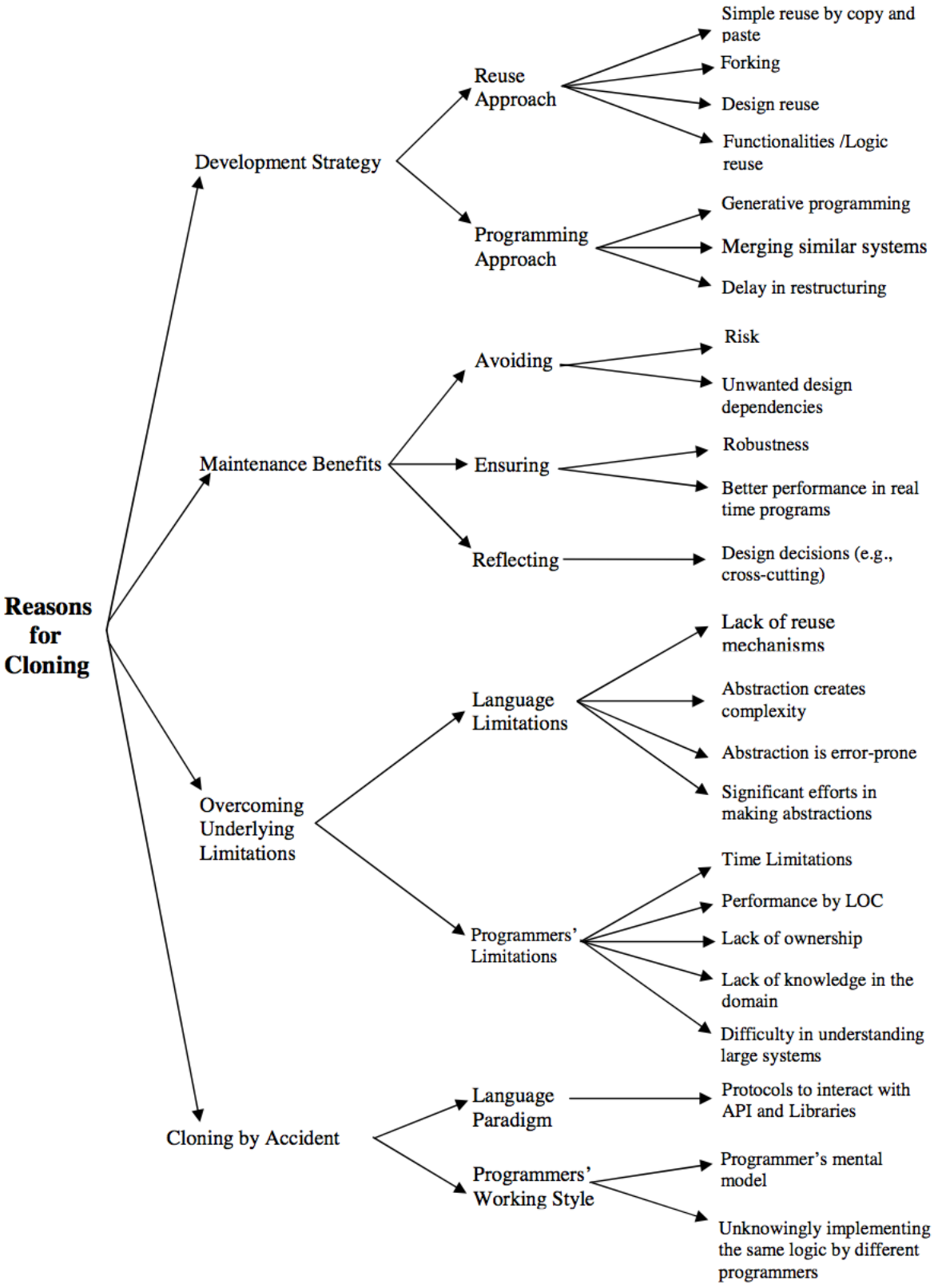


Figure 2.2: Reasons for cloning (adapted from [146])

corresponding advantages and disadvantages. Toomim et al. [170] identified a set of cases that makes the abstraction costly and leads programmers to leave the cloned code instead. A comprehensive list of factors (as shown in Figure 2.2) that introduce clones in software can be found in the survey by Roy and Cordy [146], where reasons for cloning are categorized as the following four groups:

Cloning by Accidents

Software developers may repeat common solution patterns for solving similar kinds of problems. Clones can also be created by developers because of implementing the same logic while working independently or following particular development restriction, e.g., coding under a programming language protocol or using a particular set of APIs.

Development Strategy

Different reuse and programming approaches may introduce clones in software systems. Clones can be created due to reuse of existing code, design, logic and functionality in the system. Developer's copy-paste programming practices is one of the most common forms of code cloning. Besides, merging of two software systems (of similar functionality) to produce a new one may introduce clones in the final system.

Maintenance Benefits

Developers may create or keep clones intentionally to obtain development and maintenance benefits, such as to cut the development cost off, to avoid the risk of developing new code that may introduce bugs or require additional testing to meet required QoS [37]. Clones are sometime useful in speed-up development process or keeping software architecture clean and understandable.

Overcoming Underlying Limitations

Because of having some limitations on both programming languages and programmer's ability, clones might be introduced in the system [22, 138]. Some programming languages may not have sufficient abstraction mechanism or restriction for code reuse by design or convention. On the other hand, programmers may have a lack of knowledge of the existing system, strict time constraint, lack of code ownership, lack of understanding that may lead to the creation of duplicate code in the system.

2.6 Negative Impacts of clones in Software

Despite of having some notable advantages of clones (or practice of cloning) in software, they may cause various issues in software maintenance phase. Following are a set of common impacts of clones that make software maintenance challenging.

Bug propagation

Developers may duplicate a buggy code fragment unknowingly by copy-paste operation. In doing so, the bug in the original segment may spread through all the pasted segments in the system even if the segments are pasted with minor adaptations. Besides, lack of required consistent renaming of the variables in the pasted code fragment might introduce new bug despite the fact that the original source code being bug free. Therefore, the probability of new bug generation as well as bug propagation may increase significantly in the system because of code duplication [81, 113].

Inconsistent change propagation

When removing bug in a clone fragment or making enhancement to a piece of code, a developer may forget to do the same in the other copies of the fragment, may because s/he is not aware of the presence of those similar fragments elsewhere in the system. Therefore, code clones may be one of the reasons for incomplete bug removal or inconsistent change in the source code.

Difficulties in system modification

Clones can be anywhere in the system codebase and any amount. Maintenance engineers or developers need additional time and effort in understanding the cloning status of the whole system for consistent system modification. Therefore, system modifications or adding new functionalities require more time and become more challenging when there are clones in the system [82, 126].

Software design issues

Cloning may introduce design fault in a software system, impose restrictions in software abstraction or end up with bad inheritance structure. As a consequence, it could impact negatively on the maintainability of the software [131]. Unreasonable code duplication may also yield hidden dependencies among the duplicated parts, which in turn may hinder software re-usability.

Increase maintenance cost

If a cloned code segment is found to be contained a bug, all of the duplicate copies of the segment, if exist, need to be identified and investigated for correcting the bug in question. Identification of similar fragments and bug elimination from those incur additional software maintenance cost. Same problem happens when maintaining or enhancing a piece of code, duplication multiplies the work to be done [126, 131].

Increase code comprehension effort

The presences of duplicate codes in system increase the cognitive effort by the developers to comprehend a large program. Developers or maintenance engineers may have to spend a significant amount of time

analyzing each fragment separately to understand the differences between them [81].

Increase resource requirement

Code duplication necessarily increase the growth rate of system codebase size. Industries like automobile, telecommunication, aviation and so on in general make heavy use of embedded devices. Increased system size might be a concerning issue for them since increased code size may induce costly hardware upgrade with a software upgrade. Johnson [82] described the overall effect of cloning as a form of software aging, where small changes in the architectural level become very difficult to implement in the code level.

2.7 Clone Detection

2.7.1 Anatomy of code clone detection

Naively, a code clone detector should compare every possible fragment with every other fragment to identify the level of similarity. The similarity level needs to be at least up to a pre-defined value to accept the comparing fragments as clone to each other. However, in case of a medium to large scale system, such an exhaustive comparison is computationally expensive. Thus, several measures are used to reduce either the cost of individual comparison or the domain of comparison before performing the actual comparisons. A number of clone detection techniques has been proposed in literature over the past decades. Regardless of the difference in similarity detection mechanism, from a high level perspective, they mostly follow the same end-to-end processing workflow as shown in Figure 2.3. Considering raw source code as input, a typical code clone detector may perform the following steps (usually most of them if not all) to detect clones in the input source code.

Source Pre-processing

This is the very first step of a detection approach where various uninteresting parts (e.g., embedded source code of another language, auto generated source code, etc.) are filtered out from the input source. Then the remaining source code is partitioned into a set of disjoint fragments called source units. These are the largest source fragments that may form direct clone relations with each other. Source units can have different level of granularity, for example, files, classes, functions/methods, beginend blocks, statements, or sequences of source lines. Depending on the comparison technique used in the detection approach, source units may be further subdivided into smaller comparison units represented as lines or even tokens. Comparison units can also be derived from the syntactic structure of the source unit. Approaches like metrics-based does not require this partitioning of source since metrics values can be computed from source units regardless of their granularity.

Source Transformation

In textual detection approach, similarity detection is performed among the comparison units. However, for non-textual approach, the source code of the comparison units is transformed to an appropriate intermediate representation for comparison. Additional normalizing transformations may also be performed by some approach in order to detect superficially different clones. These normalizations can vary from very simple normalizations, such as removal of whitespace and comments [11], to complex normalizations, involving source code transformations [147].

Extraction

Extraction transforms the source code to the form suitable as input to the actual comparison algorithm. Depending on the tool, it typically involves one or more of the following steps. For token-based approaches, each line of the source is divided into tokens according to the lexical rules of the programming language of interest. The tokens of lines or files then form the token sequences to be compared. All whitespace (including line breaks and tabs) and comments between tokens are removed from the token sequences. CCFinder [88] and Dup [11] are the leading tools that use this tokenization approach on the source code. In syntactic approaches, the entire source codebase is parsed to build a parse tree or abstract syntax tree (AST). Then the comparison algorithms look for similar subtrees that are considered as clones [23, 174, 181]. Metrics-based approaches may also use a parse tree representation to find clones based on metrics for subtrees [99, 126]. Some semantics-aware approaches generate program dependency graphs (PDGs) from the source code. To find clones, the techniques then look for isomorphic subgraphs [98, 105].

Normalization

This optional step is intended to eliminate trivial differences in source comparison, such as, differences in whitespace, commenting, formatting, identifier naming, etc. Almost all approaches disregard comments and whitespace, although line-based approaches retain line breaks. However, some metrics-based approaches use formatting and layout as part of their comparison. Identifier normalization is applied in most of the approaches before comparison in order to identify parametric Type-2 clones. In such normalizations, usually one single identifier replaces all the identifiers in the source code. However, Baker [11] uses an order-sensitive indexing scheme to normalize for detection of consistently renamed Type-2 clones. In text-based clone detection approaches, source codes may also pretty printed (white space formatting) to minimize the differences in source layout and spacing. Cordy et al. [39] use an island grammar [132] to generate a separate pretty- printed text file for each potentially cloned source unit. Some other transformations may be applied to change the structure of the code so that minor variants of the same syntactic form can be treated as similar [88, 135, 147].

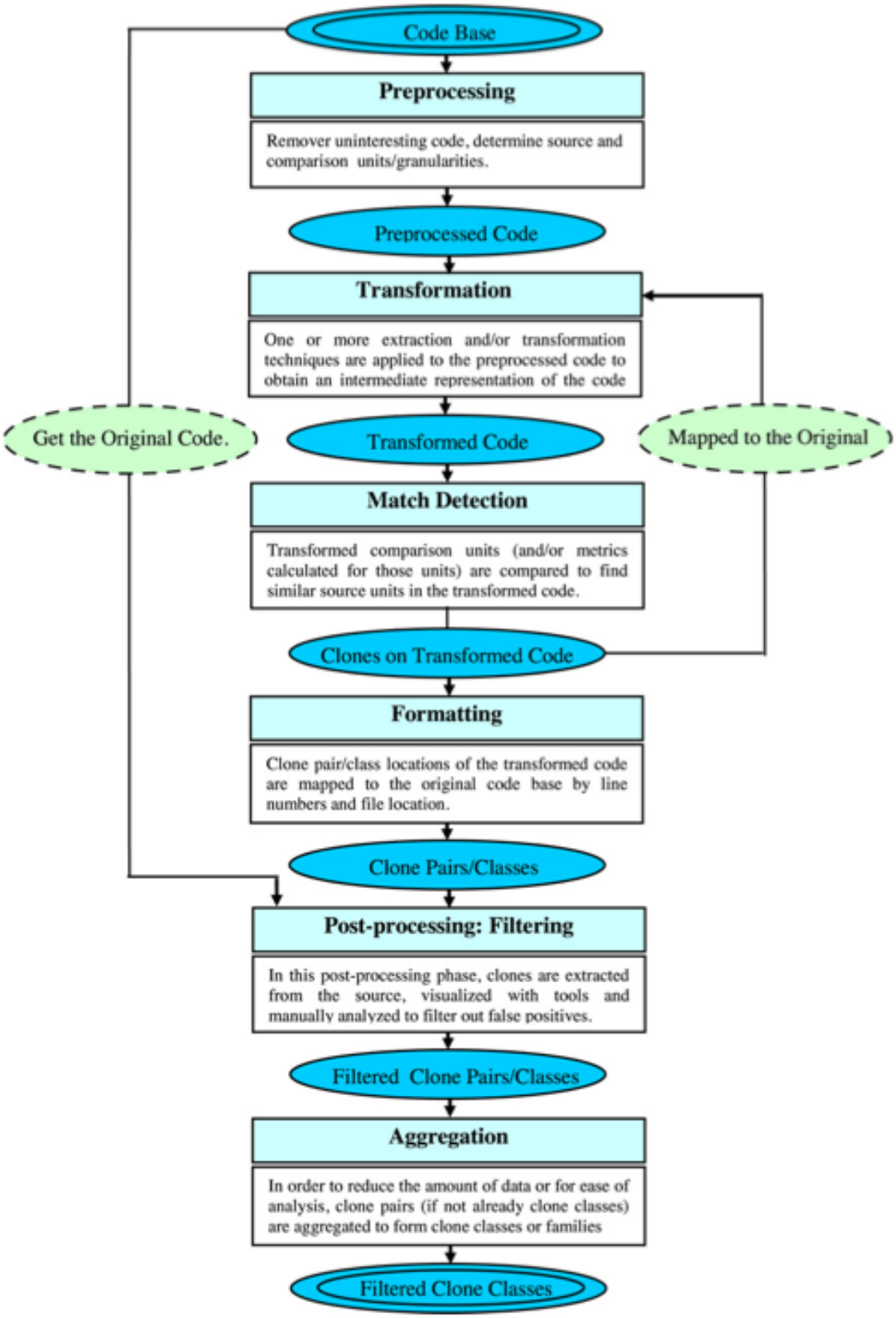


Figure 2.3: Generic clone detection process (adapted from [151])

Similarity detection

A similarity detection algorithm is applied to the transformed comparison units to find potential matches among them. Matches are recorded as a pair of comparison units. Similar units are often grouped together to form a large cluster. All the comparison units of such a cluster are pairwise similar to each other under the defined similarity measure by the algorithm. For detection output, the match records are represented as matched pair or cluster of transformed code corresponds to the comparison units in the match record. In addition to simple normalized text comparison, similarity matching algorithms used frequently in clone detection include suffix-trees [11, 100, 127], dynamic pattern matching (DPM) [47, 99] and hash-value comparison [23, 126].

Code Formatting

In this phase, the transformed code in the matched clone pair or clone cluster found in the detection phase are replaced by corresponding the original source code. Source coordinates (file path, start/end line number, etc.) of each element in the clone cluster (or pair) are also mapped to their positions in the original source files.

Clone Filtering/Post-processing

In this phase, filtered using automated or manual analysis where false positive clones or spurious clones [103] are filtered out by a human expert. Besides, heuristics can often be defined based on length, diversity, frequency, or other characteristics of clones in order to rank or filter out clone candidates automatically [88].

Aggregation

Most of the detection tools return only clone pairs as the detection result. Clones available in pairs may be aggregated into clone classes by those tools in this step. Besides, some overview statistics or analytical result can be obtained by doing subsequent analysis on the detection result, which can be presented as consolidated detection summary.

2.7.2 Existing clone detection techniques and tools

Clone detection is a fundamental part of code clone research since all other clone related research areas are dependent on the clone detection results, i.e., need to know which are the clones in a system. Besides, developers or maintenance engineers need to know the exact location of the clone codes in the software systems; otherwise it would be challenging to resolve the problems caused by the clones. Various clone detection techniques are presented in the literature. Based on the level of analysis applied to the source code, the techniques can roughly be classified into the following six categories:

Textual approaches

In this approach, the target source program is considered as a sequence of lines/strings. Two code fragments are compared with each other to find sequences of same text/strings. Approaches in this category use little or no transformation/normalization on the source code before the comparison, and in most cases raw source code is used directly in the clone detection process. The pioneering text-based clone detection approach was presented by Jonson [81] that uses fingerprints on substrings of the source code to find clones in source code. However, one of the main drawbacks of textual approaches is that they do not work well when the same syntactical structure is represented differently in different places by the developers. To identify similar source files, Manber [122] also used fingerprints, based on subsequences marked by leading keywords.

Ducasse et al. [47] developed a language-independent technique that uses line based transformation along with scatter plot visualization to detect duplication visually. Wettel and Marinescu [179] used an extension of the Ducasse et al.'s approach to find near-miss clones using dot plots. Starting with lines having the same hash value, the algorithm chains together neighboring lines to identify some kinds of Type-3 clones. Lee et al. [112] developed an algorithm named SDD (Similar Data detection), another text based technique that uses $N - neighbor$ distance concept to detect similar code fragments. Marcus and Maletic [124] use latent semantic indexing (LSI) to the source text in order to find high level concept clones (e.g., abstract data types (ADTs)) in the source code. This IR approach limits its comparison to comments and identifiers, returning two code fragments as potential clones or a cluster of potential clones when there is a high level of similarity between their sets of identifiers and comments.

Text based clone detection tool Simian [63] can detect clones in different programming languages. If Simian does not recognize programming language of the source file, then it treats it as a plain text file to find clones. Barbour et al. [18] used the KnuthMorrisPratt string comparison algorithm to update the clone information from the server incrementally in a client server setup. Later on, only relevant clones are retrieved by individual developers. However, because of being string based technique, it fails to detect clones even with minor changes [146].

In general, textual approaches are independent of programming languages and input source code does not need to be syntactically correct. However, without having advanced normalization and filtering of differences in layouts, textual approaches may susceptible to even minor changes in the source code [102].

Lexical approaches

This is also called token-based detection approach, where the entire source system is lexed/parsed/transformed to a sequence of tokens. Detection algorithm then scan this this token sequence for finding duplicated subsequences of tokens and finally, the original code portions representing the duplicated subsequences returned as clones.

CCFinder [88] is one of the leading state of the art token-based techniques developed by Kamiya et al. First, each line of source files is divided into tokens by a lexer and the tokens of all source files are

then concatenated into a single token sequence. The token sequence is then transformed based on the transformation rules of the language of interest aiming at regularization of identifiers and identification of structures. A suffix-tree based sub-string matching algorithm is then used to find the similar sub-sequences on the transformed token sequence. The similar sub-sequence pairs are returned as clone pairs/clone classes. Finally, a mapping is performed to obtain the clone pair/clone class information with respect to the original source code. CCFinder is widely used in clone research community for code clone analysis, code clone management, etc. VisCad [8] is a clone analysis and visualization tools that use output of CCFinder. Livieri et al. [117] developed D-CCFinder, a distributed version of CCFinder for large systems by using 80 workstations in master slave configuration. CCFinderX [87] was used to study code clone genealogies [95] in software at release level.

Baker's Dup [11] uses a lexer to tokenize the source code. A suffix-tree based algorithm is used to compare the token sequences of each line from the source. However, unlike in CCFinder [88] where some transformation rules are applied on the token sequence before matching, here parameterized token matching is applied by a consistent renaming of the identifiers. Basit et al. [21] developed a token based detector RTF that uses a suffix array based flexible tokenization strategy for efficient memory handling and allows the user to tailor token strings for better clone detection. The user can suppress insignificant token classes (e.g., access modifiers of Java) that may cause noise in detection, and there is an option for equating different types of token by assigning same ID on them.

Another state of the art token-based clone detection technique CP-Miner [113]. CCFinder and Dup are in general fragile to statement reordering and code insertion due to sequential analysis of tokens. These limitations are overcome in CP-Miner that uses a frequent subsequence mining technique to determine the similar sequence of tokenized statements. CP-Miner used an extended version of CloSpan [180] to support gap constraints in frequent sub-sequences. This allowed CP-Miner to tolerate one to two statement insertions, deletions, or modifications in copy-pasted code while ignoring arbitrarily long different copy-pasted segment that is unlikely to be copy-pasted. Cordy et al. [39] also developed a token and line-based technique to detect near-miss clones HTML web pages where an island grammar is used to identify and extract all structural fragments and then Unix *diff* algorithm to assess similarity among the fragments. FCFinder is a new token based tool developed by Sasaki et al. [158] to detect file clones using hashing. The study detected 68% of the FreeBSD Ports collection as file clones.

The lexical approaches are in general more robust over minor code changes such as code spacing, renaming and formatting than textual techniques [151]. These techniques even can operate on incomplete or syntactically incorrect program due to the use of lexical analysis. Clones found by token-based techniques may overlap different syntactic units because the syntax is not taken into account. However, by identifying the block delimiters or using lightweight syntactic analysis such as island parsing [132] in either pre-processing [39, 54] or post-processing [68] step, clones corresponding to syntactic blocks can be identified by a token-based detector.

PDG-based approaches

This is also called semantics-aware approaches that use static program analysis to provide more precise information than simply measure syntactic similarity. Source code of a program in this approach is represented as a program dependency graph (PDG). The nodes of this graph represent expressions and statements while the edges represent control and data dependencies. Detection algorithm then looks for isomorphic subgraphs (for which only approximate efficient algorithms exist) [98, 105, 115] to detect clones in the code.

Clone detection tools PGD-DUP [98] proposed by Komondoor and Horwitz is one of the leading PDG based approach that looks for isomorphic PDG subgraphs using (backward) program slicing [178]. Their approach groups identified clones together while preserving the semantics of the original code [97] that enables support of clone refactoring. Krinke uses an iterative approach (k-length patch matching) in Duplix [105] for detecting maximally similar subgraphs in the PDG. In a recent study, Gabel et al. [52] maps PDG subgraphs to corresponding structured syntax for finding semantic clones in source code. Liu et al. [115] developed GPLAG, a software plagiarism detection based on PDG. A statistical lossy lter is also proposed here to prune the plagiarism search space to make GPLAG scalable to large programs.

Scorpio [67] is a recent PDG based tool developed by Higo and Kusumoto that applies two-way slicing to detect clones. The tool is based on the number of PDG specializations for Java language and heuristics to speed up the overall detection process and currently works for Java language only. Higo et al. also proposed a PDG-based incremental clone detection technique in [69], where PDGs are generated from the analysis of control and data dependencies in the program code. The PDGs are preserved in the database, and clone detection is performed by approximate comparison of PDGs.

The primary limitations of PDG-based approaches are similar to those of tree-based approaches. Besides, this approach is programming language dependent, requires syntactically correct program and has high time complexity. Although, PDG-based approaches can partly handle statement reordering, insertion, deletion and non-contiguous code, they are not much scalable to large size programs [95].

Tree-based approaches

Tree based approaches use a parser to convert source programs into parse trees or abstract syntax trees (ASTs) where clones can be found by identifying similar subtrees. Variable names, literal values and other tokens in the source can be abstracted in the tree representation that allows sophisticated detection of clones.

Yang [181] proposed one of the early approaches in this category that can find the syntactic differences between two versions of the same program. The technique builds a variant of a parse tree for both versions and looks for longest common subsequence between both the trees. Baxter et al.'s CloneDR [23] is another pioneering tree matching approach that detects exact and near-miss clones using hashing and dynamic programming. It uses a compiler to generate annotated parse trees from the input source. Subtrees are then hashed into buckets. A tree matching algorithm compares subtrees with each other within the same bucket to identify similar subtree and reports the matched subtrees as clones. The hashing drastically reduces the

number of necessary tree comparisons and enables parameterized matching to detect gapped clones and clones with some reordered statements.

CloneDR has been adapted by the AST-based clone detector SimScan [29] that applies subtree comparison on the parsed source code. Here, ANTLR parser is used to parse the source code. Tool ccdiml [142] is another variant of CloneDR with some differences like the avoidance of the similarity metric, the handling of sequences and the hashing. Unlike CloneDR, ccdiml represents ASTs in IML(Intermediate Language) [104] rather than directly using those in the comparison. Both SimScan and ccdiml have been used in studying the evolution of clones in programs written in Java and C language [168].

Deckard [78] and ClemanX [136] are novel approach based on computing characteristic vectors from the AST to approximate the structure of ASTs in a Euclidean space. They use locality sensitive hashing (LSH) [41] to cluster similar vectors using the Euclidean distance metric and thus finds corresponding clones. Saebjornsen et al. [153] also used the same set of techniques to detect clones in assembly code. Deckard has been used in detecting behaviorally similar code [85] and assessing the impact of clones in software defects [140].

Recent tree-based approaches use alternative tree representations to avoid high computational cost for full subtree comparison. Asta [49] works on the phenomenon of structural abstraction of arbitrary subtrees of an AST and can detect exact and near-miss clones with gaps. Chilowicz et al. [34] used syntax tree fingerprinting to detect exact clones only. Koschke et al.'s [50, 103] approach constructs a suffix tree using serialized AST node sequences build from AST subtrees. This idea allows finding syntactic clones at the speed of token-based techniques. Wahler et al. [176] proposed a tree based approach based on frequent itemset mining applied on XML representation of source code. Anti-unification is used in [31] to discover common sub-expressions in source code represented as a tree. CloneDigger [30] is a language independent tool in which anti-unification is applied to XML representation of source code. Biegel and Diehl [27, 28] developed JCCD, a flexible and customizable AST based clone detector using pipelines. JCCD API can parallelize the detection process using multiple cores that significantly improves the total detection time.

The tree-based techniques suffer from the overhead of invoking a parser to generate parse tree. Again, due to the parser dependency, these approaches are language dependent and require syntactically correct program. In general, parser based syntax comparison techniques are very accurate in detecting clones that have similar syntactic structure, but suffer from large execution times when analyzing a large source code base. Since tree-based representations of programs are insensitive to formatting, and comparatively less sensitive to programming style, tree based techniques can detect some clones that are not usually detected by text-based or token-based methods.

Metric-based approaches

Techniques in this category detect similarity in code by comparing various software metrics vectors instead of comparing the code directly. The idea is similar code fragments should yield similar values for metrics vectors

(e.g., token frequencies, hash fingerprint, cyclomatic complexity, fan- in, fan-out). Metrics are calculated for one or more syntactic units such as a class, a function, or a method or even statement and then the metrics values are compared to find clones over these syntactic units. The source code might be parsed to its AST/PDG representation for calculating such metrics.

A widely used tool CLAN [126] developed by Mayrand et al. is one of the early approaches that compare AST based metrics to identify function clones. Metrics are calculated from names, layout, expressions, and control flow of functions. Kontogiannis et al. [99] have proposed two different ways of detecting clones. One approach uses direct comparison of metrics values to identify similarity at the granularity of beginend blocks. The second approach is applying dynamic programming (DP) on source code lines using minimum edit distance. The assumption is that duplicate code fragments originated from cut-and-paste activities are likely to have small pairwise edit distance.

Li and Sun [114] studied a novel approach by viewing the source code clones in the metric space. Here, the measured distance between members across same metric space reflects similarity between code fragments. Davey et al. [42] detects exact, parameterized, and near-miss clones by using neural networks based search technique on extracted features from the source code blocks. Lavoie et al. [110] proposed a technique based on graphics processing unit (GPU) algorithms to compute many instances of the longest common subsequence (LCS) problem using classic dynamic pattern matching (DPM) on a generic GPU architecture. The study recommends clone detection techniques using string matching with suffix trees could take advantage of the GPU algorithm.

Metrics-based approaches have been applied in finding duplicate web pages and clones in web documents [32, 45]. They have also been used successfully in clone analysis [14], clone visualization [79], clone evolution [5, 6], etc.

Hybrids approaches

In addition to the above, there are also clone detection techniques that use a combination of syntactic and semantic characteristics.

Balazinska et al. [15] propose a hybrid approach of detection of method/function clones using characterization metrics and dynamic pattern matching (DPM). Characteristic metric values are computed for each of the method bodies and compared to find a cluster of similar methods. In the approach proposed by Koschke et al. [103], the AST nodes are serialized in preorder traversal, a suffix tree is created for these serialized AST nodes. The approach compares the tokens of the AST nodes instead of the AST nodes using a suffix tree-based algorithm in linear time and space.

Tairas and Gray [166] developed a function-level clone detector for Microsoft's new Phoenix framework that uses AST and suffix trees. AST nodes are used to generate a suffix tree, which allows analysis on the nodes to be performed in linear time and space. This approach can find exact and a subset (with identifier renaming, not type changes) of parameterized function clones. A novel hybrid approach is presented by Jiang

et al. [78] that combines tree and metric based approach. Here, a set of characteristic vectors is computed from ASTs corresponding to the source code. A locality sensitive hashing (LSH) based similarity detection is used to cluster similar vectors and thus, code clones.

Sutton et al. [163] applied an evolutionary algorithm to find clones in large code-bases. Detection algorithm looks for longest common subsequence (represents clustering of similar code fragments) on a search space of variable sized metric vector derived from the source code. Grant and Cordy [61] introduced a technique using an existing information retrieval (IR) method, namely independent component analysis (ICA) to analyze some characteristics vectors representing methods in software. The distance between any two vectors measures code similarity between corresponding methods. Maeda [121] introduced a technique based on PALEX [120] source code representation. The technique is language independent and uses a suffix tree for comparison. Using frequent itemset mining, Basit and Jarzabek developed a hybrid clone detection tool Clone Miner [20] that is a variant of token based detector RTF [21].

NICAD [147] is a text-based hybrid clone detection tool which can detect *Type-3* clones efficiently. It is based on a two-stage process, namely extracting and normalizing source code fragments using pretty printing and source transformation, followed by code comparison using longest common subsequences. Uddin et al. [172, 171] proposed a similar approach *SimCad* that uses a locality sensitive hashing (LSH) technique named *simhash* [33] to generate hash fingerprint of source code. Similarity in code is measured by the pairwise hamming distance of simhash fingerprints correspond to the source code. The approach includes a two-level indexing scheme to organize the hash values for fast similarity detection. A popular data clustering algorithm DBSCAN [48] is used to cluster similar simhash values into groups based on their pairwise distance. Each of the groups having two or more members represents a clone group in the source code.

A hybrid incremental index-based clone detector proposed by Hummel et al. [73] takes input in the form of token sequences and build a hash based index on a group of source lines. Source regions having maximal continuous match of the corresponding indexed hash values are reported as clones. The index allows updating on addition, deletion or modification of source.

Hybrid approaches usually exhibit fast execution time and wider detection coverage of clone types than others.

2.8 Clone Visualization

The result of clone detection is essentially a large volume of textual information about the clones such as the file name, line number, starting position, ending position, etc. Besides, depending on the detection techniques used, the returned clones also differ in several contexts such as types of clones, degree of similarity, granularity and size. Comprehending the clones or understanding the cloning status of the system by manually exploring the detection result in textual form is rather difficult and error prone. For the proper use of the detected clones, especially for clone management, the aid of a sound visualization tool is crucial. A number of

visualization techniques, filtering mechanisms and support environments are proposed in the literature.

Clone visualization tools like Gemini [173], VisCad [8], Aries [65] and CLICS [89] read the output of CCFinder. These tools filter uninteresting clones and navigate to the clones having features the users are interested in. One of the widely accepted formats for displaying code cloning relations is scatter-plot [8, 36, 38, 117]. Scatter-plots are useful to select and view clones, as well as zoom in on regions of the plot. However, the scalability issue limits its applicability to visualize clones of many software units. Higo [66] proposed an enhanced scatter plot to overcome the scalability issue and understand the state of the clones over different versions of software.

Johnson [83] has applied Hasse diagrams for visualizing cloning relationships between files. The diagram is consisted of nodes and edges, where the copied source text and the source files are shown as nodes and the relation between clones are shown as edges. A HTML web page based approach proposed by Cordy et al. [39] provides interactive presentation of clones with an overview of the clone classes. Although, the approach offers quick navigation of clones among the clone classes through hyperlinks, it is unable reveal the high level cloning relations in the system. Rieger et al. [144] incorporated polymetric views [109] in visualizing clones that allow one to investigate the clones of a system at different levels of abstraction.

Using visualization of the clone relations in the architecture level, Jiang et al. [80] introduced the concept of coupling and cohesion to clone code. The framework is useful in investigating and managing cloning activities within and across subsystems. Adar and Kim [3] developed SoftGUESS that supports exploration and visualization of code clones. It supports the analysis of code-clones over single and multiple version of software in the context of system dependencies, authorship information, package structures and other system features.

Jiang and Hassan [79] developed *Clone System Hierarchical Graph* using a data mining technique framework that mines clone information from the clone candidates produced by CCFinder. An interactive graph is used to select nodes to highlight how the clones are scattered at different levels of the source directory. Tairas et al. developed an Eclipse plug-in CeDAR (Clone Detection, Analysis and Refactoring) [165] by extending the *AspectJ Development Tool*[1] visualizer to display the results of CloneDR. In this plug-in, one clone instance displays the properties of all the clones in the clone group, which greatly helps in clone refactoring.

2.9 Clone Evolution

Successful software systems in practice evolve over time. Evolution of software refers to the process of its initial development, and then repeatedly updating it in its maintenance phase. Reasons for which software needs to be updated includes but no limited to bug fixing, increase reliability, performance improvement, and above all, addition of new features to meet the demands of the users. Therefore, during the evolution of software, cloned codes inside the codebase also experience evolution from version to version. The study

[1] http://www.eclipse.org/ajdt

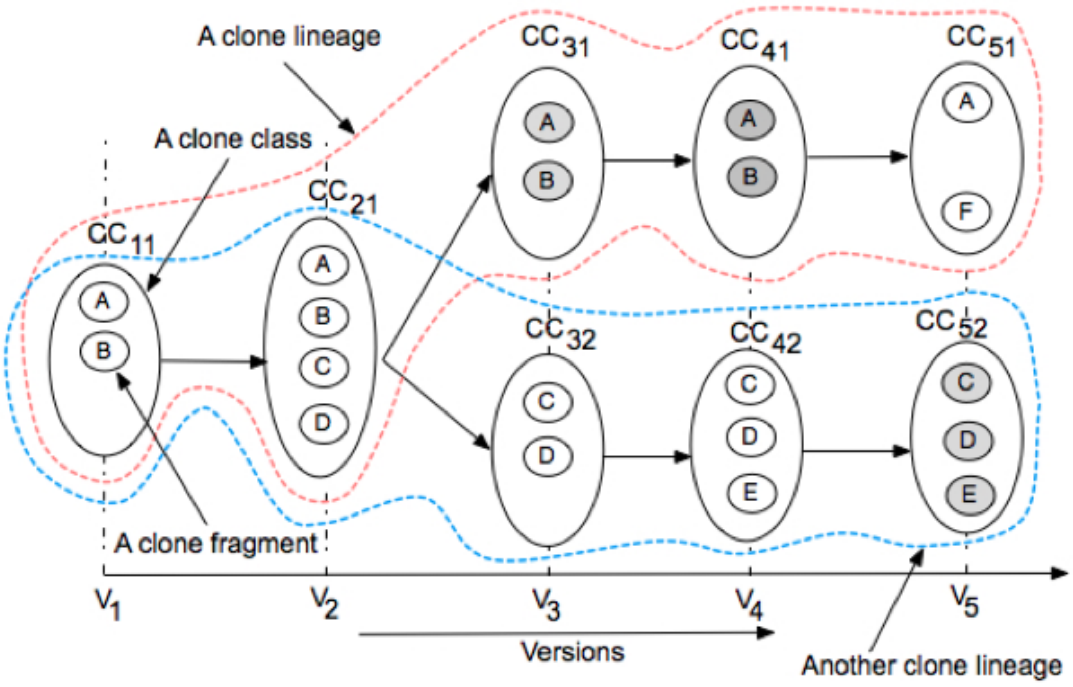


Figure 2.4: An example of a code clone genealogy (adapted from [7])

of evolution of clones is thus concerned with understanding the changes of clones, causes and the effects of those changes. Understanding this evolution can help software developers to reason about cloning, improve software processes for managing clones, support developers in copy-paste programming practices and allow maintenance engineers to make expert decisions about removing or refactoring clones.

The study of clone evolution receives much attention in the software clone research community. Several studies have been conducted to understanding the overall evolution of clones in software [55, 116, 185]. The relation of clone evolution with software faults has been investigated in [17, 59, 159]. Studies in [57, 71, 106, 107, 128] explored the stability of cloned code throughout the lifecycle of the software. Pate et al. [137] conduct a comprehensive study that provides a systematic literature review of the current state of knowledge about code clone evolution.

2.9.1 Clone Genealogy

A clone genealogy defines how a clone group evolves during the evolution of a software system. The term *clone genealogy* is first coined by Kim et al. [94] that refers to a non-empty set of *clone lineage* originating from the same clone group. A *clone lineage* is a sequence of clone groups evolving over a series of versions of a software system. Thus, a clone genealogy represents the evolution history of a clone group over subsequent versions of software's end-to-end lifecycle. Figure 2.4 shows an example of a clone genealogy consisting two clone lineages.

Clone Change Patterns

In clone genealogy, the evolution of individual clone group is characterized with some change patterns. Recent studies [95, 154] proposed following categories of clone change patterns.

- *Same:* A clone group in a version of software moves to the next version without any change in number or contents of the clone fragments.
- *Add:* A clone group gets one or more new clone fragments in the next version.
- *Delete/Subtract:* One or more existing clone fragments get disappeared from a clone group in the next version.
- *Consistent Change(CC):* Exactly same modifications are applied to each the clone fragments in a clone group in the next version.
- *Inconsistent Change(IC):* At least one clone fragment in a clone group modified differently than other fragments in the clone group's next version.

Types of Genealogies

Clone genealogies can be categorized into the following groups based on the change patterns mentioned previously:

- *Static Genealogy(SG):* A genealogy where the clone fragments in a clone group do not experience any change throughout the evolution.
- *Consistently Changed Genealogies(CCG):* A non-static genealogy that exhibits one or more consistent change pattern(s) and no inconsistent change pattern during the evolution.
- *Inconsistently Changed Genealogies:* A non-static genealogy that exhibits at least one inconsistent change pattern anywhere during the evolution. This is further categorized as the following two:
  - *Independent Evolution(IE):* A genealogy where the clone fragments of a clone group, once changed inconsistently, evolve independently across versions. This may cause branching with multiple lineages in a genealogy.
  - *Late Propagation(LP):* A genealogy where one or more clone fragments in a clone group disappear from a clone group in the next version (exhibiting inconsistent change pattern), but re-appear in the same clone group in a later version.
- *Dead Genealogy(DG):* A genealogy that does not survive to the final version of software in the evolution.
- *Alive Genealogy(AG):* A genealogy that survives to the final version of software in the evolution.

Genealogy Construction/Extraction

Construction of clone genealogies involves mapping of clones across subsequent versions of a program. There are four basic approaches as follows to constructing clone genealogies (or similarly, fragment traces).

- In the first approach [13, 95, 154], clone detection is performed for each of the versions of the target program. Clone genealogies are then constructed by mapping the clones between consecutive versions. This approach requires the use of a heuristic to determine whether a clone in version i is indeed the modified version of a clone in the version $i-1$.
- In the second approach [9, 106, 168], clones are detected in the first version of interest and then they are tracked through the subsequent versions using change information mined from a repository or IDE. A downside of this approach is that clones introduced in versions following the initial version are ignored.
- The third approach [26] combines the first and the second approaches. At first, clones are detected in all source code versions of interest. Then the detected clones are transformed to clone region descriptors (CRDs), which are traced across versions.
- Finally, in the fourth approach, clone fragments are mapped during clone detection using the changed information between versions [55, 58]. The mapping is performed at the code fragment level, allowing analysis of code fragment evolution patterns as well as analysis of clone group evolution patterns.

2.9.2 Clone Evolution Visualization

Similar to code clone visualization, evolutions of clone can be better understood through evolution visualization tools. Such tools usually can provide a high level overview of evolution as well as the flexibility to focus on a preferred area of the software with desired level of details. Following are some techniques and tools that have been proposed for visualizing properties of clone evolution including the genealogy model.

SoftGuess [3] is a system for clone evolution exploration developed by Adar and Kim that supports three different views. The evolution is modeled using a the graph exploration system named GUESS [2]. In SoftGuess, the genealogy browser offers a simple visualization of clone evolution. Here, nodes represent clones are arranged from left to right, and the clones that belong to the same class are arranged vertically in the same position. The dependency graph view shows how the nodes (package, class or method) within a version are evolved from other nodes and how they evolve in the next version. In addition, SoftGUESS also supports charting and filtering mechanisms based on a SQL type query language Gython.

Harder and Go de [62] developed CYCLONE, a multi-perspective tool for clone evolution analysis. It offers five different views to analyze clone data stored in a RCF file. RCF is a binary format to encode clone data including the evolutionary characteristics. The evolution view in CYCLONE visualizes clone genealogies using simple rectangles and circles that represent software entities. Each row represents a version of the software, and each circle arranged in a set of rows represents a clone fragments. A rectangle bounds the

clone fragments that belong to the same clone class. Finally, lines represent the evolution of a clone fragments. In addition, the view employs colors to distinguish types and the changes of the clones. VisCad [8] offers a similar visualization with additional flexibility of metric based genealogy filtering.

Saha et al. [156] proposed a visualization approach for clone evolution using the popular scatter plot. In their approach, scatter plots show the clone pairs associated within a pair of software unit (file, directory or package). Clone pairs are rendered with different colors based on associated clone genealogy type. User can select a clone pair and see the associated genealogy in a genealogy browser. The proposal facilitates developers or maintenance engineers to identify evolutionary change patterns of the clone classes in a version and call genealogy browser to dig deeper down the evolution. However, the approach does not provide overall characteristics of the genealogies. Besides, due to the large number of clone pairs, selection and useful pattern identification in such a scatter plot can be difficult.

2.10 Clone Management

In order to facilitate the software maintenance activities, clones in software need to be managed efficiently. The goal of clone management is to take the advantages of cloning as much as possible while overcoming the threats posed by them. Clone management covers a broad set of activities including clone detection, tracking of clone, clone evolution, and clone refactoring, etc. Various clone management approaches that proposed in the literature can be categorized as follows.

Preventive Clone Management:

The objective of preventive clone management is to prevent the creation of new clones in the system rather than detecting and removing them afterwards. Use of this approach suits best for a new system (from the very beginning of its development) rather than for an already developed system.

Corrective Clone Management:

Corrective clone management aims for removal/refactorization of existing clones from the system. The suspicious clones are refactored to reduce potential sources of errors resulting from duplicated code and increase the understandability of software systems. Therefore, this approach may be effective for the software systems where clones were not maintained from the beginning.

Compensative Clone management:

It might not be worthy or possible to refactor all the clones in a system. Compensatory clone management deals with applying techniques (such as annotation, documentation) to minimize the negative impacts of clones in the system staying for some valid reasons. This approach tries to facilitate a consistent evolution of the clone groups, for example, use of simultaneous editing while changing any fragment in a clone group.

Chapter 3

On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems

3.1 Introduction

Cloning is a common phenomenon found in almost all kinds of software systems. The larger the system, the more people involved in its development and the more parts developed by different teams result in an increased possibility of having evolving clone code. Several studies suggest that as much as 7-23% of code of a software system is cloned code [11, 126]. The presence of clones may lead to unresolved bug and/or maintenance related problems by increasing the risk of update anomalies [146].

Existing popular techniques [23, 88, 113] have several deficiencies, such as not supporting the detection of Type-3 near-miss clones where lines could be modified, added and/or deleted in the copied fragments, and not scaling adequately to handle clone detection in large systems [78]. One significant problem for clone detection on large corpora is the performance of querying and retrieving possible clones. One way to improve this performance is to use near constant time techniques of querying a dataset for similar entities. For exact-duplicate matching, a simple match of fingerprints suffices and a common hash function is suitable for that, but it does not work that well for inexact matches. If one wants partial matches and yet wants a constant time operation, the hash function has to map similar documents close together. Manku et al. [123] showed that Charikar's simhash [33] is a very efficient hashing technique for developing a near-duplicate detection system for a multi-billion page repository. Simhash has been used successfully in different areas of research, such as text retrieval, web mining and so on [60, 64, 139]. However, as of yet, no clone detection study has been conducted using simhash. The objective of this study is to see how effective simhash is for software clone detection, especially in detecting Type-3 near-miss clones from large scale software systems. Thus, the research questions we would like to answer in this study are:

1) Is simhash feasible to be used in code clone detection, especially for Type-3 clones in large codebase?

2) Does a simhash-based technique yield faster detection of clones in large codebases?

3) What are the possible problems/limitations of using simhash in clone detection and how can we overcome those limitations?

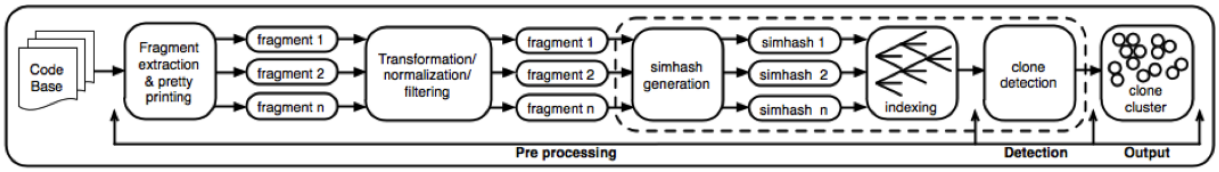


Figure 3.1: Clone detection process in *SimCad*

A *simhash* based clone detection approach is designed in this study. The performance evaluation of the approach will in turn be a measure of the effectiveness of *simhash* in this area. To evaluate that effectiveness, an existing clone detection tool NiCad [147] is chosen for comparative performance analysis. NiCad uses the UNIX *diff*[1] algorithm to measure code similarity. NiCad is a state of the art clone detection tool that returns structurally significant clone fragments (e.g., functions or blocks). It has powerful features for source code pre-processing and normalization along with context sensitive transformations. Diff based comparison along with these features enables NiCad to detect both exact (Type-1) and near-miss (Type-2 and Type-3) clones with high precision and recall [149]. NiCad's diff based detection engine is replaced with the *simhash* based approach developed in this study (marked as a dashed rectangle in Figure 3.1. The new approach named *SimCad* includes an indexing strategy and uses a data clustering algorithm to detect the clones. Using NiCad as the benchmark, a comparative performance evaluation is done to measure the effectiveness of *SimCad* for large scale clone detection in terms of detection time and the number of cloned fragments detected. This evaluation provides some insight into the effectiveness of the *simhash* based technique for clone detection and helps answer the research questions as stated above. Third party clone detectors were not used intentionally since the primary objective of this study is to evaluate the feasibility of *simhash* the detection of structural (such as functions or blocks) clones. The study compares *SimCad* with NiCad, not only because NiCad gives high precision [150, 145] and recall [148, 164, 149] but also because both NiCad and SimCad use the same pre-processed structural code fragments in the comparison phase and thus comparing the results of using the simhash technique is straightforward, which is one of the major objectives of the study.

This paper presents a novel approach for structural clone detection based on fingerprinting of source code using a hash technique called simhash. A multi-level indexing scheme is also proposed to organize the pre-processed codebase on which the clone detection algorithm is applied. The indexing scheme speeds up the potential clone search and allows the approach to be scalable by maintaining the indices in persistent storage (e.g., a database). This makes the proposed approach capable of being used by an incremental clone detection tool or a real-time code search engine. The experimental analysis shows the effectiveness of simhash in clone detection and how it enables faster near-miss clone detection in large corpora.

The rest of this paper is organized as follows. Section 3.2 covers some general terms and definitions of clones. Section 3.3 describes the proposed technique. Section 3.4 provides the complexity analysis of the

[1]http://en.wikipedia.org/wiki/Diff (accessed on August 3, 2013)

detection process. Section 3.5 presents the implementation and experimental results analysis. Section 3.6 covers some related work on the application of fingerprint based similarity and finally, Section 3.7 concludes the study.

3.2 General Terms and Definitions

3.2.1 Software Clone and Clone Detection

There is as yet no universal definition for a software clone. It is usually described as portions of source code or code fragments at different locations in a software project/program that are identical or very similar. Being 'similar' may also be defined in various ways and can refer to textual, structural or semantic aspects of the source code. Selim et al. [160] defined code clones as sets of syntactically or semantically similar code segments residing at different locations in the source code. In the words of Baxter et al. [23]: Clones are segments of code that are similar according to some definition of similarity.

There is also no precise minimum size for a code clone. Clone studies define this size differently in terms of either number of lines, tokens or AST/PDG nodes with respect to their experimental context [151]. Thus, given the various definitions of a software clone, defining and measuring code similarity is an important aspect of any clone detection technique. That is, a key issue is the manifestation of source code similarity and the success of such a tool mainly involves how efficiency and accurately it can perform the task of identifying similar code fragments based on its self-defined similarity measurement.

3.2.2 Clone Types and Clone Groups

In recent literature, clones are broadly classified as one of the following four types.

Type-1 (Exact Clones): Code fragments, which are identical without considering the variations in white space and comments.

Type-2 (Renamed/Parameterized Clone): Code fragments, which are structurally/syntactically similar but may contain variations in identifiers, literals, types, layouts and comments.

Type-3 (Gapped Clones): Code fragments with modifications in addition to those defined for Type-2 clones, such as insertion, deletion or modification of a statement. Note that for Type-3 clones, the detection tool may requires setting up a boundary of acceptable differences (between two fragments of a Type-3 clone pair) that occur for such modifications in terms of line numbers, token numbers, amount of text and so on.

Type-4 (Semantic Clone): Code fragments with the same functionality with or without being textually similar.

In this paper, by near-miss clones we mean both Type-2 and Type-3 clones with an emphasis on Type-3. Detection of Type-4 clones is not within the scope of this paper. The output of a clone detection tool is usually in terms of code fragment groupings, which is either by clone pair or clone cluster along with their

location information. A Clone Pair (CP) is a pair of code portions or fragments that are similar to each other under a defined similarity measure. A Clone Cluster (CC) is a group of code portions or fragments which are pair-wise similar (inside that group) under a defined similarity measure.

3.3 Proposed Technique

As noted in the introduction, we will evaluate the effectiveness of simhash by evaluating the performance of *SimCad* compared to NiCad. Figure 3.1 gives an overview of end-to-end clone detection process in *SimCad* which uses a distance based similarity detection mechanism developed by Manku et al. [123] based on the fingerprinting technique called simhash. This mechanism has been proven effective in various domains such as data mining and web engineering, where it usually requires dealing with billions of dataset records [123]. In the proposed approach, we tried to deal with the clone detection process from a data mining perspective. We employ a data clustering algorithm with multi-level index based searching which enables fast detection of clones. From the experimental results presented in Section 3.5, we see that *SimCad* is very effective at fast clone detection in a large dataset. Running on a standard desktop computer, the experimental implementation shows that the detection part of the process takes a fraction of a second (cf. Table 3.4) to detect all the Type-1/Type-2 function/block clones in the Linux kernel v-2.6.38, which has 12.5 million lines of code.

A clone detection tool may follow several phases in its detection process [151]. Similar to NiCad, *SimCad* has three phases: pre-processing, detection and output generation (cf. Figure 3.1), which are discussed as follows.

3.3.1 Pre-processing

The pre-processing step sets up the environment and organizes the data over which the detection algorithm is applied. There are four sub-steps in pre-processing as shown in Figure 3.1 by solid rectangular boxes. The first two sub-steps: fragment extraction with pretty printing and source transformation/normalization (renaming the data-types and identifiers) are similar to those in NiCad, details of which are available in [147]. The remaining two sub-steps: simhash generation and indexing are new to this approach which are discussed below in detail.

(i) *Simhash generation:*

The core idea of the proposed approach is to determine code similarity using fingerprinting. Fingerprinting is a well-known approach in data processing that maps an arbitrarily large data item to a much shorter bit sequence (the fingerprint) that uniquely identifies the original data. A typical use is to avoid the comparison or transmission of bulk data. For a large dataset, this technique reduces the data comparison time and effectively decrease the overall running time of a clone detection process.

```
Algorithm simhash(doc, n)

  doc: document for which simhash is computed
  n: length of the desired hash size in bits

 1.  doc is split into tokens (words for example) or super-tokens (word tuples)
 2.  weights are associated with tokens (for example: frequency count)
 3.  v = vector of size n, initialized to 0
 4.  for each token t in doc
 5.      token_hash  =  make_n-bit_simple_hash (t)
 6.      for i = 1 to n do
 7.         if( i-th bit of token_hash == 1)
 8.           v[i] = v[i] + weight(t)
 9.         else
10.           v[i] = v[i] - weight(t)
11.  bit_vector  =  vector of size n, initialized to 0
12.  for i = 1 to n do
13.      if(v[i] > 0)
14.         bit_vector [i] = 1
```

Figure 3.2: *simhash* algorithm[2]

In the proposed approach, each code block will be transformed into an n-bit fingerprint, the simhash value of that block.

Charikar's simhash [33] is a dimensionality reduction technique that maps high-dimensional vectors to small-sized fingerprints [123]. Apart from being an identifier of source data, this technique has the property that fingerprints of near-duplicate data differ only in a small number of bit positions. As for a *simhash* fingerprint f, Manku et al. [123] developed a technique for identifying whether an existing fingerprint f' differs from f in at most k bits. Their experiment shows that for a repository of eight billion pages, 64-bit *simhash* fingerprints and $k = 3$ are reasonable parameters. In the proposed technique, we will refer this notion of bit difference (value of k) using a term called *SimThreshold* as a measurement of similarity, i.e., the lower the value of *SimThreshold*, the more similar the code blocks are. Figure 3.2 shows the pseudocode of the *simhash* algorithm using which a hash fingerprint is generated for each of the extracted code fragments. We have considered words (separated by whitespace) in pretty-printed source as tokens. For generating an n-bit simple hash (Figure 3.2, line 5), we have used a 64-bit Jenkin hash function[3] (from a choice of several cryptographic and non-cryptographic hash functions) based on an empirical observation that it yields better simhash values than others considering similarity preserving nature (a small change in the source results in a small change in the *simhash* bits). This hash value is then used in the detection algorithm to detect code similarity, thus saving significant time by avoiding raw source string comparison.

[2]http://d3s.mff.cuni.cz/ holub/sw/shash (accessed on June 30, 2011)
[3]http://www.burtleburtle.net/bob/hash/doobs.html

(ii) *Multi-level indexing:*

In the proposed approach, a two-level indexing scheme has been introduced to organize the *simhash* data generated in the previous step. The goal of the data organization is to speed up the neighbour search query in the clone detection phase presented in the following section. The indexing is done first by the size of the code fragment in terms of lines of code, and then by the number of 1-bits in the binary representation of the *simhash* fingerprint. Thus, each of the first level indices points to a list of second level indices. Each of the second level index in turn points to a list of simhash values having the same number of 1-bits in its binary form (note, the position of the 1-bits might be different, which is not considered for this count). These simhash values in the final list are corresponding to the code fragments which are similar in size (lines of code). Figure 3.4 illustrates how the two-level indexing scheme works.

At the first level (line based index), the size of a code fragment in lines of code is used as a key to map the corresponding simhash data item. As mentioned earlier, the code has been pretty-printed in the pre-processing phase before computation of *simhash*. This allows us to use a data indexing strategy to store the *simhash* values and helps improve performance by avoiding computations that end up being unsuccessful. Since the *simhash* values are indexed according to their actual line sizes, a search for Type-1 and Type-2 clones does not require to compare the *simhash* of two fragments that have different line sizes. That is, we only need to consider *simhash* data for potential candidates that are computed from the code fragments having a line size that is the same as the new/unclassified *simhash* data item for which a search is being made. However, for Type-3 clones, the algorithm neither uses one particular size nor all sizes; instead it searches within a range of sizes. To limit the search, we put a cap on the differences in number of lines between two code fragments of a Type-3 clone pair. Thus, the neighbour search is limited to a window of indices having a range, for example, 30% of the line size of the corresponding item for which a search is being made.

At the second level (bit based index), the count of 1-bits in the code fingerprint (*simhash* value) is used as an index for ordering those fingerprints. Since we are using *Hamming Distance* [123] between *simhash* values (equal to the number of bit positions at which the corresponding bits are different in their binary form, i.e., the value of k as discussed in the previous section) as the distance measure, the total count of the number of '1' or '0' bits in the simhash value can be a useful index. For example, if a neighbour search is made for a *simhash* value having 10 1-bits in its binary representation, all the Type-1 clones should have the same number of 1-bits in their fingerprint. Same applies for Type-2 clones since when a transformation is applied in the pre-processing step, changes in identifier and function names are eliminated. However, the fingerprint of some other dissimilar code fragments might have the same number of 1-bits, but they would not be considered Type-1 clones because they might have those bits in different bit positions, yielding a *Hamming Distance* greater than zero. For Type-3 clones, this

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

(a) Code Fragment A

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

(b) Code Fragment B

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

(c) Code Fragment C

```
int sum (int num[])
{
 int total = 0;
 int len = sizeof(num) / sizeof(int);
 for (int i = 0; i < len; i++)
  total +=  num[i];
 return total;
}
```

(d) Code Fragment D

```
void swap ( int & a, int & b )
{
  int temp = a;
  a = b;
  b = temp;
}
```

(e) Code Fragment E

Fragment	32-bit simhash	Hamming Distance	Clone Type
A	11111101001111011111010101110100 0	—	—
B	11111101001111011111010101110100 0	sim (a, b) = 0	Type-1
C	11011101001111011010101011101010	sim (a, c) = 3	Type-2
D	11110101001111100101010101110100	sim (a, d) = 4	Type-3
E	10001011110111000101100101111011	sim (a, e) = 18	Not a clone

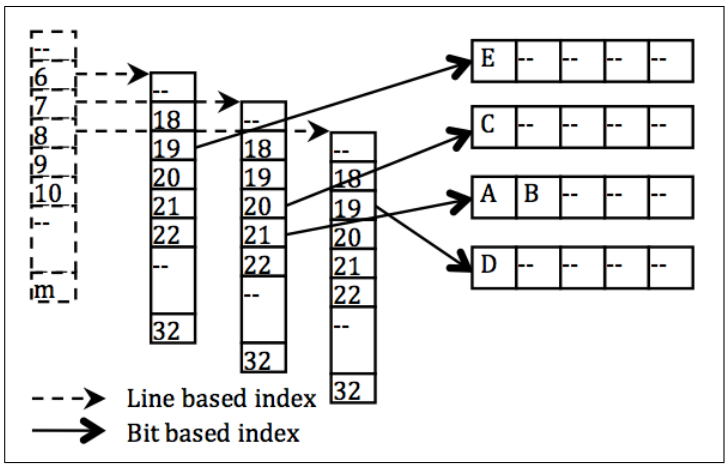
(f) Clone Types based on Hamming Distance of simhash

Figure 3.3: Distance based similarity detection

indexing scheme will also accelerate the searching by allowing a search window range (1-bit count in search item *SimThreshold*). For example, when *SimThreshold* = 5 and the number of 1-bits in the search item is 10, then the algorithm will consider comparing fingerprints having 1-bits within a range from 5 to 15 and ignore others. In Section 3.5 we have shown how this indexing strategy improved the runtime performance of the clone detection process.

3.3.2 Clone Detection

This phase is responsible for identifying and grouping similar code fragments into clusters, which are called *Clone Clusters*. The grouping is done based on the *Hamming Distance* among the *simhash* values of code fragments extracted from the codebase, and the value of *SimThreshold* controls the boundary of a *Clone Cluster*. The process is similar to a typical data clustering algorithm that partitions the data based on the



(a) Final organization of simhash data

Fragment	Lines	No. of 1-Bit
A	7	21
B	7	21
C	7	20
D	8	19
E	6	19

(b) Indexing properties for fragments A-E from Figure 3.3

Figure 3.4: Indexing strategy used in *SimCad*

similarity of individual records; the more similar the data, the more likely that they belong to the same cluster. The main goal is to identify clusters that maximize the inter-cluster distance and minimize the intra-cluster distance so that we obtain clearly distinct groups of similar entities. Therefore, in this case, once the *simhash* values for all the code fragments are available, the problem of detecting code clones is essentially clustering the *simhash* values so that, in a cluster, the pair-wise similarity distance remains below to a pre-defined threshold value, *SimThreshold*, while restricting the cluster size to be no less than another pre-defined value, *MinClusterSize*. Figure 3.3 represents an example scenario showing how the similarity detection mechanism works in the proposed approach. Note that, the example scenario uses 32-bit hash fingerprint. However, in the actual experiment we have used 64-bit hash. In summary, a *simhash* dataset S of size N is defined as:

$$S = \{s_i\}_{i=1}^{N} \tag{3.1}$$

where s_i is the simhash of a code fragment.

For any pair of simhash values in S, the pair-wise distance is:

$$D_{ij} = hamming\_distance(s_i, s_j) \tag{3.2}$$

The output of the algorithm is a cluster set C of size R such that:

$$C = \{c_i\}_{i=1}^{R} \tag{3.3}$$

where cluster c is a subset of *simhash* data elements s_j of size L ($MinClusterSizeLN$):

$$c = \{s_j\}_{j=1}^{L} \tag{3.4}$$

such that, for any pair of simhash values in cluster c,

$$\forall\ (i,j)\ \in\ (1 \cdots L) : D_{ij} \leq SimThreshold \tag{3.5}$$

```
Algorithm DBSCAN(S, eps, minPts)
   S: set of data item
   eps: distance threshold
   minPts: minimum cluster size
1. for each unvisited data s in dataset S
2. mark s visited
3. N = getNeighbours (s, eps)
4. if(sizeOf(N)) >= minPts
5.   C = a new cluster
6.   expandCluster(s, N, C, dt, minPts)
```

(a)

```
expandCluster(s, N, C, eps, minPts)
   s: a data item
   N: set of neighbors for s
   C: data cluster
   eps: distance threshold
   minPts: minimum cluster size
1. add s to C
2. for each data s' in N
3.     if s' is not visited
4.        mark s' visited
5.        N' = getNeighbours (s', eps);
6.        if(sizeOf(N')) >= minPts
7.            N = N joined with N'
8.      if s' is not yet member of any cluster
9.          add s' to cluster C
```

(b)

Figure 3.5: *DBSCAN* (from [48]) algorithm

Thus, the results of the clustering algorithm over simhash values imply that, similar code fragments are grouped into the same cluster (or form their own cluster), which can be considered as a *Clone Cluster/Clone Group*.

The value of *MinClusterSize* is set to 2 and the value of *SimThreshold* is varied as required. For example, a $SimThreshold = 0$ identifies exact clones and a $SimThreshold > 0$ identifies renamed and near-miss clones. Note that the higher the value of *SimThreshold* the higher the possibility of false positive results (i.e., not an actual clone pair/group). We can empirically determine a cap for *SimThreshold* by evaluating the results at different increasing values until the presence of false positives in the detection result is acceptable. Section 3.5 describes the details of this process.

In this experiment, a popular density based clustering algorithm named *DBSCAN* [48] is used. Here, the formation of a cluster is based on the notion of density reachability. Basically, a point q is directly density-reachable from a point p if their pair-wise distance is no greater than a given distance $\varepsilon - neighborhood$, and if p is surrounded by sufficiently many points such that one may consider p and q to be part of a cluster. Figure 3.5 shows the pseudo-code of the algorithm. It requires two parameters: $\varepsilon - neighborhood$ (mentioned as *eps* in Figure 3.5, which we call *SimThreshold* in the proposed approach) and a minimum cluster size in number of points (mentioned as $minPts$ in Figure 3.5, which we call *MinClusterSize*). The algorithm starts with an arbitrary starting point that has not been visited. This point's $\varepsilon - neighborhood$ is searched, and if it contains sufficiently many points, a cluster is started. Otherwise, the point is labeled as noise. This point might later be found in a sufficiently sized cluster of a different point and hence is made part of that cluster. In summary, the output of *DBSCAN* is some grouping of similar *simhash* values based on their *Hamming Distance*. Therefore, using this grouping we will get all the groupings of similar code fragments, i.e., all the *Clone Clusters*.

3.3.3 Output Generation

From the previous phase, we saw that the detection algorithm delivers clone detection output as lists of *Clone Clusters* each containing more than one code fragment (in its original form from the source code and its start and end line location in the file it is from along with the name of the file) which are pair-wise similar to each other. Additional post-processing steps might be required in some cases to filter out some type of clones from the output. The output is in XML format which is convenient for displaying as a webpage using XSLT[4] or for importing into a clone visualization tool.

3.4 Runtime Complexity of the Process

In this section, the runtime complexity of the overall process is discussed. As mentioned in the previous section, the process includes three phases: pre-processing, detection and output generation. The source code pre-processing and output generation time is linear to the size of the codebase. For the additional two pre-processing operations (mentioned in Section 3.3), generation of simhash requires $O(q)$ time where q is the number of tokens in the code fragment for which simhash is being calculated and thus for the complete codebase overall complexity would be $O(n.q)$, where n is the number of code fragments extracted from the codebase. The indexing setup complexity is $O(n)$ since this step is also linear with respect to the input. The detection time of *SimCad* depends on the *DBSCAN* algorithm that visits each of the n (to be more exact, the remaining non-clustered) data points of the dataset, possibly multiple times (e.g., as candidates to different clusters). However, the time complexity is mostly governed by the number of *getNeighbours()* queries (cf. Figure 3.5(a) - line 3, Figure 3.5(b) - line 5) that is executed for each data point. If a binary search scheme can be used that executes such a neighbourhood query in $O(logn)$, an overall runtime complexity of $O(nlogn)$ is obtained [48]. In the proposed approach, the *simhash* values of the code blocks cannot be organized so that a binary search can be applied; instead a two level indexing scheme has been introduced to speed up the neighbour search query. First, by the size in lines of code and then by the number of 1-bits in the fingerprint; details of which was discussed in the previous section. For Type-1 and Type-2 clones, the search query requires $O(1)$ for the 1st level index, $O(1)$ for the 2nd level index and $O(m\_max)$ for a linear search where m is the number of elements/fragments that share the same number of 1-bits in their hashes. Therefore, the overall runtime complexity is $O(n * m_max)$. Here, the maximum value of m could be n but only when all the code fragments have the same number of lines and all the corresponding simhash values contain the same number of 1-bits, which is extremely rare. On average, m can take a value of $(n/(avg\ (l) + 64))$, where l is the size of a code fragment in lines. For Type-3 clones, the search query requires $O(l)$ for the 1st level index where l is the size of a code fragment in lines (note: the value of $l \ll n$ for a large system and typically has a maximum value not exceeding a couple of hundreds), $O(k)$ for the 2nd level index where k ($0 \leq k \leq 64$)

[4]http://www.w3.org/TR/xslt

Table 3.1: Subject Systems Used in Our Experiment

Subject Systems	Version	Language	Physical-LOC	URL
Eclipse-jdt	3.6.2	Java	289678	www.eclipse.org/jdt/core/
Jboss-AS	5.1.0	Java	563585	www.jboss.org/jbossas/
Firefox	2.0.0.4	C	2711444	ftp.mozilla.org/pub/mozilla.org/firefox
Linux	2.6.38	C	15720527	www.kernel.org

is the number of 1-bits in the hash and $O(m\_max)$ for a linear search where m is the number of elements that share the same number of 1-bits in the hash and thus the overall runtime complexity is the same as $O(n * m\_max)$.

3.5 Implementation, Analysis and Evaluation

This section summarizes the implementation and execution of *SimCad*, the correctness measure of the detection, the evaluation of the detection results over four open source projects (cf. Table 3.1), the performance comparison with the original tool NiCad and finally some additional enhancements are suggested. The case studies were performed on a Linux OS (Ubuntu 10.10), which is running on a desktop PC with an Intel Core i7 3 GHz processor and 4 GB of RAM. We have implemented the proposed algorithm in Java. The size of *simhash* we used was 64-bit. This is good enough to be represented by the Java long data type, which is also a 64-bit, signed two's complement integer. Using a 32-bit hash improves the pre-processing and detection time to some extent. However, precision was found very low because of increased false positive detection even with lower *SimThreshold* values. Granularity was chosen both at the function and block levels.

3.5.1 Detecting different types of clones using *SimCad*

The *SimCad* tool can be configured to detect different types of clones, individually or all at once. To detect only Type-1 clones, the value for *SimThreshold* is set to 0, i.e., the *Hamming Distance* between two simhash values is 0. Thus, *simhash* values of a potential Type-1 clone pair must be equal and only exact copies of code fragments will have equal simhash values. No source normalization (identifier/variable renaming) is required during the pre-processing phase for the detection of Type-1 clones except pretty-printing of the code. To detect Type-2 clones, the *SimThreshold* is also set to 0, but additional source normalization [147] is required over the pretty-printed source in the pre-processing phase in order to avoid the changes due to renaming of identifiers and/or function names. Note that, in this case the output will also contain Type-1 clones. The original, non-normalized source code is used to identify and filter out Type-1 clones and find the accurate Type-2 clone count.

To detect Type-3 clones, *SimThreshold* is set to an optimum value called *MaxSimThreshold*, which has been determined through an empirical process that will be described in the next subsection. Source normal-

Table 3.2: Summary of Clone Detection Setup for *SimCad*

Runtime Parameter	Value	Found Empirically
MinClusterSize	2	No
MinSimThreshold	0	No
MaxSimThreshold	12	Yes
Simhash size	64-bit	Yes
Granularity	fixed (function/block)	No

(a)

Clone Type	SimThreshold	Source Normalization	Post processing
Type-1	0	No	No
Type-2	0	Yes	Yes, removal of Type-1
Type-3	MaxSimThreshold	Optional, yes will yield better results	Yes, removal of clone class of Type-1 and Type-2
All at once	MaxSimThreshold	Optional, yes will yield better results	No

(b)

ization is optional during the pre-processing phase, but applying normalization will yield better detection results. Clone clusters containing only Type-1 and Type-2 clones are filtered from the output to get the exact count of Type-3 clones. Table 3.2b summarizes the complete setup for detecting different types of clones.

3.5.2 Finding the optimal value for SimThreshold

In the clustering algorithm, *SimThreshold* is used as a limit of acceptance for the neighbour search query (cf. *eps* in Figure 3.5(a) - line 3, Figure 3.5(b) - line 5) that looks for similar code fragments with respect to a candidate fragment (s in Figure 3.5(a) - line 3, Figure 3.5(b) - line 5). As noted earlier, the higher the value of *SimThreshold*, the more Type-3 clones will be detected. But there is a limit to this value over which the algorithm will start including false positive clones in the detection results (cf. Figure 3.6). So the optimal value is a value that maximizes the detection of Type-3 clones and minimizes the presence of false positives in the detection result. Another significant finding was that the optimum value for *SimThreshold* is not the same for all candidate code fragments (the fragment for which a neighbour search query is performed in the detection algorithm) of all sizes, i.e., it is size sensitive and choosing a larger value for smaller candidate fragments will allow the algorithm to pick some totally dissimilar code fragments as a cluster member.

Figure 3.6 shows the distribution of the optimal threshold values for different size groups of fragments based on the two smaller (to make manual verification easier) projects among the four test projects used in this experiment. For each of the line size groups, we performed manual analysis to identify the presence

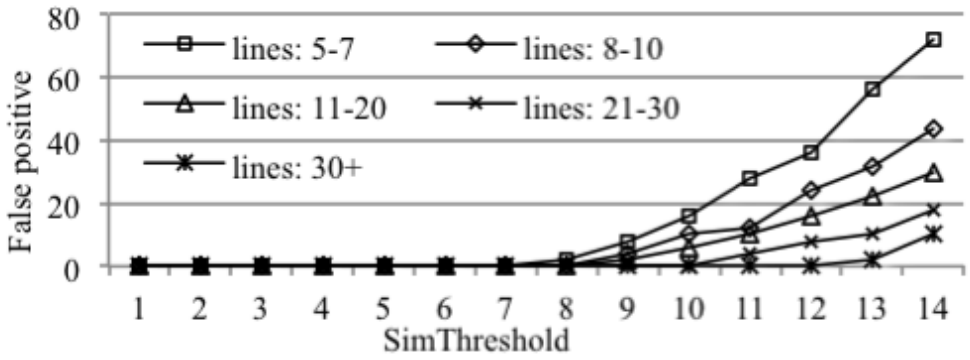


Figure 3.6: Relation between fragment sizes vs. SimThreshold

and count of false positives in the detection results. In an implementation of the proposed approach, the maximum allowed value for *SimThreshold* was found to be 12. However, it is clear from the graph that for smaller size code fragments the optimal threshold value is also smaller and for larger sizes it can have a larger value. This dynamic nature of *SimThreshold* has been incorporated into the implementation of *SimCad*, i.e., the threshold value for candidate code fragment is adjusted with respect to its size at runtime when it searches for its neighbour code fragments. For example, when the algorithm executes the neighbour search query using the instruction *getNeighbours (s, eps)* (cf. Figure 3.5), if the size of s, $|s|$ is between 5-7 lines, the value of *eps* is set to 7; if $|s|$ is between 8-10 lines, *eps* is set to 8 and so on.

3.5.3 Measurement of correctness in clone detection

In this study, a measure of correctness of *SimCad* has been done on the detection result, which is an important step for the reliability of any clone detection approach. The quantitative evaluation used here is based on three criteria: *precision*, *recall* and *f-measure*. Precision is the measure of actual clones in the detection result, recall is the measure of clones detected among the available clones and finally f-measure is the measure of a test's accuracy interpreted as a weighted average of precision and recall; for all the cases the score reaches its best value at 1 and worst at 0. Following are the three equations for the measurement of correctness criteria.

$$precision = \frac{reference \cap detected}{detected} \tag{3.6}$$

$$recall = \frac{reference \cap detected}{reference} \tag{3.7}$$

$$f - measure = \frac{2 * precision * recall}{precision + recall} \tag{3.8}$$

To evaluate the correctness of *SimCad*'s detection result, we used a variant of the mutation/injection framework for evaluating clone detection tools as proposed by Roy and Cordy [149]. Instead of completely automating the framework, we followed a semi-automated approach and a test was carried out with function

Table 3.3: Mutation-based Effectiveness of *SimCad*

		Type-1	Type-2	Type-3	Overall
clone	Reference	20	30	50	100
	Detected	20	30	49	99
	Missed	0	0	1	1
	False positive	0	0	0	0
effectiveness	precision	1	1	1	1
	recall	1	1	0.98	0.99
	f-measure	1	1	0.989	0.99

clones only. We chose a small sized open source system (Apache-Ant) for this test. To create a mutated codebase, some functions were arbitrarily selected from the original codebase and from those a total of 100 different types of reference clones (cf. Table 3.3) were created manually by copying and/or modifying the code using diverse change patterns. These reference clones were then injected into original codebase at random locations that yielded the mutated codebase for the test. The location information of the injected reference clones was recorded for the future use for a validator program. After building the mutated codebase, we applied *SimCad* on it and the clone detection results were analyzed by a validator program that looks for the reference clones in the result and then measures the values of the correctness criteria defined by equations 1, 2 and 3. Table 3.3 shows the result of the output analysis and the measured values of the three criteria. From the data, we can see that *SimCad* correctly detects all the Type-1 and Type-2 reference clones but failed to detect one Type-3 clone among the 50 Type-3 reference clones. An investigation revealed that the code modification done while building this reference clone was so much that it went beyond the chosen maximum *SimThreshold* value of 12. Now that we know *SimCad* is working correctly according to this mutation/injection based test, we next see how *SimCad* does in comparison with another tool. As we said earlier, we will compare the performance of *SimCad* considering NiCad as a benchmark over the same subject systems in order to find the answers to the research questions defined in Section 3.1. However, before doing that, we need to come up with equivalent settings for both the tools to make the performance comparison fair.

3.5.4 Finding detection threshold equivalency between NiCad and SimCad

Here the goal is to compare the performance of NiCad and *SimCad* with respect to time and the number of cloned fragments detected. Each of the tools has its own definition of threshold for setting the boundary of similarity. Similar to SimThreshold in *SimCad*, detection in NiCad is governed by a threshold value called UPI-Threshold that puts a limit to the acceptable dissimilarity between two code fragments in a clone cluster. The value ranges from 0 to 1, where 0 means exactly similar and 1 means totally dissimilar. For *SimCad*, it is

SimThreshold, the detail of which was given in Section 3.3. A value of 0 for both the thresholds means exact similarity whereas a value greater than zero means some dissimilarity is acceptable; the larger the value the more dissimilarity is acceptable for being a member of a clone cluster.

From Table 3.2b, we see that *SimCad* uses a threshold value 0 for both Type-1 and Type-2 clones, which is the same for NiCad. Thus, comparing the results of NiCad and *SimCad* is easier in these two cases. However, for Type-3 clones, both the tools use a threshold value greater than 0 and hence to conduct a fair comparison in this case an equivalency between these two thresholds needs to be defined. To do that, we first detect all the three types of function clones using *SimCad* with SimThreshold = 12, which we found to be the highest optimum threshold value (cf. Figure 3.6). The output has been analyzed by a separate program, which measures the coverage of the clones under the definition of NiCad's UPI-Threshold with increasing threshold values starting from 0. It has been found that, at UPI-Threshold = 0.4 almost all the clones (detected by *SimCad* at SimThreshold = 12) are covered and hence in this experimental setup two thresholds are considered as equivalent to each other. However, NiCad considers 0.3 as a standard value for UPI-threshold in Type-3 clone detection and here we came up with an equivalent threshold value more than that standard value. This is an important decision to make since this might be a threat to the validity of the proposed approach. For the tool comparison, we have decided to go with the value 0.4 for obvious reasons. Equivalency in the other direction could have used in this experiment, i.e., consider finding an equivalent SimThreshold value corresponding to the standard UPI-Threshold value which is 0.3. However, in that case, a lower or higher value than 12 for SimThreshold would make *SimCad* miss potential Type-3 clone or reduce clone quality by including more false positives in detection result, and neither of which are expected. With *SimCad*, the target is to get as many near-miss clones as possible while minimizing the presence of false positives in the detection results since we are considering this approach to be effective for large scale Type-3 clone detection. Thus, we have decided to continue the comparison with the value 0.4 for the UPI-Threshold. Below we will present the comparative performance analysis of both the tools running on these two equivalent thresholds.

3.5.5 *SimCad* vs. NiCad: head-to-head comparison

The study examines the effectiveness of simhash by evaluating *SimCad*'s detection capability. The performance of *SimCad* in terms of detection time and number of clones has been compared with NiCad based on four medium to large scale open source applications. The test data was taken with an equivalent similarity measurement setting for both the tools defined in Section 3.5.4. Let us look at the detection time first. Table 3.4 and Table 3.5 show the comparison on detection time between NiCad and *SimCad* for three types of function and block clones respectively. Since we are trying to compare the efficiency of the detection engine of both the tools, the time data presented here is for the detection process only (excluding any pre or post processing time). The experimental outcome clearly shows that the proposed approach significantly outperforms NiCad in detection time for all the three types of clones.

Table 3.4: *SimCad* vs. NiCad function clone detection time (ms)

Subject Systems	No. of functions of size 5 lines or more	Type-1		Type-2		Type-3	
		NiCad	*SimCad*	*NiCad*	*SimCad*	*NiCad*	*SimCad*
Eclipse-jdt	9832	309	6	313	7	7592	655
Jboss-AS	17601	667	8	760	10	15102	2737
Firefox	15285	358	5	364	6	32324	2392
Linux	198146	13343	713	14319	720	7660091	625563

Table 3.5: *SimCad* vs. NiCad block clone detection time (ms)

Subject Systems	No. of functions of size 5 lines or more	Type-1		Type-2		Type-3	
		NiCad	*SimCad*	*NiCad*	*SimCad*	*NiCad*	*SimCad*
Eclipse-jdt	10903	1383	8	389	8	15022	1255
Jboss-AS	31086	2213	10	2238	12	76022	5433
Firefox	39990	1972	9	1985	10	195215	11382
Linux	418605	250050	7851	256117	4865	20702947	2194557

Table 3.6: *SimCad* vs. NiCad function clone fragment count

Subject Systems	Type-1		Type-2		Type-3		All at once	
	NiCad	*SimCad*	*NiCad*	*SimCad*	*NiCad*	*SimCad*	*NiCad*	*SimCad*
Eclipse-jdt	555	555	1719	1719	3190	3319	3613	3802
Firefox	1273	1273	1679	1679	3593	3741	4317	4614
Jboss-AS	1463	1463	2102	2102	6780	6925	7806	7991
Linux	2860	2860	18230	18230	40150	41114	52970	54534

Table 3.7: *SimCad* vs. NiCad block clone fragment count

Subject Systems	Type-1		Type-2		Type-3		All at once	
	NiCad	*SimCad*	*NiCad*	*SimCad*	*NiCad*	*SimCad*	*NiCad*	*SimCad*
Eclipse-jdt	2002	2002	2863	2863	4054	4211	5749	5913
Firefox	5962	5962	7250	7250	12726	12996	14019	14318
Jboss-AS	5674	5674	6847	6847	10148	10353	10511	10957
Linux	53967	53967	63458	63458	142638	143479	160988	162043

Table 3.8: Common and unique clones in NiCad vs. *SimCad*

Subject Systems	Function Clone			Block Clone		
	Common	***Unique***		***Common***	***Unique***	
	NiCad/SimCad	*NiCad*	*SimCad*	*NiCad/SimCad*	*NiCad*	*SimCad*
Eclipse-jdt	2804	386	515	3795	259	416
Firefox	3233	360	498	12297	429	699
Jboss-AS	6494	286	431	9788	360	565
Linux	38740	1410	2374	141086	1522	2393

Note that, for a large scale system such as the Linux Kernel, *SimCad* is 19 times faster than NiCad for Type-1/Type-2 clones and 12 times faster for Type-3 clones in case of functional clone detection. Similarly, for block clone detection on Linux, SimCad is 31 times faster than NiCad for Type-1/Type-2 clones and 9 times faster for Type-3 clones.

Table 3.6 and Table 3.7 show the comparison in number of detected function and block clone fragments respectively. We see that both the tools are equal for Type-1 and Type-2 clones but differ for Type-3 clones, and again *SimCad* performs slightly better than NiCad in this case. Analyzing the outcome of Type-3 clones for both the tools, we have identified that, although *SimCad* detects more Type-3 clones than NiCad, it does not cover all the clones detected by NiCad. That means under the equivalent threshold setting, each of the tools is detecting some unique clones that are undetected by the other tool. Common and unique Type-3 clone fragment count for both tools is shown in Table 3.8. We analyze the unique clone fragments for both the tools and investigate the cause of why one tool detected some of the clones that the other tool failed.

3.5.6 Analyzing the unique clones

Table 3.9 presents the categorization of unique clone fragments detected by *SimCad* with different UPI-Thresholds. The important observation from the data distribution presented here is that, NiCad should have detected the clones falling under UPI-Thresholds from 0.1 to 0.4 since the UPI-Threshold for NiCad was set to 0.4 for Type-3 clone detection. The likely reasons are summarized as follows:

i) *Single data reference for cluster:* NiCad uses a single data reference for cluster membership. Therefore, it might miss those potential clones which fall under the UPI-Threshold in comparison to other members of the cluster but not for the reference member.

ii) *Coarse grain change detection:* NiCad uses the UNIX *Diff* in the detection engine to measure the dissimilarity between two code fragments. *Diff* works on line level changes, i.e., it does not care whether the change in a line is small or significant. Thus if there are small changes in multiple lines of a code fragment, the dissimilarity measure might go beyond the accepted threshold limit which will cause the fragment to be undetected by NiCad.

iii) *Sensitive to ordering of line:* The *Diff* command puts another limitation on NiCad since it is sensitive to line ordering. Therefore, re-ordering among the statements of a code fragment might result in a high dissimilarity value that might cause the fragment to be undetected. In contrast to NiCad, *SimCad* considers each of the cluster members as cluster references, uses fine grained (token level) change detection and is insensitive to instruction reordering except for Type-1 clones. Thus, the proposed approach overcomes some of the limitations in NiCad while taking much less time to detect the clones.

Table 3.10 shows the number of clones that are not detected in *SimCad* but detected in NiCad with equivalent settings and categorized by different SimThresholds. These clones were detected in NiCad with UPI-Threshold 0.4, but it is clear from the table that those were not detected in *SimCad* because they were beyond the equivalent SimThreshold value 12. Note, for each of these clones, the minimum distance with a cluster member was considered for categorization of that clone fragment. For example, in a clone cluster, if the distance of a clone fragment x_1 from other cluster members x_2, x_3, x_4 and x_5 are computed as 14, 13, 17 and 19 respectively, then x_1 is categorized (in Table 3.10) under 13. From Table 3.10, it is clear that the majority of the undetected clones are just above the SimThreshold value of 12. Hence, by increasing the SimThreshold value those clones can be detected. However, according to an earlier analysis (cf. Figure 3.6) a threshold value greater than 12 will increase the chances of false positive detection. As a solution to this problem, we have implemented a couple of additional techniques to assist the original detection process, which allow *SimCad* to use a high SimThreshold value while minimizing the presence of false positives. Those are presented as follows.

(a) *Multiple simhash:* If we look at the simhash algorithm (Figure 3.2, line 5), it uses a simple hash function to compute a hash for each token. In this study which was the Jenkin Hash Function. It is possible to use another simple hash function here to generate a second simhash value for each fragment. Note that the second simhash will also hold the basic principle of simhashing although it might be a completely different hash value with respect to the first one and this second simhash must be compared with the second simhash of another fragment and never with the first one. Thus, a combination of these two simhash values will allow choosing a higher SimThreshold value while minimizing the presence of false positives.

We have used a variant of the Jenkin hash in order to generate the second simhash and then again gone through the procedure discussed in Section 3.5.2 to determine the optimum value for SimThreshold. This time the highest value was found to be 16. Now, if we see the statistics in Table 3.10, it is clear that with this SimThreshold value, *SimCad* does not cover 100% of the unique clones detected by NiCad (it is still missing those at SimThreshold 17 and 18) but it does cover most of them (95-98%). Detection with multiple-hash also costs additional processing time in detection. From the experiment it has been found that this double simhash version requires 25-30% more time than the original one. The detection time comparison among the single simhash based *SimCad*, dual simhash based *SimCad* and NiCad is shown in the first four columns of

Table 3.9: Unique clone fragments in *SimCad* with different UPI-Thresholds

Subject Systems	Function Clone								Block Clone							
	UPI-Threshold								***UPI-Threshold***							
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	total	0.1	0.2	0.3	0.4	0.5	0.6	0.7	total
Eclipse-jdt	0	8	162	221	88	26	10	515	0	3	98	196	82	29	8	416
Firefox	0	6	113	250	86	29	14	498	0	4	185	351	97	48	14	699
Jboss-AS	0	2	125	208	58	26	12	431	0	3	134	248	124	39	17	565
Linux	0	16	576	1085	448	182	67	2374	0	22	671	992	587	72	49	2393

Table 3.10: Unique clone fragments in NiCad with different SimThreshold

Subject Systems	Function Clone							Block Clone						
	SimThreshold							***SimThreshold***						
	13	14	15	16	17	18	total	13	14	15	16	17	18	total
Eclipse-jdt	164	96	68	42	14	2	386	108	65	46	28	10	2	259
Firefox	124	72	83	66	9	6	360	96	88	145	78	14	8	429
Jboss-AS	98	62	67	44	10	5	286	98	93	82	64	17	6	360
Linux	713	316	218	92	44	27	1410	713	348	266	114	49	32	1522

Table 3.11. We can see, despite additional processing, *SimCad* with dual simhash still performs better than NiCad.

(b) *Additional text-based comparison in the detection process:* In the detection algorithm, distance-based hash similarity comparison at higher thresholds can be assisted by some additional text-based comparison (for example the diff algorithm) to detect some potential clones that are undetected by *SimCad* with its current settings. This text- based comparison will be applied conditionally. For example, with respect to the categorization presented in Table 3.10, the text-based comparison will be applied to check similarity only when the distance between two simhash falls between 13-18. We have integrated an implementation of Myer's [134] diff algorithm in the detection process of *SimCad* and used it to measure text similarity during the condition mentioned above. With this approach, *SimCad* was successfully able to detect all the unique clones detected in NiCad (cf. Table 3.10). Table 3.11 shows the time comparison of *SimCad* including diff (last column) with the single and dual simhash based *SimCad* and NiCad. We see that the approach takes 60-65% more time than the original version of *SimCad*. Although it requires a bit more time than the multiple simhash based approach, it is still much faster than NiCad.

These workarounds can be seen as a trade-off between the time efficiency and detection accuracy of the proposed approach for Type-3 clone detection, and can be applied as per the detection requirement.

3.5.7 Time performance gain from multi-indexing

To measure the efficiency of our proposed multi-level indexing strategy we have implemented the following four versions of *SimCad* with different indexing strategies:

Table 3.11: Time (ms) comparison of *SimCad* including *diff* with others in Type-3 function clone detection

Subject Systems	**NiCad**	**SimCad with single-simhash**	**SimCad with dual-simhash**	**SimCad with single-simhash and diff**
Eclipse-jdt	7592	655	845	1106
Firefox	32324	2392	3011	4075
Jboss-AS	15102	2737	3388	4642
Linux	7660091	625563	769432	1000846

Table 3.12: Time (ms) comparison for different indexing strategies

Subject Systems	**Type 1 Function Clone**				**Type 3 Function Clone**			
	Naive	*L-index*	*B-index*	*M-index*	*Naive*	*L-index*	*B-index*	*M-index*
Eclipse-jdt	2712	281	266	6	5564	4182	5041	655
Jboss-AS	4270	462	430	8	6419	5704	6235	2737
Firefox	6645	657	615	5	10153	7877	9532	2392
Linux	218761	28975	27537	713	2239618	1815249	2033079	625563

Naive: No indexing

L-index: Line based index only

B-index: Bit based index only

M-index: Multi (both line and bit based) index

Table 3.12 summarizes the detection time for these variants of *SimCad* based on the indexing strategies. It is clear from the data that the multi-level indexing used in the proposed approach clearly sped the detection proccess up to a notable extent.

3.5.8 Addressing the research questions

We see that the experimental results and analysis strongly supports the effectiveness of simhash's use in the detection of exact and near-miss clones in a software system. The proposed simhash based approach allowed *SimCad* to work significantly faster than NiCad, which is a challenge for a large system such as the Linux Kernel. *SimCad* also overcomes some of the shortcomings of NiCad that allowed *SimCad* to be able detect some new potential Type-3 clones. On the other hand, *SimCad* has its own shortcomings for Type-3 clone detection, which can be overcome with a couple of additional techniques. In summary, simhash:

1. is feasible to be used for software clone detection,

2. enables faster detection of clones in large software systems, and

3. has some limitations in detecting Type-3 clones that can be overcome with a few additional techniques.

3.6 Related Work

Fingerprinting techniques have been used in different areas of computing research. In software clone detection research, a number of approaches used fingerprinting with normalized source code or Abstract Syntax Trees (ASTs). Johnson [81] presents a detection mechanism that uses fingerprints to identify exact repetitions, which is not applicable for near-miss clone detection. Chilowicz et al. [34] proposed an approach that uses hash fingerprints on ASTs. State of the art approaches proposed by Hummel et al. [73] and Li et el. [113] are also based on statement fingerprints and very good in detecting Type-1 and Type-2 clones. However, since these techniques do not use a similarity preserving hash, Type-3 clone detection would be hard to support or not possible at all. Smith and Horwitz [162] present a similarity preserving fingerprinting technique, while Baxter et al. presents CloneDR [23] uses a hash based AST comparison, both having some success in detecting Type-3 clones. However, the measure of effectiveness of their approaches in detecting Type-3 clones was not evident from the experimental results they provided. Jiang et al. [78] presents an approach that uses matching of characteristic vectors on ASTs for identifying clones including Type-3. However, the claim of the approach being scalable in the case of very large datasets was not clearly explained. In summary, existing fingerprinting techniques are either incapable or have not been proven effective in the detection of Type-3 clones and/or are not scalable to large datasets. Type-3 clone detection in large datasets is one of the main concerns of the proposed approach since for Type-1 and Type-2 clones, there are a number of fast and elegant solutions available [73, 88, 113], but for Type-3 clones success is limited [151, 151].

Recently, Yuan and Guo developed Boreas [182], a scalable token-based clone detector, especially designed to detect those clones with swapped lines or added/removed tokens. They introduce a novel counting-based method to define the characteristic matrices, which are able to describe the program segments distinctly and effectively for the purpose of clone detection. A code clone detection approach based on formal methods is proposed by Cuomo et al. [40], where Process Algebra [5] was used for equivalence checking to detect whether two fragments of code are clones. Their approach can detect Type-1 and Type-2 clones only. Abd-El-Hafiz [1] proposed a metrics-based data mining approach for clone detection. A data mining algorithm, Fractal Clustering [16], is used on some metrics (collected for all functions in the software system) to partition the software system into a relatively small number of clusters. Each of the resulting clusters encapsulates functions that are within a specific proximity of each other in the metrics space from which clone classes, rather than pairs, are extracted. When the methods or blocks in source code are partially duplicated, existing metric-based techniques and text-based techniques using the LCS algorithm unable to detect code

[5] http://en.wikipedia.org/wiki/Process_calculus

clones. Murakami et al. [133] proposes a new method that detects those gapped code clones using the Smith-Waterman algorithm [161] that can identifying similar alignments between two sequences even if they include some gaps.

Outside the area of clone detection, Google is using simhash for finding near duplicate webpages [64]. Gong et al. [60] presents an approach for detecting near-duplicates within a huge repository of short messages. Similarly, SimFinder [139] is a fast algorithm proposed by Pi et al. to identify all near-duplicates in large-scale short text databases. The experimental outcome in this study shows that the simhash based fingerprinting has great potential for use in the area of clone detection.

3.7 Summary

Detection of clones provides several benefits in terms of maintenance, program understanding, reengineering and reuse [108]. We took an existing code cloning system and improved the time performance by an order of magnitude using simhash and demonstrated its feasibility for use with large systems such as the Linux Kernel. As well, we adapted simhash to a code cloning framework and demonstrated its viability for the clone detection of Type-1, Type-2 and Type-3 clones in large-scale systems. The experiment confirms that diff-based comparison (as used in NiCad) works well for finding Type-3 clones, which was also observed by Tiarks et al. [169]. However, this comes with both a cost in time complexity, and lower recall (for the possible optimizations used in the comparison). On the other hand, simhash has significant potential for the fast and large scale Type-3 clone detection but comes with the increased possibility of false positive clones. However, we have shown that it is possible to overcome the limitations of simhash using techniques like multiple hashing or a conditional diff-based comparison. We believe that this study supports a call for further research in using/adapting simhash for clone detection research or similar studies.

Chapter 4

SimCad: A Highly Scalable and Configurable Clone Detection Tool

4.1 Introduction

Are clones harmful in software development and evolution or are they not? Despite a decade of active research in software clone, arguably there is no such conclusive finding that significantly favors one argument over the other. However, researchers commonly agreed that in order to make best use of the good aspects of code cloning and to reduce the possible harmful effects, we need an efficient and cost-effective way to manage clones. Although, a number of clone detection tools have been proposed in past studies, only a few of them can perform well against current diverse requirements such as fast detection of near-miss clones and adaptation/integration of a third party clone detection tool to a clone management system. Thus the need for designing better clone detection techniques is still considered an important problem. Considering that as a motivation, *SimCad* is developed as a fast and scalable clone detection tool that works well in detecting near-miss clones even for a large-scale dataset and non-source code data.

4.2 Motivation

Although there has been considerable research in the field of clone detection over the past years, detection of Type-3 clones is still an open research issue due to the inherent vagueness in their definition [169]. A recent study by Saha et. al [157] shows that Type-3 clones should be managed more carefully than Type-1 and Type-2 clones due to their more inconsistent nature.

Existing popular techniques [23, 88, 113] have several deficiencies. Although these tools are good in detecting Type-1 and Type-2 clones, many of them do not support the detection of Type-3 near-miss clones, or not scale adequately to handle clone detection in large systems [78]. Tools like CCFinder [88], Dup [11], ConQAT [84] are frequently used in both academia and research. They showed the usefulness of the fast speed with which suffix trees are used in detecting clones. Falke et al. [50] shows, suffix tree based tools can detect type-1 and type-2 faster, but require additional post-processing step in detecting type-3 clones [169]. Other approaches like CCDIML [142], CloneDR [23], SimScan [29] are in general computationally more expensive

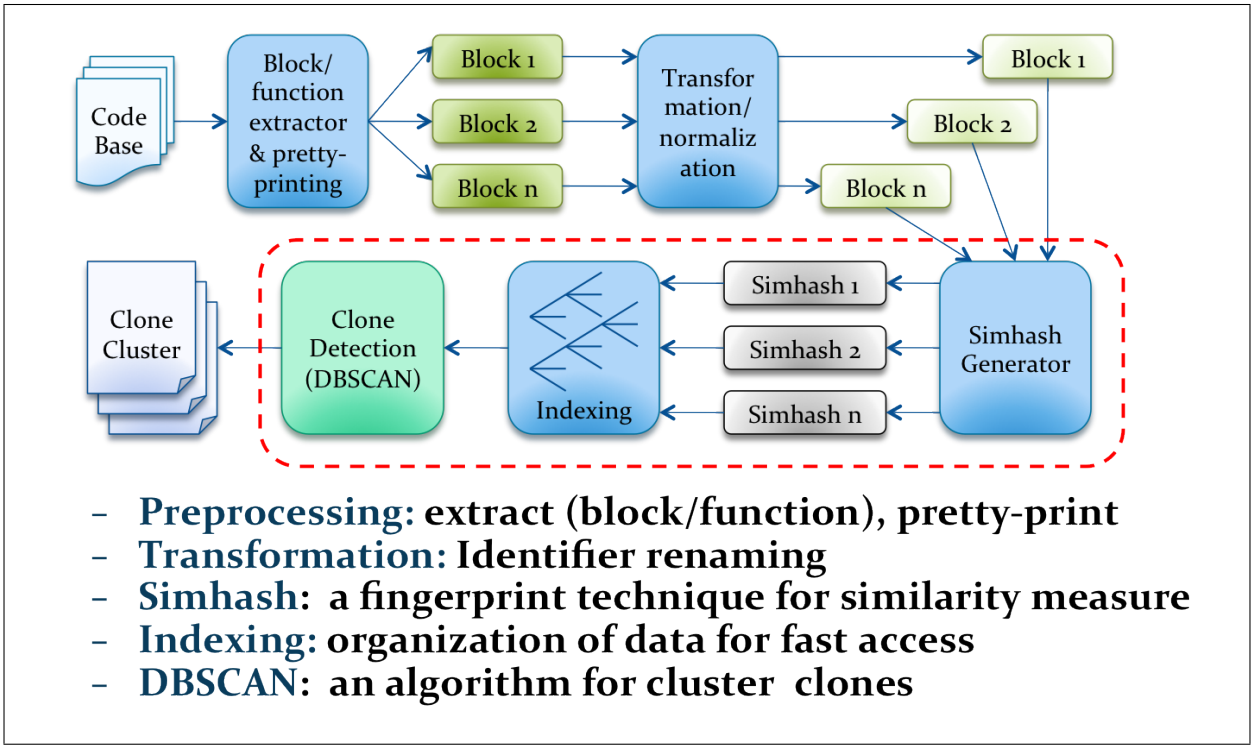


Figure 4.1: *SimCad*: Clone Detection Workflow

due to syntax-tree comparison and thus exhibit poor runtime performance in case of large scale systems. Recent AST based tool Asta [49] improved runtime performance by using structural abstraction of arbitrary subtrees of an AST, while JCCD [28] use parallel detection process that significantly improves the total detection time. Although, PDG-based approaches Scorpio [67], DUPLIX [105] can partly handle statement reordering, insertion, deletion and non-contiguous code, there runtime performance is not not good [141], and thus no scalable to large size programs [95].

In the previous study [172], simhash fingerprint based hybrid clone detection approach has been found effective in detecting both exact and near miss clones, especially for fast detection of clones in large scale project. Outcome of this study and the demand of having a fast & scalable near-miss clone detection tool motivated us to build a fully fledged standalone clone detection tool from the research prototype. The tool has been made highly configurable to be used for detecting clones in both source code and non-source code based data.

4.3 The *SimCad* Clone Detector

SimCad features a standalone clone detection tool evolved from our earlier research [172] where effectiveness of *simhash* (a similarity preserving data hashing technique) [33] has been studied for fast detection of clones in source code. The prototype implementation was found effective especially for quick detection of near-miss clones in large-scale software systems. Based on that prototype, a full-fledged clone detection tool has

```
<clone_group groupid="21" nfragments="2">
 <clone_fragment startline="160" endline="164" file="/src/draw/util/StorableInput.java">
  <![CDATA[
  private void map (Storable storable) {
    if (! fMap.contains (storable)) {
     fMap.add (storable);
    }
  }
  ]]>
 </clone_fragment>
 <clone_fragment startline="42" endline="48" file="/src/draw/util/StorableOutput.java">
  <![CDATA[
  private void map (Storable storable) {
    if (! fMap.contains (storable)) {
      fMap.add (storable);
    }
  }
  ]]>
 </clone_fragment>
</clone_group>
```

Figure 4.2: Sample Clone detection result in XML

been engineered that we present here as *SimCad*. It employs a data clustering algorithm with a multi-level index based searching that enables fast detection of clones. Figure 4.1 gives an overview of end-to-end clone detection process in *SimCad*. The whole process includes three phases named as: Pre-processing, Detection and Output generation. The pre-processing phase again consists of the following four sub-process: Extraction, Normalization, SimHash generation and Indexing; once done altogether, the result (index) can be reused as many times as needed to perform clone detection operation. Technical details of all the steps can be found in an earlier study [172].

SimCad is a structured clone detection tool. That is, it detects clones as code fragments (e.g., function or code block), the boundary of which are predefined during the source code pre-processing step. The tool provides both command-line and graphical user interface for its user. Detection outcome can be varied by providing input to a number of parameters exposed through the user interface. There is also an external configuration file named *simcad.cfg.xml* that provides more configurable parameters with their default values. User can modify these parameter values in order to fine tune the overall clone detection process. The clone detection result is exported as an XML file to a location specified by the user, otherwise to a system defined location relative to the given source location. An sample content of such XML file is shown in Figure 4.2. It also provides a XSLT[1] (Extensible Stylesheet Language Transformations) based visualization of clones in HTML5 compatible browser (Figure 4.3).

[1] http://www.w3.org/TR/xslt

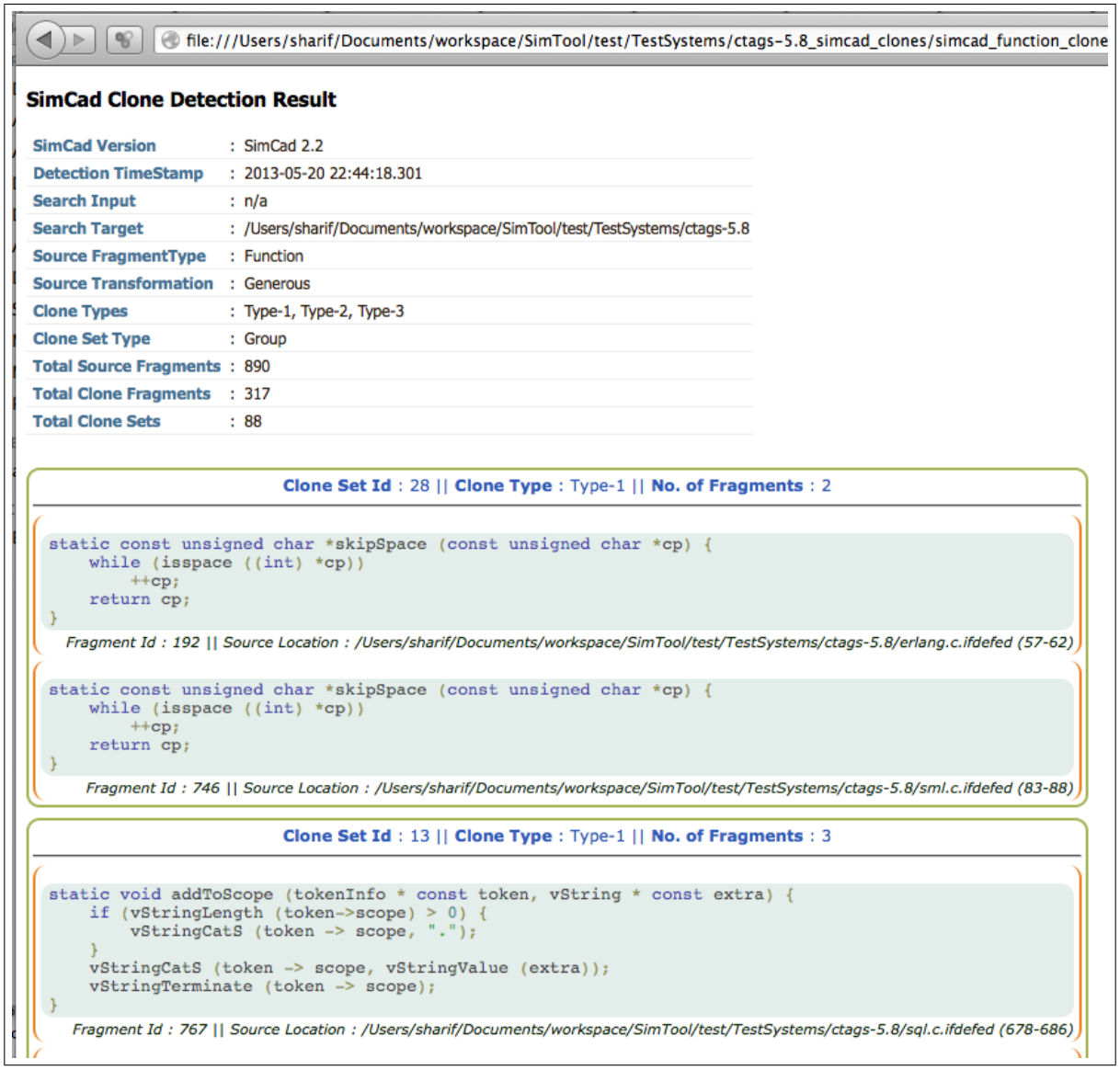


Figure 4.3: Visualization of clones detected by *SimCad* in HTML5 compatible browser

4.3.1 Command-line User Interface (CUI) of *SimCad*

SimCad provides the following two commands for detecting clones in a target system: *simcad2* (Figure 4.4a) and *simcad2xml* (Figure 4.4b). The command *simcad2* takes root folder of the subject system and source code language name of that system as two mandatory inputs. The language parameter is required by *SimCad* to use appropriate TXL[2] scripts for extraction and normalization of code fragments from original source during pre-processing step. Currently it supports four popular programming languages namely: C, Java, Python and CSharp. An example of detecting clones using the command *simcad2* is shown below for a project named 'dnsjava':

[2]www.txl.ca

```
$ ./simcad2 -help

Usage: simcad2 [-version] [-v] [-f] [-help] [-gx] [-i item_to_search] -s source_path -l language
               [-g granularity] [-t clone_type] [-c clone_grouping] [-x source_transform]
               [-o output_path]

 -version         : display simcad version
 -v               : verbose mode, shows the detection in progress
 -f               : force detection to discards previous pre-processed
                    resources if exist
 -help            : print this run instruction
 -gx              : display graphical user interface
 language         : name of the source language [ c | java | cs | py ]
 granularity      : source fragment type [ (block | b ) | (function | f) :
                    default = (function | f) ]
 clone_type       : types of clone to seek [ 1 | 2 | 3 | 12 | (23 | nearmiss)
                    | 13 | (123 | all) :  default = (123 | all) ]
 clone_grouping   : grouping of clones [ (group | cg) | (pair | cp) :
                    default = (group | cg) ]
 source_transform : source transformation strategy [ (generous | g) | (greedy
                    | G) : default = (generous | g) ]
 item_to_search   : absolute path to file/folder contains search candidates
 source_path      : absolute path to source folder
 output_path      : absolute path to output folder [ default =
                    {source_path}_simcad_clones ]
```

(a) Command *simcad2*

```
$ ./simcad2xml -help

Usage: simcad2xml [-version] [-v] [-f] [-help] [-gx] [-n] -s o_source_xml [-x t_source_xml]
                  [-t clone_type] [-c clone_grouping] [-o output_path]

 -version        : display simcad version
 -v              : verbose mode, shows the detection in progress
 -f              : force detection to discards previous pre-processed
                   resources if exist
 -help             print this run instruction
 -gx             : display graphical user interface
 -n              : non-exclusive fragments, a fragment might contain in
                   another bigger fragment
 o_source_xml    : absolute path to xml file containing original source
                   fragments
 t_source_xml    : absolute path to xml file containing transformed source
                   fragments
 clone_type        types of clone to seek [ 1 | 2 | 3 | 12 | (23 | nearmiss)
                   | 13 | (123 | all) :  default = (123 | all) ]
 clone_grouping  : grouping of clones [ (group | cg) | (pair | cp) :
                   default = (group | cg) ]
 output_path     : absolute path to output folder [ default =
                   source_path_simcad_clones ]
```

(b) Command *simcad2xml*

Figure 4.4: *SimCad* command line intefaces

```
<source file=".../file-1.x" startline="x" endline="y">
     fragment 1 text
</source>
...
<source file=".../file-n.x" startline="nx" endline="ny">
     fragment n text
</source>
```

Figure 4.5: Input XML file format for command *simcad2xml*

```
$ ./simcad2 -s /TestSystems/dnsjava -l java
```

The command *simcad2* has a special optional parameter *item_to_search*. This parameter takes a file or folder that contains source code to be searched in a target project. That is, the parameter points to some source code as search candidates and execution of the command goes for detecting codes similar to the search candidates in the target project. This opens up a number of interesting possibilities in code and clone search. For example, user might want to see if some arbitrary code (or anything similar) exists in a target project. In such case, user needs to provide the arbitrary code location as input to the parameter *item_to_search* (i.e., the search candidates outside the target project location) and then execute the detection command after setting the target project. So *SimCad* in such case essentially works as a source code search engine.

```
$ ./simcad2 -i /Documents/Snippet.java -s /TestSystems/dnsjava -l java
```

Another use case scenario for this command would be *localized clone search*. That is, detection is performed into a specific region of a codebase to see if that region contains any clone code with respect to the whole project. Using this command, user can get the service by proving the subset codebase location as an input to the parameter *item_to_search* and choosing the whole project as the target project as follows:

```
$ ./simcad2 -i /TestSystems/dnsjava/org/xbill/DNS/utils -s /TestSystems/dnsjava -l java
```

In the same way, this command could also be used to detect inter-project clones in two difference projects. In case the user ignores the parameter, detection will go for searching all possible clones inside the target project.

The next command *simcad2xml* takes source input through the only required parameter *o_source_xml* as an XML file of a pre-defined format [172] as shown in Figure 4.5. Optionally user can provide a similar XML file (through parameter *t_source_xml*) that contains normalized/transformed version of the original source data if available, which would yield better detection result for near-miss clones. This interface opens a great opportunity for *SimCad* to perform clone detection on any kind of textual data since unlike the previous command it is language independent.

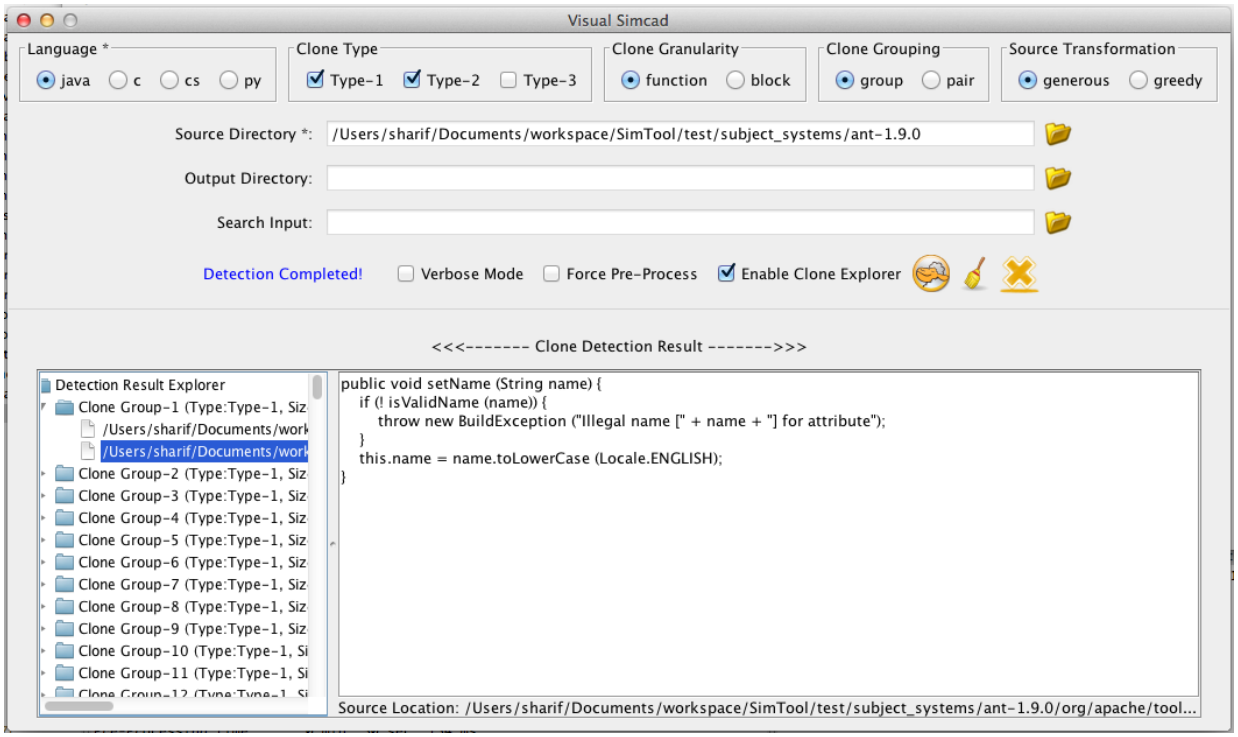


Figure 4.6: GUI corresponds to the command *simcad2*

4.3.2 Graphical User Interface (GUI) for *SimCad*

SimCad also provides GUI as a convenient way of using the tool for clone detection. Figure 4.6 shows the GUI corresponds to the command *simcad2* that can be made available using the following command:

```
$ ./simcad2 -gx
```

User can provide the other arguments here as well, but those will be propagated to the GUI as appropriate. Additionally, the GUI provides a simple clone explorer and clone code viewer (bottom part in Figure 4.6) that user can choose to display clones (optionally) while performing a clone detection on a subject system. Similar GUI is also available for the command *simcad2xml* as shown in Figure 4.7.

4.4 *SimCad* Usages

This study presents *SimCad* as a fully fledged hybrid clone detector for both source code and non-source code based data. Following scenarios illustrate the multi-purpose use of *SimCad*:

1. Large Scale Clone Detection: *SimCad* can be used to detect both exact and near-miss (function and block) clones from large systems of different languages. Currently, *SimCad* supports four languages (namely C, Java, Python and CSharp) for its first interface (Figure 4.4a, 4.6) while the second interface

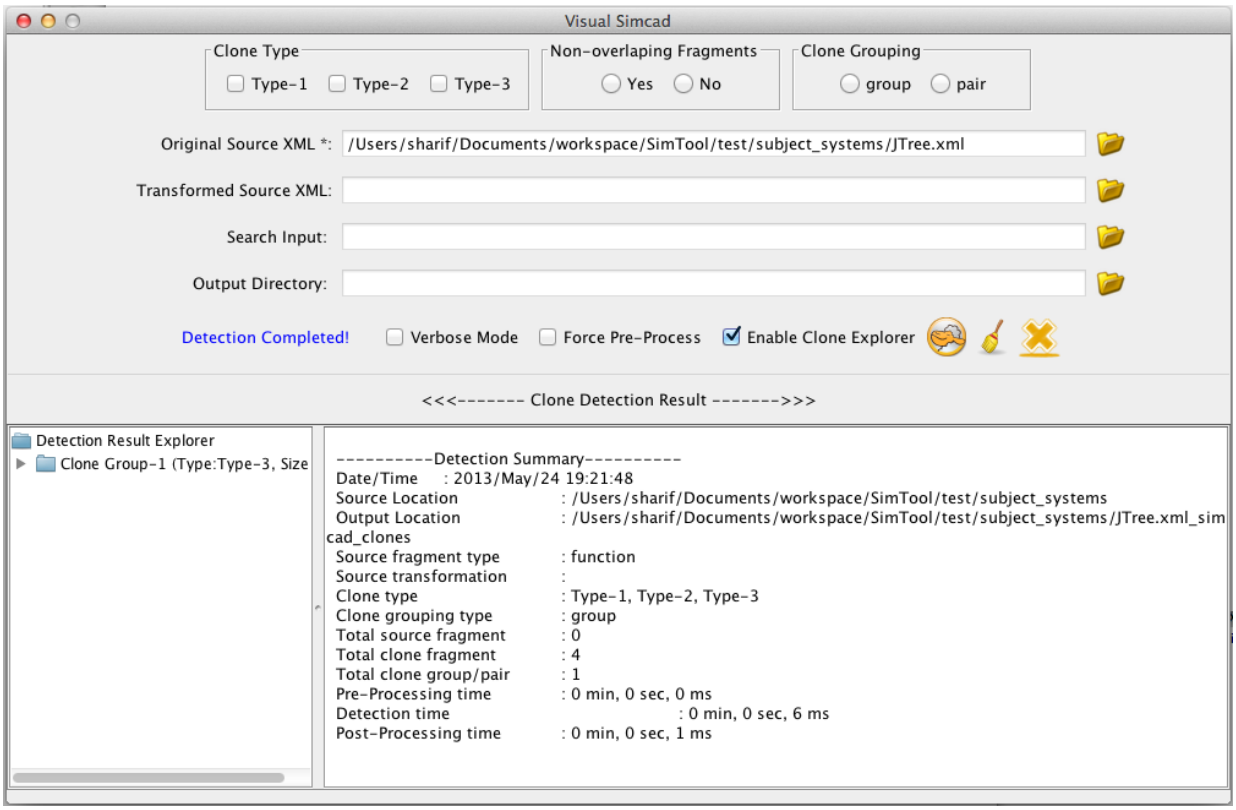


Figure 4.7: GUI corresponds to the command *simcad2xml*

(Figure 4.4b, 4.7) works language independently. For the second interface, input source code has to be provided in a pre-defined XML format.

2. Random Code Search: *SimCad* can be used as a code search tool. Given a code fragment as search candidate (Figure 4.4a, 4.6), *SimCad* can find all other similar fragments (including Type 3 fragments) in a given system.

3. Localized/Focused Clone Search: *SimCad* can effectively be used to check whether a particular region (e.g., a subfolder, file, a set of files and so on) contains any code clones with respect to the whole project. In this case, the given region works as a search candidate.

4. Cross-project Clone Detection: It is possible to detect cross-project clones using *SimCad*. In this scenario, one project is used as search candidate and the other project is used as target project where clone is sought. Here, the similar codes that exist in both the system will be reported as clone in the detection result.

5. Language-independent Similarity Detection: *SimCad* can be used not only for software code but also for any project artifacts such as the requirements or help documents by using its second interface (Figure 4.4b, 4.7). In order to do that, the user needs to create a XML file with target data according to the format shown in Figure 4.5 and then use this file as an input to the appropriate interface as

mentioned earlier. For all the scenarios above (e.g., clone detection, clone search, cross-project clone detection), this interface can use for any types of artifacts.

4.5 Summary

SimCad is a potential tool to be used in clone detection research and industry. It provides an easy to use command line interface as well as a convenient graphical user interface (GUI) where a user can explore the detected clones in a subject system. Using it, clones can be searched locally or in the whole project. Besides, it can work as a code search engine as well. In this chapter, we have demonstrated various features of *SimCad*, explained its different interfaces and outlines multi-purpose use of it in the area of code similarity detection. *SimCad* has been published [171] as a versatile tool for detecting clones in diverse situations as well as for both source and non-source code based data. It is also used in a recent study by Keivanloo et al. [92] to detect clones in Java bytecode. It is also being used in our research lab for several other ongoing studies.

Chapter 5

SimLib: A Customizable API for Portable Code Clone Detection Service

5.1 Introduction

Applying an existing clone detection tool on data with diverse characteristics and having limited configurability support in such a tool limit the applicability of the tool for both research and industry. In most of the software clone research, clone detection is the first step prior to doing further analysis. Since most of the popular tools used in the clone research community are standalone, in order to develop a tool for clone analysis studies, researchers need to obtain the clone detection results first, typically as text or XML files. Programs then need to read those files to do further processing. Creating such a program or tools using loosely coupled components (e.g., a standalone clone detection tool) is inconvenient and has the added difficulty of having to deal with different clone output formats. In order to avoid such intermediate clone data processing and to provide a portable and robust clone detection solution, *SimLib*, a customizable and scalable clone detection API is developed. This API offers a convenient solution by providing a broad set of configuration facilities to meet different detection requirements. As well, the library can be integrated with other clone management tools so that they can directly access clone detection results from memory to avoid the overhead of getting the results from the output of a standalone tool through a 'post-mortem' approach [111].

5.2 Motivation

Over the past decades, clone detection based study happens to be a highly active area in software clone research. The research trend in the area in recent time is mostly on determining what to do with the clones when they are detected. A significant portion of the research includes but is not limited to finding ways to manage clones, gaining more control of their generation and studying clone evolution and their effects on the evolution of software.

However, as mentioned in previous section, service portability is one of the major issue in use of currently available state of the art standalone tools in clone analysis and management. Since clone detection is one important pre-requisite of any clone based research, the API *SimLib* has been developed as a convenient

solution in providing off the shelf clone detection functionality where needed. The API is targed to provice the following benefits.

- On the fly clone detection
- Re-detection / import-export clone index
- Incremental clone detection
- Direct availability of detection result/no intermediate file processing
- Custom clone filtering
- Custom process injection, for example compiling some analytical report based on the detected clones
- Ready-made clone detection service
- Modular development architecture
- Java based API, portable to any platform
- Easy integration to other tools
- Highly customizable to meet diverse detection requirement
- Extensible by user code integration

5.3 Existing clone detrection API

Deissenboeck et al. [44] developed ConQAT as an integrated framework to detect clones in Simulink/Matlab models especially in automotive domain. Their clone detection module CloneDetective [84] is available for use as an API, which is part of the ConQAT framework, works by representing the model as a normalized multi-graph where labels are assigned to relevant blocks. Biegel and Diehl [28, 27] introduced a novel way for fast and configurable code clone detection using pipelines. They developed JCCD, a flexible and customizable AST based clone detection tool in which several cascaded processors perform various steps of clone detection process. JCCD API parallelizes the detection process using multiple cores.

5.4 *SimLib* Architecture

The *SimLib* application interface has been designed on top of a highly modular architecture that facilitates its extension and use in a fully customized clone detection system including a very fast clone search that can be used for detecting clones in variety of structured and non-structured data. As shown in Fig. 5.1, it contains a 3-layer architecture: *(1) Pre-processing*, *(2) Detection*, and *(3) Post-processing*. Each layer contains number

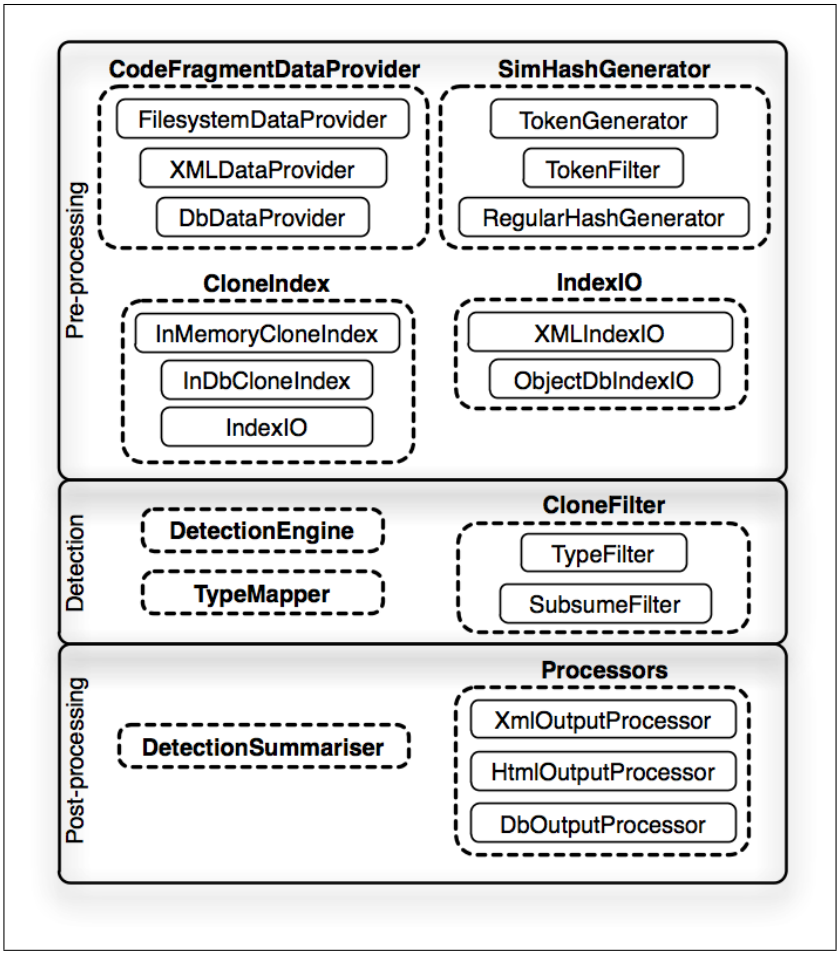


Figure 5.1: Architecture of *SimLib*

of configurable components where the user can choose different options from within the library or can create custom components and link them with the core system using an external *SimLib* configuration file to address different needs. Components in each layer are discussed as follows:

5.4.1 Pre-processing

This module covers processing of the detection input source code as collection of soure fragments, generation of *simhash* fingerprint for each fragment and finally organize the *simhash* values using an indexing strategy. It covers the following APIs.

CloneDataProvider

This is an interface to data provider for managing raw data for clone detection from different data sources. Following implementations of CloneDataProvider are available in the API.

i. FileSystemDataProvider: Here the data source is standard source code file. The implementation uses

a TXL[1] based parser to normalize, extract and transform raw source code to source code fragment at function or block level. Because of using a parser, this implementation is language dependent and currently supports four popular programming languages, namely: c, java, c-sharp, and python.

ii. XMLDataProvider: Here the data source is a standard XML file of a pre-defined format. The XML file consists of source code fragments along with their location information in the original source file. This implementation is language independent and can be used for both source code or non-source code based data.

iii. DBDataProvider: This is an equivalent implementation of XMLDataProvider except that the data source is a database instead of XML filesystem.

SimHashGenerator

This class implements the *simhash* algorithm outlined in Figure 3.2 that is used to generate *simhash* value for each sosurce code fragment. Following helper interfaces are used in this algorithm implementation. The *SimLib* API provides some default implementation of these helper interface. However, at runtime, the API allows user to hook-up their own implementation of these interface for a versatile execution of *simhash* algorithm suited for various types of data.

i. TokenGenerator: A source code fragment need to be tokenized while generating simhash value for it. This interface is used to implement different token generating strategies depending on source data characteristics. Additionally, a TokenGenerator can take a list of TokenFilter based or source data characteristics.

ii. TokenFilter: This is an interface to implement various token filters. Token filters are used to accept, discard or transform tokens used by TokenGenerator during *simhash* generation.

iii. RegularHashGenerator: The algorithm of simhash requires user of some regular hash function for each data token. This interface is used to implement various types of 64-bit hash generators for each token provided by a TokenGenerator.

CloneIndex

Once the *simhash* value for each source fragments are generated, the values are organized in an indexing structure (Figure 3.4) to facilitate fast searching of similar code fragment. This interface is used to implement a 2-level index of source data. Following two implementations are available in the API.

i. InMemoryCloneIndex: Interface to implement clone index in runtime memory. The library contains three implementations based on the collection frameworks from Java, Apache and Google.

[1] www.txl.ca

ii. InDbCloneIndex: Interface to implement clone index in database in support of detection of clone in large-scale data and implementation of instant clone search.

IndexIO

IndexIO interfaces is used to import and export clone index. Using implementation of this interface, clone index can be saved and reused as many time as needed, which facilitates faster re-detection and incremental clone detection.

i. XMLIndexIO: Interface to implement clone index in runtime memory. The library contains three implementations based on the collection frameworks from Java, Apache and Google.

ii. ObjectDbIndexIO: Interface to implement clone index in database in support of detection of clone in large-scale data and implementation of instant clone search.

5.4.2 Detection

This module receives a clone index and detection input from the previous module and performs the clone detection process. It covers the following APIs.

DetectionEngine

This class implements the clone detection algorithm, which is mainly the data clustering algorithm as outlined in Figure 3.5. The implementation is build on an *Observer* design pattern where the observers of the detection process are being notified about different runtime state or event, for example, current status of completion, detection of a new clone group, etc.

CloneFilter

Interface to implement various clone filters to discard uninterested clones during detection process. The library includes following two such filters.

i. TypeFilter: This filter is used to discard a particular type of clones from the detection result and keep those for which users are interested in.

ii. SubsumeFilter: This filter is particularly used during the detection of block clones to discard the clone fragments that are subsumed by some bigger fragments (the former clone fragment is fully contained in the later one).

TypeMapper

Once a clone group/pair is fully discovered by the detection algorithm, TypeMapper class is used in defining the type of the clone group/pair.

```
----------Detection Summary----------

Date/Time                : 2013/May/20 19:35:00
Source Location          : /TestSystems/junit-4.11
Output Location          : /TestSystems/junit-4.11_simcad_clones
Source fragment type     : function
Source transformation    : generous
Clone type               : Type-1, Type-2, Type-3
Clone grouping type      : group
Total source fragment    : 387
Total clone fragment     : 110
Total clone group/pair   : 47
Pre-Processing time      : 0 min, 2 sec, 963 ms
Detection time           : 0 min, 0 sec, 77 ms
Post-Processing time     : 0 min, 0 sec, 5 ms
```

Figure 5.2: Clone Detection Summary

5.4.3 Post-processing

This module accepts detection results from the detection module and does some additional processing on the result as needed. This includes generation of a clone detection summary and exportation of the detection result in several output form for various purpose, for example: persistent storage, visualization, etc. It covers the following APIs.

DetectionSummariser

This class is used to generate a complete detection summary of an invocation of clone detection using this API. An example of such detection summary shown by this class is shown in Figure 5.2.

Processors

Interface to implement classes for various post-processing activities to be performed on clone detection results. The library contains several clone data output processors for different output strategies, for example:

i. XmlOutputProcessor: This processor writes Clone detection results as an XML file.

ii. HtmlOutputProcessor: This processor writes Clone detection results as an HTML file.

iii. DbOutputProcessor: This processor writes clone detection results as a database. We call it '*zero configuration database output*' since the library itself will create the predefined schema in a database and will write the clone detection results to it. Using Hibernate[2] ORM, the library provides a variety of database options for the user both in embedded and client/server mode as shown in Table 5.1. *SimLib*

[2]www.hibernate.org

Table 5.1: Database selections for DbOutputProcessor

Database	Source	Embedded	Client-Server
H2	www.h2database.com	Yes	Yes
DerbyDB	http://db.apache.org/derby	Yes	Yes
HSQLDB	http://hsqldb.org	Yes	Yes
MySQL	www.mysql.com	N/A	Yes
PostgreSQL	www.postgresql.org	N/A	Yes

provides an external configuration file *hibernate.cfg.xml* from which the user can select a particular database and provides the credentials to access the database as required.

5.5 *SimLib* Modification and Extension

SimLib provides options to modify its default behavior or to add new features in its pre-processing and post-processing layers (Fig. 5.1). Using such customization facilities, a user can adapt *SimLib* for clone detection on data with different characteristics, add functionality to process the clone detection results, and its even possible to develop a full customized clone detection tool to target processing a particular type of data. The modification and extension options are discussed in the following subsections.

5.5.1 Extension Points

SimLib provides three extension points where users can plug in their own code to be executed during the detection process. What users need to do is to write their own implementation of the corresponding interface and point the class name in the *SimLib* external configuration file *simcad.cfg.xml* through the appropriate property key. At runtime, the user provided code will be executed in place of those in the library.

TokenGenerator

An implementation of the interface *ITokenGenerator* takes a data fragment and builds a list of tokens. A user can define the boundary of a single token based on the target data and implement that strategy in their own implementation because a single strategy might not work for all types of data. Such a strategy could be based on a single line, a sequence of characters separated by whitespace characters, a fixed length character sequence, a fixed length sub token, a tokenization by special characters (e.g., comma, period), and so on.

RegularHashGenerator

The detection algorithm used in *SimLib* is based on *simhash* [33]. In order to generate *simhash* for a data fragment, a regular data hashing[3] mechanism is required to convert each token generated from that fragment. By default *SimLib* uses an implementation of JenkinHash[4] for this purpose. User can use an alternate algorithm by implementing the interface *IRegularHashGenerator* if the default one is not considered good for the target data.

CloneIndexHolder

The interface *ICloneIndex* defines the multi-level Clone Index [172] that provides the context for the detection process. The library provides three different implementations based on collection frameworks provided by Java, Google and Apache; the default is set to Java. A user can choose from the alternate implementations or use their own.

5.5.2 Programatic Manipulation

IFragmentDataProvider

The detection algorithm works on pre-defined source fragments extracted during its pre-processing step (Figure 5.1). *SimLib* provides a group of classes to implement the interface *IFragmentDataProvider* for extracting fragments from source data. It contains a basic implementation of the interface *IFragmentDataProvider* named *XMLSourceFragmentDataProvider* that accepts an XML file in a pre-defined format (Figure 4.5) containing marked up source fragments. For clone detection in a target dataset, users need to organize the source data fragments into the predefined XML format. *SimLib* also provides an advanced Data Provider implementation named *FileSystemFragmentDataProvider* that takes a filesystem location of a dataset and generates an XML file from the source code using TXL[5]. A user can develop their own implementation of *IFragmentDataProvider* or modify an existing implementation that can extract fragments from the target dataset or generate the XML file of source fragments from the target dataset through other means and then feed it to *XMLSourceFragmentDataProvider*.

IProcessor & ProcessorDispatcher

SimLib's post-processing tasks are performed through 'processor': an implementation of the interface *IProcessor*. Processors are added to a *ProcessorDispatcher* to coordinate their execution. Processors perform activities, such as : generating a detection summary; writing a log file for the detection process; and, writing a detection result to an XML file or a database. For example, in the sample code snippet shown in Figure 5.3,

[3] http://en.wikipedia.org/wiki/Hash_function
[4] http://en.wikipedia.org/wiki/Jenkins_hash_function
[5] www.txl.ca

```
1. IProcessor xmlOutputProcessor = new XmlOutputProcessor( detectionSettings, output_dir,
       "Writing xml file..." );
2. IProcessor logWriter = new DetectionSummaryPrinter( detectionSettings, logPrinter,
       "Writing log file..." );
3. IProcessor consoleWriter = new DetectionSummaryPrinter( detectionSettings, consolePrinter );
4. ProcessorDisptacher.getInstance().addProcessor( xmlOutputProcessor ).addProcessor( logWriter )
       .addProcessor( consoleWriter ).applyOn( detectionResult, detectionSettings );
```

Figure 5.3: Example execution of IProcessors in post-processing.

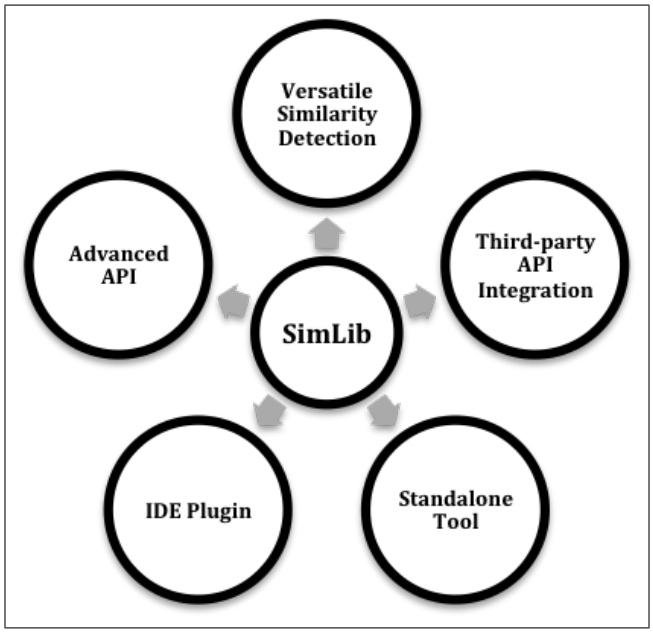


Figure 5.4: Different uses of *SimLib*

the statements 1, 2 and 3 use an implementation of three different processors used to write detection results in XML file, write a log file for the detection process and print a detection summary in the console respectively. Users can also write their own implementation of a processor for a particular post-processing task and add it to *ProcessorDispatcher* to execute it.

5.6 Target Use

SimLib has a great potentiality to be used both in industry and research as a configurable and ready-made clone detection service provider. Figure 5.4 shows some multi-dimensional use of *SimLib* in providing clone detection facility. Following are high-level overview of *SimLib*'s target use.

Third-party API Integration:

Both in academia and industry, researchers and developers develop various types of clone analysis,

visualization or management tools. Clone detection service can be considered as a basic requirement for any such clone based software. *SimLib* can be great tool in this respect for providing portable and customizeable clone detection service where needed. Therefore, integration of *SimLib* as a 3rd part API with other clone based applciation would be a convenient way to provide on the fly cone detection support.

Advanced API:

In order to make a software successfull in practice, it needs to be updated continuously by adding new features and or enhancing capabilities. One important aspect of API design is the capability of its future extension with minimal modification of existing core components/architecture. Because of having a moduler architecture, *SimLib* can easily be modified and/or extended by incorporating additional feature and thus enhancing its overall performance and capability. Therefore, it would be possible to developed more advanced clone detection API based on *SimLib*'s core architecture.

IDE Plugins for clone management:

Since clones are evolved mainly because of various development activities of developer, integration of clone detection and analysis facility within software development environment can provide better support in clone management in software. Towards this goal, *SimLib* can be used to develop an IDE plugin intended towards on the fly clone detection, visualization and management.

Standalone Tool for Clone Research:

In software engineering study, researchers frequently use or develop various standalone tools to generate or process data for further analysis. In clone based study, *SimLib* could be a useful library in developing such tools. *SimLib* can be customized for detecting clone in variour types of textual of data. Therefore, development of a standalone clone detection tool for that target data using *SimLib* would be faster and convenient.

Versatile Similarity Detection API:

The primary target of *SimLib* is to be used as a portable library for source code clone detection in software projects written in popular high level programming language like C, Java, etc. However the similarity detection capability can be tailored and extended to be used for multi-purpose needs, for example, detection of clones in intermediate source code (java/.net byte code) or non-source code data like code documentation, source code search engine, plagiarism detection in source or non-source code based data, etc.

5.7 Evaluation on *SimLib* Installation, Integration and Use

A user study based evaluation on *SimLib* API has been performed to gain some insights about the operation and target use of the API. In this study, the participants get to install, integrate and use the API through a

demo program and at the end they were required to answer some questions regarding their experience using the API and provide their opinions on improvements and future use of the API.

5.7.1 Study Design

A set of tasks was designed to have the participants install *SimLib* using the documentation available in public site and integrate the API to a demo program that can call and execute clone detection methods from the API. Once installation and integration is done, they were instructed to execute the demo program with varying detection settings using the options available in both the API and the external detection configuration file. After performing these tasks, participants were requested to answer a set of questionnaires capturing their experience on integration and use of *SimLib* as an API.

Questionnaire Design

For this study, the following two sets of questionnaire have been designed.

Questions on API Installation and Use: This set of questions is to capture overall user experience of the tool operation. User's feedback on each question was captured based on a five level likert scale containing options as: *Strongly agree, Agree, Neutral, Disagree and Strongly disagree.*

Q1: Do you think the API installation/integration/use was simple and easy?

Q2: Do you think the API documentation for installation/integration/use was helpful?

Q3: Do you think the API would be useful in detecting clone from data with various characteristics?

Q4: Do you think the API would be useful in providing clone detection support as a 3rd party tool?

Q5: Do you think the configurability and extendibility feature made the API more versatile in clone detection?

Q6: Did you find the tool satisfying overall?

Issues, Target use and Future Improvements Questions: This final set of questions is to identify the issues that the participants felt while integrating and using the API and find the scope for further improvements in their opinion.

(a) What whould be the three most important target uses of *SimLib*?
Option: 1) Third-party API Integration, 2) Advanced API, 3) IDE Plugin, 4) Standalone Tool and 5) Versatile Similarity Detection

(b) What difficulty did you face while using the tool?

(c) What portion of the tool you think requires improvement?

(d) What additional features do you think should be implemented in the tool?

Feature based Task Design

The following tasks has been defined to guide the user from installing to using the API as a developer.

T1: Download and install pre-requisites of *SimLib*.

T2: Setup/Install the sample *SimLib* integration project.

T3: Change and execute the demo program 'CloneDetectionDemo.java' as shown in integration documentation for the given project source code.

T4: Execute the demo 'CloneDetectionDemo.java' with varying detection settings for type and transformation.

T5: Customize the detection behaviour by changing external detection configuration parameters in 'simcad.cfg.xml'.

T6: Change and execute the demo 'CloneDetectionDemo.java' for the given non-source code data (code documentation) provided in XML file.

Runing Study Session

This is a one-to-one study session that allows the participants to do a set of task towards using the API and answer the related questions afterwards. They were given introduction about the nature and purpose of the study. Then they were briefed about the different features of *SimLib* API and the motivations behind it. The participants were then allowed to perform the tasks specified in previous section.

After the demo session the questionnaire was handed to the participants and the participants' opinion based on their level of agreement on the performed tasks were recorded. Answering the 'Overall Evaluation Question's was an open discussion session and the participants had the option of asking questions regarding any difficulty they faced during the task completion and knowing their views in further improvements of the API.

5.7.2 Summary of Findings

User Experience on API Installation and Use

We had 5 participants in this study each of them having at least three years of software development experience, have basic knowledge of software clones and have experience of using clone based tools and technologies in academic research. Participants' feedback on the User Experience questions are analyzed and presented in Figure 5.5 with separate component for each question. From the feedback chart, it is clear that the *SimLib* got most of the remarks on the positive side of the likert scale from the study participants on all the evaluation questions. While 60% of the participants found the API overall satisfying, the rest being found neither satisfying not dissatisfying and advised valuable suggestion for further improvements.

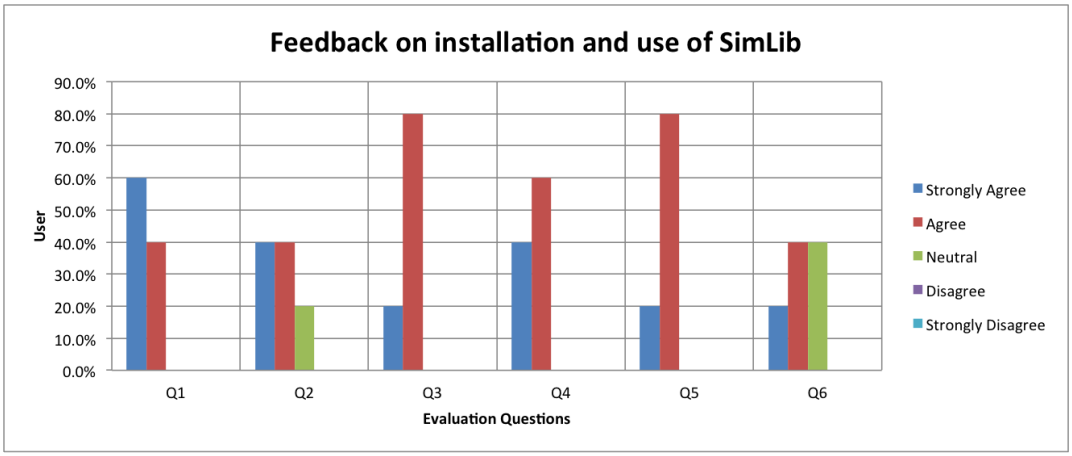


Figure 5.5: Consolidated user feedback on *SimLib* API use

At the end of the study session, the *Issues, Target use and Future Improvements Questions* were enabled us to get some advise on how can we improve the API for convenient and versatile use. We got valuable suggestion on existing feature improvements, specially for support of more programming languages and use of source code transformation through integrated API instead of loosely coupled external tool. For additional features, notable requests we got were: generic way to adapt source transformation for any programming language, similar API in other language as the one currently available in Java. To investigate the users' opinion on the possible target use of *SimLib* API, we asked them to choose best three options that they thought *SimLib* API could be used for. The distribution of their choice has been presented in Figure 5.6, which shows the use of *SimLib* API for *Third-party API Integration* has been choose by 100% for the participants, followed by 60% of the participants supported for *Advanced API* and *IDE Plugin*. This user feedback shows, *SimLib* could be a potential library towards providing support for clone/similarity detection needs of other applications.

5.8 Summary

Portability of clone detection service is one of the major usability issues of standalone clone detection tools. The clone detection library *SimLib* helps to overcome limitations of minimally configurable standalone tools, such as *SimCad*. *SimLib* allows its default behaviour to be modified through a higher level of customization. For example, *SimLib* can be configured for detecting clones in software source code as well as it can be configured for detecting clones in non-source code based documents, where most of the existing tools might not be a good fit. We performed a user study based evaluation on *SimLib* to get some feedback on installation and use of the API and to uncover its potential use as a third-party clone detection library. The user feedback we received from the study indicates its potential use as a third-party clone detection library towards providing support for various clone/similarity detection needs to the other applications. Beside, considering the operation benefit as mentioned in Section 5.2 and by addressing the user feedback on further

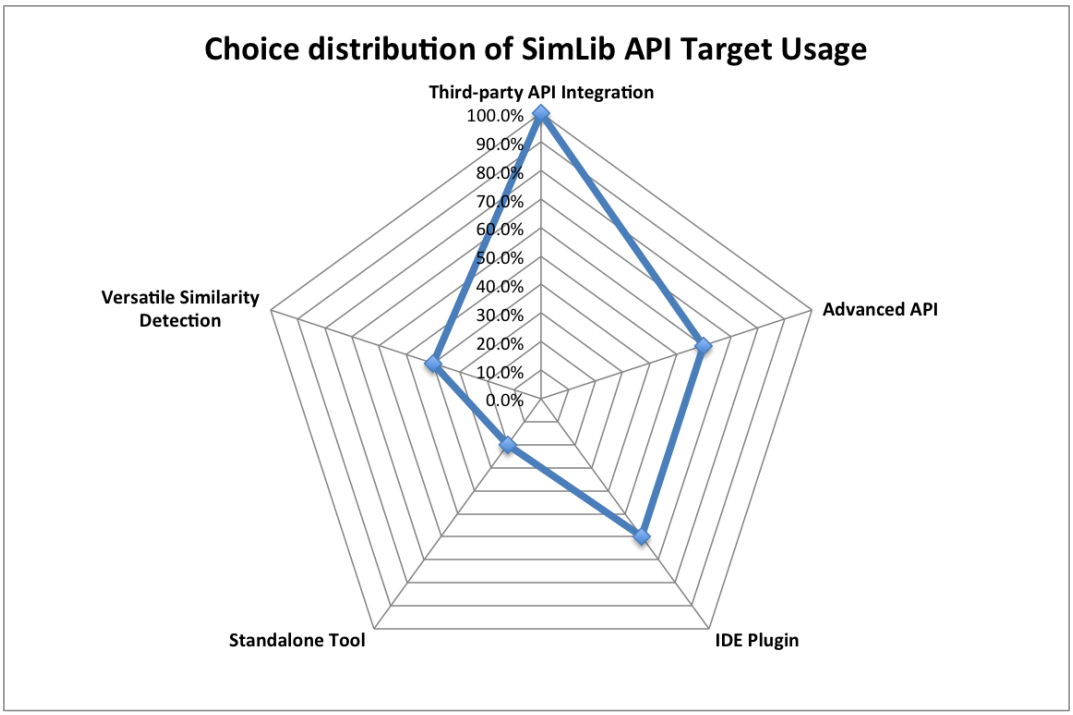


Figure 5.6: Distribution of top three target uses for *SimLib* API chosen by the study participants

improvements with a regular API maintenance, *SimLib* could be a viable tools in software clone based research as well as in industry.

Chapter 6

SimEclipse: Towards Managing Code Clones in Software Development and Evolution

6.1 Introduction

The research described in this chapter focuses on the activity of maintaining clones in code from a software developers' perspective. Göde [56] found that most of the clones detected by state of the art clone detectors need not be removed. The study has significant implications from the point of view of software maintenance. Besides it is problematic to identify consistently changing clones over time. Thus a clone tracking and awareness tool is essential to assist software developers in efficient maintenance of software. The chapter covers IDE integration of state of the art techniques for detection, visualization and tracking of clones in software with a focus on preventive clone management. In addition, the study also considers integration of novel techniques in clone based research, for example, extraction and visualization of clone genealogy in IDE to assist in the process of clone comprehension by the software developers. The overall idea of supporting clone management, is gathering some clone management primitives under the same hood where clones gets evolved. As a result, we are interested in addressing the following three research questions:

RQ1: What clone based technologies (tools and techniques evolved in clone study) can be grouped together in support of better clone management?

RQ2: How effective can clone management be when the clone based technologies are brought together under the same platform?

RQ3: Could an IDE be the preferable platform for integrated clone management functionalities rather than having the same functionalities as a standalone tool?

In order to answer some of the research questions, we conducted a user study based on a prototype IDE plugin with integrated clone management features. In this chapter, we also present that prototype named *SimEclipse* - an Eclipse IDE plugin to provide developer support for detecting, visualizing and tracking clones in projects being developed using Eclipse IDE. It uses a fast source code similarity detection algorithm we previously studied that makes it possible to implement a real time clone detection and management system for exact and near-miss clones in large-scale software systems. In support of understanding the evolution of

both clone and non-clone source code in a multi version software system, *SimEclipse* also provides source code evolution history viewer and clone genealogy viewer. We show how *SimEclipse* can support a software developer in dealing with clones and be integrated with a typical software development environment.

6.2 Motivation

No matter what the cause, in most cases it is the developer's activity that directly (e.g., by copying, pasting or modifying code) or indirectly (e.g., by using code generation programs or tools) introduces cloned code in a system. It is very hard to manually track, especially in large systems, whether those activities unknowingly cause clones. Convenient access to clone information from within the development environment is a key factor in managing clones. Because, without an efficient use of clone information it is also hard for a developer to efficiently manage clones in the system (e.g., refactoring/removing cloned code) or to reduce their negative effects during software evolution. Even if the knowledge of clones in a system is available (e.g., using a standalone clone detection tool), in most cases a developer needs to apply that knowledge by manual investigation and identification inside the working codebase to locate and process the clones.

From a developer's perspective, there is a gap between state-of-the-art software clone management support tools and software development environments. Besides, manual investigation of clones in system would make the task of clone management both challenging and inefficient. In support of code clone management, a number of standalone tools are proposed in the literature and are available for use in clone detection, visualization, analysis, etc. However, using these standalone tools for managing clones in evolving software might not be practically useful as opposed to having the similar facilities directly from within a software development platform (for example, IDE) as some integrated features. A growing need for further research towards an integrated clone management system is also addressed in a recent study by Roy et al. [152]. Towards that goal, we developed *SimEclipse*, an Eclipse IDE plug-in to assist developers in managing clones in the softwares they are developing/maintaing using one of the most popular software development IDEs, Eclipse. The current implementation of *SimEclipse* would assist clone managenemt by providing support for:

1) Just-in-time clone detection to identify the presence of cloned code in whole or any part of the project, e.g., in files, directories, etc.

2) Visualization of the clones detected in the project, linking them to their code location and, if interested, comparing clones side-by side from within the IDE.

3) Mark/Annotation of clones in the editor for easy and convenient access to existing clones throughout the project.

4) Automatic tracking of changes made in the code and notifying the developer if a change introduces cloned code; that is, a clone-aware development environment.

5) Inspection of code code change history for both cloned and non-cloned code

6) And, finally detection and visualization of clone genealogy

6.3 Clone Management

Clone management is a cross cutting topic concerning different domains of software clones. As an umbrella activity it covers various aspects regarding clones including but not limited to clone detection, classification, visualization, evolution analysis, tracking, refactoring, etc.

Current studies show various benefits of clone management including improved customer satisfaction, improving the quality of the system with a positive impact on maintainability of the software [108, 90]. Moreover, clone management in an Integrated Development (IDE) found effective in increasing clone awareness and understanding the evolution of duplicated code over time [75, 93].

6.3.1 Management Strategy

Clone management summarizes all process activities which are targeted at detecting, avoiding and/or removing clones [53]. Therefore, there are essentially two mainstream strategies of managing clones:

Preventive Clone Management featuring avoidance of introduction of clones, or being aware of the existing unavoidable clones and understand their evolution to avoid their harmful effects. In practice, clones are unavoidable and thus expectation of a clone-free system can be unrealistic. Therefore, preventive clone management might also termed as *proactive* clone management [72] that aims to deal with the clones during their creation or anytime soon after they are introduced in the system.

Corrective Clone Management featuring refactoring/removing clones after they are detected. While it might not be possible to remove/refactor all the clones in the system especially the near-miss clones, but exact clones (Type-1) are typically good candidates for being refactored.

Figure 6.1 shows a high level overview of clone management workflow. The figure also shows two clone management strategies mentioned earlier covering various clone management primitives.

- *Detection:* To manage clones in a system the very first requirements is that clones need to be detected. Although clone detection is a fundamental requirement to a clone management strategy, this feature might not always be integrated into a clone management system. In case where this feature is not present natively, detection results from third-party standalone clone detectors are used. Such feature dependency, or in another way, lack of native detection support would make a clone management system less interactive to the developers and also might be incapable of supporting other features like focused/on-demand clone search, clone tracking, and so on. Therefore, a versatile and user-friendly clone management system could be benefited from having its own clone detection capability rather than depending on some external detectors.

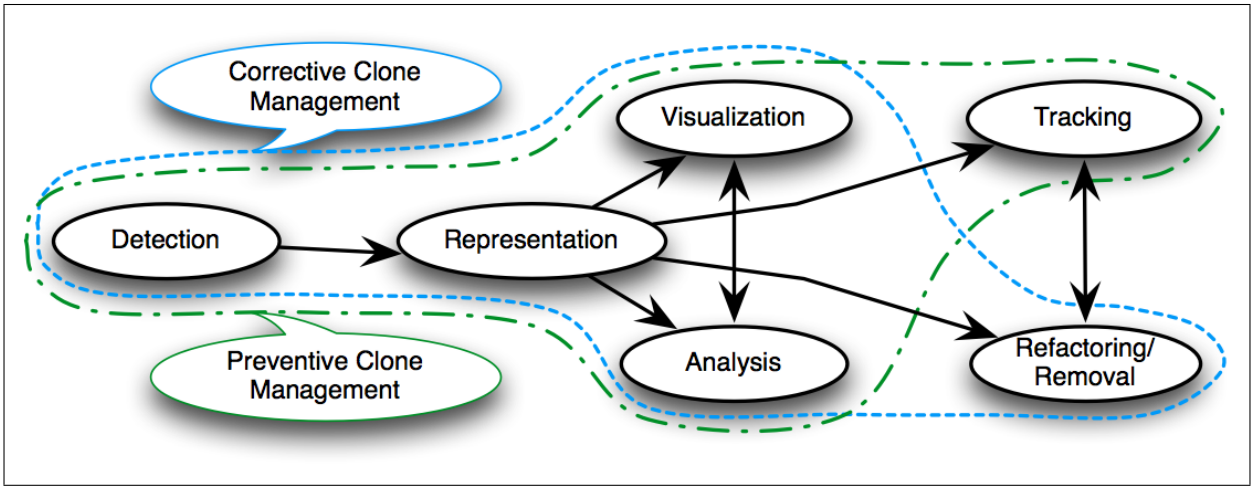


Figure 6.1: Clone management workflow

- *Representation/Documentation:* This feature captures and stores the result of clone detection in the form of clone documentation that records the location of code segments and their clone relationship. Once created, it can be re-used as many times as needed to get the clone information of the system without invoking the detection process. The clone documentation may be analyzed to determine justification of clones or to find potential clones for removal.

- *Visualization:* Visualization is a powerful technique that can aid understanding and analysis of clones after detection. This feature may offer various types of clone visualization with (for example, tree-view) or without (for example, scatter-plot, tree-map, etc) using internal features of the IDE. In most of the visualization approaches, developers can navigate to the actual clone code in the system and possibly compare the clones in the same group to see the differences.

- *Analysis:* This module analyzes the clones in the system and provides various insights about the cloning status of the system. Such analysis may include but is not limited to various statistical analysis featuring types of clones, clone density in various regions in the system or historical analysis, for example, extract and explore the evolution clones a software system having versionized source code (throughout the source revisions or releases in their development lifecycle). Besides, analysis could be performed to determine clone candidates for refactoring with the purpose of removing duplication. Therefore, this module basically helps in understanding various aspects of the clones in the system, which in turn helps developers in managing those.

- *Tracking:* This module provides clone aware functionality in a clone management system. An evolving software system goes through frequent changes in the code-base. Such changes may introduce new code segments that might form new clones or invalidate some existing clones in the system. Both situations require corresponding update in clone documentation that can be supported through this feature. It tracks existing clones in the system and follow their evolution throughout the lifecycle of the software.

Besides, it looks for source code changes and notifies developer if new clones introduced in the system because of the changes made.

- *Refactor/Removal:* A number of patterns in the general software refactoring patterns [51] are found to be suitable for clone refactoring, as suggested by earlier research [65, 184]. This module allows developers to delegate the code structure changes to the refactoring engine of IDE and reduces errors that may occur if changing the code was done manually. However, the clone refactoring step of feeding the clone information to the refactoring engine is still required as a manual process.

6.3.2 Dimensions of Clone Management

Most of the clone detectors [105, 164, 91, 78] are implemented as standalone tools and typically search for all the clones in a given code-base. Although, such tools can help clone management in a post-mortem approach, from a management perspective, this approach is not too convenient in practice. Researchers and practitioners [62, 71, 83, 84, 132, 154, 204, 205] believe that clone management activities should be integrated with the development process to enable effective clone management. Giesecke [53] addressed the following dimensions of clone management offered by a software tool as a Clone Management System.

Operational Structure

Most of the software systems are implemented in teams where members of the teams work in collaboration on different, possibly overlapping parts of the system. The environment of such teamwork can be characterized as a distributed system with local, decentralized software development environments for team members and a central source repository, which often provides configuration and version management functionalities. Therefore, a clone management system can be implemented either at the central repository (Centralized Architecture) or at the local development environments (Decentralized Architecture).

In a *Decentralized Architecture*, the clone management functionalities, when augmenting the features in local programming environments, can enable the individual programmers to exploit the benefit of clone management. Developers can use different tools here; even some of them can get the flexibility to completely or partially disregard clone management at their respective situations. However, this might require additional setup for establishing means for communication between distributed developers, as well as combining and synchronizing clone information across all the developers.

In a *Centralized Architecture*, the clone management functionalities can be implemented as a client-server application on top of the central version control systems. Due to the client/server architecture, a centralized implementation will affect both the repository and the local programming environments. Such a centralized clone management system may require greater effort and offer less flexibility than a decentralized implementation.

Locality of clone management data

The data building the foundation of the clone management may be used locally only or globally. In a decentralized implementation a local-only use of the data will be easier to implement, while in a centralized implementation, it would require sharing the data among the distributed developers.

Triggering of clone management activity

A clone management activity can be human triggered (for example initiated by the developer or some other person involved in the software maintenance process) or system triggered (for example as a response to certain actions performed by the developer in the system).

In a human-triggered initiative, a developer, after writing or modifying a piece of code, may invoke search for its clones in the system, and upon finding the clones, she may analyze and decide how to deal with them. An instance of clone management activity may also be periodically scheduled in advance as part of a larger plan of process activities.

In a system-triggered initiative, the development environment can trigger clone management activities in response to certain events, such as saving changes in the code, or the check-in of modified code to the central repository. Such events may notify and suggest the developer to perform the required clone management operations.

Scope of Clone Management

An instance of clone management activity may be more focused towards the clones or the subject system itself. While a clone-focused activity deals with a narrow set of clones of a particular code segment of interest, a system-focused clone management activity aims to deal with broad collection of clones in the entire code base, or particular portions of the subject system.

6.3.3 Implementation options for a Clone Management System

Clone management activities may be considered as a part of regular code-centered development activity [125]. A developer typically works inside an IDE running on his/her workstation, for fairly large projects, especially in industrial settings, a team of developers collaboratively work on a shared code base kept in a version control system (e.g., SVN, CVS) set up on a central server. Thus, the need for the integration of clone management activities with the development process suggests that the IDEs should include features to support clone management activities during their actual development phase.

While, ideally, all clones should be managed proactively, in practical settings, proactive treatment to all clones may not be feasible or possible. Therefore, a versatile clone management system should focus on the support for proactive management, while at the same time, should also facilitate corrective clone management [35]. Therefore, an IDE would be a better option than a standalone management tool to implement

both clone management strategies. Because managing clones using some standalone clone management tool, especially for implementing proactive clone management strategy would simply mean making the developers decoupled from an IDE, which would be nothing but a post-mortem approach.

6.4 Existing work on IDE based clone management

There are many clone detection tools, each has its own strengths and weaknesses. However, for proactive clone management, the support for clone detection should be integrated with the development process. Therefore, we focus on those tools that realized the clone detection feature integrated with an IDE or a version control system.

Tools for managing copy-paste clone: This very first category covers the tools that only deal with clones resulting from developers copy-paste operation. Hou et al. developed their clone management tool CnP [72] that is tightly coupled with the clone detection technique based on the programmers' copy-paste operations. Thus, the scope of the clone management is limited to copy-pasted code only, and not applicable to clone management based on similarity based clone detection. Jablonski and Hou introduced CReN [74] tool as an Eclipse plug-in for Java programs to help programmers avoid making copy-paste error, while renaming various instances of identifiers, such as variable names. It tracks the code clones involved when copying and pasting occurs in the IDE and infers a set of rules based on the relationships between the identifiers in these code fragments. So, it is only a subset of Type-1 clones (i.e., only the clones arising from the developer's copy-paste action in IDE) that are being managed using this tool. The plug-in CSeR (Code Segment Reuse) was developed by Jacob et al. [76] to check copy and paste induced clones in an integrated development environment. The tool was designed to compute the clone differences interactively by checking whether a piece of code is copy-pasted while the programmer edits and/or types the code. A similar IDE plugin is proposed by Venkatasubramanyam et al. [175] for proactive moderation of the genesis of clones through copy-paste-modify operations. The approach is guided by associating constraints formulated from predefined guidelines, and checking for their satisfaction at the time of copy and upon modifications. CloneBoard [43] and CPC [177] are another two Eclipse plugins that can detect and track clones based on clip-board/copy-paste activities of the developer. Both CPC and ClonkeBoard support linked editing of clone pairs. While the above mentioned approaches based on programmers' copy-paste activities may be able to handle intentional clones (i.e., the clones that someone purposely introduced in the system or being aware that the intended action will introduce new clone), they are unable to deal with unintentional clones. Moreover, such tools may not be suitable for distributed development, as they may fail to combine information about clones separately created by distinguished developers working in a distributed environment [184].

Tools with integrated clone detection and visualization only: This second groups of tools provide native clone detection support and offer various visualization of clones from within the IDE. SHINOBI [91] is an add-on to the Microsoft Visual Studio 2005. It internally uses CCFinderX's preprocessor,

and thus it can detect Type-1 and Type-2 clones only [184]. It was developed as a client-server application to mainly relocate the clone detection overhead from the client (local programming environment) to a central server (source code repository). SHINOBI only displays clones of a code fragment underneath the mouse-cursor, no further support for clone management is offered. Such a plug-in is CloneDR [23], an AST-based clone detector that can detect Type-1 and Type-2 clones, but it also fails to detect Type-3 clones in many scenarios [151]. Zibran and Roy [184] presented an IDE-integrated focused clone search tool for Type-1, Type-2 and Type-3 clones. The implementation is based on a suffix-tree based k-difference hybrid algorithm. Bahtiyar developed JClone [10] as a plugin to the Eclipse IDE for detecting Type-1 and Type-2 clones only from Java projects. JClone applies an AST based technique to detect clones. It enables the user to trigger the detection of clones from one or more selected files or directories. It also offers a few visualization (i.e., TreeMap and CloneGraph views). Plug-ins in this category may aid clone analysis to some extent, but they offer no further support for clone management beyond the detection and visualization of clones.

Versatile clone management tools: This final group of tools cover additional clone management services beyond those mentioned in the previous two groups. CloneTracker [46] is a more versatile clone management system as an Eclipse plug-in presented by Duala-Ekoko et al. The plug-in allows tracking and simultaneous editing of clones. It relies on the output of the SimScan [29] clone detection tool and requires the programmer to manually select the clone groups of interest to be documented. Once the clone groups are identified, CloneTracker translates the location of all clone regions from a file name and line range notation into Clone Region Descriptors (CRDs). Instead of using the clone's exact text or its physical location in the file, the CRD technique uses syntactic, structural, and lexical information (the clone region's alignment with code blocks) to determine the clone's relative location in a file. While this technique has some benefits, it only gives an approximate location. Recently, Tairas and Gray developed CeDAR [167], a plug-in for the Eclipse IDE, where they introduced an approach to unify the processes of clone detection, analysis, and refactoring. They integrated a clone detection approach in the plug-in and presented a visualization technique in which one clone instance displays the properties of all the clones in the clone group. This representation helps in refactoring as clone group representation displays the differences among clone instances.

In next section we are going to present *SimEclipse* as a versatile IDE plug-in covering a rich set of features in support of better clone management in software development and evolution.

6.5 *SimEclipse*: The plug-in for clone-aware software development

A number of state-of-the-art tools have been evolved in software clone research during past decade covering clone detection, visualization, analysis and management. From software development perspective, the question is, are those tools really useful in practical context? That is, how conveniently developers can make use

Figure 6.2: Enable/Disable *SimEclipse*

of those tools in their development workflow. In most cases the existing approaches partially cover the scope of clone management or do not integrate well in a software developer's development workflow. In this section, we present *SimEclipse* - an Eclipse plugin focusing on preventive clone management strategy (Figure 6.1) by providing developer support for detection, visualization, analysis and tracking clones in projects being developed using Eclipse IDE. The goal of 'SimEclipse' is to make state-of-the-art clone management tools available in the IDE so that clones in software can be managed in a better way at the place where they are introduced and evolved. It uses a fast source code similarity detection algorithm we previously studied that makes it possible to investigate real time management of exact and near-miss clones in large-scale software systems. On the fly detection and tracking of clones in the IDE has been made easier in the plug-in using the versatile clone-index implemented in the *SimLib* API (Chapter 5). Besides, an implementation of the genealogy extractor proposed by Saha et. al. [155] has been integrated into the plugin for extraction and visualization of clone genealogies that would help developer in understanding the evolution of clones. Various features of SimEclipse are presented as follows.

6.5.1 Startup and Configuration

Once *SimEclipse* is installed in the Eclipse environment, it can be enabled or disabled for any open projects shown in the Project/Package Explorer by right clicking on the project and navigating to the *SimEclipse* menu item in the context menu. More options are available for the *SimEclipse* enabled project in the same menu location (Fig. 6.2). These new options are also available from other views (Fig. 6.4.a, Fig. 6.4.b). Each project has a customizable *SimEclipse* settings that defines the detection and runtime behaviour of the plugin on that project.

View for *SimEclipse* Projects

This view shows all the projects in the current workspace for which *SimEclipse* has been enabled (Fig. 6.3.b). It also shows the 'Enable/Disable' status of the two background services for the corresponding project. This

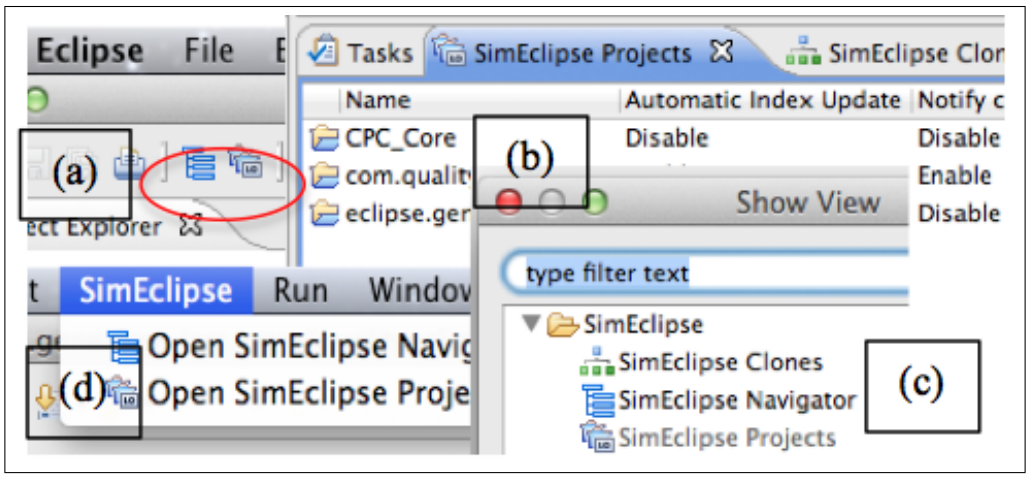


Figure 6.3: *SimEclipse* Views

view can be made available anytime from the 'Workbench's Main Menu' (Fig. 6.3.d), 'Workbench Toolbar' (Fig. 6.3.a) and 'Show View dialogue' (Fig. 6.3.c). For each of the projects shown in *SimEclipse Projects View*, the following actions are available to perform from the context menu available by right clicking on any project in that view (Fig. 6.4.a):

6.5.2 Just-in-time Clone Detection

Clone detection can be performed on a *SimEclipse* enable project from various views, e.g., 'Eclipse Project/-Package Explorer View', 'SimEclipse Projects View', 'SimEclipse Navigator View,' etc. (Fig. 6.4). 'SimEclipse Navigator View' (Fig. 6.4.c) also allows developers to explore the codebase of a *SimEclipse* enabled project using a tree based hierarchical view. Developers can see the directory, files and fragments maintained in the clone-index [172], which reflects the similar hierarchy of the actual code-base. Non-related files and folders are not shown in this view although they are available in the actual codebase. For example, if Java is the language for a project, only Java files and container folders will be shown in this view. Clicking on any file or fragment entry here will display the actual source file in the workbench using the integrated source editor associated with the file. For fragments, the entire fragment will be highlighted in the editor (Fig. 6.6). Besides, user can also perform clone detection from editor by selecting an arbitrary portion of code of interest and then go for that selection only or for corresponding complete functions or blocks touched by that selection (Fig. 6.7).

6.5.3 Clone Visualization

Detected clones are visualized through the 'SimEclipse Clones View' (Figure 6.8) where user can see grouping of clones based on the detection settings for the project. Clones are displayed in different colours based on their types. Users can click on any clone to see it highlighted in the editor (Fig. 6.9). From this view, the user can also compare any two clones in a clone group to see the actual similarities and dissimilarities (Fig. 6.10).

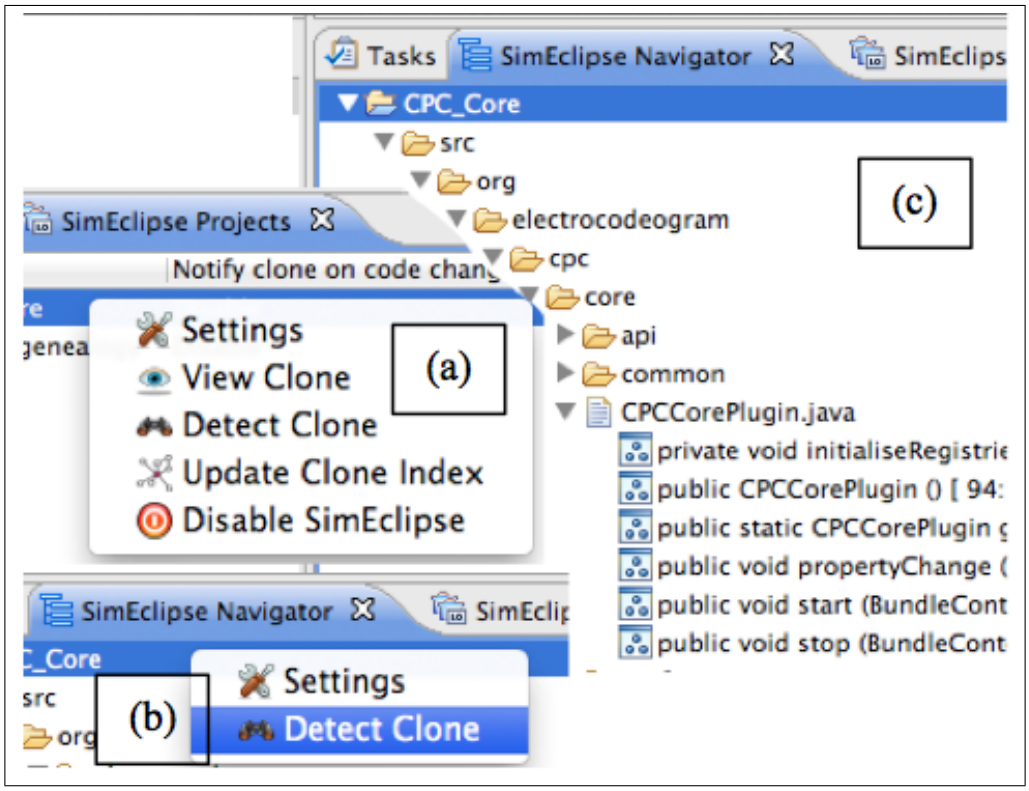


Figure 6.4: *SimEclipse* Views and Actions

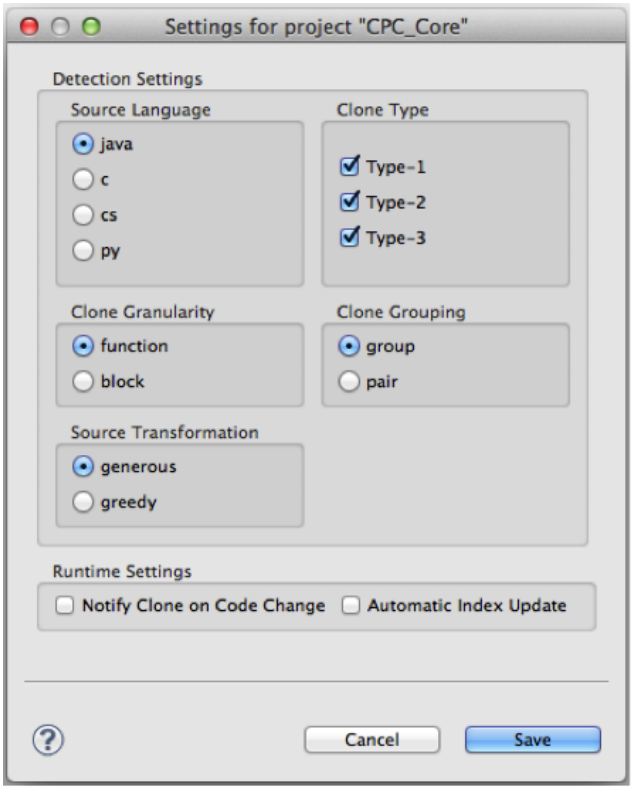


Figure 6.5: *SimEclipse* Settings

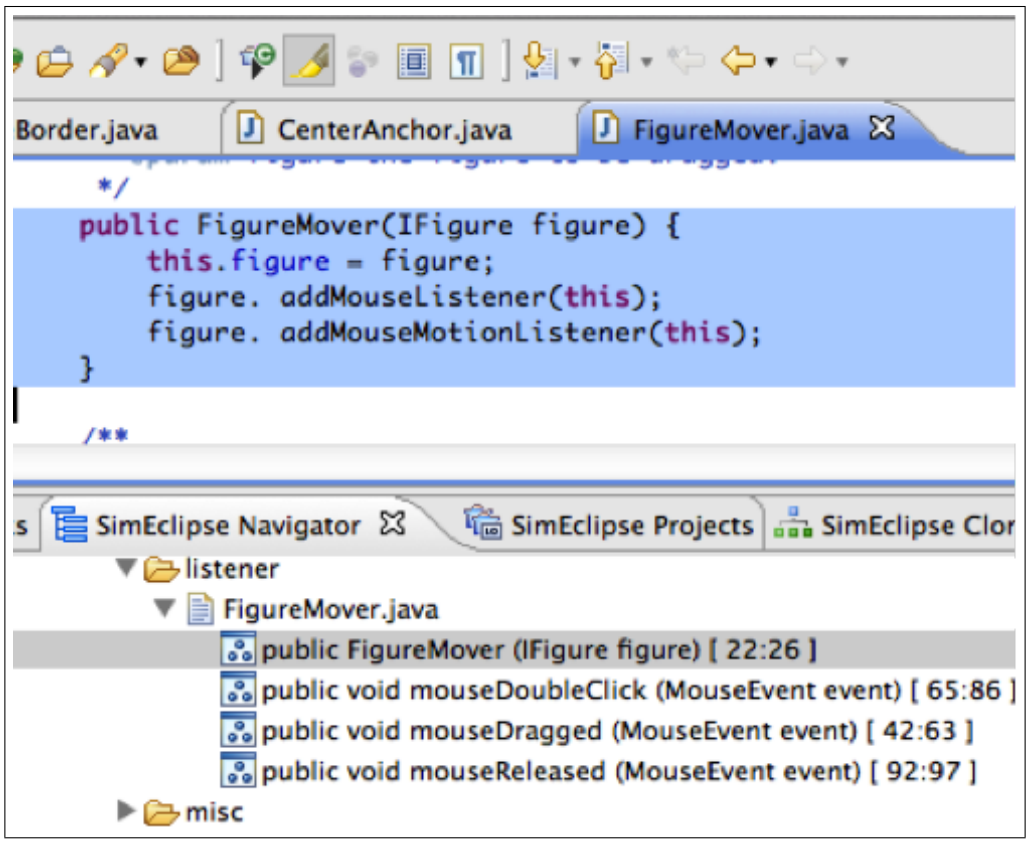


Figure 6.6: Navigation of source in *SimEclipse Navigator View*

6.5.4 Clone Analysis

Clone Genealogy Viewer

For a multi-version project, *SimEclipse* is able to extract and display clone genealogies. From the *SimEclipse* settings option (Figure 6.5), users can add multiple versions of the project and then invoke for extract/view clone genealogy for the lifecycle project covered by the versions choosen. From 'SimEclipse Clone Genealogy View' (Figure 6.11) user can explore the all the genealogies version by version and get to know their evolution.

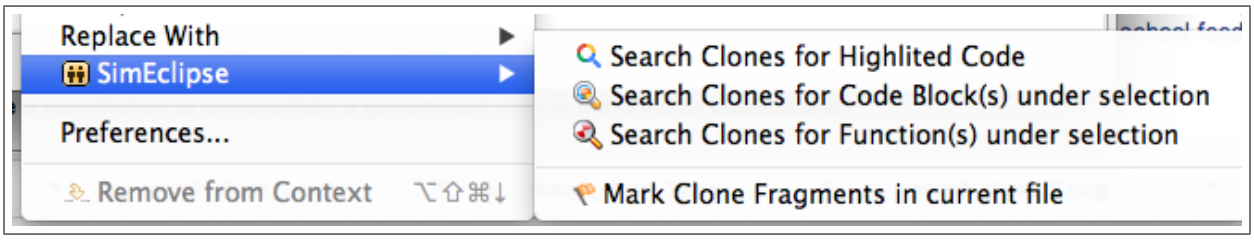


Figure 6.7: Search for Clones From Editor

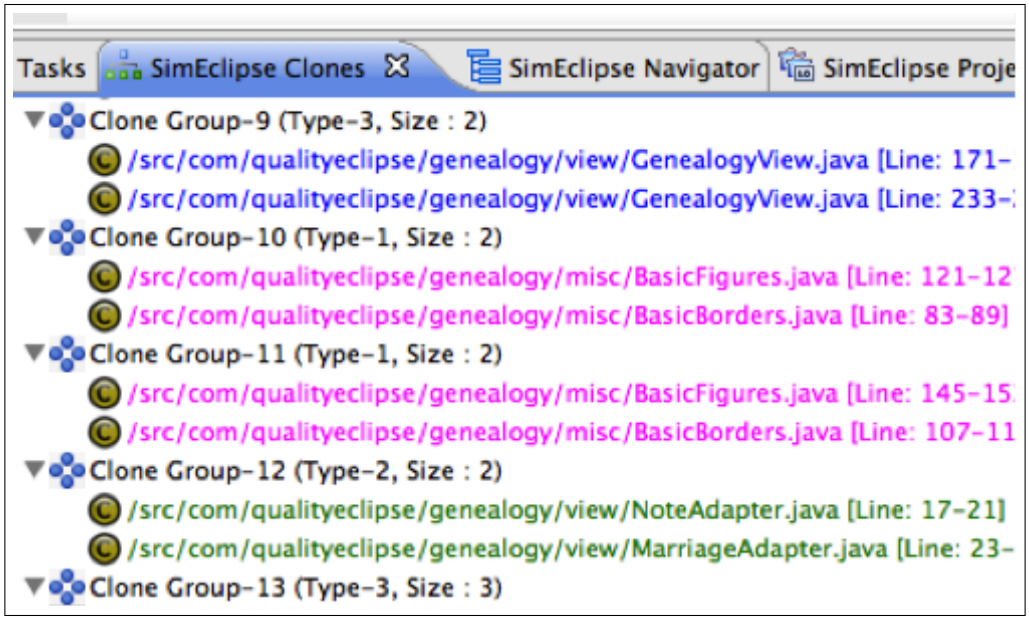


Figure 6.8: *SimEclipse Clones View* for display clone detection result

Code History Exploration

Similar to Clone Genealogy, for a multi-version project, *SimEclipse* allows developers to inspect the source modification history for both clone and non-clone source fragments (Figure 6.12). If the original source code includes source code ripository information along with it, *SimEclipse* would be able to display the developer/author information as well so that the developer who is currently working with the code get to know which other developer worked on this code in the past. This feature would help developer in understandig changes in source code throughout the lifecycle of the project.

6.5.5 Clone Tracking

Mark/Annotation of Clones in Editor/Existance Tracking

Developers can have their editor marked for a file that contains clones by choosing the corresponding option in the context menu as shown in Fig. 6.7, which can be made available by right-clicking on the source file in editor. Each yellow flag mark shown in the left vertical bar of the editor (Fig. 6.13) represents the location of a clone fragment in the subject file. Location information of other clone fragments related to this fragment can be seen by double clicking the flag. This will open a pop-up window (Fig. 6.14) from which developers can explore and see the actual source of those fragments in the editor.

Clone Tracking in Project/Change Tracking

Clone Tracking is an automated service provided by *SimEclipse* to track generation of new clones in project due to the developer's activity on changing project source code. Developers can have this service enabled

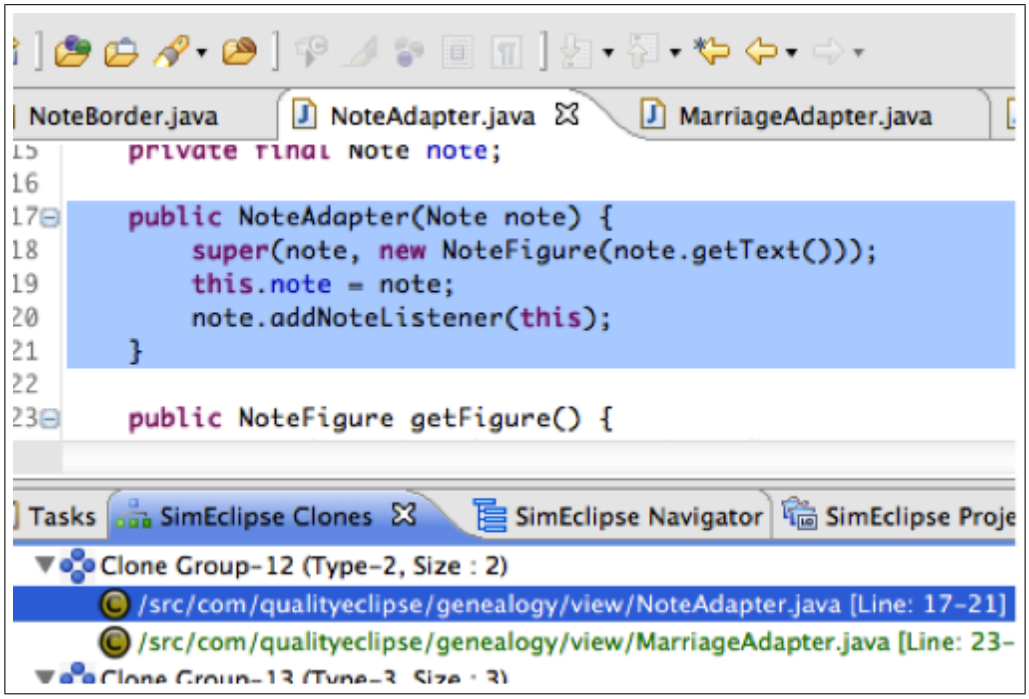


Figure 6.9: Inspect Clone Code in *SimEclipse Clones View*

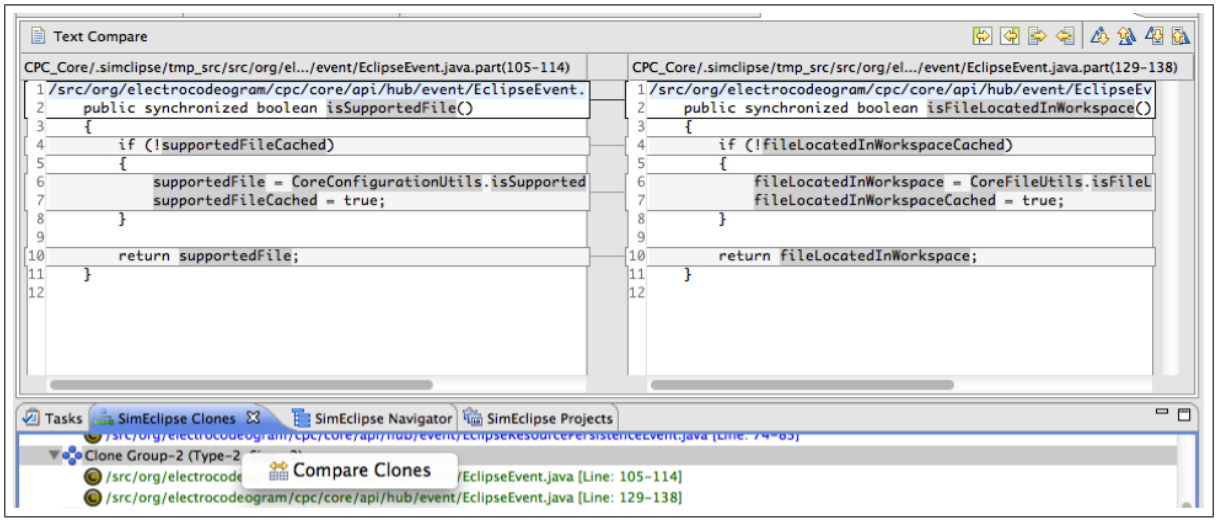


Figure 6.10: Text compare between two clone fragments

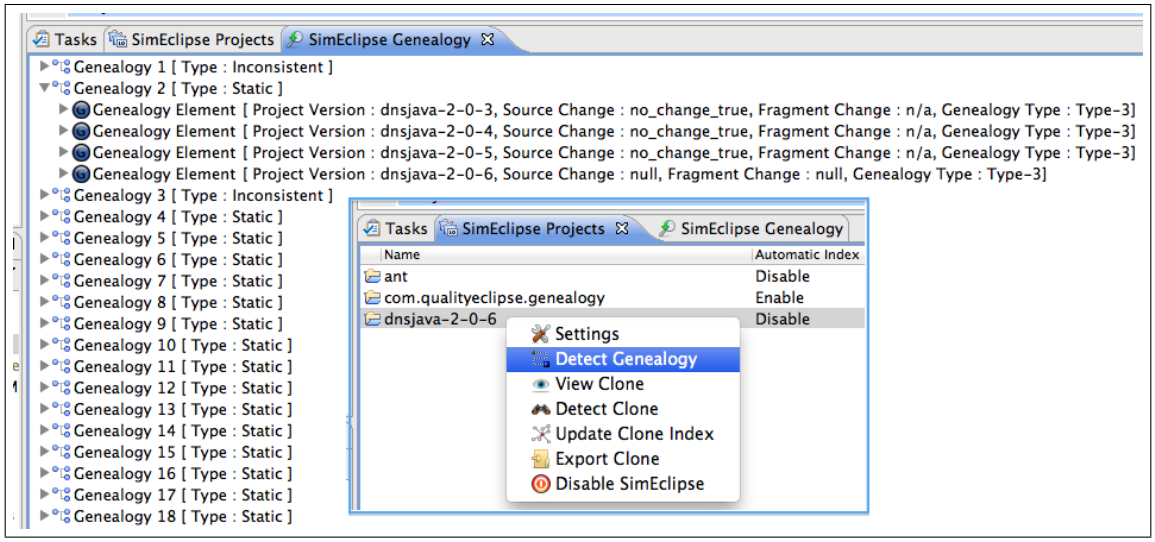


Figure 6.11: Clone Genealogy Viewer in *SimEclipse*

or disabled from the project specific settings of the plugin. This workflow of service is shown in Fig. 6.15. It provides real-time notifications to the developers when any new clone has been evolved in the system because of developer activity. If the changed code is originally a clone but the recent changes made on it is not quite enough for it to no longer be considered as a clone under the defined detection setting, it will also be reported. Both the new and old (but changed) clone fragment entry will be highlighted in red text in *SimEclipse Clones View*, while for the others it remains in the default colour scheme for the particular type of clone (Fig. 6.16). As a background service, it senses changes made on the codebase after every save operation made on the project files from the IDE and reports if changed or new portion of code matches with any other existing code in the codebase under the defined detection settings. Using this, developers can avoid accidental generation of clones, for example, because of not knowing existence of similar code somewhere else in the project. If source code gets changed outside the project, that may affect the cloning state of the system, updating the clone index for that project in *SimEclipse* would cover those changes in the clone tracking process. Current implementation of *SimEclipse* does not provide any automatic refactoring or removal feature of clones. Instead, it lets the developers do this based on his/her own rationale using manual approach or general code refactoring features available in IDE. *SimEclipse* however does make sure whether any changes made to clone fragments make them disappear or not.

6.6 Scenario Based Study for Identifying Effectiveness of Integrated Clone Technologies

The motivation of this study is to capture the user experience in dealing with software clones in two different working environments (experimental setup). In one setup, the study participants were told to perform some

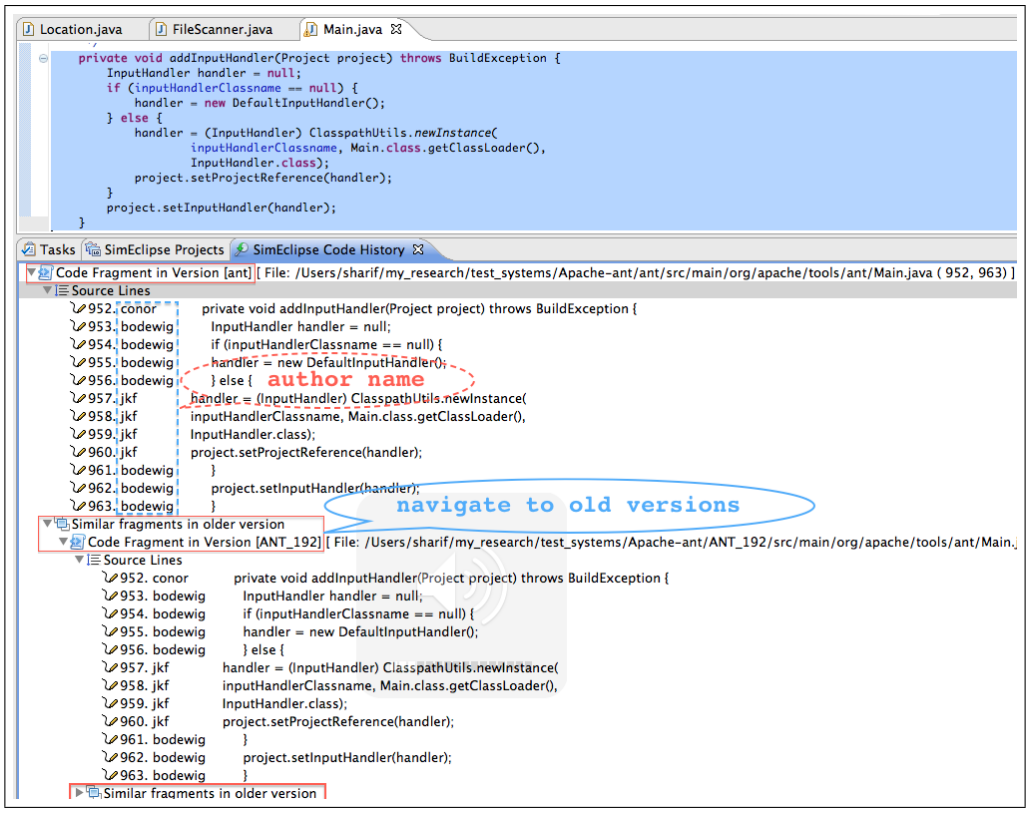


Figure 6.12: *SimEclipse* Code History Explorer

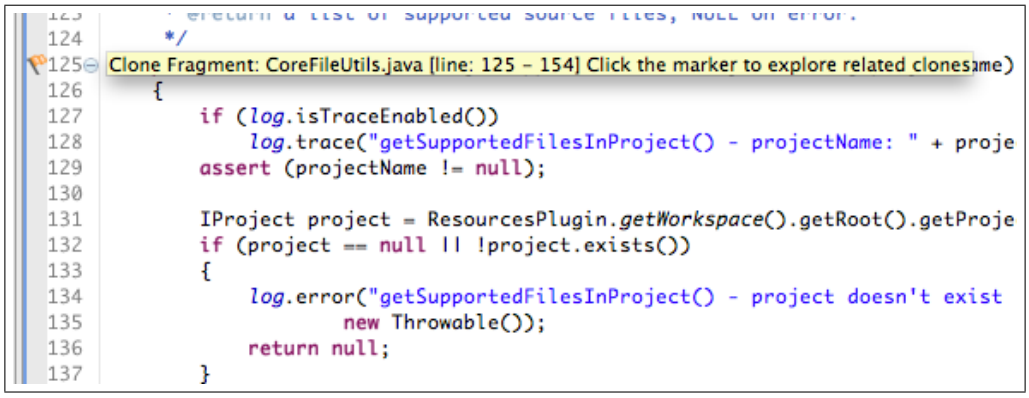


Figure 6.13: Mark Location of Clones in Editor

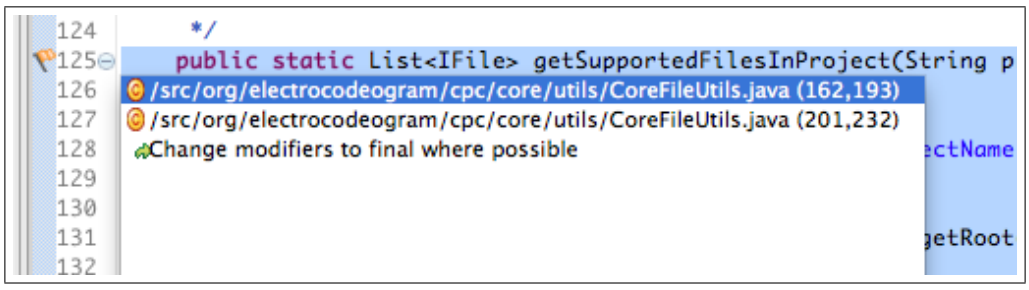


Figure 6.14: Information of Other Clones in Marked Location

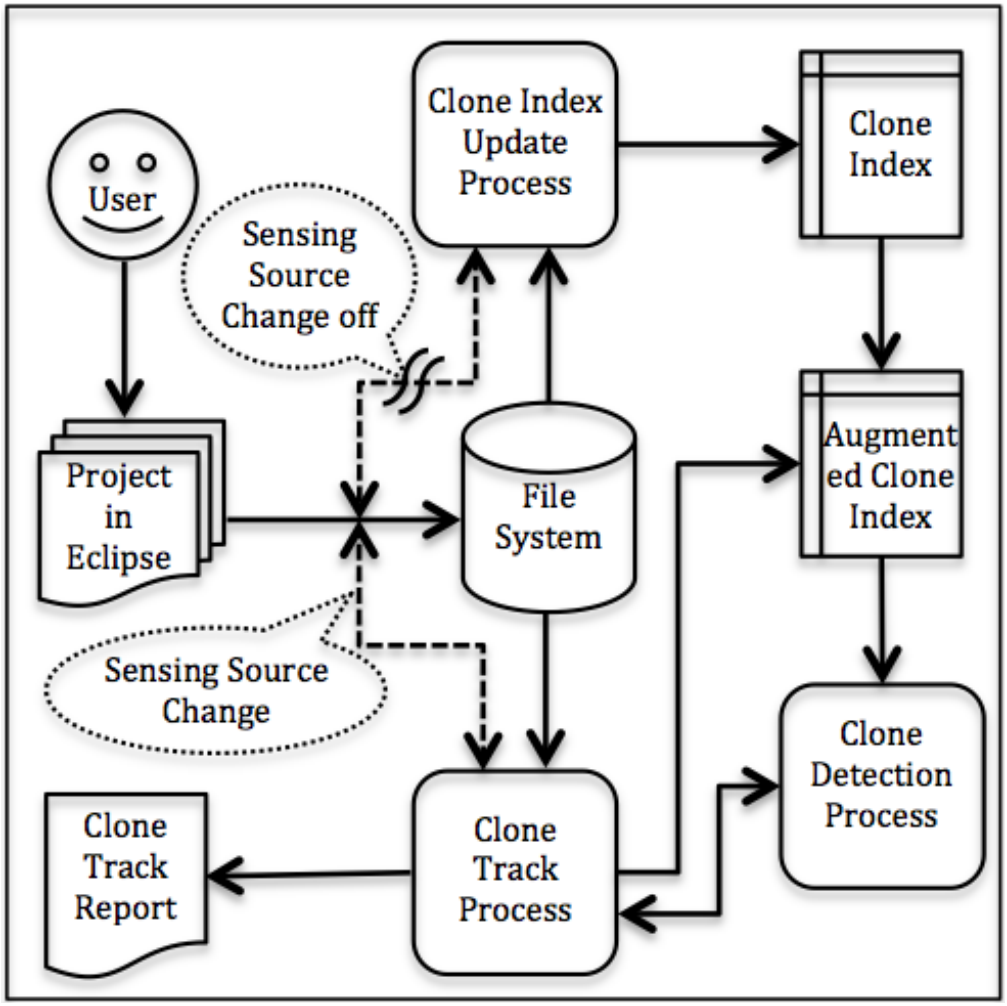


Figure 6.15: Clone Tracking Service in *SimEclipse*

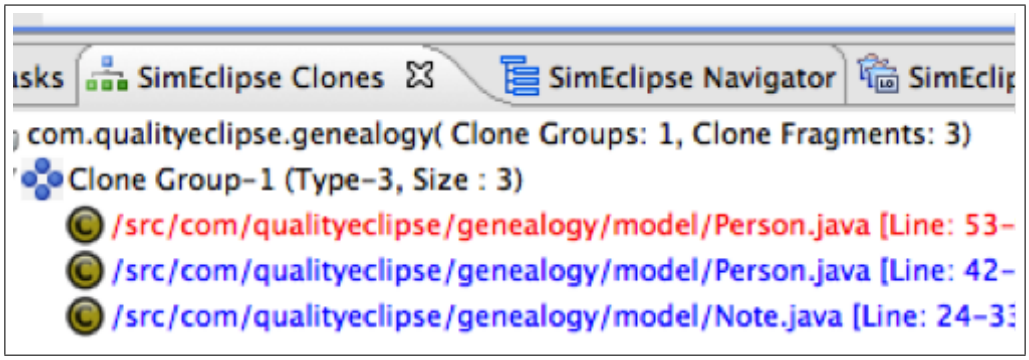


Figure 6.16: Notification of newly introduced clone

pre-defined code clone related tasks on a subject system using some standalone tools. In second setup, they were told to perform the same tasks using *SimEclipse* as a tool with integrated clone management features. User experiences on both the environments are contrasted to measure the effectiveness of using integrated clone technologies in working with software clone. To fulfill the experiment we have conducted a series of steps starting from experimental setup to analysis of the results.

6.6.1 Experimental Setup

To perform the study, we required a set of participants, a set of tasks, a questionnaire, methods of evaluation, and analysis of the result. The following are the steps that constitute our setup for the experiment.

Tasks Design

In this experiment, the participants get to perform the same set of tasks in two different working environments as stated earlier. The tasks for this study have been designed from easy to fairly complex. Each of the tasks is associated with a question and the participants are expected to lead towards some answers by performing the task. All the required input information to perform the given tasks are also provided to the user. The idea of having the participants performing these tasks in two different environments is to capture whether they feel any importance of having a platform with integrated clone management system rather than using some separate standalone tools to accomplish the tasks. Following are the tasks that the participants have been performed in this study.

T1: Given three functions, identify if they are clone or not in the given system.
Target Usage: Clone Detection

T2: On-demand/Focused clone search, identify number of Type-1 clones in three source package/folder.
Target Usage: Clone Detection/Visualization

T3: Apply given source modification to three existing functions, if there are clones; apply the modification to all clone fragments.
Target Usage: Clone Detection/Visualization/Tracking

T4: Add three functions on a specified source file (both code to write and source file location are provided), determine which of the functions newly added are fall into existing clones in the system, if there are existing clones to the new code, then identify the location and type of the clones.
Target Usage: Clone Detection/Visualization/Tracking

T5: Identify clone genealogy of two given function (the function is being a member of an existing clone class).
Target Usage: Clone Analysis/Understanding

T6: The clone genealogy of one of the above two function is inconsistent. Identify that inconsistent genealogy and locate the change that makes the clone class inconsistent. Locate the developer who made the change..

Target Usage: Clone Analysis/Understanding

Questionnaire Design

For this study, following two sets of questionnaire have been designed.

User Profile Based Questionnaires: This first set of the questionnaire is designed to acquire the background of the participants with respect to their knowledge about the domain clones in software development, which includes the following questions:

1. Software development experience (in years).
   Options: <1, 1, 2, 3, 4, 5, 5+.

2. Industrial experience.
   Options: Yes/No

3. Experience in Programming languages.
   Options: Java, C/C++, Python, C-Sharp

4. Experience in lDE.
   Options: Eclipse, Netbeans, VisualStudio, Other

5. Do you have basic knowledge of software clone?
   Options: Yes/No

6. Do think "Clones should be managed during software development"?
   Options: Strongly agree, Agree, Neutral, Disagree and Strongly disagree

7. Do think "Preventive clone management is preferable than Corrective clone management"?
   Options: Strongly agree, Agree, Neutral, Disagree and Strongly disagree

8. Do think "Clone understanding is important for managing clones efficiently"?
   Options: Strongly agree, Agree, Neutral, Disagree and Strongly disagree

9. Do you have any experience of using clone based tools for research?
   Options: Yes/No

10. Do you have any experience of using clone based tools in software development (for managing clones)?
    Options: Yes/No

Task Based Questionnaires: The second set extracts information about user experience in performing the given tasks. At the end of each task, the participants were required to answer the following questions. This set of questions again sub-divided into following two groups:

1. Per task based question: Participants were required to answer the following questions at the end of each task mentioned earlier in this section:

   (a) Completion status of the task.
   Options: 1) Yes, 2) No (task fails if not complete within 15 minutes)

   (b) Time taken to complete the task.
   Options: 1) 0-5 min, 2) 5-10 min, 3) 10+ min (max 15 mins/task)

   (c) Correctness of the answer found in the task.
   Options: 1) Wrong/Incomplete, 2) Partially correct, 3) Fully correct

   (d) Difficulty to accomplish the task.
   Options: 1) Very easy, 2) Easy, 3) Moderate, 4) Somewhat hard, 5) Hard, 6) Very Hard

2. Overall user experience question: Finally, participants were asked following two questions to understand their overall experience in the study:

   (a) In which environment you felt comfortable and confident in performing the given tasks?
   Options: Environment-1, Environment-2

   (b) Which platform do you think would be preferable for managing clones in software? If you prefer one over the other, please mention your reasons for support if any.
   Options: IDE Plugin, Standalone Application Suite, Either would be fine

Run Study Session

It is the second phase of our experiment modeling, which initiates the user participation in one-to-one session with each participant. At the very beginning, the participants were given an introduction about the nature and purpose of the study followed by a small orientation on the software clones if the term is new to the participant. After that, a short survey was performed using *User Profile Based Questionnaires* that captures their familiarity with software development and various aspects of clones in software. The next part allows the participants to do specific tasks using some code clone related software tools and answer the related questions. This part of the study session went in following three phases:

Phase 1:

Study Environment-1:

1. A candidate Java project with multiple versions (at release level), each version is checkout from SVN source repository to separate folder with version name.

2. Eclipse IDE with latest version of the candidate project imported as Java project
3. Standalone clone detection, visualization, analysis tools (we have used *SimCad*, *VisCad*, *gCad*) and SVN command-line client.

Study Session: The study participants have been shown the use of supporting standalone tools (using video demonstration) and then asked to perform the tasks mentioned earlier in this section.

User Feedback: After end of each task, the participants gets to match his/hew own answer to the actual answer, fill out the 'Task Based Questionnaires' for this task on Study Environment-1, which records his/her experience and any outcome of the task in this environment setup.

Phase 2:

Study Environment-2:

1. A candidate Java project with multiple versions (at release level), each version is checkout from SVN source repository to separate folder with version name.
2. Eclipse IDE with latest version of the candidate project imported as java project
3. *SimEclipse* plugin installed and enabled for candidate project imported in Eclipse

Study Session: The study participants (the same group as earlier) have been shown the features of *SimEclipse* (using video demonstration) and then they are asked to perform the same tasks (with different input set) again using various features of *SimEclipse*.

User Feedback: After end of each task, the participants fills out the 'Task Based Questionnaires' for Study Environment-2, which records his/her experience and any outcome of the task in this environment setup.

Phase 3:

After finishing the tasks on both the study environments, user fills out the answers to the question covering "Overall User Experience" in performing the tasks in two different environments.

6.6.2 Summary of Findings

General Feedback

This feedback is about user profile and their knowledge/experience with software clones and related tools. All of the participants found having some software development experience throughout their academic curriculum and 40% of them were found having few years of experience in industries. The participants have experience in developing software with at least one or more popular programming language using well-known IDEs used for software development. Among the participants, 70% were found having at least basic knowledge of software clones prior to the introductory session of the study. However, all were found being at least 'agree' that clones should be managed and *understanding of clones* is important to manage them efficiently. On the question of

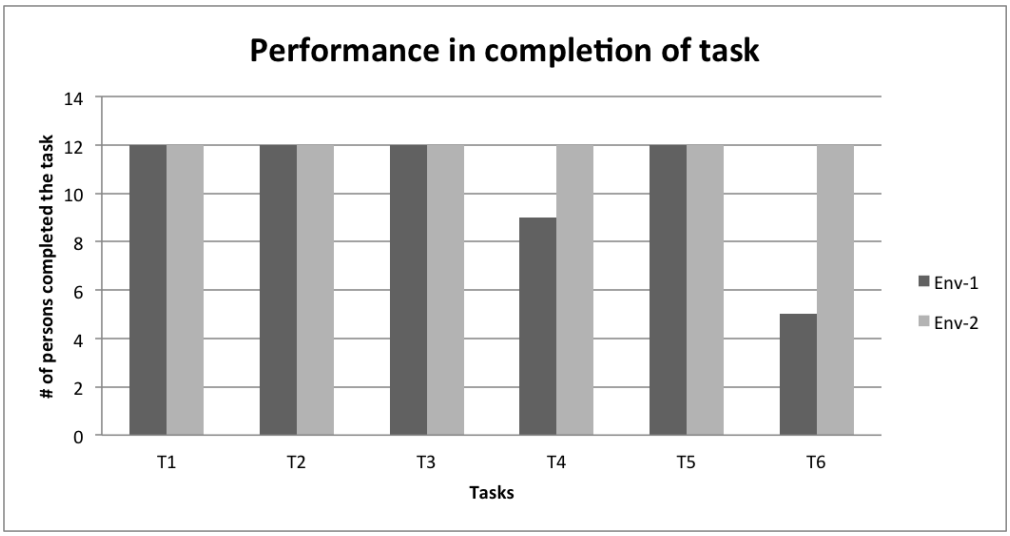


Figure 6.17: User performance in completion of task in different environments

clone management strategy, 70% of them found being at least 'agree' that *Preventive clone management is preferable than Corrective clone management.*

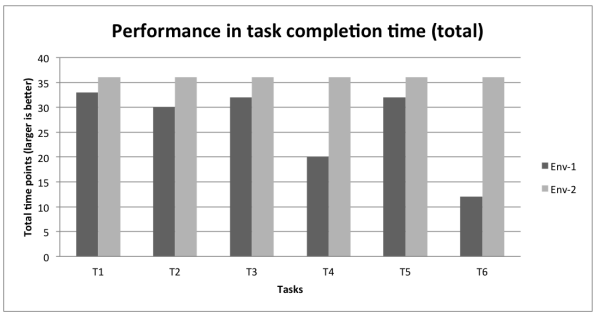
Interestingly, 50% of the participants were found having some experience in using various clone based tools for academic research, but none of the participants were found to have experience of using any clone management tool in their own software development process. Possible reasons for that as expressed by the participants' opinion include: worked with small projects, did not stuck into a situation so far for which the reason was being identified as clone, lack of established standard clone management tools in industry, lack of default clone management support in IDE, etc.

Task Based Feedback

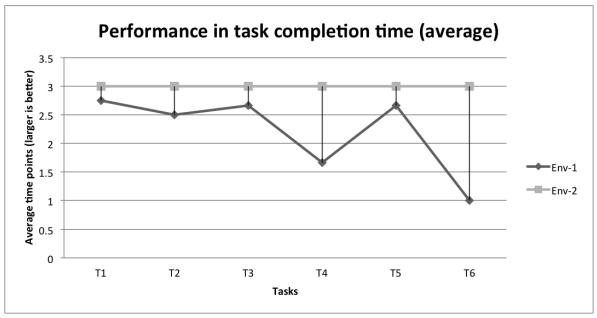
User feedback from the 12 participants upon performing the given tasks in two different environments has been analyzed. The result is presented here in the form of contrasting their task performance in those environments based on the four criteria: completeness, time taken, correctness and difficulty.

Task Completeness Analysis: Here we comapre how much tasks have been completed successfully by the participants in two different study environments. Figure 6.17 shows, in *Environment-1*, three (25% of the total) and seven (58% of the total) participants were not able to complete the task *T4* and *T6* respectively in the given timeframe. On the other hand, all the participants in Environment-2 were able to complete all the tasks successfully.

Task Completion Time Analysis: In the task completion time analysis, we tried to investigate and compare the time taken by the participants in completing the tasks in different environments. The recorded time slot data in the study are normalized into *Time Points* according to the Table 6.1. The normalization scale



(a) Users' total completion time points per task



(b) Users' average completion time points per task

Figure 6.18: Comparison on completion time of tasks taken by users in different environment

Table 6.1: Time Point scale for task completion time analysis

Time Slot	Time Point
0-5 minutes	3
5-10 minutes	2
10+ minutes	1

assigns higher points to the task completed in lesser time. For each tasks, total and average *Time Points* are calculated for all the participants. Figure 6.18 shows the comparison of total and average *Time Points* of the tasks in two different environments. From the figure it is clear that all the tasks being performed in *Environment-2* associated with higher total and average *Time Points* than in *Environment-1*, which means participants were able to complete the tasks faster in Environment-2 than in Environment-1.

Table 6.2: Correctness Point scale for task correctness analysis

Question Answered	Correctness Point
Wrong/Incomplete	0
Partially correct	1
Fully correct	2

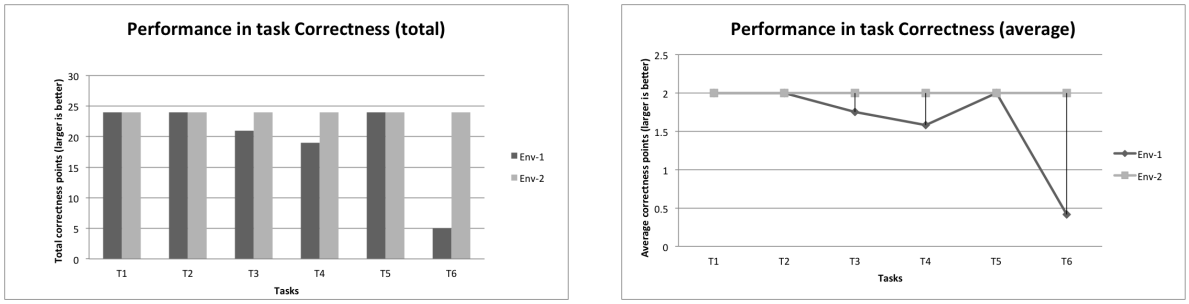
Task Correctness Analysis: Each task has a question associated it, and by performing the task, user are leading towards some answer to that. This analysis covers user performance in finding correct answer to the question associated with the tasks. Based on the user's answer, a *Correctness Point* is assigned according to Table 6.2. For each tasks, total and average *Correctness Points* are calculated for all the participants. Figure 6.19 shows the comparison of total and average *Correctness Points* of the tasks in two different environments, which clearly indicates that the participants did well in Environment-2 than in Environment-1 towards finding correct answers to the questions.

Table 6.3: Difficulty Point scale for task difficulty analysis

Difficulty Level	Difficulty Point
Very easy	1
Easy	2
Moderate	3
Somewhat hard	4
Hard	5
Very Hard	6

Task Difficulty Analysis: This part of the analysis investigated on how difficult it was to perform the given tasks using the resource provided in each environment. The participants reported their opinions in a six level difficulty scale, where each level is assigned a *Difficulty Point* according to the Table 6.3. For each tasks, total and average *Difficulty Points* are calculated for all the participants. Figure 6.20 shows the comparison of total and average *Difficulty Points* for the tasks performed by the participants in two different environments, which again shows that participants performing the tasks in *Environment-2* felt that the tasks were less difficult to complete than it is in *Environment-1*.

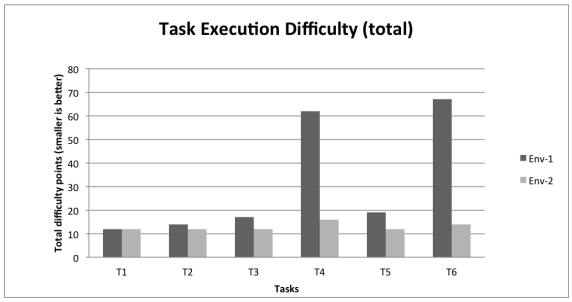
Finally, from the feedback received on *Overall user experience question*, 100% of the participants felt more comfortable and confident in performing the given tasks in the *Integrated Environmnet* than in *Discrete Environment*. Moreover, eight (66%) participants mentioned 'IDE' as a preferred platform over a 'Standalone Application Suite' for providing tool support for clone management. Some major reasons mentioned in support to their opinion were: avoidance of context switching between IDE and some external tool to transfer clone knowledge, less manual or intermediate data processing and reducing the possibility human error. Among the rest, two (17%) participants were fine with either approaches, and the remaining two (17%) preferred *Standalone Application Suite* over IDE with supporting reasons including: possibility of IDE being bogged down with memory/time intensive tasks (specially for large systems), freedom for developers to choose working with different IDEs, etc.



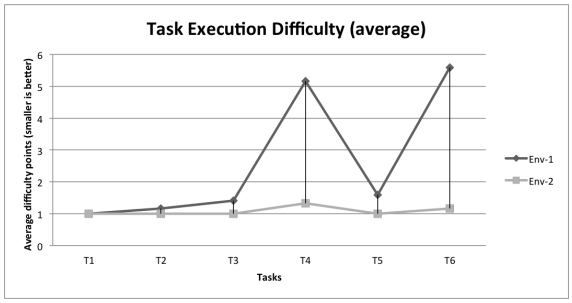
(a) Users' total correctness points per task

(b) Users' average correctness points per task

Figure 6.19: Comparison on correctness of the tasks performed by the users in different environment



(a) Users' total difficulty points per task



(b) Users' average difficulty points per task

Figure 6.20: Comparison on difficulty of performing the tasks by the users in different environment

From here we can conclude that majority of the participants were in favour of doing clone management activity in *Environment-2*, which in this study was an IDE integrated clone management tool that makes the management tasks less difficult, less error prone and leads to faster processing.

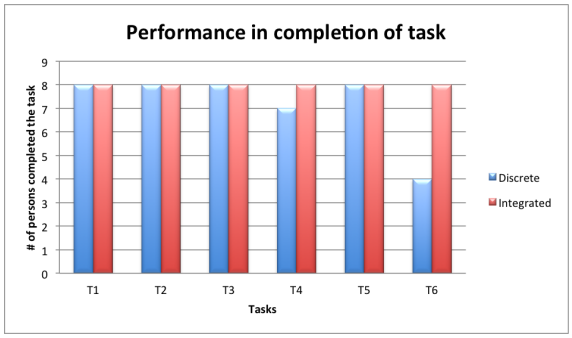
Investigating a Possible Biasness in Task Performance

In the study, the participants were given to perform the tasks first in the discrete environment and then in the integrated environment. Although, for the same task, the inputs were different for different study environments, there might be a chance that participants would do better in performing the tasks in later environment because of being experienced in doing things in former one and thus the study result would be biased to the later environment. In order to investigate that, we have conducted the same study with eight new participants, where they performed the tasks in integrated environment first and then in discrete environment. User performance in this supplementary study has been shown in Figure 6.21. Comparing the results in both the studies, we found almost similar pattern of task performance and no significant sign of biasness as discussed earlier.

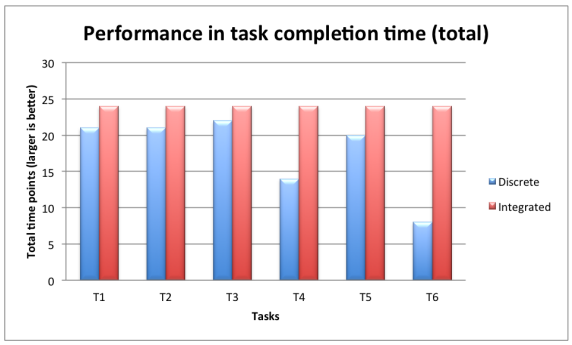
Besides, in both the studies, we have asked the participants whether changing the order of the task environment would make any different in their task performance. In response to that, we have found, all the participants had a common understanding that, since the way of performing the tasks in the two study environments are completely different, there is a very small chance that participants would get any advantage in performing the tasks in the later environment, and thus changing the order would not make any influence in the task performance as defined.

6.6.3 Threats to Validity of the Experiment

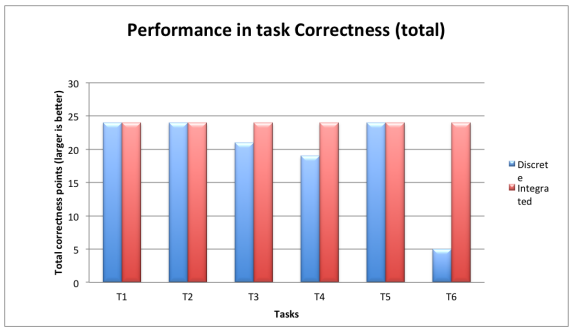
- In the 'Task Correctness' measurement, the state of answer termed as 'partial correct' has not been weighted (for task correctness points) as per the number of the correct answers. Therefore, there would be no difference whether the answers for a task found by the participants were almost correct or almost incorrect. Considering this would make the comparison more realistic.



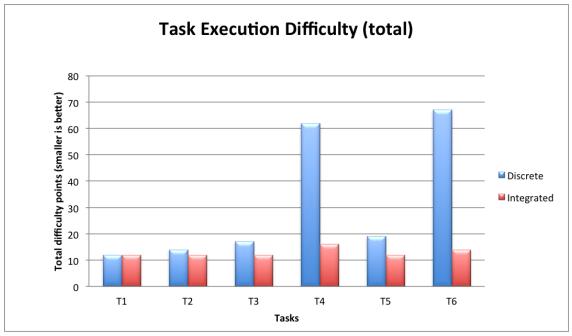
(a) Comparison on completion status of the tasks performed by users



(b) Comparison on completion time of tasks taken by users



(c) Comparison on correctness of the tasks performed by the users



(d) Comparison on difficulty of performing the tasks by the users

Figure 6.21: User performence in supplementary study

- The participants size of our study is not sufficient enough to draw a general conclusion. Furthermore, the level of technical expertise of the involved participants might also have some effects on the experimental results.

6.7 Scenario Based Feature Evaluation of *SimEclipse*

In this section we will describe how *SimEclipse* is evaluated based on its usefulness. This is an extension of the previous study where user gets to play with different features of *SimEclipse* as a tool for managing clones in software. This study included the actual validation of the outcomes of the tool by the expert developers.

6.7.1 Study Motivation

Since the correctness evaluation of the core technologies behind the *SimEclipse* (e.g., clone detector [172], genealogy detector [155], etc) has been done in their respective study, we did not conduct similar evaluation for *SimEclipse* here. Instead, we tried to focus on investigating its usefulness and acceptability to its target users, as in *SimEclipse*, we attempted to integrate those core technologies to facilitate management of software clones in IDE.

6.7.2 Study Design

The evaluation study was based on the following criteria, which were measured from the task based users' experience on the tool:

Usability: How effective the tool is to manage clones in system during software development?

Operability: How simple and easy the tool is to operate for source code clone management?

Presentation: How presentable the information is in the tool for the participants to understanding and analysis of clone?

A set of tasks was designed to cover all the features of SimEclise and after performing the task the participants were required to answer a set of questionnaires capturing their experience based on the above mention criteria.

Questionnaire Design

For this study, following three sets of questionnaire have been designed.

Feature Evaluation Questions: The questions in this set are solely designed to get user satisfaction level. The satisfaction level is represented using a likert scale. Five different levels such as *Strongly agree, Agree, Neutral, Disagree and Strongly disagree* define how satisfied the participants are about the usefulness of the tool after the tasks are done.

(a) The feature is helpful for understanding and/or managing clones.

(b) The feature is easy to operate.

(c) The information presented on this feature was relevant and well presented.

Overall Tool Experience Questions: This set of questions is to capture overall user experience of the tool operation. User's feedback on each question was captured based on a five level likert scale containing options as: *Strongly agree, Agree, Neutral, Disagree and Strongly disagree.*

(a) Did you find the tool satisfying overall? [Overall Satisfying]

(b) Do you think the overall performance of the tool was good? [Overall Good Performance]

(c) Do you think the clone aware software development planform provided by the tool would help developer in managing clones in their projects? [Helpful in Clone Management]

Issues and Improvements Questions: This final set of questions is to identify the issues that the study participants felt while operating the tool and find the scope of further improvements based on their observation.

(a) What difficulty did you experience while using the tool?

(b) What portion of the tool you think requires improvement?

(c) What additional features do you think should be implemented in the tool for better clone management?

Feature based Task Design

To evaluate *SimEclipse* and address the three evaluation criteria we have defined a set of tasks based on different clone management features, which will allow the participants to express their opinions accordingly. Each task intends to gather statistics about the usefulness and the usability of the tool. Usefulness signifies how the tool can bridge the gap between the complexity of a system's implementation and the program comprehension by the participants. Usability portrays how simple and easy the tool is to operate and therefore assists the participants in understanding and convenient management of clones. Table 6.4 provide a summary of the task the participants were provided with.

Runing Study Session

This experiment modelling is a continution of the previous study with separate task and goal. This part also allows the participants to do specific tasks using the tool and answer the related questions. They were given introduction about the nature and purpose of the study. Then they were briefed about the different features of *SimEclipse* and the motivations behind it. The participants were then allowed to perform the tasks specified in Table 6.4 over a given subject system and operate for a while to get familiar with it.

After the demo session the questionnaire was handed to the participants. Each individual task was followed by a question that recorded the participants' opinion based on their level of agreement on the performed tasks. Answering the 'Overall Evaluation Question's was an open discussion session and the participants had the liberty to ask questions regarding any difficulty they faced during the task completion and knowing their views in further improvements of the tool.

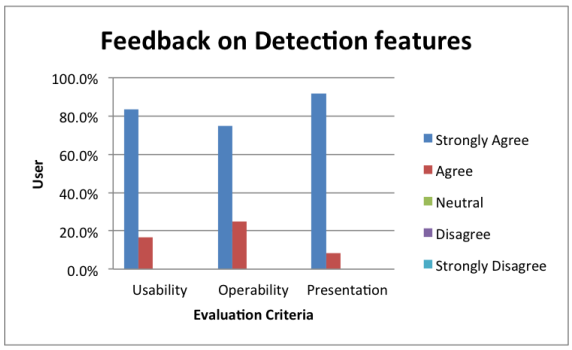
6.7.3 Summary of Findings

Feature Evaluation

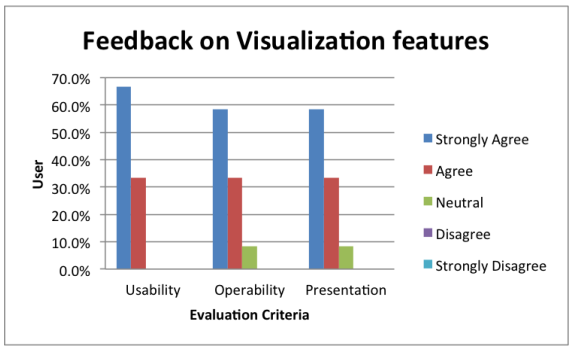
This section presents an analytical overview of the user evaluation of different features of *SimEclipse* in managing clones in software. User feedback on the feature evaluation questions are analyzed and presented in Figure 6.22 with separate component for each feature. From the feedback chart, it is clear that the features of *SimEclipse* got most of the remarks on the positive side of the likert scale from the study participants on all the three evaluation criteria (*Usability, Operability and Presentation*). Figure 6.23 presents a consolidated view of the positive feedback (Strongly Agree' or 'Agree') recived for all the features of *SimEclipse*. The

Table 6.4: Feature based tasks on *SimEclipse* for user study

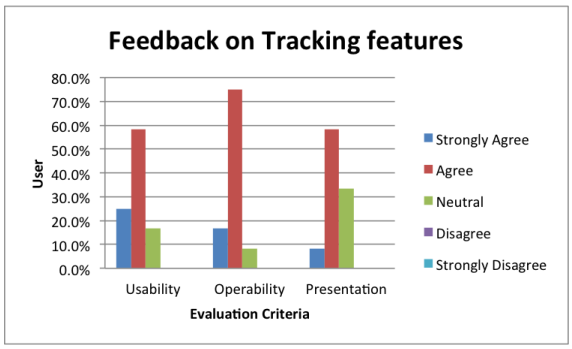
Feature	Task	Task Description
Clone Detection	- Full clone detection - Focused clone detection - On-the-fly/Instant clone detection from editor	- Detect clone in the whole project. - Explore the project hierarchy in SimEclipse *Navigator View* and detect clones for any package/-folder, file or source fragment. - Open a source file in editor, select/highlight one or more function or block of code and detect clone for the selected code.
Clone Visualization	- Detection result exploration - View clone code	- Explore detected clones in SimEclipse *Clones View*. - View/highlight the clone code in editor. - Compare clone fragments.
Clone Tracking	- Detect copy-paste clone - Detect non copy-paste clone - Existing clone code modification	- Enable Clone Tracking feature - Copy a function from a source file, paste the function in another source file and save the file. Make sure the new function is reported as clone with the source function from where it was being copied. - Pick a function from any source file in the project. Type code for a new function from the scratch in another source file, keeping some source code similarity with the previously picked function. Save the new function and make sure it is reported as clone with the picked function. - Select a clone class from the clone detection result. Select a clone fragment and do as much as source modification such that it should not be part of that clone class. Save the source file and perform clone detection on that source file. Make sure the modified source fragment does not appear in the detection result. Undo the chage and perform the detection again. Make sure the fragment appears again in the detection result.
	- On demand Clone Marking/Annotation (useful when clone tracking is disabled)	- Select a function from a file that is known to be a part of a clone class. Invove clone Annotation for that function. Make sure an *Information Marker* appears in the editor marker bar (as shown in Figure 6.13 and Figure 6.14) next to selected function with location information to other clone fragments of this clone class.
Clone Analysis	- Clone Genealogy View - Source Change History View	- Enable clone genealogy extraction - In settings, configure source location of the previous versions of the project - Detect clone genealogy for the project - Explore the clone genealogies shown in *SimEclipse Genealogy View* - Select a particular function from a source file and invoke for *Source Change History* from context menu. - Explore the source of the function in previous versions of the project from *SimEclipse Code History View*.



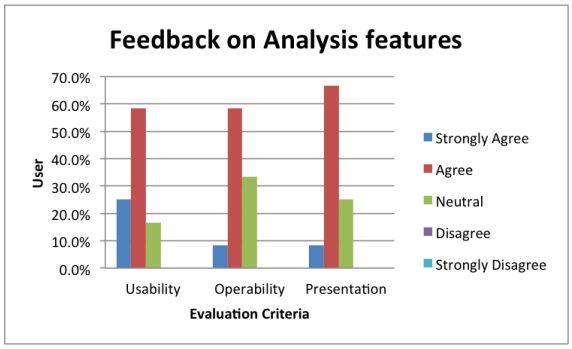
(a) Detection Feedback



(b) Visualization Feedback



(c) Analysis Feedback



(d) Tracking Feedback

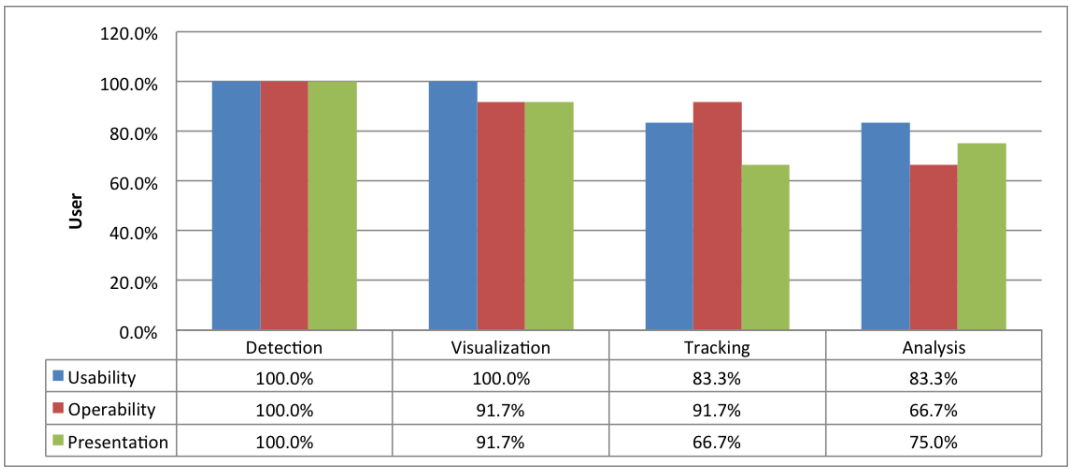
Figure 6.22: User opinion on Usability, Operability and Presentation for the features of *SimEclipse*

observation shows, 100% of the participants expressed their positive feedback (answering 'Strongly Agree' or 'Agree') on all the evaluation criteria for the *Detection*, and for *Visualization* features it was found 91.7%. For the remaining two features, at least 66.7% of them were being positive in the three evaluation criteria for *Analysis* and *Tracking* features. This indicates the *Analysis* and *Tracking* features need to be improved to further enhance the clone management experience using *SimEclipse*. The study also received various recommendations from user in improving and adding new features to *SimEclipse*, which will be discussed in the following section.

Feedback from the *Overall Experience Questions* has been analyzed and presented in Figure 6.24. Among the study participants, 83.3% of them agreed that *SimEclipse* would be *Helpful in Clone Management*. While 66.7% of the participants were found being agreed on *SimEclipse*'s *Overall good Performance*, 75% were found being *Overall Satisfied* in using the tool.

Future Improvements

At the end of the study session, the *Issues and Improvements Questions* were enabled us to get some advice on how can we improve the overall clone management experience by the developers using *SimEclipse*. We got valuable suggestions on existing feature improvements, especially for interactive visualization in the user interface part for Clone Tracking and Clone Analysis part. Although *SimEclipse* is capable to do focused



	Detection	Visualization	Tracking	Analysis
Usability	100.0%	100.0%	83.3%	83.3%
Operability	100.0%	91.7%	91.7%	66.7%
Presentation	100.0%	91.7%	66.7%	75.0%

Figure 6.23: Consolidated feedback on all the features of *SimEclipse*

clone search, a few users request for post-detection filtering of clone when detection is done in the whole project, search and filtering option in *Clone Genealogy View*. To enhance the clone visualization experience, implementation of model/graph based visualization also suggested by some participants. For additional features, notable requests we got were: enhanced highlighting of difference in clone comparison, portable clone documentation sharable among developers, clone refactor scheduling, linked-editing of clones, all of which we believe would be a great addition in our future improvement plan for *SimEclipse* to enhance the overall clone management experience of the developer.

6.8 *SimEclipse* Feature Comparison

In this thesis, we looked for what clone based technologies can be incorporated into the software developer's development workflow along with convenient solutions to make those technologies readily available in practice. The motivation of the study is to allow developers to deal with clone in software they are developing both in time and effort efficient ways. Towards that goal, we developed *SimEclipse* as a solution for making a number of clone based technologies readily available to software developers and clone researchers. Table 6.5 shows the feature matrix of *SimEclipse* compared to existing clone management tools discussed in Section 6.4. From the table, it is clear that *SimEclipse* supersedes the rest of the tools in providing integrated features to identify and understand software clones as well as address various clone management needs.

6.9 Addressing Research Questions

In this section we are going to answer the research question presented at the beginning of this chapter based on the study and experimental analysis.

RQ1: The research question was about identifying the clone based technologies that can be grouped

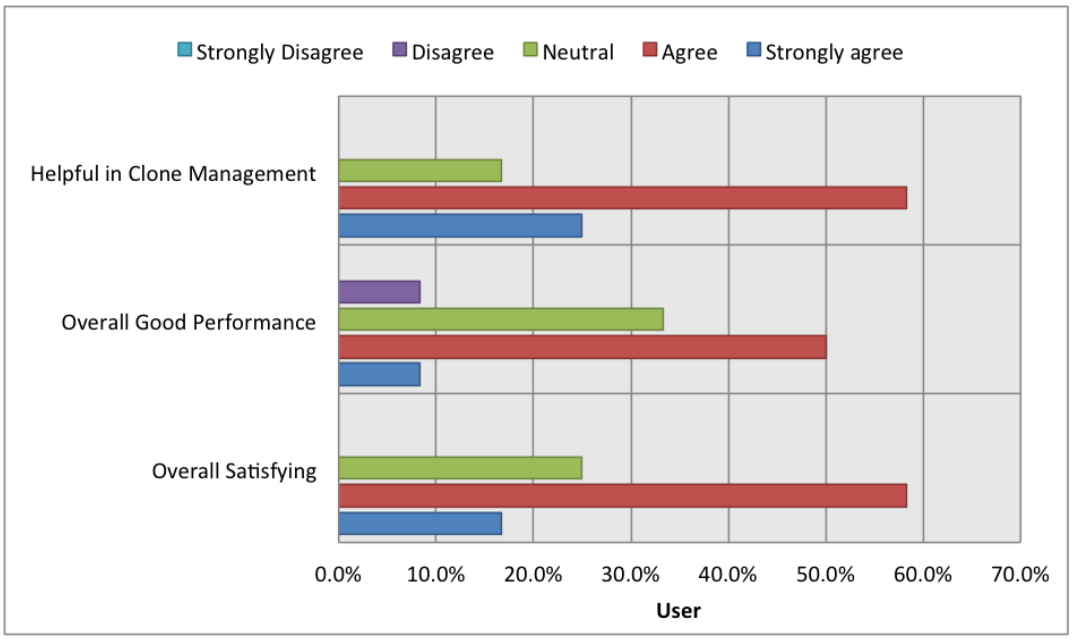


Figure 6.24: Overall user evaluation of *SimEclipse*

together in support of better clone management. This question has been addressed in Section 6.3.1. We have identified the individual components of managing clones in software and grouped them together in the form of some strategy. These strategies can be applied to deal with clones in various state of the project and its cloning situation.

RQ2: This question was about investigating the effectiveness of integrating clone based technologies into a single platform. The analysis done in Section 6.6 covers the answer to this question. Participants in the study were allowed to perform some clone based tasks using some discrete clone support tools and then using an IDE integrated clone management plugin. Based on their feedback after performing the same tasks in two different environments, it is clear that having various clone management functionalities under a single platform yields faster, convenient and less error prone clone management.

RQ3: This research question looks for feasible choice of platform that can be used for integration of clone based technologies towards effective clone management. Answer to this question is based on our analysis done in Section 6.6 and Section 6.7. Based the feedback gained from the 'Overall user experience questions' at the end of the study done in Section 6.6, we have seen that 100% of the participants felt more comfortable and confident in performing the given tasks using an integrated clone management tool in IDE other than using some discrete standalone tool. In this part of study, more relevant finding to this research question is, 60% of the participants expressed that they would prefer IDE as a platform over a Standalone Application Suite for providing tool support for clone management. So, we have developed *SimEclipse*, a clone aware IDE plugin to provide the software developers a platform for managing clones from within IDE. In the second part of our study done in Section 6.7, we tried to evaluate *SimEclipse*

Table 6.5: Feature Comparison of *SimEclipse* with other tools/plug-ins

Features	*ChP [72]*	*CSeR [76]*	*CloneBoard [43]*	*SHINOBI [91]*	*CloneDR [23]*	*Zibran and Roy [184]*	*JClone [10]*	*CloneTracker [46]*	*CeDAR [167]*	*SimEclipse*
Integrated Detection Engine	N	N	N	Y	Y	Y	Y	N*	N*	Y
Copy-paste Detection Only	Y	Y	Y	N	N	N	N	N	N	Y
Type-1 Clone Detection	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Type-2 Clone Detection	N	N	N	Y	Y	Y	Y	Y	Y	Y
Type-3 Clone Detection	N	N	N	N	Y	Y	N	Y	Y	Y
On-the-fly/Focused Detection	N	N	N	N	Y	Y	Y	N	N	Y
Clone Visualization	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Code Change History View	N	N	N	N	N	N	N	N	N	Y
Clone Genealogy View	N	N	N	N	N	N	N	N	N	Y
Clone Tracking	N	N	Y	N	N	N	N	Y	N	Y
Clone Refactoring	N	N	N	N	N	N	N	N	Y	N

* Use clone detection results from other standalone tool

based on the three evaluation criteria (*Usability, Operability and Presentation*) upon its features. We found, 83.3% of the participants considered *SimEclipse* would be *Helpful in Clone Management*. While 66.7% of the participants found agreed on *SimEclipse*'s *Overall good Performance*, 75% were found being *Overall Satisfied* in using the tool.

6.10 Summary

Lack of integration of clone management functionalities in a developer's work environment makes the task of managing clones in software a challenging and potentially error prone task. This chapter presents a user study in order to answer some research questions on finding a way to provide efficient clone management support to the developers. Here we also present *SimEclipse*, a tool intended to promote the practical use of code clone research. Our main contribution is an approach for integrating clone detection functionality and clone management techniques in particular with the software development environment. The highly configurable clone detection library developed in our previous study [172] provides us a fast software code clone detector. Based on that, *SimEclipse* was developed to investigate how we can integrate clone detection with software development to better support clone management and software evolution. By integrating some

other clone analysis and management features, it provides a clone detection friendly and clone-aware software development environment that forms a strong basis for developing IDE based tools for managing clones. Our small scale user study shows that *SimEclipse* has great potential to be used both in research and industry in dealing with clones.

Chapter 7

Conclusion and Future Work

Cloning is a common phenomenon found in almost all kinds of software systems during their evolutions. Investigating and understanding a software system's code clones is important to manage clones effectively. Previous studies reported that duplicate code in software systems ranges from 9-17% [183] to 50% [144]. Cloning can be a substantial problem during development and maintenance unless special care is taken to find and track existing clones and their evolution [78]. Source code clone detection has been greatly studied over the past decade. A number of state of the art clone detection tools are in use in software clone research [151], and the majority of the work focused on the detection and analysis of code clones. Current research trend in the area is mostly on determining what to do with the clones when they are detected, i.e., finding the ways and means to manage the clones, gaining more control of their generation and studying clone evolution and their effects on the evolution of software. Roy et al. [152] conducted the very first comprehensive survey on software clone management, which points to the achievements so far in this area, and reveal the opportunities for further research necessary towards an integrated clone management system, and developing a prototype of such a system is one of the main parts of this thesis.

In this thesis, we first presented a fast and scalable near-miss structural software clone detection process. We made this process available for practical use in the form of a standalone clone detection tool *Simcad* as well as a portable clone detection library *SimLib*. For the evaluation, we performed a user study on *SimLib* to get some feedback on installation and use of the API and to uncover its potential use as a third-party clone detection library. Our small scale user study demonstrates that *SimLib* could be a potential library towards providing support for various clone/similarity detection needs to the other applications. Finally, we have also conducted an empirical study in order to answer three research questions on finding a way to provide efficient clone management support to the developers. In support of the study, we presented a clone-aware software development platform *SimEclipse* as a handy tool to accomplish the task of clone detection and various clone management activities in an easier and meaningful way. The objective of developing such a tool was to make the clone management activities as a part of the software development lifecycle. As clones in software are mainly evolved from various coding activities of the developer, we have shown that ready-made tool support for clone based technologies in a typical software development tool (e.g., Integrated Development Environment) as an IDE plugin would make a great impact in getting efficient control of managing evolution of clones in software systems. By using the clone tracking facility in *SimEclipse*, some of the prominent

causes of clone origination like accidental code copying or not knowing of the previous existence of code can be avoided. Identifying clones in the IDE will also make the clone refactoring task easier. We conducted a user study based evaluation of *SimEclipse* plugin considering the three evaluation criteria (*Usability, Operability and Presentation*), and majority of the study participants found it to be 'Helpful in Clone Management'. We hope *SimEclipse* will help developers deal with cloned code from within an IDE and support clone management from where a clone evolves. Having these clone based services under the same hood would be handy in detection, visualization and management of clones in software.

We have presented three tools in this thesis that are readily available to be used in practice by various types of users. Following are some target users of these tools:

- *SimCad:* Developer/researcher for clone analysis in software source code or non source code based data.
- *SimLib:* It would help a developer/researched in developing 1) more advanced clone detection tool customized for particular type of data, 2) other clone analysis/visualization/management tool where clone detection is a pre-requisite of the tool.
- *SimEclipse:* It would help a software developer managing clones in the software s/he in working on.

We hope that both researchers and developers would enjoy and utilize the benefit of using these tools in different aspect of code clone research and easily manage cloned codes in their projects.

Since current research trend in software clones states strong emphasis on clone management, in the future, we would like to focus mainly on improving *SimEclipse* by addressing the feedbacks and improvement suggestions we received from our tool evaluation study. The target of this future improvement will be to enhance the overall clone management experience of the developers using *SimEclipse*. Our evaluation study on *SimEclipse* shows that the clone 'Tracking' and 'Analysis' features require further improvements to meet users expectations, which would be first thing to work on in our future plan. Besides, implementation of new features like enhanced highlighting of difference in clone comparison, portable clone documentation sharable among developers, clone refactor scheduling, linked-editing of clones, all of which we believe would be a great addition to our future improvement plan to establish *SimEclipse* as a viable tool for integrated clone management.

References

[1] K. Abd-El-Hafiz. A metrics-based data mining approach for software clone detection. In *Computer Software and Applications Conference*, pages 35–41, 2012x.

[2] E. Adar. GUESS: a language and interface for graph exploration. In *CHI*, pages 791–800. ACM, 2006.

[3] E. Adar and M. Kim. SoftGUESS: Visualization and exploration of code clones in context. In *ICSE*, pages 762 –766, 2007.

[4] G. Alkhatib. The maintenance problem of application software. In *Journal of Software Maintenance*, volume 4, pages 83–104, 1992.

[5] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *ICSM*, page 273, 2001.

[6] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755 – 765, 2002.

[7] M. Asaduzzaman. Visualization and analysis of software clones. M.Sc. thesis, University of Saskatchewan, Canada, 2011.

[8] M. Asaduzzaman, C. Roy, and K. Schneider. VisCad: flexible code clone analysis support for NiCad. In *IWSC*, pages 77–78. ACM, 2011.

[9] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *CSMR*, pages 81 –90, 2007.

[10] M. Bahtiyar. JClone : Syntax tree based clone detection for java. Master's thesis, Linnaeus University, 2010.

[11] B. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

[12] B. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, pages 86 –95, 1995.

[13] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *ICSM*, pages 24–33, 2007.

[14] M. Balazinska, E. Merlo, M. Dagenais, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *WCRE*, pages 98–107. IEEE Computer Society Press, 2000.

[15] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS*, pages 292 –303, 1999.

[16] D. Barbara and P. Chen. Using self-similarity to cluster large data sets. In *Data mining and Knowledge Discovery*, volume 7, pages 123–152, 2003.

[17] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *ICSM*, pages 273 –282, 2011.

[18] L. Barbour, H. Yuan, and Y. Zou. A technique for just-in time clone detection. In *ICPC*, pages 76–79, Washington DC, USA, 2010.

[19] H. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. *SIGSOFT Softw. Eng. Notes*, 30:156–165, 2005.

[20] H. Basit and S. Jarzabek. A data mining approach for detecting higher-level clones in software. In *Transactions on Software Engineering*, volume 35 (4), pages 497– 514. IEEE, 2009.

[21] H. Basit, S. Puglisi, W. Smyth, A. Turpin, and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *ESEC-FSE companion*, pages 513–516. ACM, 2007.

[22] H. Basit, D. Rajapakse, and S. Jarzabek. Beyond templates: a study of clones in the stl and some general implications. In *ICSE*, pages 451–459, 2005.

[23] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, page 368, 1998.

[24] S. Bellon. Vergleich von techniken zur erkennung duplizierten quellcodes. Diploma thesis, Universität Stuttgart, 2002.

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

[26] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan. An empirical study on inconsistent changes to code clones at release level. In *WCRE*, pages 85–94, 2009.

[27] B. Biegel and S. Diehl. Highly configurable and extensible code clone detection. In *WCRE*, pages 237–241, Beverly, MA, USA, 2010.

[28] B. Biegel and S. Diehl. Jccd: a flexible and extensible api for implementing custom code clone detectors. In *Automated Software Engineering*, pages 167–168, Antwerp, Belgium, 2010.

[29] Blue Edge Bulgaria. *SimScan - Similarity Scanner*. http://blue-edge.bg/download.html, last access: Aug 2011.

[30] P. Bulychev. *CloneDigger Tool*. http://clonedigger.sourceforge.net, Last Accessed August 2013.

[31] P. Bulychev and M. Minea. Duplicate code detection using anti-unification. In *Colloquium on Software Engineering*, pages 51–54, St. Petersburg, Russia, 2008.

[32] F. Calefato, F. Lanubile, and T. Mallardo. Function clone detection in web applications. In *Journal of Web Engineering*, volume 3 (1), pages 3–21, 2004.

[33] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.

[34] M. Chilowicz, E. Duris, and G. Roussel. Syntax tree fingerprinting for source code similarity detection. In *ICPC*, pages 243–247, 2009.

[35] A. Chiu and D. Hirtle. Beyond clone detection. CS846 Course Project Report, University of Waterloo, 2007.

[36] K. W. Church and J. I. Helfman. Dotplot: A program for exploring self-similarity in millions of lines for text and code. In *Journal of American Statistical Association*, volume 2(2), pages 153–174, June 1993.

[37] J. Cordy. Practical challenges to software maintenance automation. In *IWPC*, pages 196–206. IEEE, 2003.

[38] J. Cordy. Live scatterplots. In *IWSC*, pages 79–80. ACM, 2011.

[39] J. Cordy, T. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *CASCON*, pages 1–12. IBM Press, 2004.

[40] A. Cuomo, A. Santone, and U. Villano. A novel approach based on formal methods for clone detection. In *International Workshop on Software Clones*, pages 8–14, 2012.

[41] M. Datar, N. Immorlica, Indyk P., and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoGG*, pages 253–262, Brooklyn, New York, USA, June 2004.

[42] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3/4):219 – 236, 1995.

[43] M. de Wit. *Managing Clones Using Dynamic Change Tracking and Resolution.* M.Sc. thesis, Delft University of Technology, 2008.

[44] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ICSE*, pages 603–612. ACM, 2008.

[45] G. Di Lucca, M. Di Penta, and A. Fasolino. An approach to identify duplicated web pages. In *COMPSAC*, pages 481 – 486, 2002.

[46] E. Duala-Ekoko and M. Robillard. CloneTracker: tool support for code clone management. In *ICSE*, pages 843–846, 2008.

[47] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109–118, 1999.

[48] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[49] W. Evans, C. Fraser, and F. Ma. Clone detection via structural abstraction. In *WCRE*, pages 150–159. IEEE Computer Society, 2007.

[50] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Softw. Engg.*, 13:601–643, 2008.

[51] M. Fowler, K. Beck, J.Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley Professional, 1999.

[52] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE*, pages 321–330. ACM, 2008.

[53] S. Giesecke. Generic modelling of code clones. In *DRSS*, pages 1–23, 2007.

[54] D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. In *SIGCSE*, pages 266–270. ACM, 1999.

[55] N. Göde. Evolution of type-1 clones. In *SCAM*, pages 77–86, 2009.

[56] N. Göde. Clone removal: Fact or fiction. In *International Workshop on Software Clones*, pages 22–40, Cape Town, SA, 2010.

[57] N. Göde and J. Harder. Clone stability. In *CSMR*, pages 65 –74, 2011.

[58] N. Göde and R. Koschke. Studying clone evolution using incremental clone detection. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 1–28, 2010.

[59] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *ICSE*, pages 311–320. ACM, 2011.

[60] C. Gong, X. Huang, Cheng, and S. Bai. Detecting near-duplicates in large-scale short text databases. In *PAKDD*, pages 877–883, 2008.

[61] S. Grant and J. Cordy. Vector space analysis of software clones. In *ICPC*, pages 233–237, Vancouver, BC, Canada, 2009.

[62] J. Harder and N. Göde. Efficiently handling clone data: Rcf and cyclone. In *IWSC*, pages 81–82. ACM, 2011.

[63] S. Harris. *Simian - Similaruty Analyzer*. http://www.harukizaemon.com/simian, last access: Aug 2011.

[64] M. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR*, pages 284–291, 2006.

[65] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support environment based on code clone analysis. In *IASTED-SEA*, pages 222–229. ACTA Press, 2004.

[66] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. *Code Clone Analysis Methods for Efficient Software Maintenance*. PhD thesis, Graduate School of Information Science and Technology, Osaka University, 2006.

[67] Y. Higo and S. Kusumoto. Enhancing quality of code clone detection with program dependency graph. In *WCRE*, pages 315 –316, 2009.

[68] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On software maintenance process improvement based on code clone analysis. In *PROFES*, pages 185–197. Springer-Verlag, 2002.

[69] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental code clone detection: A PDG-based approach. In *WCRE*, pages 3 –12, 2011.

[70] W. Hordijk, M. Ponisio, and R. Wieringa. Harmfulness of code duplication: A structured review of the evidence. In *EASE*, pages 88–97, 2009.

[71] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software. In *IWPSE-EVOL*, pages 73–82. ACM, 2010.

[72] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *ICPC*, pages 238–242, 2009.

[73] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *ICSM*, pages 1 –9, 2010.

[74] P. Jablonski and D. Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *ETX*, pages 16–20, 2007.

[75] P. Jablonski and D. Hou. Aiding software maintenance with copy and paste clone awareness. In *ICPC*, pages 170–179, 2010.

[76] F. Jacob, D. Hou, and P. Jablonski. Actively comparing clones inside the code editor. In *IWSC*, pages 9–16. ACM, 2010.

[77] S. Jarzabek and Y. Xue. Are clones harmful for maintenance. In *IWSC*, pages 73–74, 2010.

[78] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, pages 96–105, 2007.

[79] Z. Jiang and A. Hassan. A framework for studying clones in large software systems. In *SCAM*, pages 203 –212, 2007.

[80] Z. Jiang, A. Hassan, and R. Holt. Visualizing clone cohesion and coupling. In *APSEC*, pages 467 –476, 2006.

[81] J. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*, pages 171–183. IBM Press, 1993.

[82] J. Johnson. Substring matching for clone detection and change tracking. In *ICSM*, pages 120 –126, 1994.

[83] J. Johnson. Visualizing textual redundancy in legacy source. In *CASCON*, pages 32–41. IBM Press, 1994.

[84] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective - a workbench for clone detection research. In *ICSE*, pages 603–606, 2009.

[85] E. Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy and paste. In *CSMR*, pages 78–87, Madrid, Spain, 2010.

[86] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE*, pages 485–495, 2009.

[87] T. Kamiya. *CCFinderX Tool*. http://www.ccfinder.net, Accessed August 2013.

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

[89] C. Kapser and M. Godfrey. Improved tool support for the investigation of duplication in software. In *ICSM*, pages 305–314. IEEE Computer Society, 2005.

[90] C. Kapser and M. Godfrey. Cloning considered harmful" considered harmful. In *WCRE*, pages 19–28, 2006.

[91] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. SHINOBI: A tool for automatic code clone detection in the IDE. In *WCRE*, pages 313–314, 2009.

[92] I. Keivanloo, C. Roy, and J. Rilling. ava bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *IWSC*, pages 36–42, 2012.

[93] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *International Symposium on Empirical Software Engineering*, pages 83–92. IEEE, 2004.

[94] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR*, pages 1–5. ACM, 2005.

[95] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.

[96] E. Kodhai, V. Vijayakumar, G. Balabaskaran, T. Stalin, and B. Kanagaraj. Method level detection and removal of code clones in C and Java programs using refactoring. In *IJJCET*, pages 93–95. Gopalax Publications & TCET, 2010.

[97] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *POPL*, pages 155–169, Boston, MA, USA, January 2000. ACM.

[98] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56. Springer-Verlag, 2001.

[99] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. In *ASE*, volume 3 (1–2), pages 77–108, 1996.

[100] S. Kosaraju. Faster algorithms for the construction of parameterized suffix trees. In *FOCS*, pages 631–638, 1995.

[101] R. Koschke. Survey of research on software clones. In *DRSS*, pages 1–24, 2006.

[102] R. Koschke. Frontiers of software clone management. In *FoSM*, pages 119 –128, 2008.

[103] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE*, pages 253–262, 2006.

[104] R. Koschke, J. Girard, and M. Wrthner. An intermediate representation for reverse engineering analyzes. In *WCRE*, pages 241–250, Hawai, USA, 1998.

[105] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301–309, 2001.

[106] J. Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE*, pages 170 –178, 2007.

[107] J. Krinke. Is cloned code more stable than non-cloned code? *SCAM*, 0:57–66, 2008.

[108] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM*, pages 314–321. IEEE Computer Society, 1997.

[109] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. In *Transactions on Software Engineering*, volume 29(9), pages 782–795, Semptember 2003.

[110] T. Lavoie, M. Eilers-Smith, and E. Merlo. Challenging cloning related problems with gpu-based algorithms. In *IWSC*, pages 25–32, Cape Town, SA, 2010.

[111] M. Lee, J. Roh, S. Hwang, and S. Kim. Instant code clone search. In *FSE*, pages 167–176, 2010.

[112] S. Lee and I. Jeong. SDD: high performance code clone detection system for large scale source code. In *OOPSLA*, pages 140–141, 2005.

[113] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176 – 192, 2006.

[114] Z. Li and J. Sun. A metric space based software clone detection approach. In *Software Engineering and Data Mining*, pages 111–116, Chengdu, China, 2010.

[115] C. Liu, C. Chen, J. Han, and P. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *KDD*, Philadelphia, USA, August 2006. ACM.

[116] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Analysis of the linux kernel evolution using code clone coverage. In *MSR*, page 22, 2007.

[117] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *ICSE*, pages 106–115, 2007.

[118] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *ICSM*, pages 227–236, 2008.

[119] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *MSR*, pages 18–18, 2007.

[120] K. Maeda. Experience of xml-based source code representation with parsing actions. In *SoMeT*, pages 330–339, 2007.

[121] K. Maeda. Syntax sensitive and language independent detection of code clones. In *World Academy of Science, Engineering and Technology*, volume 60, pages 350–354, 2009.

[122] U. Manber. Finding similar files in a large file system. In *Usenix Technical Conference*, pages 1–10, 1994.

[123] G. Manku, A. Jain, and A. Sarma. Detecting near-duplicates for web crawling. In *WWW*, pages 141–150, 2007.

[124] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *ASE*, pages 107 – 114, 2001.

[125] J. Mayrand, B. Lague, and J. Hudepohl. Evaluating the benefits of clone detection in the software maintenance activities in large scale systems. In *WESS*, 1996.

[126] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, pages 244 –253, 1996.

[127] E. McCreight. A space-economical suffix tree construction algorithm. In *Journal of the ACM*, volume 2 (2), pages 262–272, 1976.

[128] M. Mondal, C. Roy, M. Rahman, R. Saha, J. Krinke, and K. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *ACM-SAC (SE Track)*, pages 1–8, 2012 (to appear).

[129] M. Mondal, C. Roy, and K. Schneider. Dispersion of changes in cloned and non-cloned code. In *IWSC*, pages 29 – 35, 2012.

[130] M. Mondal, C. Roy, and K. Schneider. An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study. In *Science of Computer Programming*, page 44, 2014.

[131] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *METRICS*, pages 87–94, Ottawa, Canada, 2002. IEEE.

[132] L. Moonen. Generating robust parsers using island grammars. In *WCRE*, pages 13–22, 2001.

[133] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Gapped code clone detection with lightweight source code analysis. In *International Conference Program Comprehension*, pages 93–102, 2013.

[134] E. W. Myers. An o (nd) difference algorithm and its variations. In *Algorithmica*, volume 1(1), pages 251–266, 1986.

[135] S. Nasehi, G. Sotudeh, and M. Gomrokchi. Source code enhancement using reduction of duplicated code. In *IASTED*, pages 192–197. ACTA Press, 2007.

[136] T. Nguyen, H. Nguyen, J. Al-Kofahi, N. Pham, and T. Nguyen. Scalable and incremental clone detection for evolving software. In *ICSM*, pages 491 –494, 2009.

[137] J. Pate, R. Tairas, and N. Kraft. Clone evolution: a systematic review. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 1–23, 2011.

[138] J. Patenaude, E. Merlo, M. Dagenais, and B. Lagu e. Extending software quality assessment techniques to java systems. In *IWPC*, pages 49–56, Washington, DC, USA, 1999. IEEE.

[139] B. Pi, S. Fu, W. Wang, and S. Han. Simhash-based effective and efficient detecting of near-duplicate short messages. In *ISCSCT*, pages 020–025, 2009.

[140] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell. In *Mining Software Repositories*, pages 72–81, Cape Town, South, Africa, 2010.

[141] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. volume 55(7), pages 1165–1199. Information and Software Technology, July 2013.

[142] A. Raza, G. Vogel, and E. Plödereder. *Project Bauhaus.* http://www.bauhaus-stuttgart.de, Last Accessed August 2013.

[143] M. Rieger. *Effective Clone Detection Without Language Barriers.* Phd thesis, Institut fuï Informatik und angewandte Mathematik, Germany, 2005.

[144] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *WCRE*, pages 100–109, 2004.

[145] C. Roy. Detection and analysis of near-miss software clones. In *ICSM*, pages 447–450, 2009.

[146] C. Roy and J. Cordy. A survey on software clone detection research. Tech Report TR 2007-541, School of Computing, Queens University, Canada, 2007.

[147] C. Roy and J. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.

[148] C. Roy and J. Cordy. Towards a mutation-based automatic framework for evaluating code clone detection tools. In *C3S2E*, pages 137–140, 2008.

[149] C. Roy and J. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *ICSTW*, pages 157–166, 2009.

[150] C. Roy and J. Cordy. Near-miss function clones in open source software: an empirical study. *J. of Softw. Maintenance and Evolution: Research and Practice*, 22(3):165–189, 2010.

[151] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74:470–495, 2009.

[152] C. Roy, M. Zibran, and R. Koschke. The vision of software clone management: Past, present and future. In *Software Evolution Week (SEW'14)*, page 16. IEEE, 2014.

[153] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *ISSTA*, pages 117–128. ACM, 2009.

[154] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, and K. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *SCAM*, pages 87–96, 2010.

[155] R. Saha, C. Roy, and K. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *ICSM*, pages 293 –302, 2011.

[156] R. Saha, C. Roy, and K. Schneider. Visualizing the evolution of code clones. In *IWSC*, pages 71–72. ACM, 2011.

[157] R. Saha, C.. Roy, K. Schneider, and D. Perry. Understanding the evolution of type-3 clones: an exploratory study. In *MSR*, pages 139–148, Piscataway, NJ, USA, 2013. IEEE.

[158] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue. Finding file clones in freebsd ports collection. In *Mining Software Repositories*, pages 102–105, Cape Town, South, Africa, 2010.

[159] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, and Y. Zou. Studying the impact of clones on software defects. In *WCRE*, pages 13 –21, 2010.

[160] G. Selim, K. Foo, and Y. Zou. Enhancing source-based clone detection using intermediate representation. In *WCRE*, pages 227–236, 2010.

[161] F. Smith and M. Waterman. denti fi cation of common molecular identification of common molecular subsequences. In *Journal of Molecular Biology*, volume 147, pages 195–197, 1981.

[162] R. Smith and S. Horwitz. Detecting and measuring similarity in code clones. In *IWSC*, pages 28–34, 2009.

[163] H. Sutton, A. andKagdi, J. Maletic, and G. Volkert. Hybridizing evolutionary algorithms and clustering algorithms to find source-code clones. In *GECCO*, pages 1079–1080, Washington DC, USA, 2005.

[164] J. Svajlenko, C. Roy, and J. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *IWSC*, pages 8–9, 2013.

[165] R. Tairas. Centralizing clone group representation and maintenance. In *OOPSLA*, pages 781–782, Orlando, Florida, USA, 2009.

[166] R. Tairas and J. Gray. Phoenix-based clone detection using suffix trees. In *ACM-SE*, pages 679–684, 2006.

[167] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. In *Information and Software Technology*, volume 54, pages 1297–1307, 2012.

[168] S Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15:1–34, 2010.

[169] R. Tiarks, R. Koschke, and R. Falke. An extended assessment of type-3 clones as detected by state-of-the-art tools. In *Software Quality Journal*, volume 19(2), pages 295–331, 2011.

[170] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *VLHCC*, pages 173–180. IEEE Computer Society, 2004.

[171] M. Uddin, C. Roy, and Schneider K. Simcad: An extensible and faster clone detection tool for large scale software systems. In *ICPC*, San Francisco, CA, USA, May 2013.

[172] M. Uddin, C. Roy, K. Schneider, and A. Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *WCRE*, pages 13 –22, 2011.

[173] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *METRICS*, pages 67–76. IEEE Computer Society Press, 2002.

[174] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *APSEC*, pages 327 – 336, 2002.

[175] R.D. Venkatasubramanyam, H.K. Singh, and K. Ravikanth. A method for proactive moderation of code clones in ides. In *IWSC*, pages 62–66, 2012.

[176] V. Wahler, D. Seipel, J. Wolff, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM*, pages 128 –135, 2004.

[177] V. Weckerle. CPC: an eclipse framework for automated clone life cycle tracking and update anomaly detection. Master's thesis, Freie Universität Berlin, Germany, 2008.

[178] M. Weiser. Program slicing. In *IEEE Transactions on Software Engineering*, volume 10(4), pages 352–357, July 1984.

[179] R. Wettel and R. Marinescu. Archeology of code duplication: recovering duplication chains from small duplication fragments. In *SYNASC*, pages 63–70, Sep 2005.

[180] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets. In *SDM*, pages 166–177, San Francisco, CA, USA, May 2003.

[181] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21:739–755, 1991.

[182] Y. Yuan and Y. Guo. Boreas: an accurate and scalable token-based approach to code clone detection. In *Automated Software Engineering*, pages 286–289, 2012.

[183] M. Zibran and C. Roy. Towards flexible code clone detection, management, and refactoring in IDE. In *IWSC*, pages 75–76, 2011.

[184] M. Zibran and C. Roy. IDE-based real-time focused search for near-miss clones. In *ACM-SAC (SE Track)*, pages 1–8, 2012.

[185] M. Zibran, R. Saha, M. Asaduzzaman, and C. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *ICECCS*, pages 295–304, 2011.