DESIGN AND DEVELOPMENT OF AN EFFICIENT SOFTWARE CLONE DETECTION TECHNIQUE

A THESIS

Submitted in fulfillment of the
requirements for the award of the degree of

Doctor of Philosophy

Submitted By

Dhavleesh Rattan
(950903019)

Under the Supervision of

Dr. Rajesh Bhatia
Professor and Head,
Computer Science and Engineering
PEC University of Technology,
Chandigarh -160012

Dr. Maninder Singh
Associate Professor
Computer Science and Engineering
Thapar University,
Patiala-147004

COMPUTER SCIENCE & ENGINEERING DEPARTMENT

THAPAR UNIVERSITY, PATIALA – 147004 (INDIA)

July 2015

CERTIFICATE

I hereby certify that the work which is presented in this thesis entitled **DESIGN AND DEVELOPMENT OF AN EFFICIENT SOFTWARE CLONE DETECTION TECHNIQUE,** in fulfillment of the requirements for the award of degree of **DOCTOR OF PHILOSOPHY** submitted in Computer Science and Engineering Department (CSED), Thapar University, Patiala, Punjab is an authentic record of my own work carried out under the supervision of Dr. Rajesh Bhatia and Dr. Maninder Singh, and refers the work of other researchers, which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

(Dhavleesh Rattan)
Registration No. 950903019

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge and belief.

(Dr. Rajesh Bhatia)
Professor and Head
Computer Science and Engineering
Department, PEC University of Technology
Chandigarh – 160012
Supervisor

(Dr. Maninder Singh)
Associate Professor
Computer Science and Engineering
Department, Thapar University
Patiala – 147004
Supervisor

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisors Dr. Rajesh Bhatia and Dr. Maninder Singh for their encouragement, painstaking supervision, innovative suggestions and invaluable help during the entire period of my Ph. D. thesis. It has been a great honour and pleasure for me to work under their supervision. I learned a great deal from them, not only about research but also matters touching many other aspects which will benefit me in future life and career.

I would like to acknowledge the Director and Dean (RSP), Thapar University for the academic and technical support which has been indispensible. I express my gratitude to Doctoral Committee comprising of Dr. Deepak Garg, Dr. R. K. Sharma and PG Coordinators Dr. Inderveer Chana and Dr. Parteek Bhatia for monitoring the progress and providing valuable suggestions for improvement from time to time. I am grateful to anonymous referees of international research journals in the field of software engineering.

Finally, I could not reach the important milestones of my life without the support and encouragement of my family. Thanks to my parents who have made it possible for me to reach where I am today. I express my sincere appreciation to my wife Geetanjali and children Rishika and little Kaku for their love and patience when it was most required.

And above all, I am thankful to Almighty whose divine grace gave me the required courage, strength and perseverance to overcome various obstacles that stood in my life.

Dhavleesh Rattan

ABSTRACT

Reusing software by means of copy and paste is a frequent activity in software development. In source code and other software artifacts, the original (code) fragment is copied and pasted with or without modifications. The pasted (code) fragment is said to be a clone and this activity is known as (code) cloning. The presence of code clones in the software may increase the post implementation maintenance (preventive and adaptive) effort. Code cloning increases the probability of bug propagation. Software clones are classified depending upon the type of similarity between two code fragments and the level of granularity. There are many reasons which promote software cloning. Complexity of the large systems makes it difficult for the software developer to understand the functionality. It promotes copying the existing functionality and logic. Sometimes programmers are forced to copy and paste code due to limitations of code reuse in programming languages. Moreover, programmers often fear to bring in new ideas in existing software. It is easier to reuse the existing code than to develop a fresh solution since new code may introduce new errors. There is an urgent need to detect clones in various software artifacts. Now-a-days, model driven development has become standard industry practice, so the objective of the proposed work is to detect clones in object oriented systems by using Unified Modeling Language (UML) models. In the proposed work, two techniques are presented to detect clones in UML models.

In our work, we surveyed wide range of literature. 213 articles out of a collection of 2039 are surveyed using the standard systematic literature review guidelines. We put an emphasis on clone management, model clones, and semantic clones and classified the literature in different key areas. The focus of our survey is broader than the earlier surveys and includes the latest research work related to software clones. In addition to clone detection tools and methods, we have addressed other issues related to software clone research such as clone analysis, clone evolution and impact of clones on software quality. We used a systematic method to develop a clone management map which identifies how clone management papers overlap with clone detection method papers and clone detection tool papers. We explored the model based and semantic clones in detail and compared the state of the art techniques.

The first approach detects clones in UML class models. The technique accepts the XMI file of a UML class model as input. The core of our technique is the construction of a labeled, ranked tree by carefully mapping the elements parsed from the XMI file to the tree representation. The duplicate subtrees are grouped and clustered with the aim to detect exact and meaningful clones. The major contributions of the first approach are:

- Detection of model clones in UML class diagrams at different levels of granularity i.e. single attribute/operation, set of attributes/operations and recurring classes with their members.
- Detection and classification of model clones as:
    - Type-1 : model clones due to standard modeling/coding practice
    - Type-2 : model clones by purpose
    - Type-3 : model clones due to design practices

Since UML modeling has got inherent object oriented features, thus our classification of model clones is inspired from these object oriented characteristics of UML class model. We carried out the empirical evaluation on reverse engineered open source systems due to the unavailability of standard repository of UML models recognized by modeling community. Moreover, we are also considering forward designed models for evaluation to capture the essence of model driven development and to check the practical relevance of the proposed approach. We believe that the results of the tool are accurate and relevant for practical purposes and demand further investigation.

The second technique is based on computing similarity across object oriented programs at different levels of granularity. The tool is able to detect concept level similarities by applying latent semantic indexing and principal component analysis. Detection of high level similarities can help in comprehending the design of the system for better maintenance. We have extended our tool to detect similarities for UML diagrams by measuring the distance between two class models. In addition, we mined important change patterns at method level using multi-version program analysis by applying the proposed technique throughout the evolutionary history of the open source software *dnsjava*. We have validated the similarity score by applying the tool at function level in the source code.

CONTENTS

ABBREVIATIONS

ADT	Abstract Data Type
AOP	Aspect Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
CeDAR	Clone Detection, Analysis and Refactoring
ConQAT	Continuous Quality Assessment Toolkit
CPU	Central Processing Unit
CRD	Centre for Reviews and Dissemination
DP	Dynamic Programming
FIM	Frequent Itemset Mining
GPU	Graphics Processing Unit
HTML	Hyper Text Markup Language
ICA	Independent Component Analysis
IDE	Integrated Development Environment
IEEE	The Institute of Electrical and Electronics Engineers
JDK	Java Development Kit
LCS	Longest Common Subsequence
LOC	Lines of Code

LSI	Latent Semantic Indexing
LSH	Locality Sensitive Hashing
ME	Model Element
OMG	Object Management Group
PCA	Principal Component Analysis
PDG	Program Dependence Graph
RTF	Repeated Tokens Finder
SRS	Software Requirements Specification
STL	Standard Template Library
SVD	Singular Value Decomposition
UML	Unified Modeling Language
VSM	Vector Space Model
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XVCL	XML Variant Configuration Language

List of Figures

List of Tables

Chapter 1

Introduction and Statement of the Problem

This chapter provides a short introduction to the thesis. We provide a general motivation to this thesis in section 1.1. In section 1.2, we provide the problem statement. In section 1.3, we list the research objectives and section 1.4 explains the contributions of the thesis in brief. Finally, in section 1.5 we provide an outline of remaining chapters.

1.1 Motivation

Copying existing code fragments and pasting them with or without modifications into other sections of code is very frequent process in software development. The copied code is called a *software clone* and the process is called *software cloning*. Software developers tend to copy the existing functionality from the source code and paste it somewhere else either intentionally or unintentionally. The increasing use of open source software and its variants increased code reuse, too. Existing code can be modified to cater to new requirements thereby facilitating and advancing open source development. The importance of clone detection can be gazed from the fact that large percentages of different software artifacts contain duplication as shown in Table 1.1. The table shows different clone detection techniques and subject systems and it reflects the percentage by which the code can be shrunk in software.

Table 1.1 Percentage of cloned code across different systems

Sr. No.	Percentage of cloned code	Subject System	Size (LOC)	Clone Detection Technique	Citations
1.	13%- 20%	SS Subsystem	1.1 M	Token based	[12]
2.	5 % - 20%	Telecommunication Monitoring Systems	1M	Metrics based	[174]
3.	12.7%	Process Control System	400K	Tree based	[23]
4.	50%	gcc and other application projects	6.5K – 460 K	Text based	[52]

Sr. No.	Percentage of cloned code	Subject System	Size (LOC)	Clone Detection Technique	Citations
5.	8.3% - 14.8%	E-Business Website and Resource Management System	15-35 Sequence Diagrams	Model based	[158]
6.	20% - 50%	Java Source Code, C# Source Code	425K, 16K	Tree based	[54]
7.	14.9%	SRS	2500 pages	Text based	[49]
8.	12.1% - 32.1%	Open source Python Systems	9KLOC – 272KLOC	Hybrid	[197]

Fowler et al. [58] mentions duplication of code as one of the bad practices in software development increasing maintenance cost. A bug detected in one section of code therefore requires correction in all the replicated fragments of code. Thus, it is important to find all related fragments throughout the source code [168]. Considering the high maintenance cost, software clone detection has emerged as an active research area. Different programming paradigms and languages have led to number of clone variants and detection techniques.

Different domains like automotive domain, web based applications and other complex systems tend to use models in the development [45]. Model driven software development has been accepted as an industry best practice. Models developed using Unified Modeling Language (UML) improve the quality of product delivery and bring in advantages of automatic code generation, early verification and validation, etc. [143]. As UML models provide an abstract view of the system thus detecting clones in models is equally important because similar challenges as found in source code exist in the case of UML models [216], too. In this thesis, we have proposed a clone detection technique for UML class models which is discussed in chapter 4.

It is true that similarities exist at higher level of abstraction in software which will be referred as concept clones. Concept clones are defined as similar program structures within one software system or across versions reflecting similar domain concepts. Since

the graphical UML [236] is increasingly replacing conventional programming languages for developing and modeling software systems, the presence of design level similarities in form of concept clones cannot be precluded in UML models [105]. Marcus and Maletic [172] identified a scenario in which the software developer writes fresh solution to a problem knowingly or unknowingly for which the solution already exists. This scenario leads to different solutions for the same problem. Identification of high level concept clones helps the software developer in understanding and comprehending the system at abstract level [18]. The presence of structural similarities during the evolution of software system helps in improving the understanding of the system. Chapter 5 of the thesis presents a technique to detect these concept clones.

The detection of software clones will help to raise the awareness of clones existing within systems and aim to prevent the creation of new clones. Other applications of software clone detection include finding cross cutting code [30], code evolution [8], software refactoring [14,76,129,152], plagiarism and copyright infringement detection [28], software product lines [204], clones in web sites [10,147], system quality evaluation [175], locating bugs and reducing software defects [94] and to gain insight in program design and understanding.

1.2 Problem Statement

Code duplication, or cloning, is the use of existing software artifacts in the construction of new code. It is essentially a form of reuse. The cloning leads to spread of an error potentially contained within the duplicated code. Thus, it is important to find all related fragments throughout the source code. Efficient software clone detection depends upon finding fragments of source code with high similarity. In this section, few of the challenges are discussed to formulate the research problem.

1.2.1 General Challenges

A software clone detector needs to compare every fragment with every other fragment which increases the complexity of match detection. Thus, it is an apparent challenge for clone detection. Extensive research has been carried out in the field of source code clone detection. In text based techniques, the source program is considered as sequence of strings or lines and comparison is carried out based upon that only. These techniques yield type-1 clones i.e. exact matches. Token based clone detection techniques yield type-

2 clones i.e. renamed clones. These techniques tokenize the source code and the sequence of tokens is read to trace duplicate subsequences. Similarly type-3 clones are detected by converting the source code to abstract syntax tree (AST). Tree matching or tree similarity techniques are applied to the AST representation of source code. All these techniques of clone detection rely on keyword matching only.

1.2.2 Semantic Clone Detection– A Challenge

Two program fragments with different syntax may be semantically same. These types of clones are known as semantic clones or type-4 clones. It is challenging to detect semantic clones. In 1990, a key article [81] by Horwitz was published to detect textual and semantic similarities.

Program Dependence Graph (PDG) is directed attributed graph representing the statement, control flow and captures semantic information from source code. It is an abstraction of source program. Krinke [137], Komondoor and Horwitz [130] and Gabel et al. [60] presented techniques to detect semantic clones using PDG as source representation. Since PDG is in the form of graph, so graph mining techniques like sub graph isomorphism are applied and detection of isomorphic subgraphs is NP-complete. Choi et al. [36] used a set of API calls to detect similar programs but this technique is vulnerable to deobfuscation attacks, so extending birthmarks with more information and making technique robust against attacks is an upcoming challenge. In Philip Schugerl [205]'s technique, AST is normalized to description logic but the technique currently does not detect small clones and clones across methods. Jiang and Su [97] proposed a technique to detect functionally equivalent code fragments of arbitrary size depending on the input–output behaviour of a piece of code. A general method for different programming languages should be developed to detect functionally equivalent but syntactically different code fragments. Exploring future research on functionally equivalent code refactoring and reuse is another upcoming hot area.

Mostly, various representations adopted for clone detection are closely tied to the source code. These representations encode different smaller changes like reordering, intertwining of clones which usually take place during programming. Semantic clones are particularly hard to detect without a great deal of background knowledge about program construction and software design. As we move from Type-1 Clones to Type- 4 Clones, the

sophistication and complexity underlying corresponding detection increases. PDG based approaches of detecting semantic clones suffer from slow generation of clone pairs and imprecise definition of semantic clones. Developing the framework to help in fast generation of PDG from source code is a definite challenge. Moreover, PDG does not consider statement ordering, thus non-contiguous clones may turn out to be false positives during manual verification. The technique should be scalable seeing the bulk of code in latest software systems which is computationally challenging.

1.2.3 Detection of Clones in Object Oriented Systems

Number of researchers recognized the need to detect semantic equivalence in source code. Due to the assorted nature of the problem, difficulty and cost associated forced them to go for alternate methods which are rather more close to implementation. Moreover, there is a great need to understand the commonalities in domain concepts which most of the time correspond directly to implementations.

Due to the use of model driven development as the standard industry practice, we are motivated to detect clones in object oriented systems by using object oriented i.e. UML models. There is higher number of concept or domain clones than other types of clones [172]. Concepts are usually represented in model driven engineering using general purpose modeling languages like UML. Detection of concept clones can help in comprehending the design of the system for better maintenance [225]. During evolution, high level similarities emerge. The presence of structural similarities as the software system evolves helps in improving the understanding of the system. Detection of high level similarities throughout the history of the subject system helps in improving the different attributes of quality of the software like maintainability, reusability, extensibility [110].

Clone detection in models [45] [187] [158] is touched vastly by related fields like algorithms, symbolic logic, data mining, graph mining, artificial intelligence, programming languages, machine learning, etc. Moreover large systems introduce intricacies in the form of unexpected overlaps of parts and unexpected behavior of previously well behaved systems [236] as many designs can be easily shared between design teams [118]. Tim Weilkiens [236] confirms that the design methodology of any system or process has shared parts as crucial component. As the UML model of a system is generated before implementation, most of the reasons leading to clones in code based

development are also applicable to model based development. Duplications in models have negative effects on maintainability and reusability [216] and thus the study states the importance of detecting clones in models to improve quality. The detection of model clones is an active area of research [215]. There exists only one approach [216] to detect clones in UML models. No study has been carried out to explicitly examine the effects of cloning in model based development.

1.3 Research Objectives

It is common consensus and utmost important to find software clones. We understand that the presence of software clones hamper maintenance and may lead to bug propagation in source code. The detection of clones will help to raise the awareness of clones existing within the system and aim to prevent the creation of new clones. With the popularity of model driven development, duplicate parts in models i.e. model clones pose similar challenges as in source code. Thus the complete problem is divided into following major objectives:

1. To explore various software clone detection techniques
2. To identify various parameters affecting software semantic clone detection
3. To design and develop an efficient software clone detection technique for object-oriented systems
4. To verify and validate the clone detection technique

1.4 Contributions of the Thesis

We have conducted an extensive systematic literature review of software clones in general and software clone detection in particular. We used the standard systematic literature review method based on a comprehensive set of 213 articles from a total of 2039 articles published in 11 leading journals and 37 premier conferences and workshops. Existing literature about software clones is classified broadly into different categories. The importance of semantic clone detection and model based clone detection led to different classifications. Empirical evaluation of clone detection tools/ techniques is presented. Clone management, its benefits and cross cutting nature is reported. Number of studies pertaining to nine different types of clones is reported. 13 intermediate representations and 24 match detection techniques are reported. We call for an increased

awareness of the potential benefits of software clone management, and identify the need to develop semantic and model clone detection techniques. Recommendations are given for future research.

Now-a-days, model driven development has become a standard industry practice. Duplicate parts in models i.e. model clones create similar problems as in source code. We have developed a technique to detect clones in UML class models. The core of our technique is the construction of a labeled, ranked tree corresponding to the UML class model where attributes with their data types and methods with their signatures are represented as subtrees. By grouping and clustering of repeating subtrees, the tool is able to detect duplications in a UML class model at different levels of granularity i.e. complete class diagram, attributes with their data types and methods with their signatures across the model and cluster of such attributes/methods. We propose a new classification of model clones with the objective of detecting exact and meaningful clones. Empirical evaluation of the tool using open source reverse engineered and forward designed models show some interesting and relevant clones that provide useful insights into software modeling practice.

We have developed another technique to compute similarity across object oriented programs at different levels of granularity. The tool is able to detect concept level similarities by applying latent semantic indexing and principal component analysis. Previous research has shown that detection of high level similarities can help in comprehending the design of the system for better maintenance. We have extended our tool to detect similarities for UML diagrams by measuring the distance between two class models. In addition, we mined important change patterns at method level using multi-version program analysis by applying the proposed technique throughout the evolutionary history of the software. We have validated the similarity score by applying the tool at function level granularity in the source code. We assess the usefulness and scalability of the proposed technique by empirical evaluation on source code of open source subject systems and multi-version program analysis on 8 releases of *dnsjava*.

1.5 Organization of the Thesis

The remaining chapters of this thesis are organized as follows:

- The second chapter provides systematic literature review of existing work related to software clone detection. This chapter starts with background information which includes types of clones, reasons for occurrence of clones, etc. We have done rigorous study of the existing techniques of software clone detection with a complete list of available clone detection tools. This chapter also identifies the comparison and evaluation studies. Key sub areas related to software clone detection with particular emphasis on clone management is also mentioned in this chapter.

- Chapter 3 summarizes all the semantic clone detection techniques and model based clone detection techniques. Various intermediate source representations and match detection techniques are listed in comparing these techniques. We have also highlighted the process of identifying the parameters of software clone detection. The chapter concludes with general motivation for model clone detection.

- In chapter 4, we devised the clone detection technique for UML models. Various reverse engineered and forward designed models are used for empirical evaluation to verify and validate the technique.

- Chapter 5 presents another approach in which we detected high level similarities in the form of concept clone detection in source code and concept model clone detection in UML models. The results of clone detection at different levels of granularity are also mentioned.

- Finally, chapter 6 concludes the thesis along with directions for future research.

Chapter 2

Systematic Literature Review

The volume of research in the field of software clones is continually increased. This has led to a need for critical evaluation and integration of the available research in particular the need for a systematic literature review. Kitchenham et al. [125,126] and Brereton et al. [27] define a systematic literature review as a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest. This chapter presents a systematic literature review which was carried out to analyze and report the findings in clone-based research. Systematic literature reviews are time consuming but provide transparent and comprehensive view of ongoing research, and can be used to identify a number of research avenues. This review identifies different key areas of research on software clones, discusses the concepts, research method used and major findings. The systematic review reported in this paper was done following the guidelines of Kitchenham et al. [125, 27]. The steps included in the review include: development of a review protocol, conducting the review, analyzing the results, reporting the results and discussion of findings.

This chapter starts with background information which includes types of clones, reasons for occurrence of clones, etc. The way the systematic literature review is planned; sources of information and research questions framed are discussed next. We have done rigorous study of the existing techniques of software clone detection with a complete list of available clone detection tools as given in section 2.6. Section 2.7 identifies the comparison and evaluation studies. Section 2.8 and section 2.9 discusses the key sub areas related to software clone detection and clone management respectively. Various subject systems used in clone detection research are mentioned in section 2.10. Achievements of our review with the key findings are mentioned in section 2.11. The chapter concludes with a summary in section 2.12.

2.1 Background

Firstly, we define the different types of software clones, and the factors leading to software clones. We then summarize the drawbacks of software cloning and state some

points as to why software cloning is beneficial sometimes.

2.1.1 Software Clones

The Merriam-Webster dictionary defines a clone as one that appears to be a copy of an original form, thus being synonymous to a duplicate. In source code and other software artifacts, the original (code) fragment is copied and pasted with or without modifications. The pasted (code) fragment is said to be a clone and this activity is known as (code) cloning. However in software engineering field, the term code clones is still searching for a suitable definition. Its vagueness was properly reflected in Ira Baxter‟s words:

> "*Clones are segments of code that are similar according to some definition of similarity."*

Roy and Cordy [191] mentions code duplication or cloning as a form of software reuse. Baker [12] after experimenting with a sample program concluded that code can shrink by 14% based on exact matches, and 61% based on parameterized matches. The study suggests that as much as 20-30% of large software systems consist of cloned code.

2.1.2 Types of Clones

It is quite pertinent to mention that standards in case of nomenclature are still missing thereby leading to different taxonomies by different researchers. We list here basic types of clones [21,135,191].

Type-1 (Exact Clones): Program fragments which are identical except for variations in white space and comments.

Type-2 (Renamed Clones): Program fragments which are structurally/syntactically similar except for changes in identifiers, literals, types, layout and comments.

Type-3 (Near Miss Clones): Program fragments that have been copied with further modifications like statement insertions/ deletions in addition to changes in identifiers, literals, types, layouts, etc.

Type-4 (Semantic Clones): Program fragments which are functionally similar without being textually similar.

Function Clones: The clones which are limited to the granularity of a function/ method

or procedure. Several studies devised the clone detection methods that found the clones at function level which can be extracted in a different procedure.

Model Based Clones: Nowadays graphical languages are replacing the code as core artifacts for system development. Unexpected overlaps and duplications in models [45] are termed as model based clones.

Concept Clones: The concept clones are defined as similar program structures within one software system or across versions reflecting similar domain concepts.

Concept Model Clones: The presence of design level similarities in form of concept clones cannot be precluded in UML models. C**oncept model clones** may be defined as two different class models representing the same conceptual/ domain model or design model. The class diagrams may be created by different people pertaining to same domain.

2.1.3 Why Clones

Although cut-copy-paste-adapt techniques are considered bad practices from a maintenance point of view, many programmers use them. We list some of the reasons of software cloning.

- **Programmers Limitation and Time Constraints:** The software is seldom written under ideal conditions. Limitations of programmer's skills and hard time constraints inhibit proper software evolution [120]. Only way out is copying/pasting/editing.
- **Complexity of the System:** The difficulty in understanding large systems only promotes copying the existing functionality and logic.
- **Language Limitations:** Kim et al. [120] conducted an ethnographic study on why programmers copy and paste code. They concluded that sometimes programmers are forced to copy and paste code due to limitations in programming languages. Many languages lack inherent support for code reuse, leading to duplication [228].
- **Phobia of Fresh Code:** Programmers often fear to bring in new ideas in existing software. They fear that introduction of new code may result in a lengthy software development life cycle. Furthermore, it is easier to reuse the

existing code than to develop a fresh solution since new code may introduce new errors [94,154].

- **Lack of Restructuring:** Programmers delay restructuring, refactoring, abstraction, etc. of code due to time limits. Often, restructuring gets delayed until after product delivery which increases subsequent maintenance costs [140].

- **Forking/ Templating:** Forking is the reuse similar solutions, with the hope that evolution of the code will occur independently at least in short term [115]. Use of structural and functional templates is often mentioned as reuse mechanisms.

2.1.4 Advantages of Clones

Sometimes software developers intentionally introduce code clones into existing software. The study by Kapser and Godfrey [113,115] discusses this issue. Some of the points are mentioned below:

- It is a fast and immediate method of addressing change requirements.
- Some programming paradigms encourage the use of Templates in programming.
- If a programming language lacks reuse and abstraction mechanism, it is the only way left to quickly enhance the existing functionality.
- The overhead of procedure calls sometimes promotes code duplication for efficiency considerations.

2.1.5 Disadvantages of Clones

- **Higher maintenance costs:** Two studies [174,175] confirm that presence of code clones in software greatly increase the post implementation maintenance (preventive and adaptive) effort [2].

- **Bug propagation:** If a code fragment contains a bug and that fragment is pasted at different places, the same bug will be present in all the code fragments. So code cloning increases the probability of bug propagation [99,161].

- **Bad impact on design:** Code cloning discourages the use of refactoring, inheritance, etc [58,148]. It leads to bad design practice.

- **Impact on system understanding/improvement/modification:** It is quite common that the person who developed the original system is not the one who is maintaining it. Moreover the presence of duplicated code not only complicates the design but leads to decreased understanding thereby hampering improvements and modifications. In the long run, the software may become so complex that even minor changes are hard to make [135,174].

- **Strain on resources:** Code cloning increases the size of the software system thereby putting a strain on system resources [99,135]. It degrades the overall performance in terms of compilation/execution time and space requirements.

2.2 Motivations to carry out Systematic Literature Review

- Software clones are present in many different software artifacts. Therefore, our study on detection techniques goes beyond source code.

- Upon assessing state of art in software clone research, we realized the lack of systematic literature review. Thus we summarized the existing research based on extensive and systematic database search and report the research gaps for further investigation.

- Systematic literature review provides critical evaluation and integration of the available research i.e. a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest.

2.3 Planning the Review

The review protocol includes the research questions framework, the databases searched, methods used to identify and assess the evidence. Conducting the review comprises identification of primary studies, applying inclusion and exclusion criteria and synthesizing the results. Electronic databases were extensively searched and its studies are reported. Moreover, some of the leading software engineering journals and conference proceedings which fail to come in electronic search were searched manually. In total 2039

articles appeared in electronic and manual search. Study selection procedure is shown in Figure 2.1.

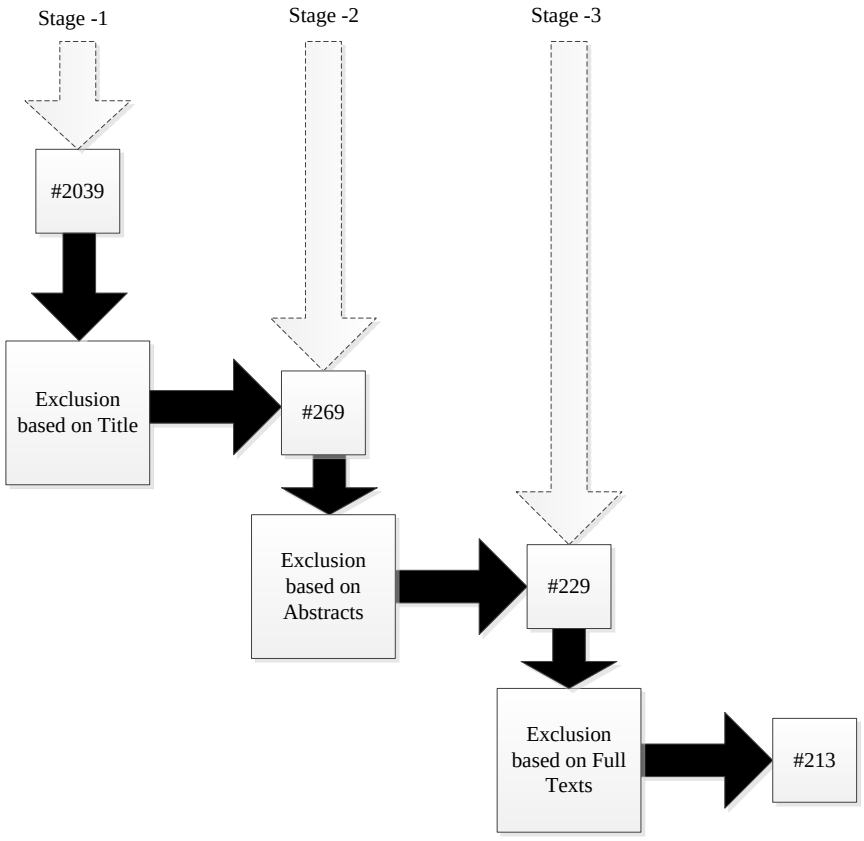


Figure 2.1 Study selection procedure

2.4 Research questions

The main goal of this systematic review was to identify and classify the existing literature focusing on clone detection, clone management, semantic clone detections and model based clone detection techniques. To plan the review, a set of research questions were needed. Table 2.1 lists the specific research questions and sub questions.

Table 2.1 Research question, sub question and motivation

Research Question	**Motivation**
1) What is the current status of clone detection? 1.1) What methods of clone detection (intermediate representation and match detection technique) are used and what granularity of clone do they use? 1.2) What tools are available for software clone detection, what method do they use, what clone type do they address, how frequently are they cited? 1.3) Which studies have evaluated methods/ tools and with what results?	It helps in understanding the clone detection techniques. Various intermediate representations and match detection techniques used in clone detection technique/ tool are reported. Various tools/ techniques for clone detection developed till date are mentioned with their chare of usage. The research question explores the studies which evaluated/ compared different clone detection technique. We mentioned studies which empirically compared different clone detection tools. The number of studies for each type of clone is also reported.
2) Research status in semantic and model based clone detection 2.1) What are different studies in semantic clone detection and their comparative analysis? 2.2) What are the different studies in model based clone detection and comparative analysis?	It is hard to detect semantic clones. Functional equivalence problem is undecidable in general and subgraph isomorphism is NP-complete. Model based clones are quite hard to locate too. So any research indication for the same is immensely helpful. It will help in devising better and highly scalable strategies.
3) Key Sub Areas 3.1) What are the important areas related to software clones, number of studies in each classified area and their findings? 3.2) A time based count to show how the area has evolved over time.	It helps in knowing the type of study carried out in the article. It is important to know the number of studies for each sub area which helps in identifying key areas for further research. A time based count shows how the key area has evolved over time.
4) What is the current status of clone management 4.1) What are the studies discussing benefits of	Clone management has turned out to be a cross cutting topic. Recent research is shifting towards efficient clone management techniques. The research questions focuses on

clone management? 4.2) What is the current status of research in clone management and visualization? 4.3) A systematic map showing cross cutting nature of clone management.	understanding the current status of research in clone management and its sub-topics like clone visualization. Different key area identified in research question 3 touch clone management. It is important to know different clone detection methods and clone detection tools overlapping with clone management topics.
5) What is the subject system used 5.1) What is the size of software used in LOC? 5.2) What is the programming language of the subject system and whether the system is open source or commercial?	It will help in building the database on which the clone detection research can be carried out. It is a step towards benchmarking and standardization of comparative analysis studies.

2.5 Sources of Information

A broad perspective is necessary for an extensive and broad coverage of the literature. Before starting the search, an appropriate set of databases must be chosen to increase the probability of finding highly relevant articles. [27,125] recommends searching widely in electronic sources and following databases were searched. In almost all the searches, the keyword "clone" is included in abstract. It is an extensive and time consuming process. Table 2.2 shows the defined search strategy from different e-resources. We tried to extract as much of relevant literature as possible.

2.6 Current Status of Clone Detection

There are a large number of clone detection tools/ techniques. For any technique/ tool, the source representation and the match detection technique are most the important characteristics.

Table 2.2 Sources of information and search strings

Sr. No.	E – Resource	Search String	Dates	Product /Content Type	Subjects	#
1.	ieeexplore.ieee.org	Abstract : Clone	1988 – 2011	Conferences, Journals and Standards	Computing and Processing (Hardware and Software), general topics for Engineers (Math, Science and Engineering), Engineering Profession	967
2.	www.acm.org	Abstract: Clone	All Dates	Journal, Proceedings, Transaction and Magazine	All Subjects	485
3.	www.sciencedirect.com	Abstract: Clone	All Years	All Sources	Computer Science, Decision Sciences and Engineering	134
4.	www.springerlink.com	Title: Clone All Text: (Software/ Code)	Entire Range of Publication Dates	All Sources	All Subjects	245
5.	www3.interscience.wiley.com	Article Title: Clone Full Text/ Abstract: code or software	All Dates	Journals, Reference works, Databases	All Subjects	208

2.6.1 Intermediate source representations and match detection techniques

Initially, source code is pre-processed to remove any uninteresting parts (e.g. comments and blank lines). Then suitable transformation techniques are applied to the pre-processed code to obtain an intermediate representation of the code. Intermediate representation is a way of extracting useful information based upon which comparison is done. Clone

granularity defines the boundary of comparison. It can be fixed e.g. function, class etc. or free e.g. number of statements. We listed granularity level for each intermediate representation. Different clone granularity levels apply to different intermediate source representations. We list all of them in Table 2.3. AST or Parse trees, Source code or text and Regularized tokens are the most frequently used intermediate transformation.

Table 2.3 Intermediate representations, relative count, granularity and citations

Sr. No.	Intermediate Representation/ Transformation Technique	Code	#	Clone Granularity Level	Citations
1.	Regularized Tokens	S1	16	Set of statements, set of tokens, fragments of sequence diagrams, files, functions	[12],[17],[18], [29],[34],[64], [84],[98],[100], [101],[108],[154], [155],[160],[203], [240]
2.	AST / Parse Tree	S2	28	Number of tokens, lines of source code, set of instructions, code regions, methods, functions, threshold set by user	[7],[8],[13],[20], [23],[26],[29], [31],[35],[40], [54],[55],[63], [89],[132],[133], [134],[148],[151], [155],[174],[180], [200],[205],[211], [220],[233],[241]
3.	Partite Sets and Vertices	S3	1	Program	[36]
4.	Source Code / Text	S4	16	Functions, methods, threshold as set by user, number of words/lines	[16],[41],[42], [52],[97],[99], [127],[147],[167], [172],[183],[186], [194],[210],[216], [235]
5.	Call Graph	S5	1	Functions	[34]
6.	Vector Space Representation	S6	3	Methods, blocks	[70],[156],[217]
7.	One Dimensional Array	S7	1	Fragments of sequence diagrams	[158]
8.	PDG	S8	5	A set of statements that can be extracted into a function, threshold as set by user, non-contiguous clones	[60],[80],[81],[130], [137]

Sr. No.	Intermediate Representation/ Transformation Technique	Code	#	Clone Granularity Level	Citations
9.	Index and Inverted index	S9	1	Set of statements	[150]
10.	Sparse, Labelled Directed Graph	S10	3	Fragments of graph	[45],[85],[187]
11.	Recorded Parsing Actions and Lexical Information	S11	1	Set of statements	[170]
12.	Abstract Memory States	S12	1	Procedures	[123]
13.	Description Logic	S13	1	Functions, methods, control blocks	[205]

Match detection algorithms are the prominent issue in clone detection process. After a suitable source code representation and granularity level is decided, an appropriate match detection technique is applied to the units of the source code representation. The output is a list of matches with respect to the transformed code. All the match detection algorithms found in our primary studies are shown in Table 2.4. We have listed primary studies that discuss each match detection algorithm. The most frequently occurring match detection techniques are Metric/Feature Vector clustering, Suffix tree based token by token comparison, Substring/Subtree/Model comparison and Dynamic programming. In post-processing, detected clones are screened for false positives manually. Since many software systems contain large duplication, the outcome of the clone detection results is reported using techniques like scatter plots and other forms of visualization.

Table 2.4 Match detection techniques

Sr. No.	Match Detection Algorithm	Code	#	Clone Granularity Level	Citations
1.	Suffix Tree based token by token comparison	M1	12	Fragments of sequence diagrams, functions, program fragments, number of tokens as set by user, number of words	[12],[55],[64],[100], [101],[108],[134], [155],[158],[160], [170],[220]

Sr. No.	Match Detection Algorithm	Code	#	Clone Granularity Level	Citations
2.	Weighted Partite Matching	M2	2	Number of blocks, program	[36],[45]
3.	LSI based clustering algorithms	M3	2	Code segments, functions, files	[167],[172]
4.	Metrics/ Feature vectors clustering	M4	17	Set of instructions, code regions, functions and methods, Threshold set by user, blocks	[7],[8],[13],[34],[42] [89],[127],[132], [133],[147],[148], [156],[174],[180], [183],[186],[200]
5.	Fingerprinting	M5	3	Number of lines, set of statements, blocks	[35],[99],[186]
6.	Anti-unification	M6	3	Threshold of tokens set by user, sub expressions	[29],[31],[151]
7.	Hashing/ LSH	M7	6	Set of instructions, code regions, files	[23],[60],[84],[180], [200],[203]
8.	FIM	M8	3	Higher level similarities i.e. ADT, classes	[18],[155],[233]
9.	Program Slicing	M9	2	procedure, non - contiguous clones	[80],[130]
10.	DP	M10	8	Methods, set of statements, programs	[13],[23],[41],[63], [101],[132],[148], [241]
11.	Suffix Arrays	M11	4	Function, sequence of tokens	[17],[34],[98],[240]
12.	LCS	M12	6	Program, variable size	[63],[84],[123], [194], [217],[241]
13.	ICA	M13	1	Methods, blocks	[70]
14.	Associative Array	M14	1	Number of tokens, lines of source code	[54]
15.	Nearest Neighbor	M15	1	Set of statements	[150]
16.	Canonical Labeling	M16	2	Fragments of graph	[85],[187]
17.	Substring/ Subtree/ Model Comparison	M17	9	Set of tokens, number of lines of code	[16],[20],[26],[52], [60],[81],[210],[211], [216]
18.	k-length patch matching	M18	1	Threshold as set by user	[137]

Sr. No.	Match Detection Algorithm	Code	#	Clone Granularity Level	Citations
19.	Partitioning Algorithm	M19	1	Different components of the program	[81]
20.	Tree Kernel	M20	1	Class, method, set of statements	[40]
21.	Levenshtein Distance	M21	1	Class, method, set of statements	[89]
22.	Semantic Web Reasoner	M22	1	Functions, methods, control blocks	[205]
23.	Random Testing	M23	1	Lines of source code	[97]
24.	Dot Plot/ Scatter Plot	M24	2	Lines of source code	[52],[235]

2.6.2 Clone Detection Tools

Numerous tools have been developed for clone detection. We have conducted an extensive survey regarding the penetration and usage of clone detection tools for instructors, research or commercial purpose. Table 2.5 shows tools name/ first author name, its source representation and match detection method, which papers have cited it. The table includes clone detection based not only in source code but also on the usage of the tool in other areas like web applications, requirement specifications, etc. As shown in Table 2.5 we have classified the techniques roughly into seven types: text based, token based, tree based, graph based, metrics based, model and hybrid clone detection techniques. Apart from model-based techniques, these techniques and the tools that use them are discussed in more detail later in this section. Model-based techniques are discussed in more detail in chapter 3.

Different types of clones are detected by different techniques. Many tools/ techniques are able to detect only a subset of clone types. In Table 2.6, we have enumerated the type of clones and identify the studies discussing each type of clone. Table 2.6 makes it clear that clone research has concentrated primarily on clones of type-2 and type-3, clones of type-1, type-4 and model based clones have been given some attention, but the remaining 4 clone types have been the subject of very little research.

Table 2.5 Tool type, tool/ first author name, source representation/ match detection technique, number of studies referring it and citations

	Tool/ 1st Author	Method	#	Citations
TEXT BASED	*Duploc*	Source Code, Substring Comparison/ Dot Plot Scatter Plot	5	[21],[52],[53],[234], [243]
	Simian	Source Code, Substring Comparison	12	[16],[22],[51],[138], [139],[140],[141],[142], [205],[206],[207],[210]
	DuDe	Source Code, Dot Plot/ Scatter Plot	2	[79],[235]
	SDD	Index and Inverted Index, Nearest Neighbor	1	[150]
	CSeR	AST, Metrics/ Levenshtein Distance	1	[89]
	NICAD	Source Code, LCS	6	[127],[173],[193],[194], [196],[197]
	EqMiner	Source Code, Random Testing	1	[97]
	Johnson	Source Code, Fingerprinting	1	[99]
	Cordy	Source Code, DP	1	[41]
	Marcus	Source Code, LSI	1	[172]
	Barbour	Source Code, Substring Comparison	1	[16]
TOKEN BASE	*Dup*	Tokens, Suffix Tree	7	[12],[21],[53],[56],[64], [234],[243]
	CCFinder (X)	Tokens, Suffix Tree	50	[5],[16],[21],[22],[26], [30],[32],[53],[59],[62], [72],[75],[78],[79],[88], [89],[93],[96],[107], [108],[109],[111],[112], [113],[114],[115],[121], [122],[123],[136],[161], [163],[175],[176],[178], [190],[201],[203],[204], [205],[206],[207],[223], [227],[230],[234],[240], [242],[245],[246]
	D-CCFinder	Tokens, Suffix Tree	2	[159],[160]
	RTF	Tokens, Suffix Array	1	[17]

D	Tool/ 1st Author	Method	#	Citations
	clones /cscope	AST, Suffix Tree	1	[134]
	iClones	Tokens, Suffix Tree	2	[64],[66]
	CP-Miner	Tokens, FIM	2	[95],[154]
	SHINOBI	Tokens, Suffix Array	1	[240]
	FCFinder	Tokens, LSH	1	[203]
	Jian-lin	Tokens, Suffix Array	1	[98]
	Chilowicz	Tokens/Call Graph, Metrics/ Suffix Array	1	[34]
	Deckard	PDG/ Parse Tree, LSH/ Subtree Comparison	9	[51],[60],[61],[95], [104],[123],[153],[189], [224]
TREE BASED	*CloneDR*	AST, LSH/DP	8	[21],[23],[32],[56],[72], [209],[222],[234]
	SimScan	AST, Subtree Comparison	11	[11],[16],[22],[50],[51], [56],[83],[161],[178], [211],[227]
	Asta	AST, Associative Array	1	[54]
	Clone Digger	AST, Anti-unification	2	[31],[40]
	Sim	Parse Tree, DP/LCS	1	[63]
	ClemanX	AST, Metrics/ Feature Vector Clustering/ LSH	1	[180]
	JCCD API	AST, Subtree Comparison	1	[26]
	Ccdiml	AST, Subtree Comparison	3	[20],[30],[227]
	CloneDetection	AST, FIM	1	[233]
	Cpdetector	AST, Suffix Tree	1	[134]
	Clast	Parse Tree, Suffix Tree	1	[55]
	Chilowicz	AST, Fingerprinting	1	[35]
	Saebjornsen	AST, Metrics/ Feature Vector Clustering/ LSH	1	[200]
	Tairas	AST, Suffix Tree	1	[220]

	Tool/ 1st Author	Method	#	Citations
	Lee	AST, Anti-unification	1	[151]
	Yang	Parse Tree, DP/LCS	1	[241]
	Brown	Tokens/AST, Anti-unification	1	[29]
	PDG-DUP	PDG, Program Slicing	2	[30],[130]
GRAPH BASED	*Scorpio*	PDG, Program Slicing	1	[80]
	Duplix	PDG, k- length patch matching	3	[21],[137],[234]
	Choi	Partite Sets and Vertices, Weighted Partite Matching	1	[36]
	Horwitz	PDG, Substring Comparison/ Partitioning Algorithm	1	[81]
	CLAN/ Covet	AST, Metrics	10	[8],[21],[32],[33],[56], [144],[174],[185],[234], [243]
METRICS BASED	Li	Vector Space Representation, Metrics/ Feature Vector Clustering	1	[156]
	Kontogiannis	AST, Metrics/ Feature Vector Clustering/ DP	2	[132],[133]
	Similar Methods Classifier	AST, Metrics/ DP	1	[13]
	Antoniol	AST, Metrics	2	[7],[8]
	Dagenais	Source Code, Metrics	1	[42]
	Kodhai	Source Code, Metrics	2	[127],[128]
	Patenaude	Source Code, Metrics	1	[183]
	Perumal	Source Code, Metrics/ Fingerprinting	1	[186]
	Lavoie	AST, Metrics/ DP	1	[148]
	Lanubile	Source Code, Metrics/ Feature Vector Clustering	1	[147]
	CloneDetective / ConQAT	Tokens, Suffix Tree/ DP	9	[46],[47],[49],[90],[100], [101],[102],[103], [104]

	Tool/ 1st Author	Method	#	Citations
MODEL BASED	*ModelCD*	Sparse labeled direct graph, canonical labeling	2	[47],[187]
	DuplicationDetector	One Dimensional Array, Suffix Tree	1	[158]
	MQ_{lone}	Model/ Source Code, Model Comparison	1	[216]
	Clone Detective	Sparse labeled direct graph, Weighted Partite Matching	1	[45]
	Hummel	Sparse labeled direct graph, canonical labeling	1	[85]
	Clone Miner	Tokens, FIM	3	[18],[19],[247]
HYBRID	*MeCC*	Abstract Memory States, LCS	1	[123]
	Maeda	Recorded Parsing Actions and Lexical Information, Suffix Tree	1	[170]
	Lucia	Source Code, LSI	1	[167]
	Li	Tokens/ AST, Suffix Tree	1	[155]
	Hummel	Tokens, LSH/LCS	1	[84]
	Sutton	Vector Space Representation, LCS	1	[217]
	Cordy	Vector Space Representation, ICA	1	[70]
	DL_Clone	AST/ Description Logic, Semantic Web Reasoner	1	[205]
	Corazza	AST, Tree Kernel	1	[40]

Table 2.6 Number of studies referring to different types of clones

Sr. No.	Type of Clone	#	Citations
1.	Type-1/ Exact Clones	11	[52],[53],[55],[84],[99],[127],[134],[155],[160],[186], [233]
2.	Type-2/ Renamed clones	23	[12],[17],[52],[53],[55],[64],[79],[84],[98],[108],

			[127],[134],[149],[155],[160],[170],[174],[186], [220],[231],[233],[240],[244]
3.	Type-3/Near Miss Clones	27	[23],[29],[35],[40],[41],[53],[55],[63],[70],[79],[80], [96],[101],[131],[133],[134],[137],[149],[151],[170], [174],[183],[194],[196],[206],[217],[220]
4.	Type-4/ Semantic Clones	8	[36],[60],[97],[123],[130],[137],[172],[205]
5.	Structural Clones	2	[18],[19]
6.	Model based clones	7	[45],[47],[85],[100],[158],[187],[216]
7.	Function Clones	3	[34],[54],[196]
8.	File Clones	1	[203]
9.	Contextual Clones	1	[173]

2.6.3 Text based clone detection techniques

In text based clone detection techniques, two code fragments are compared with each other in the form of text/ strings/ lexemes and similar fragments are reported as code clones. Johnson [99] applied a fingerprinting technique for comparison of source code. Ducasse et al. [52] developed a language independent clone detection tool *duploc* which required no parsing. Line based comparison is done using dynamic pattern matching and results are displayed in the form of dot plot.

DuDe [235] is another line based clone detection tool which is able to detect duplication chains consisting of a number of smaller size exact clones. *Simian* [210] is able to 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. The *SDD* (Similar Data Detection) [150] tool is helpful in detecting code clones in large size systems. The technique is based on generating index and inverted index for code fragments and their positions. Then an n-neighbor distance algorithm is used to find similar fragments.

NICAD [194] is a text based hybrid clone detection tool which is able to detect type-3 clones effectively. It is based on a two stage process, viz. identification and normalization of potential clones using pretty printing and code normalization and code comparison using longest common subsequences. Variants of *NICAD* [193] have been used to calculate recall and precision by applying a mutation/ injection based framework. *NICAD* has been used to detect function clones in open source systems written in Python to discover any changes in cloning patterns as compared to traditional software systems [197]. Martin and Cordy [173] use *NICAD* to detect and analyze similar web services in the form of contextual clones. Cordy et al. [41] detected near-miss clones in HTML web pages using island grammar to identify and extract all structural fragments and applying UNIX diff as comparator. Barbour et al. [16] used the Knuth-Morris-Pratt algorithm for string comparison to update the clone information from the server incrementally to save time in a client server setup. Later on, only relevant clones are retrieved by individual developers. The technique is found to be faster than string based *Simian* and AST based *SimScan*. However, since it uses string based technique, it fails to detect clones with minor and major changes.

2.6.4 Token based clone detection techniques

It is more meaningful in parameterized clone detection as tokens are better than simple keyword matches. In token based clone detection techniques, firstly, tokens are extracted from the source code by lexical analysis. Then some set of tokens at a specific granularity level is formed into a sequence. Suffix tree or suffix array based token by token comparison is the heart of token based clone detection algorithms. This match detection technique is the most frequently cited in the literature and was used in one of the first clone detectors *dup* [12]. Token sequences are fed into a suffix tree. The approach used "functor" as an abstraction of concrete values of identifiers and literals that maintains their order. The study reported a greater number of parameterized matches than exact matches in the same subject system. *CCFinder* [108], a token based clone detection tool uses suffix tree matching algorithm to find identical subsequences. It is a popular tool among researchers and has been widely used for code clone analysis, code clone management, etc. Many researchers have worked to enhance the output of *CCFinder* by devising clone visualization tools. In their work, they increased the threshold in *CCFinder* to filter small clones. Livieri et al. [160] developed a distributed version of *CCFinder*,

viz. *D-CCFinder* for large systems by using 80 workstations in master slave configuration. *CCFinderX* [109] was used to study code clone genealogies at release level. Two studies [136,201] used it for analysis of the relations between open source software quality and code cloning. Monden et al. [176] concentrated on detecting type-2 clones using *CCFinderX*. The number of tokens of largest clone pair and the percentage of duplication within the most suspicious source file pair are important metrics in distinguishing type-2 clones in their study.

CP-Miner [154] is a token-based tool using frequent itemset mining to detect bugs in the software induced by cloning. *Clone Miner* [18] detects structural clones which are high level abstractions. *RTF* [17] works by applying suffix array on tokens. Suffix array on source code tokens is also used by Jian-lin and Fei-peng [98].

Koschke et al. [134]'s tool *clones* uses a parser and generates abstract syntax tree. Then the AST is serialized and input to suffix tree. The technique is able to detect syntactic units which are not possible by applying suffix tree only. Göde and Koschke [64] developed the incremental tool *iclones* by extending *clones* to detect clones for multiple versions. Li and Thompson [155] used two techniques, viz. combination of token stream and the AST to detect and remove code clones. The two representations are used for their accuracy and speed. Match detection is done with the help of a suffix tree. Another modern multi-input open source clone detector framework *ConQAT* [100,101] detects clones using a suffix tree on tokens. The tool is based on a pipelined approach for extensible token based clone detection. *ConQAT* has been used to detect behaviourally similar code [104]. A number of approaches [45,85] using *ConQAT* have been proposed to detect duplications in Matlab/Simulink models. Acceptance of tools like *CCFinder*, *dup, ConQAT* in academia and research showed the usefulness of the fast speed with which suffix trees detect clones.

Yamashina et al. [240] proposed a novel clone detection/ modification tool to support the software maintenance process. The study reported a substantial difference between novice and experienced programmers regarding motivation and behaviour in handling clones. Using *CCFinderX*'s preprocessor, tokens are gathered from the source code. Then they are input to a suffix array for fast retrieval. Clone retrieval and ranking is performed by the *SHINOBI* server. The evaluation shows *SHINOBI* to be fast and accurate. In the pre-experiment, a large number of programmers were interviewed to analyze their behavior.

SHINOBI still needs improvement in ranking algorithm. However, it can be extended to support other useful information in addition to code clones.

Sasaki et al. [203] developed a new token based clone detection tool *FCFinder* to detect file clones i.e. files which are copied across projects using hashing. The study detected 68% of the FreeBSD Ports collection as file clones. However, *FCFinder* took a long time to detect file clones in the 10GB collection.

2.6.5 Tree based clone detection techniques

Abstract syntax trees and parse trees are frequently used representations when source code is to be transformed into tree structures. However, tools based on this approach suffer from large execution times when analysing a large source code base. The output is purely syntactic units of source code which are ready for refactoring. Tree based clone detection is capable of detecting clones in which the code is inserted or deleted i.e. type-3 clones.

Yang [241] proposed one of the first approaches for finding the syntactic differences between two versions of the same program. The technique was based on grammar and builds a variant of a parse tree for both the versions. Detection is applied synchronously to both the trees and is based on the longest common subsequence method of dynamic programming. A limitation of this approach is that his differential comparator can only work for syntactically correct programs conforming to the grammar.

Semantic Designs' *CloneDR* [23] is another tool which is able to detect exact and near miss clones using hashing and dynamic programming. The tool has different variants for different programming languages. The study reported the use of clone detection in finding commonalities in the form of domain concepts in source code which will help analysts in understanding the design of the system for better maintenance. *SimScan* [211] and *ccdiml* [20] are variations of *CloneDR*. *ccdiml* transforms the source to intermediate representation and *SimScan* applies subtree comparison on the parsed source code. The source code is parsed with the help of ANTLR parser generator. *SimScan* and *ccdiml* have been used to classify the evolution of source code clone fragments in Java and C source code files [227]. Falke et al. [55] and Tairas and Gray [220] used suffix tree to detect clones in code transformed into an AST. The technique has advantage of precision of syntax tree and high speed of suffix tree.

Gitchell and Tran [63] developed *Sim* which converts source programs to parse trees. Viewing parse trees as strings, the tool applied longest common subsequence and dynamic programming to assess similarity. *Deckard* by Jiang et al. [95] is based on computing characteristic vectors from the AST and clustering vectors which are close in Euclidean space by locality sensitive hashing. *Deckard* has been used in localizing the representation of clone groups [224] and has been used to detect behaviorally similar code [104]. Another application of *Deckard* is to assess the impact of code clone on defects in source code [189]. *Asta* [54] is an AST based tool which works on the phenomenon of structural abstraction of arbitrary sub trees of an AST. *ClemanX* [180,181] is an incremental AST based framework. The tool constructs characteristic vectors from AST subtrees and used locality sensitive hashing. Saebjornsen et al. [200] also used the same set of techniques to detect clones in assembly code.

Anti-unification is used in three studies [29,31,151] to calculate the distance between two AST's and grouping the similar classes in one cluster. Anti-unification helps to discover common sub-expressions in source code represented as a tree. *CloneDigger* [31] is a language independent tool in which anti-unification is applied to XML representation of source code. *CloneDetection*, a tree based tool by Wahler et al. [233], is based on frequent itemset mining applied on XML representation of source code. Chilowicz et al. [355] developed a new technique to detect exact clones based on syntax tree fingerprinting.

Shifting our focus to code clone management, *CSeR* (Code Segment Reuse) was developed by Jacob et al. [89] to check copy and paste induced clones in an integrated development environment. The tool was designed to compute clone differences interactively by checking if some piece of code was copy-pasted as the programmer was editing and typing the code. It works on the phenomenon of converting the immediate clone to an AST and computing the difference with the original in a bi-directional manner using metrics like the Levenshtein distance.

Biegel and Diehl [26] introduced a novel way for fast and configurable code clone detection using pipelines. They developed *JCCD*, a flexible and customisable 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.

2.6.6 Graph based clone detection techniques

PDG represents control and data flow dependencies of a function of source code. Horwitz [81] used this method to identify syntactic and semantic differences between two versions of a program. Graph representation of source program is partitioned depending upon the behaviour of source code fragment. The partitioning algorithm and substring comparison are used to detect similarity. *Duplix* [137] works on the k-limiting approach of finding maximal similar subgraphs. *PDG-DUP* [130] is also a PDG-based tool which uses program slicing to find isomorphic subgraphs. It helps in detecting non-contiguous clones. *Scorpio* [80] by Higo and Kusumoto applied two-way slicing to detect clones. The tool currently works on for Java and is based on number of PDG specializations for Java language and heuristics to speed up the overall detection process. It was developed to address the problem that the existing PDG based clone detection approach is slow in detection of contiguous clones.

2.6.7 Metrics based clone detection techniques

Metrics based clone detection techniques are applied after the source code is represented in some suitable form such as an AST. In the systematic literate review, we found a large number of articles using this technique. Characteristic vectors are usually applied for the sub tree matching in an abstract syntax tree or parse tree to detect type-3 clones. The metrics [3, 4] which are extracted from source code are compared to assess similarity. Mayrand et al. [174] developed *CLAN*, one of the first approaches to compare metrics obtained from an AST of source code. Metrics are calculated from names, layout, expressions and control flow of functions. *CLAN* has been widely used in the last decade. Patenaude et al. [183] detected metrics from source code organized in five themes, viz. classes, coupling, methods, hierarchical structure and clones. Kontogiannis et al. [132,133] report a technique to detect code clones using metrics extracted from an AST representation of code. Match detection is done by applying dynamic programming on source code lines using minimum edit distance. Balazinska et al. [13] found metrics and applied dynamic matching to an AST representation of source code. Metrics were used by Lanubile and Mallardo [147] to detect function clones in web applications. Perumal et al. [186] used metrics and fingerprinting technique to detect clones in source code. Kodhai et al. [127, 128] and Dagenais et al. [42] applied metrics on textual representations of source code. Li and Sun [156] explored a novel approach by viewing source code clones in

metric space with coordinate values. The distance between members across same metric space is measured and the distance reflects similarity between code fragments. This study was accurate and scalable but the technique is yet to be verified for different subject systems. Metrics have been used successfully in clone analysis [14, 78], clone evolution [7, 8, 185], clone visualization [96], etc.

Lavoie et al. [148] presented a novel technique based on graphics processing unit (GPU) algorithms to compute many instances of the longest common subsequence problem on a generic GPU architecture using classic DP-matching. Because the algorithm is parallelized on a GPU using dynamic pattern matching algorithm, it leads to an opportunity of increased performance. The tool is useful to address the problem of finding more false positives compared to metrics-based clone detection methods. It also compared clone identification problem on CPUs and GPUs hardware architectures. The study recommends clone detection techniques using string-matching with suffix trees could take advantage of the GPU algorithm.

2.6.8 Hybrid clone detection techniques

Sutton et al. [217] applied an evolutionary algorithm to search for clones in large code bases. The source code is represented as variable size vector. Clustering of similar code fragments is done with the longest common subsequence. Cordy and Grant [75] introduced a technique using an existing information retrieval method, namely independent component analysis (ICA) to analyze vector representations of software methods. Firstly, singular value decomposition is applied on original method token matrix. Then ICA is applied and points in new vector space that correspond to the input data are recognized. The distance between any two vectors is considered a measure of their similarity.

Maeda [170] introduced a technique based on PALEX source code representation. The PALEX source code includes lexical information and parsing actions recorded from the compiler as it processes the source program. The technique is language independent and uses a suffix tree for comparison. The hybrid approach by Corazza et al. [40] uses a tree kernel which is a class of functions for computing similarities among information arranged in tree structure. It works in a recursive fashion on a tree, starting from similarity measure of the nodes and aggregating the results. The technique is language

independent and works at method level for Java programs. The results are compared with AST based tool *CloneDigger*. A disadvantage of this approach is that it takes a substantial time to execute.

Chilowicz et al. [34] developed a technique to detect function clones in source code represented as call graph using suffix array and metrics. The technique starts with collecting tokens using lexical analysis. Basit and Jarzabek [18] developed *Clone Miner* using frequent itemset mining which works on the output of token based clone detection tool , *RTF* [17].

Hummel et al. [84] developed a hybrid incremental index-based clone detector which takes input in the form of token sequences. The index is used to lookup for all clones in a single file and allows updating on addition, deletion or modification. The tool in implemented in *ConQAT* and runs in a pipeline fashion thereby highly scalable, incremental and provides excellent run time performances. In one case study, 100 machines performed clone detection in 73MLOC in 36 minutes. The authors proposed the use of locality sensitive hashing in the current implementation to support the detection of type-3 clones. The technique is used in code clone detection as well as model based detection. Keivanloo et al. [117] introduced *Se-Byte* a clone detection and search model which works on intermediate code representation i.e. bytecode. The technique is based on token matching and applies multidimensional comparison heuristic i.e. Jaccard coefficient. The technique is scalable and provides reliable ranking of clones.

2.7 Comparison and evaluation of clone detection tools and techniques

Comparison and evaluation of clone detection techniques is a challenging task. Diverse subject systems and the absence of standard similarity measures complicate the comparison task. So studies covering comparison and evaluation are immensely important to identify an efficient clone detector.

In all the studies barring one by Burd and Bailey [32], where we mentioned exact values, we wrote ordinal scales (--, +, ++,+++) where --reports lowest and +++ reports highest depending upon the value of parameters in the paper. The reader should refer to the relevant research paper to find the exact values. Different studies evaluated the tools using different parameters. We have left some of the entries blank as the paper does not report the relevant results. These studies are discussed below.

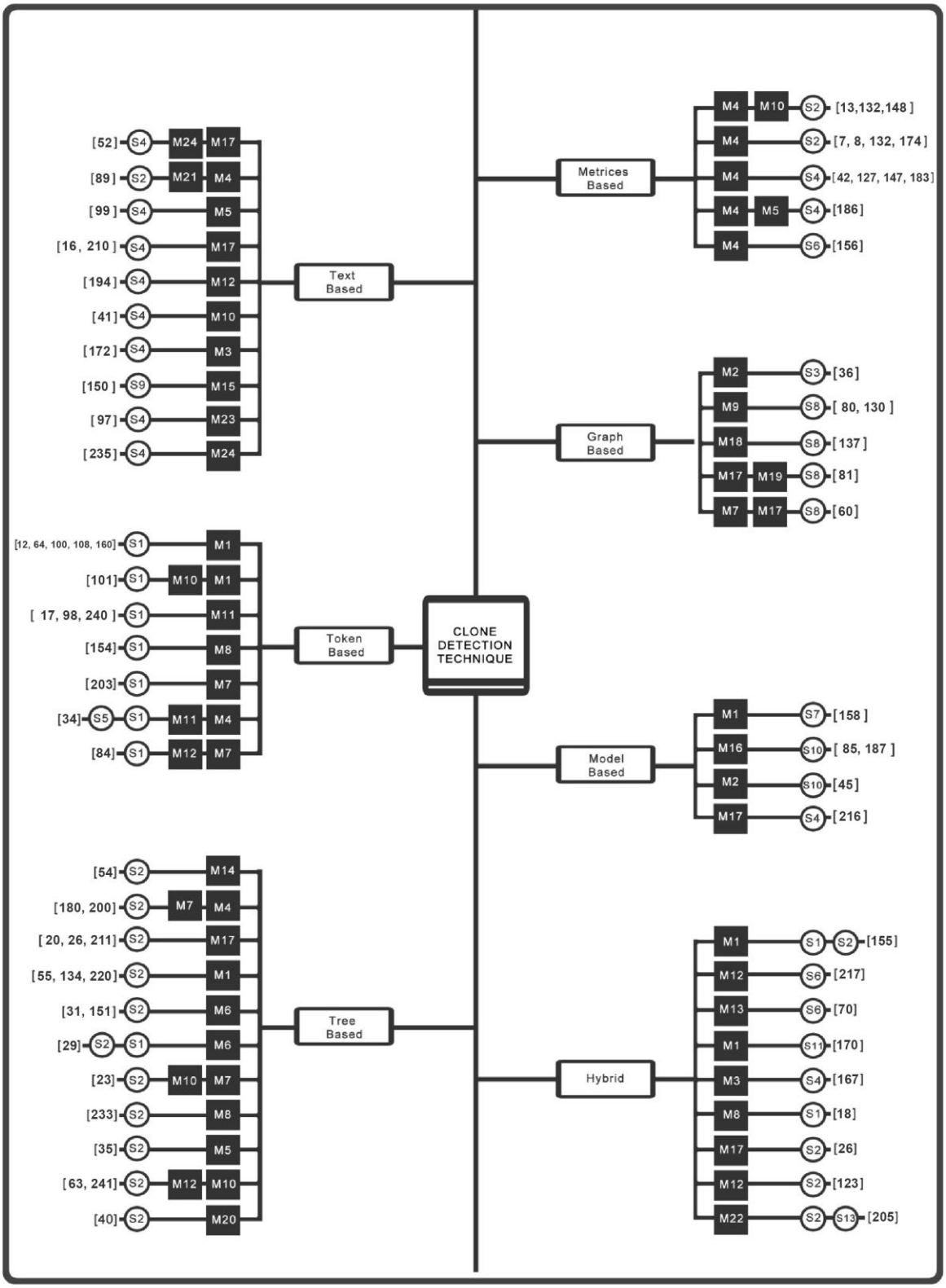


Figure 2.2 Different clone detection techniques, match detection algorithms, source representations and citations

Burd and Bailey [32] conducted the first experiment to compare three clone detection tools, *CCFinder*, *CloneDR* and *Covet* and two plagiarism detectors *JPlag* and *Moss*. They validated all the clone candidates of the subject system obtained by all the tools and made a human oracle. Then they used the human oracle for comparing the different techniques in terms of precision and recall. The result of their experiment shows 100% precision for syntax-based technique *CloneDR* indicating that no false positives in its detected clones are reported by this tool, however, recall was very low. The token-based tool *CCFinder* shows the highest recall 72% and reasonable precision 72%. The metric based tool *Covet* showed the minimum precision i.e. 63% compared to the other tools. The limitation of the case study was in terms of system size. The case study used source code of GraphTool written in Java which was only 16 KLOC.

Rysselberghe and Demeyer [243] compared text-based *duploc*, token-based *dup* and metric based clone detector *CLAN*. The goal of this comparative study was to identify the most appropriate clone detection tool for refactoring. They use five small to medium (under 10KLOC) sized cases for evaluating the techniques. These techniques were evaluated qualitatively rather than quantitatively in terms of suitability, relevance, confidence and focus. The results show that metric fingerprint techniques were best suited to work with a refactoring tool. No significant difference was found between the approaches with respect to relevance and focus. Simple line matching gives the highest confidence as it finds exact matches whereas all the other techniques requires a manual inspection to reject false positives. Their study also found that parameterised matching techniques returns more clones.

Koschke et al. [134] evaluated their AST-suffix tree based tool *cpdetector* in terms of precision, recall and runtime against existing tools using Bellon's experimental procedure and additional tools are *cpdetector, ccdiml, clones* and *cscope*. Since their developed tool is suitable for C systems only, therefore they worked only with 4 C systems of Bellon's experiment and provided detail results for only one system. The results show that the AST-based tool *ccdiml* shows a good recall (53%). However, the average recall for the token-based tools is almost double (54%) than the AST-based tools (30%). Their experiment result shows that clones found 71% more clones than *CCFinder*. Metrics based tool *CLAN* is good and *duplix* is worst in terms of runtime.

Table 2.7 Empirical Comparison studies

Sr. No.	Reference	Tools Compared	Method	Outcome				
1.	Burd and Bailey [32]			Precision	Recall	No. of Candidates		
		CCFinder	Tokens, Suffix Tree	72	72	77		
		CloneDR	AST, Hashing/ DP	100	9	6		
		Covet	AST, Metrics	63	19	19		
2.	Rysselberghe and Demeyer [243]			Suitability	Relevance	Confidence	Focus	
		duploc	Text, Substring Matching			++		
		dup	Text / Tokens , Suffix Tree			-		
		CLAN	AST, Metrics	++		-		
3.	Bruntink et al. [30]			Error Handling	Tracing	Range Checking	Null Value Checking	Memory Error Handling
		CCFinder	Token, Suffix Tree	++			++	
		ccdiml	AST, Tree Matching	++		++	++	
		PDG-DUP	PDG, Program Slicing		++			++
4.	Koschke et al. [134]			No. of Candidates	Rejected Candidates	True Negatives	Recall	
		CCFinder	Token, Suffix Tree	++	++	+	++	
		CloneDR	AST, Hashing/ DP	-	+	+++	-	
		CLAN	AST, Metrics	-	-	+++	-	
		dup	Text / Tokens , Suffix Tree	+	++	++	+	
		duplix	PDG, k length patch matching	++	+++	+++	-	
		duploc	Text, Substring Matching	+	++	+++	+	
		cpdetector	AST, Suffix Tree	+	++	++	+	
		ccdiml	AST, Tree Matching	++	++	++	++	
		clones	Tokens, Suffix Tree	+++	+++	++	+	
		cscope	Tokens, Suffix Tree	++	+++	++	+	
5.	Bellon et al. [21]			Recall	Precision	Time Requirements	Space Requirements	
		CCFinder	Tokens, Suffix Tree	++		+	+	
		CloneDR	AST, Hashing/ DP		++	-	-	
		CLAN	AST, Metrics		++	++	++	
		dup	Text / Tokens , Suffix Tree	++		++	+	
		duplix	PDG, k length patch matching			--	+	
		duploc	Text, Substring Matching	++				
6.	Falke et al. [55]			No. of Candidates	Rejected Candidates	Recall		
		ccdiml	AST, Tree Matching	+	++	++		
		cpdetector	AST, Suffix Tree	+	++	+++		
		clast	Parse Tree, Suffix Tree	+	++	+++		
		clast-ba	Parse Tree, Suffix Tree	+	++	+++		
		clones-uk	Tokens, Suffix Tree					
		clones-ba	Tokens, Suffix Tree					
		cscope	Tokens, Suffix Tree	+++	+++	+++		
7.	Roy and Cordy. [193]			Recall	Precision			
		Basic NICAD	Text, LCS	+	+			
		FlexP NICAD	Text, LCS	+	+			
		Full NICAD	Text, LCS	++	++			
8.	Pham et al. [187]			Precision	Completeness			
		CloneDetective	Labeled multigraph, maximum weighted bipartite matching	+	+			
		ModelCD	Sparse labeled directed graph, canonical labeling	++	++			
9.	Deissen-boeck et al. [47]			Time				
		CloneDetective	Labeled multigraph, maximum weighted bipartite matching	++				
		ModelCD	Sparse labeled directed graph, canonical labeling	+				
10.	Juergens et al. [104]			Partial Clone Recall	Full Clone Recall			
		Deckard	AST, Characteristic Vector/ Hashing	++	++			
		ConQAT	Tokens, Suffix Tree	+	+			
11.	Selim et al. [206]			Recall	Precision	Rejected Candidates		
		Selim et al.	Jimple, Suffix Tree/Subtree Comparison	++	++	++		
		CloneDR	AST, Hashing/ DP	+	+	+		
		CLAN	AST, Metrics	+	+	+		

Table 2.7 shows the empirical studies for comparison and evaluation of clone detection tools.

Bellon et al. [21] conducted a tool comparison experiment with the same three clone detection tools that were used in Burd and Bailey's study and with three additional tools, viz. *dup, duplix* and *duploc*. They also used software systems in Java and C with 4 Java and 4 C systems totalling almost 850KLOC. Their experiment showed that precision and recall was complementary for each of the tool except the PDG-based *duplix* where both as attributes exhibited the lowest values. The AST based *CloneDR* and the metrics based *CLAN* had high precision but their recall was very low. The token based tools *dup* and *CCFinder* and the text based tool *duploc* had the highest recall but low precision. The experiment shows that the PDG-based tool *duplix* performed very badly in terms of execution time compared to the other clone detection tools.

Falke et al. [55] empirically compared *ccdiml*, *cpdetector*, *clast*, *clast-ba*, *clones-uk*, *clones-ba* and *cscope* using subject systems in C and Java. The results show that token based tools yield large number of clones. AST matching shows lower rejection rate, but also has a lower recall. The result indicates that detection based on suffix trees is faster than detection based on tree matching. Parse tree based tools show lower rejection rate than AST-based tools for C systems but a higher rate for Java systems. Also parse tree based tools are faster than AST based tools for Java systems. In contrary to previous experiments, this study does not show high recall for token based tools. According to this study the advantage of token based techniques is that it is easier to implement lexer than parser and requires less space than AST. On the other hand AST based techniques are good to find syntactic clones and help to filter those syntactic structures which are of little interest.

Roy and Cordy [193] propose a mutation/injection based framework for evaluating clone detection techniques. In this method mutant versions of code fragments are created. Then these mutant versions are injected into original source code. Different variants of tool *NICAD* viz. *Basic NICAD, FlexP NICAD* and *Full NICAD* are run on these mutant versions to compare tool on basis of precision and recall. The result of this study shows that *Basic NICAD* has poor recall for clones generated by mutant operators of type-2 clones. *FlexP NICAD* is also not good to find clones generated by mutant operators of type-2 clones. *Full NICAD* is best to find clones generated by mutant operators of all type

of clones. The advantage of this framework is to evaluate and compare recall of different clone detection tools without manual intervention and similarly precision can be evaluated either automatically or by using an interface with minimal manual intervention. The experiment [47] compared the techniques of Pham et al. [187] and Deissenboeck et al. [45]. The study proposes reduction in branching to avoid multiple occurrence of sub graph in the search tree. This is done to prune the search space to make it relevant and fast. The techniques to remove obvious clones and branch reduction lead to lower recall.

Juergens et al. [104] demonstrated the applicability of state of the art tools in detecting behaviourally similar code. The authors stated that behaviourally similar code is highly unlikely to be syntactically similar and such code results due to independent development. *Deckard* and *ConQAT* cannot detect more than 1% of such code. Selim et al. [206] proposed a hybrid technique in which clone detection is performed simultaneously for source code and intermediate code which are merged to detect near miss clones in Java. Using Bellon benchmark [21], comparison with other state of the art tools is done. The technique gives lower precision and higher recall than *CCFinder* and *Simian,* when used standalone. The technique detected some clones which are not useful. Use of the technique on large code bases is still to be done.

An assessment of type-3 clones as detected by the clone detection tools namely *ccdiml*, *clones, clast, duplix* and *CLAN* is performed by Tiarks et al. [226]. The study apparently points out that existing type-3 clone detectors need to be improved as only 25% of detected clones are accepted by human oracle. An empirical study [196] is carried out regarding function clones in open source software.

Upon assessing each of studies, it is difficult to find out the best tool for each study. After a complete review of all studies, we are able to write following general remarks:

Token based clone detection tools like *CCFinder* detected large number of clones. They have high recall and reasonable precision. These tools do not help the developer in refactoring in a straightforward manner.
Tree based tools like *CloneDR* have high precision. Though these tools detect very less number of clones with low recall, but the detected candidates are ready for refactoring thus helping the developer in clone management.
Metrics based tools like *CLAN* have good precision but suffer from low recall and less

number of candidates detected. These tools run fast and detect function clones ready for refactoring.

Tools which apply suffix tree in AST representation of code like *cpdetector* works faster than other AST based tools. One PDG based tool named *duplix* is able to recognize type-3 clones only but suffer from high time complexity.

We have not found any empirical study comparing various semantic clone detection tools/ techniques. There are two papers comparing model based clone detection approaches i.e. *ModelCD* and *CloneDetective and ConQAT*. Benchmark [21] has been used by [134], [21], [55] and [206] in their empirical studies. It consists of clone pairs validated by humans for eight software systems written in C and Java from different application domains.

We have mentioned the empirical studies comparing different clone detection tools in the above paragraphs. These studies are undertaken by taking one or more subject systems to evaluate the tools. Evaluation is done with parameters like precision, recall, etc. as shown in Table 2.7. Other studies comparing clone detection tools/techniques qualitatively are explored in the following paragraphs. Studies devising clone oracle are also discussed in following paragraphs.

Ducasse et al. [53] compared their string based clone detection technique *duploc* with Baker's token based tool, *dup* and Kamiya's token based tool, *CCFinder* and confirms that this inexpensive clone detection technique can also yield high recall and acceptable precision. This study uses Weltab of size 9 KLOC and Cook of size 46 KLOC as subject systems. The detailed results of the study are shown in Table 2.8. For *duploc* - means no normalization; C means constants normalized; I means identifiers normalized; F means functions normalized; CI means constants and identifiers normalized; IF mean identifiers and functions normalized; CIF means full normalization; and 0,1,2 specifies the maximum gap size in the comparison sequence. The results of the study show that for Weltab, normalizing identifiers and function names is important to achieve similar results as that of *dup* and *CCFinder*. For Cook, normalizing identifiers can lead to too many clones but normalizing constants only with maximum gap size zero gives good precision. The study confirms that this inexpensive clone detection technique can also yield high recall and acceptable precision, although the tool needed to be extensively calibrated to each system. However, the results also show clearly that the effectiveness of clone

detection tools is strongly influenced by the specific system to which the algorithms are applied. This confirms that studies attempting to compare different types of clone detection tools must evaluate them on a variety of different systems.

Table 2.8 Comparison of three clone detection systems on two software products

Study/tool	WELTAB			COOK		
	Candidates	**Precision (%)**	**Recall (%)**	**Candidates**	**Precision (%)**	**Recall (%)**
Dup	2742	80	80	8593	29	70
CCFinder	3888	99	93	2388	42	43
Duploc	2378(IF,1)	90	86	9043(CIF,0)	26	71
	2609(IF,1)	90	88	7661(-,2)	42	64
	3761(CIF,0)	91	92	276(C,0)	49	26

Roy and Cordy [192] compared clone detection techniques on the basis of several criteria like language support, comparison granularity, clone similarity, code representation etc. They also propose a set of hypothetical editing scenarios for different clone types and evaluate the clone detection techniques based on their estimated potential to accurately detect clones that may be created by those scenarios. Their studies results shows that hybrid technique based on tree-based techniques e.g. *cpdetector* and text based techniques e.g. *duploc* can detect type-1 clones efficiently. Hybrid approach of token based i.e. *dup* and *RTF* and AST based i.e. *cpdetector* and *Asta* best suits to find type-2 clones. Type-3 clones are best found by hybrid approach based on text based *DuDe* and *SDD* and AST based *Deckard* techniques and graph based techniques *duplix* and *GPLAG* are best suitable to find semantic clones. The results of this study are predictive rather than empirical but assist to understand and find interesting combinations of techniques.

In another study, Roy et al. [195] conducted a large case study to classify and compare clone detection approaches based on a number of facets, each of which has a set of attributes. They qualitatively evaluated the classified techniques and tools with respect to taxonomy of editing scenarios designed to model the creation of Type-1, Type-2, Type-3 and Type-4 clones. So this case study compares the clone detection techniques and tools qualitatively and helps to understand the potential of each technique and tool to find clones generated by different scenarios.

Walenstein et al. [234] carried out a study to highlight the problems in devising

universal oracle. The study involves researchers' view in classifying the candidate clones into clones and non-clones categories as detected by tools, viz. *dup*, *CloneDR*, *CCFinder*, *duplix*, *CLAN* and *duploc*. Authors observed high level of disagreement. This exploratory study is a step towards clone detector benchmarks. The study pointed out that inter-rater reliability measures should be calculated for human generated reference data. Lakhotia et al. [145] stressed on the need to create a community supported open benchmark suite to help researchers in evaluating and comparing clone detectors and choosing standard subject systems for the study. Lavoie et al. [149] presented a novel technique to construct clone oracles automatically in large systems based on the Levenshtein metric. Compared to manual oracles, this oracle is of good quality for type-3 clone detection assessment and is able to deal with large code base in reasonable time. The oracle generates good quality clones and aims for comparison and evaluation of clone detection techniques in a reasonable time.

Barring one comprehensive study by Roy et al. [195], comparative studies of clone detection techniques pertain to some subset of tools. We found only eleven empirical studies as shown in Table 2.7 which shows apparent lack of work in comparison and evaluation of clone detection techniques. We found two studies by Deissenboeck et al. [47] and Pham et al. [187] on comparison of existing model based clone detection techniques.

2.8 Key Sub Areas

We categorized the literature related to software clones in 6 different yet allied areas. We realized that different identified categories are very important from the research perspective in the field of software clones. Although many sub areas overlap each other, we have tried to separate them into self contained topics. Cross cutting studies in some of these key areas are included in a clone management systematic map constructed to answer research question 4 in section 2.9.4 see Figure 2.4. Details of these studies have been discussed in their respective key areas to make the corresponding key area complete in itself. This is done to help researchers to pursue further research in these sub areas.

2.8.1 Code Clone Evolution

Code clone evolution depicts the patterns in which the code is developed throughout the history of the software. Other interesting patterns of clones by evolution of software

through different versions are also covered in clone evolution.

As software evolves with time and introduction of newer versions, so clones present in the software evolve too. Laguë et al. [144] pioneered clone evolution analysis using metrics based function clone detection technique for large telecommunication monitoring system of 1 million LOC. Two changes, viz. preventive control to keep the introduction of newer clones in system under control and problem mining to cope with the existing code clones in a system under continuous development and maintenance are studied to assess their impact on clone detection design. Both the changes can effectively improve clone management. Antoniol et al. [7, 8] analyzed different versions of the Linux kernel to check the changes in cloning patterns. In an earlier study [7], the author modeled the time series by analyzing clones over several versions of software. Shawky and Ali [209] modelled clone evolution using chaos theory. The study predicted clones in new versions of open source systems with high prediction accuracy. Di Penta et al. [184,185] devised a framework i.e. *Evolution Doctor* to control software system evolution. It defines a set of methods and tools to deal with removal of clones, restructuring and reorganizing the source code.

Code clone genealogies approximate how programmers create, propagate and evolve code clones [122]. Saha et al. [201] carried out an empirical study to evaluate and understand clone genealogies in 17 open source software systems in 4 different languages at release level. Their study is an extension to a study by Kim et al. [122]. Unlike [122], this study analyzed the evolution of clones at release level. It used *CCFinderX*, a token based clone detection tool. A location independent approach was used to match identifier names across releases. The clone genealogies were classified as alive genealogy, dead genealogy, syntactically similar genealogy and consistently changed genealogy.

Clone Evolution View [15] was developed to work with a metrics based clone detection tool to study how developers copy across different versions of a program. Livieri et al. [159] carried out evolution analysis of 136 versions of Linux kernel using code clone coverage metrics. They used *D-CCFinder*, a distributed extension of code clone detection tool *CCFinder*. Aversano et al. [11] performed an empirical study using *SimScan*, a syntax based clone detector, on two open source Java systems. They defined three clone evolution patterns to study the effect of maintenance activities on clones.

Krinke [138] pioneered the argument that clones have a bad impact on software

maintenance. In five large open source systems, the author traced the changes across 10 weeks. The study showed that clone groups are continually changed in 50% cases. Lozano et al. [161] analyzed clones at method level. *CloneTracker* was applied to measure the number and density of changes in two cases i.e. during the presence of cloned code in methods and not. The study observed that cloned code methods need to be changed more frequently than non-cloned code. Lozano et al. [162] presented a way to perform origin analysis and assess the effect of cloning on methods' maintenance effort. The study computed measures of likelihood and the impact of change as they represent the work required for maintaining a method. The study concluded that change effort increases when a method has clones. In another study [163] a resilient approach to track clone instances over time is presented. Extension, persistence and stability in methods were the chosen metrics to assess the cloning imprint and impact of clones on changeability of the application. In sample subject systems, the study concluded that cloning extension remains stable in 10-20% of the application. Otherwise cloning presents low stability, high persistence and low extension.

An ethnographic study of clones in object oriented programming paradigm was carried out by Kim et al. [120]. Software developers' copy and paste programming behavior was captured with the help of a logger. The study observed that language limitations and programmers intentions promote clones. The study pointed out the use of copying and pasting in understanding and restructuring source code. Detailed evolution analysis of type-1 clones was carried out by Göde [65]. The study included 200 revisions of nine open source systems. He found that the lifetime of clones decreased on average. The minimum clone length and programming language had little impact on the results but the lifetime of fragments differed between the systems.

2.8.2 Code Clone Analysis

This topic covers studies related to refactoring of code clones based on code clone classification or detection. Many software renovation frameworks work on improving software by miniaturization of code i.e. clone removal. Some studies investigated cloning patterns. These are also included in this domain.

Fanta and Rajlich [56] used *CloneDR*, *dup*, and *CLAN* to develop a re-engineering process intended to eliminate clones from a proprietary software of the Ford Motor Company i.e. Powertrain Engineering Tool sized at 120 KLOC. The study pointed out

disadvantages of clones. Balazinska et al. [14] developed a clone re-engineering tool *CloRT* which works with any AST based clone detector. Their analysis focused on two aspects of clones i.e. the meaning of their differences from programmers' point of view and context analysis which help in refactoring. The study concluded that clones are good candidates for refactoring. The clone analysis tool *Gemini* [231] takes the output of *CCFinder* and displays the results in the form of scatter plot and metrics graph. The tool provides users with useful functions for analysis, maintenance and refactoring of code.

Higo et al. [76, 78] suggested code clone analysis based on a refactoring perspective. The tool *ARIES* gives indicators for certain refactoring methods in the form of metrics from the output of *CCFinder*. It helps in merging code clones. In one study [80], a tool *Libra* is developed for simultaneous modification based support method. *Libra* helps in finding clone candidates for the input file selected by the maintainer.

Kapser and Godfrey [112,113] observed that code clone detection tools produce large result sets hindering in-depth investigation of any subject system. They carried out the study on the Apache web server to gain insight into cloning patterns and understand cloning behaviour. The study concluded that cloning usually occurs in related sub-systems. Kapser and Godfrey [115] used *CLICS* to determine clone characteristic and location patters. Cloning patterns are defined by what, why and how cloning takes place. In the sample subject system, the study concluded that patterns have good impact on a software system if used carefully. Quo et al. [188] stressed the need to shift the onus of pattern mining of clones from spatial space analysis to logic domain analysis. The authors use a PDG as the source code representation and propose a joint space-logic domain framework for pattern mining. The proposed approach mines two lists of patterns: one ordered by reused times and other by size of patterns. Former patterns provide good inputs for code optimization. The tool helps in locating related defects across identical pieces of code.

Tairas et al. [222] developed *CeDAR* (clone detection, analysis and refactoring) which takes the output from *CloneDR* and other clone detection tools. It helps in refactoring of detected code clones. Tairas and Gray [223] applied latent semantic indexing to find relationships in clone classes, thereby helping in clustering and maintenance of clones. Clone detection was done with the help of *CCFinder* on a sample subject systems viz. MS Windows NT source code kernel. In one study, Tairas et al. [224] used *CeDAR* to represent clone groups in a localized manner and information about

each clone in a group could be viewed in one location. Each clone in the group is represented as an AST, after then suffix tree is used to trace similarities and differences. The technique is capable of finding near miss clones and uses *Deckard* as a back end.

Jarzabek and Li [91] found out that 68% of code in Java Buffer library, JDK1.5 was contained in cloned classes or class methods. Manual investigation of situations leading to clones revealed difficulties in their elimination. The authors proposed a generative programming technique of XML based variant configuration language (*XVCL*) to analyze and represent clones. Tiarks et al. [226] found a large number of false positives i.e. as high as 75%. This has an adverse impact of clone understandability. It also hinders the use of clone detection for practical purposes. Respective studies were carried out to detect and remove code clones from Erlang/OTP [155] and Haskell [29] programs. Jacob et al. [89] used different mechanisms to compute differences in clones. Their study used metrics and the Levenshtein distance to display the changes interactively in the code to the developer.

Juergens and Göde [102] successfully used clone coupling to detect relevant clones from a large number of false positives. The clone detector is re-run to improve accuracy by removing false positives. Shawky and Ali [208] assessed a set of metrics for similarity prediction for clone detection. Precision and recall were calculated for every experiment. They concluded that the order in which metrics are used for clone detection affects results.

Krinke et al. [139,140] analyzed the version control system to distinguish the original from the copied code. Each clone pair is classified as identical, copied or unclassifiable. The clones of a clone pair are said to be classifiable with a tolerance based on the Levenshtein distance. In a particular case study [140] for GNOME projects, more than 60% of the clone pairs could be separated into original and copy. Increasing the minimal clone size in the clone detection tool i.e. *Simian*, the number of clone pairs decreased asymptotically. In another work, Basit et al. [19] carried out a study to analyze the presence of simple clones in structural clones. They used the clone detection tool, *Clone Miner*. Across a comprehensive set of 11 subject systems, structural clones are analyzed to identify their location, similarities, etc. The analysis of structural clones i.e. high level similarities is intended to help in understanding, refactoring and maintenance of the software system. The study concluded that over 50% simple clones were a part of structural clones. Bian et al. [25] proposed *SPAPE* which is a procedure extraction

method for near miss clones. The method works by building PDG and then transforming it to AST. After that the differing nodes of AST are combined for refactoring.

2.8.3 Impact of Software Clones on Software Quality

Do clones have adverse effect on system quality i.e. maintainability, reliability, etc.? This has been a matter of extensive discussion in research community. This section summarizes the related studies on whether the clones are harmful or not in the form of a table. In this section, we also include studies concerning detection of bugs as side effect of cloning and inconsistent modification.

Monden et al. [175] used *ARIES* to discover the relation between software clones and software quality attributes like reliability and maintainability. The study concluded that modules or files having large code clones of more than 200 LOC are less reliable and maintainable than non-clone modules. Imai et al. [86] estimated the maintenance cost caused by clones. The study measured functional redundancy which is a degree of propagation of clone potential function. After clustering, a functional redundancy tree is constructed where the weight given to each node of the tree indicates cost.

Krinke [141] carried out an important study to compare the stability of cloned code and non-cloned code. He studied changes particularly deletions, additions and changes of five open source systems across 200 weeks in cloned code and non-cloned code. This study was extended by Göde and Harder [67] using different parameters and detailed measurements. They concluded that clone detection parameters influence the results but do not change the relation between stability and non-stability. They also concluded that type-1 clones are less stable than type-2 and type-3 clones. In addition to [141, 67], Krinke [142] investigated the age of code as a parameter for clone stability. He calculated the average age of cloned code and concluded that cloned code in a file is usually older than non-cloned code, thus more stable.

Table 2.9 Studies related to whether the clones are harmful or not

Sr. No.	Study	Focus of the Study	Findings/ Results
1.	Monden et al. [175]	Relation between code clones and software reliability and maintainability	Clone included modules are more reliable and less maintainable than non cloned modules

Sr. No.	Study	Focus of the Study	Findings/ Results
2.	Kim et al. [120]	Analysis of programmers' copy and paste programming practices	Cloning is not always harmful as many of the clones are intentionally introduced by the programmer which helps in fast development and program understanding
3.	Lozano et al. [161]	Analysis of harmfulness of cloning based upon change based experiment at method level granularity	Clones are harmful and have bad impact on maintenance
4.	Kapser and Godfrey [115]	Categorization of cloning patterns in programs to know whether they are useful or not	Cloning is not always harmful and some of the cloning patterns are beneficial to software development and maintenance
5.	Krinke [141]	Analysis of stability in terms of changes to the system from an analysis of 200 weeks of evolution	Cloned code is more stable than non-cloned code
6.	Juergens et al. [101]	Analysis of inconsistent changes to code clones and whether these changes lead to defects	Inconsistent clones are a major source of faults thus increasing maintenance
7.	Juergens et al. [103]	Clone detection is done in requirement specification documents by customizing code clone detection tool	Cloning is harmful in requirement specification documents
8.	Rahman et al. [189]	Analysis of relationship between cloning and defect proneness	Cloning do not introduce more bugs thus not harmful
9.	Selim et al. [207]	Whether cloning is harmful? What features of cloned code make it error prone?	Harmfulness of cloning is subject system dependent.
10.	Bettenburg et al. [22]	Analysis of effect of inconsistent changes to code clones on software quality at release level	Cloning do not effect post release quality of the subject system
11.	Göde and Harder [67]	Analysis of stability of the software system in terms of deletions to the cloned code and non- cloned code	Cloned code is more stable than non-cloned code
12.	Krinke [142]	Analysis of stability of the software system using average age of cloned code in comparison to non cloned code	Cloned code is more stable than non-cloned code

Table 2.9 lists the related studies on whether the clones are harmful or not. In addition to the studies which analysed the relation between code clones and software quality to know the impact of clones, we also listed some studies like Kapser and Godfrey [115] which categorised the cloning patterns to be harmful or harmless. Some of the studies mentioned in the table are discussed in other sections of this chapter.

It is always 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. Jablonski and Hou [88] identified features of *CnP* which help in cutting software maintenance costs. Clone information during software development helps programmers in modification and debugging tasks. Visualization and renaming features i.e. *CReN* and *LexID* of *CnP* were tested thoroughly. The study showed the effect of clone information on maintenance. Bettenburg et al. [22] explored the effects of inconsistent changes to code clones on software quality at release level through an empirical study. Developers usually carry numerous changes in different parts of software systems across releases. They are usually interested in source code at release level.

Earlier studies investigated the impact of code clones at much finer granularity level. Duala-Ekoko and Robillard used *Clone Region Descriptor* [51] approach to track code clones across releases. All clone genealogies found were manually inspected for inconsistent change. Juergens et al. [103] carried out clone detection to find copy and paste activities in requirements specifications documents. Large documents are used to detect any redundancy, thereby assessing the quality of specifications. The comprehensive study was carried out across 28 documents with 9000 pages, fixing clone length to be 20 words. Rahman et al. [189] carried out an empirical study to investigate the impact of code clones on defects in code. The clones were not found to be particularly error-prone. Clone detection tool, *Deckard* and data mining from version control and bug repositories were used to carry out the study. Göde [66] found that most of the clones detected by state of the art clone detectors need not be removed. His study has significant implications from point of view of software maintenance. With standard subject systems as input and an incremental clone detector tool, the study identified procedure extraction as the most commonly used refactoring method. Kozlov et al. [136] explored internal quality attributes and code clone detection metrics. For all 117 releases of peer to peer open source software namely, eMule, a software project fork, internal quality attributes were measured using *SoftCalc* tool and code clone detection metrics were extracted using *CCFinderX*. Statistically significant correlations across groups were identified based on the Pearson product moment correlation. The study was based on a number of hypotheses taken from prominent clone detection studies using metrics. The study concluded that internal quality attributes act as explanatory variables for code clone detection metrics. Selim et al. [207] used survival models to study the impact of clones on software defects. The probability of occurrence of a defect at any time was modeled using Cox's

proportional hazard function. The study used a set of predictors to classify clones as helpful or harmful. The study was the first of its kind to use Spearman correlation between actual and predicted occurrences of defects. It explored a new set of predictors related to code siblings at method revision level. They concluded that cloned code is not always more risky than non-cloned code.

Copied code leads to inconsistencies at multiple places when a bug is introduced in the original code fragment. Automatic detection of these bugs has turned to be an allied area effecting the quality and maintainability of source code. Li et al. [154] pioneered defect detection in clones using *CP-Miner*. The technique was able to detect code duplications and related bugs using a frequent subsequence mining technique. However, the tool reports many false positives. Jiang et al. [94] stated that inconsistencies emerge when code is copied to a new place and appropriate changes are not made with reference to new context. They used *Deckard* to detect context based clone related inconsistencies or bugs. The approach was able to discover previously unknown bugs and the results were compared with *CP-Miner*. *CCFinder* was used to find identifier naming inconsistencies and *CP-Miner* was used to filter code clone related bugs. Yoshida et al. [244] used lexical analysis and identifier similarities to detect duplicate code. The technique helps in detecting code fragments with similar defects. In a similar study, Yoshida et al. [246] detected similar defects in source code based on comparison of synonymous identifiers using the Jensen-Shannon divergence method. Clustering of identifiers was based on distance between identifiers. They also compared their results with *CCFinder* on the same subject systems. Juergens et al. [101] introduced the *CloneDetective* framework to detect inconsistencies. The study pointed out that inconsistent clones lead to faults. In the sample subject system, 58% of the clones contain inconsistencies. Gabel et al. [61] introduced *DejaVu*, a tool to detect inconsistent bugs. Firstly similar code fragments are found using a *Deckard* based clone detection framework. Later on, buggy change analysis is done to classify benign and buggy inconsistencies from the clone detection data set. The potential bugs are classified as: bugs, code smells, style smells, unknown, false reports. Jalbert and Bradbury [90] used clone detection and rule evaluation to identify bugs in concurrent software. They tried to reduce the domain of testing by using clone detection. An identified bug is input to find similar bug patterns in concurrent software.

2.8.4 Clone Detection in Websites

Websites i.e. web applications are usually multi-lingual and suffer from short development life cycles. Changing requirements frequently leads to introduction of clones. Many studies confirm the presence of clones in websites. The extent of cloned code varies from 30% [218] to as much as 63% [190].

Aversano et al. [10] proposed reuse of existing sites by means of cloning and adaptations from a repository of conceptual views and code components. Lucca et al. [164] presented an approach to detect duplicate web pages using similarity metrics. They proposed refining the results of similarity metrics using the Levenshtein distance between each pair of analyzed HTML strings. Lanubile and Mallardo [147] conducted a study to detect script function clones using metrics. The study was conducted on three sample subject systems and found 39%-50% of the total script functions to be clones. Synytskyy et al. [218] and Cordy et al. [41] used island grammar for simultaneous parsing of multilingual web applications. They used UNIX *diff* command to compare potential clones. The method is able to detect near miss clones. Lucia et al. [165, 166] proposed a method to identify cloning patterns in a web application. The method was based on clone detection using similarity thresholds. The technique was helpful in clone analysis and reducing the code size and navigational patterns of a web application.

Rajapakse and Jarzabek [190] used *CCFinder* to detect patterns of clones in websites. Authors extended the study further to unify most of the clones using server pages. This helped in reducing code size and the possibility of update anomalies. Different clustering algorithms and latent semantic indexing were used by Lucia et al. [167] to detect clones in web applications. The techniques were tested on different static web sites. Different clustering algorithms produced comparable results. Jung et al. [106] explored three levels of views namely, relationships between web applications, passed parameters and target applications for detecting clone pairs in a web application. Clone pair candidates were selected based on static and dynamic approaches. The combined approach was successfully validated on two open source projects. Martin and Cordy [173] introduced the concept of contextual clones i.e. clones that can only be found by augmenting code fragments with related information referenced by the fragment to give its context. They proposed a technique to leverage the idea of contextual clones to detect similarities in web service description language.

2.8.5 Cloning in Related Areas

Duplication is also prevalent among other software artifacts like requirement specifications, binary executables, etc. Such studies are included in this category. Studies of different applications of clone detection are included in this category too.

Software archives contain large amount of similar software. Kawaguchi et al. [116] applied a code clone based similarity metric, a decision tree based approach and latent semantic analysis based approach to categorize similar software in an archive. The technique helps to identify relationships among software systems.

Domain analysis is usually carried out in device drivers to search for similar implementations. Mao et al. [171] used table recognition technology and clone detection to convert conventional table-based websites to modern cascading style sheet websites. Software clone detection was used to detect common layout styles in pages. German et al. [62] studied technical and legal implications of cloning in code siblings when there is license compatibility between copyright owner and destination. Monden et al. [176] chose metrics with a lower bound on code clone measurement threshold to determine violations in open source licensing. 50 open source subject systems were chosen from free software directory to trace violations. Brixtel et al. [28] used clone detection tool for checking plagiarism in student projects and assignments. They tested their technique on projects using different languages. Davis and Godfrey [43] used assembly instructions to source code clones.

Software product lines have a core part around which its different components are built. Schulze et al. [204] identified clones in feature oriented software product lines. Using *CCFinder*, a significant number of clones were detected in 10 subject systems. They discussed reasons for cloning in feature oriented software product lines and the way most of the clones can be refactored. Domann et al. [49] used the *CloneDetective* [100] framework to find instances of clones in 11 software requirement specification documents of 2500 pages. Clones in binary executables were detected by Saebjornsen et al. [200]. This tree based approach worked by clustering of characteristic vectors on labeled trees. Ciancarini and Favini [38] carried out a study to detect clones in game playing software. They found a criterion to judge similarities in game playing software with chess as an example. Juergens et al. [103] investigated the amount and nature of duplicated text in software requirement specifications using *CloneDetective* tool. After detecting

duplications in 28 software requirements specification (SRS) documents of 8667 pages, recommendations were given for working with SRS in practice. Whaley and Lam [237] applied cloning in pointer alias analysis by creating clones of methods. They used binary decision diagrams to achieve context sensitive results.

2.8.6 Software Clone Detection in Aspect Oriented Programming/ Cross-cutting Concerns

Aspect-oriented programming (AOP) was a response to the problem of cloned code relating to cross-cutting concerns e.g. logging, and error-handling in object-oriented systems. The detection of code that should be refactored into an aspect is an important part of AOP.

Yokomori et al. [242] analyzed the relationships between aspects to understand how the behavior of existing code clones in original classes spread to aspects. The study used *CCFinder* to detect code clones. The study concluded that number of clone relations between class and aspect increases if only one part of clone group is extracted. Bruntink et al. [30] conducted a study where the token-based *CCFinder*, AST-based *ccdiml* and PDG-based *PDG-DUP* were evaluated in terms of finding cross-cutting concerns in C programs with homogeneous implementations. Some well known cross-cutting concerns such as error handling, tracing, range checking, null-value checking and memory error handling were found. Their study showed that both *ccdiml* and *CCFinder* are best suited for null-value checking and error handling concerns while *ccdiml* is also suitable for the range checking concern. *PDG-DUP* can find tracing and memory handling concerns. The study confirmed that code clones contribute 25% of the code size in aspects. Code clone classification was done by adding semantic information to clone detection tool.

Figure 2.3 shows the number of publications in different key areas in the years 1997 to 2011. Different trends can be seen for different key areas. Research in the area of impact of clones on software quality rose most dramatically in 2010 from a single study in 2009 to ten studies in 2010. The number of publications in area of code clone analysis grew sharply, almost doubling from about one study in 2007, two studies in 2008, five studies in 2009 and eight studies in 2010. The research in the key areas of code clone analysis peaked in 2010. It shows the improvement in recognition and need of research in these areas recently. On the contrary, research in area of clone detection in AOP comprised the

smallest numbers. On the other hand, number of publications in the key areas of clone detection in websites, cloning in related areas and code clone evolution remained stable throughout the years.

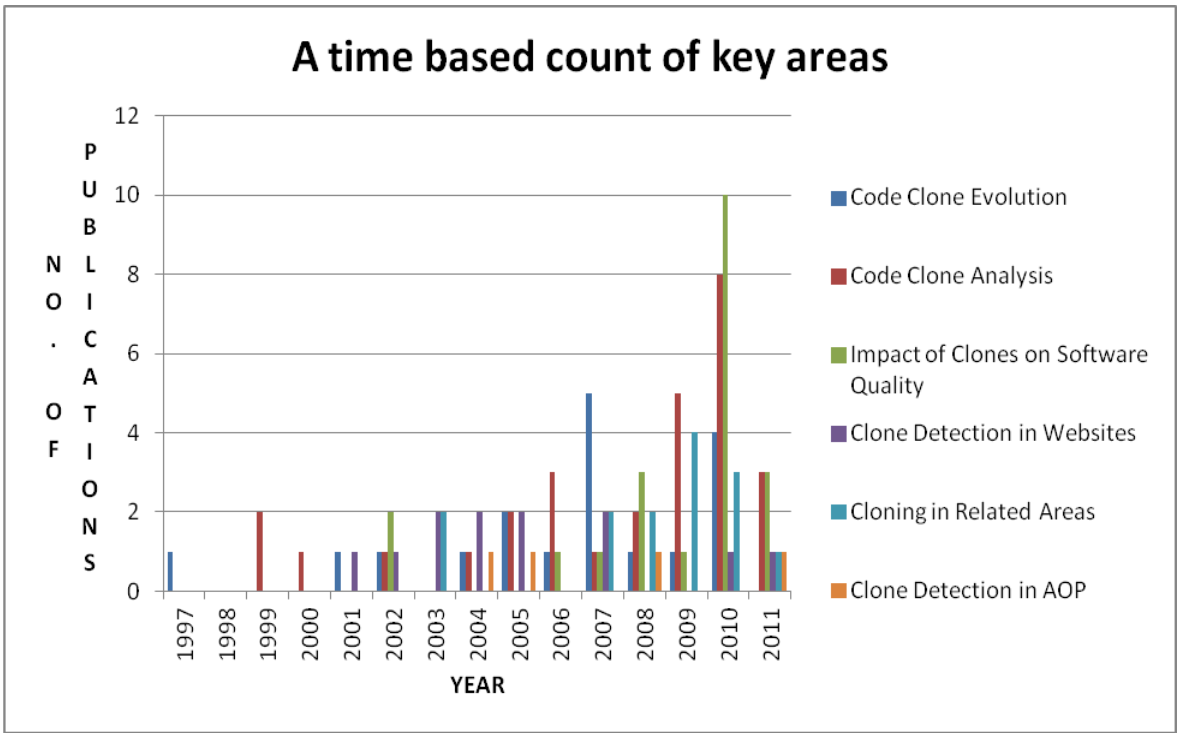


Figure 2.3 A time based count for key areas

2.9 Current status of Clone Management

Clone management is a set of activities like clone classification, refactoring, visualization, tracking and evolution. It plays a pivotal role in development and maintenance process. Some of the benefits of clone management are discussed in sub section 2.9.1. However, it also overlaps with clone analysis, clone evolution, and impact of clones on software quality which are discussed in section 2.8.

2.9.1 Benefits of clone management

In literature, we came across many studies which discussed benefits of clone management. Some of the findings are:

- Laguë et al. [144] identified that an effective clone management strategy improves customer satisfaction and software system quality.
- Kim et al. [120] pointed out that many clones are intentionally introduced in code due to language restrictions. Thus refactoring after clone detection may not

improve the software. So it is more important to manage the clones to see how they evolve over time. Moreover, clone management in an Integrated Development (IDE) makes developers concerned about duplication. The developer should be informed in the IDE about all the clones which are introduced deliberately due to hard time constraints, etc.

- Kapser and Godfrey [115] identified cloning patterns which are helpful in improving the quality of the system. They found that as many as 71% clones had a positive impact on the maintainability of the software. They stressed the importance of managing code clones using synchronous maintenance of code clones.
- Duala-Ekoko and Robillard [51] proposed managing code clones by notifying developers of modifications to clone regions. They developed a clone tracking system which works as the system evolves.
- Jablonski and Hou [88] concluded that clone awareness with visualization and consistent identifier renaming support help developers during debugging and modifications. The study highlights the importance of clone management.

2.9.2 Clone management – A cross cutting and an umbrella activity

Code clone management is an umbrella activity covering all aspects regarding clones. It is a superset including clone analysis, clone evolution, code clone taxonomies, code clone classification, etc. It also covers code clone visualization. We discussed here those studies which focused on clone management and clone visualization. Moreover, the relevant information is spread across many themes addressing research question 3 as shown in table 2.1.

Balazinska et al. [13] focused on restructuring and reengineering software after successful clone detection. Using six open source subject systems written in Java, method clones were manually analyzed for reengineering opportunities. Higo et al. [75] provided a technique to identify meaningful blocks in code clones that are easy to merge. They used *CCFinder* to detect code clones. The technique is helpful in reducing number of clone pairs detected in two open source Java systems. Kapser and Godfrey [111] investigated two large subject systems to classify code clones, viz. function clones and partial function clones. The authors suggested management of code clones by classification and filtering false positives. *CReN* [87] developed by Jablonski and Hou

provides programmers with identifier renaming support and tracking of clones in an integrated development environment. The authors extended their work in the form of *CnP*. *CnP* [82, 83] is intended to support and manage clones proactively as they are created and evolved. The tool was developed as Eclipse plug-in.

Tairas [221] proposed a technique to unify clone detection, analysis and refactoring. The study presented a method to improve clone maintenance by eliminating redundant code by identifying refactoring opportunities. Nguyen et al. proposed *Cleman* [178], a framework for comprehensive code clone group management in evolving software. They developed *Clever* [179], a clone aware software configuration management system which works with any AST based clone detection tool. It performs umbrella activities like clone detection, clone change management, clone consistency validating, clone merging, etc. Duala-Ekoko and Robillard developed *CloneTracker* [50] to track clones during the evolution of source code. It uses *SimScan* as clone detection tool. The authors used *CloneTracker* to produce clone region descriptors (CRD) [51] which are an abstract combination of lexical, syntactical and structural information. Clones are tracked using CRD which represents a clone region for different clone groups which are of interest to a developer. This approach goes beyond code based clone descriptors in integrated development environments like *CReN*, etc. *CloneBoard* [238] is a clone management tool which infers clone relations dynamically by monitoring clipboard activity. The tool is integrated in Eclipse to support copy pasted clones. Lee et al. [153] introduced a scalable and instant code clone search technique for use during software development. The technique works by extracting characteristic vectors from the source code. Then a multidimensional indexing tree structure R*tree is used. In the index, filtering and ranking is used to evaluate code clone detection queries in order. The authors also devised an approximate clone detection technique which is fast but less accurate. Both the algorithms gave sub second response time for processing a million lines of source code.

2.9.3 Clone Visualization

Code clone visualization is one of the most important areas of code clone management after successful code clone detection. With the increase in duplication in various software artifacts e.g. source code, proper representation of software clones has become a challenge. This domain covers presentation of duplicated code which helps in fast analysis of clone detection results.

Several clone detection tools report the presence of clones in form of starting and ending line number, file name, etc. One of the widely accepted formats for representation of clones is scatter plot. Tairas et al. [219] extended the *AspectJ Development Tool* visualizer to display the results of *CloneDR* in Eclipse framework. The main application of the tool is to display the cross cutting concerns in aspect oriented programs. Adar and Kim [1] developed *SoftGUES*S which supports exploration and visualization of code clones. Their system supported different views of analysis of clones over single and multiple versions, through analysis of graphs. Jiang and Hassan [96] developed a *Clone System Hierarchical Graph,* an interactive graph used to select nodes to highlight how the clones are scattered in a particular directory. They used clone mining to highlight clones at different levels of abstraction.

Most of the clone visualization tools like *Gemini* [77] read the output of *CCFinder*. These tools filter uninteresting code clones and navigate to clones having features the users are interested in. Zhang et al. [247] developed a standardized graphical representation as an Eclipse plug-in called *Clone Visualizer* to filter and visualize the output of *Clone Miner*. Fukushima et al. [59] used a code clone graph to visualize the output of *CCFinder*. Nodes of a code clone graph correspond to code clone sets and edges represent clone sets in the same file. The technique helps in representing diffused clones in which clone set clusters are located in files having different functionality. Tairas [222] presented a technique in which one clone instance displays the properties of all the clones in the clone group. The technique was integrated as an Eclipse plug-in called *CeDAR* (Clone Detection, Analysis and Refactoring). This representation help in refactoring as clone group representation displays the difference among clone instances.

2.9.4 Clone Management: A Systematic map

Clone detection tools are applied to detect clones after the software development is completed in a post-mortem approach of clone management. But even the leading clone detection tools report a large number of false positives. So recently, researchers and practitioners have tried to make clone detection an integrated part of development environment to increase the effectiveness of clone management. Developers need to be informed online as and when clones proliferate.

Clone management is a cross cutting topic touching different domains of software clones. We identified four key areas in this systematic literature review i.e. clone analysis,

clone evolution, impact of clones on software quality and clone visualization which touch clone management. The first of these three key areas has been discussed in detail in the context of research question 3 See table 2.1. The papers from these key areas were studied to identify relevant sub topics of clone management facet below:

1. **Code clone classification**

2. **Code clone refactoring**

3. **Code clone visualization**

4. **Code clone tracking**

5. **Code clone evolution**

We used the systematic mapping method to map clone management papers with clone detection method papers and clone detection tool papers. From the set of primary studies, we identified 49 relevant papers. Papers were classified based on these three different facets and the results are presented in the form of bubble plot as shown in Fig 2.4.

The bubbles show the number of publications identified for each clone management facet with clone detection method facet and clone detection tool facet. Total number of papers on either side of the map is not equal as many clone detection tools use more than one method for clone detection. Our bubble plot shows that the majority of research is in clone refactoring using the suffix tree and dynamic programming methods of clone detection. The metrics/ characteristic vectors method of clone detection is frequently used in different sub topics of clone management. We observed the use of clip board operations as clone detection method in clone tracking and visualization. This is due to the fact that clip board activity captures in the initial creation of a clone in an IDE. *CCFinder (X)* is the most frequently cited tool. It has been used for clone classification, clone refactoring and clone visualization as it is able to detect large number of clone candidates with high recall. The map suggests a lack of research in clone classification. Barring *CCFinder(X)*, *SimScan*, *CloneDR*, *Deckard*, *Simian*, and *CLAN*, the rest of the tools are only used in one or two papers.

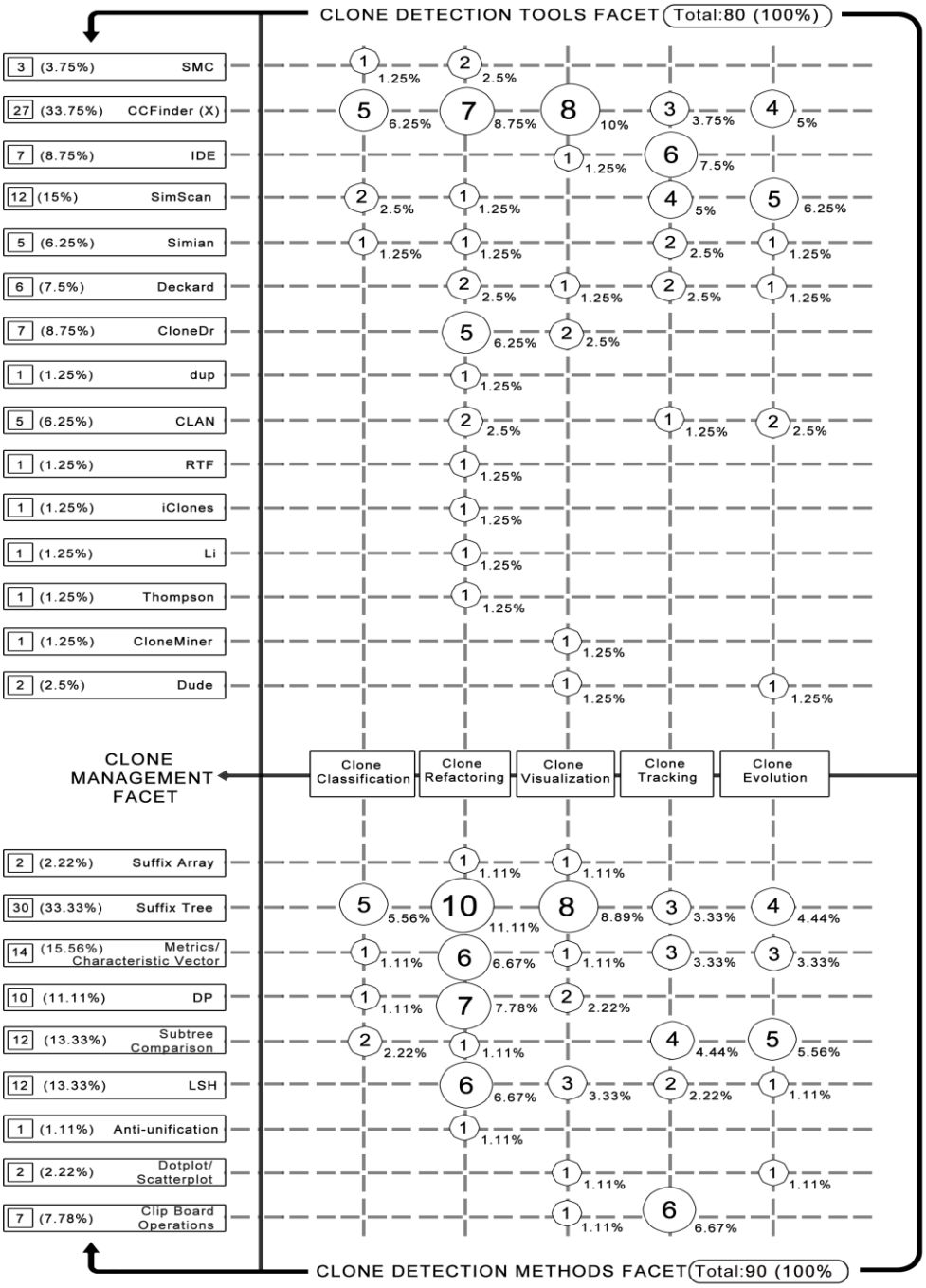


Figure 2.4 Clone management map

2.10 Subject Systems

We have observed that different subject systems are being used in clone detection research. We are hopeful that the following table may help researchers in choosing most commonly used subject system as benchmark for evaluation and empirical studies. We list 28 open source subject systems that were subjects of different studies. Table 2.10 lists all open source software systems and Table 2.11 lists commercial systems. We also list

approximate size in LOC, programming language of the system and usage count. The usage of the subject system according to our classification of literature and citations is also mentioned in the table.

Table 2.10 Open source subject systems

Sr. No.	Subject System	Size (LOC)	Language	#	Classification	Citation
1.	JDK	3200K	Java	10	Clone Detection	[35],[108],[153], [180],[183],[205], [233]
					Clone Analysis	[13],[91]
					Comparison and Evaluation	[205]
2.	Apache-httpd	343K	C	9	Clone Detection	[64],[123],[151], [154],[196],[197]
					Clone Analysis	[115],[139],[188]
3.	Apache-Ant	1.41M	Java	4	Clone Analysis	[19],[78],[224]
					Impact of Software Clones	[207]
4.	ArgoUML	1.76M	Java	10	Clone Evolution	[11]
					Clone Analysis	[145],[224]
					Impact of Software Clones	[22],[66],[67],[141], [142],[207]
					Clone Detection	[64]
5.	Linux	6.2M	C	10	Clone Detection	[17],[33],[70],[84], [108],[154],[196], [197]
					Clone Analysis	[96]
					Impact of Software Clones	[244]
6.	Weltab	11K	C	7	Clone Detection	[53],[127],[186], [196],[197],[220]
					Clone Analysis	[208]
7.	Netbeans-javadoc	19K	Java	7	Clone Detection	[16],[55],[80],[196], [197],[206]
					Comparison and Evaluation	[54]
8.	jEdit	157K	Java	5	Clone Detection	[26]
					Clone Analysis	[19],[224]
					Impact of Software Clones	[22],[142]
9.	Bison	16K	C	5	Clone Detection	[55],[130],[134], [136],[200]
10.	PostgreSQL	937K	C	6	Clone Detection	[51],[121],[134], [154],[196],[197]
11.	Snns	105K	C	5	Clone Detection	[55],[134],[186], [196],[197]
12.	FreeBSD	403M	C	3	Clone Detection	[108],[154]
					Clone Analysis	[160]

Sr. No.	Subject System	Size (LOC)	Language	#	Classification	Citation
13.	ANTLR	61K	Java	3	Clone Detection	[183]
					Clone Analysis	[13],[19]
14.	Eclipse-Ant	35K	Java	7	Comparison and Evaluation	[54],[196],[197]
					Clone Detection	[16],[55],[64],[80]
15.	Wget	17K	C	3	Clone Detection	[55],[196],[134]
16.	Cook	70K	C	3	Clone Detection	[53],[186],[196]
17.	Abyss	1500K	C	3	Clone Detection	[196],[220]
					Clone Analysis	[211]
18.	Tomcat	130k 167K	Java	3	Comparison and Analysis	[149]
					Clone Detection	[26],[148]
19.	SQuirreL	218K	Java	3	Clone Analysis	[224]
					Impact of Software Clones	[67],[141]
20.	GCC	1.2M	C	3	Clone Detection	[52],[64],[99]
21.	FileZilla	90K	C++	3	Impact of Software Clones	[141]
					Clone Evolution	[209]
					Impact of Software Clones	[103]
22.	Tcsh	45K	C	2	Clone Detection	[132]
					Clone Analysis	[133]
23.	Bash	40K	C	2	Clone Detection	[132]
					Clone Analysis	[133]
24.	CLIPS	34K	C	2	Clone Detection	[132]
					Clone Analysis	[133]
25.	Jabref	114K	Java	2	Clone Detection	[84],[104]
26.	DNSJava	25K	Java	2	Clone Evolution	[11]
					Clone Analysis	[19]
27.	Eclipse-jdtcore	148K	Java	8	Comparison and Evaluation	[54]
					Clone Detection	[16],[55],[80],[89] [196],[197]
					Impact of Software Clones	[141]
28.	j2sdk1.4.0-javaxswing	204K	Java	7	Comparison and Evaluation	[54]
					Clone Detection	[16],[55],[80],[196], [197]
					Clone Analysis	[19]

Table 2.11 Commercial subject systems

Sr. No.	Subject System	Size (LOC)	Language	#	Classification	Citation
1.	Government system	1 M	COBOL and PL/1	1	Clone Detection	[108]
2.	HagerROM (Würzburg University)	87K	Java	1	Clone Detection	[233]
3.	DISLOG Development Kit (DDK)	95K	PROLOG	1	Clone Detection	[233]
4.	SPARS-J	47K	C	1	Clone Analysis	[160]
5.	Commercial	460K	ABAP	1	Clone Detection	[84]
6.	Commercial CAD	1.6M	C	1	Clone Detection	[240]
7.	Graph-layout program (IBM)	11K	C	1	Clone Detection	[130]

Recently clones in Matlab/Simulink models are detected by [45],[47],[187]. Studies [46],[47] have used common models i.e. SIM,MUL,SEM,ECW available from Matlab central. The size of the model in blocks varied from 440–18000 blocks. Domann et al. [49] and Juergens et al. [103] applied clone detection to 11 and 28 requirement specifications respectively from a total of 2500 to 8667 pages.

We have omitted those open source subject systems from Table 2.10 which are only used in one study. Different versions of the same program are mentioned in one place and the size of latest version used in any study is included. For instance, different versions of Linux have been used by [17], [108], [154], [196] etc. These are written in the one place and the largest size among them is reported.

JDK has been used many times for clone detection and clone analysis. The Apache web server and Linux kernel and its different versions have been used extensively as subject systems to carry out the clone detection and analysis study. Similarly, the software system ArgoUML has been used in recent studies to investigate the impact of software clones.

Subject systems in C namely, weltab, snns, postgresql and in Java namely, netbeans-javadoc, eclipse ant, eclipse-jdtcore and j2sdk1.4.0-javaxswing used by Bellon's experiment [21] are frequently used by researchers. Most of the research is carried out using subject systems written in C and Java. This may be due to a lack of suitability of tool for other languages. The efficiency of existing tools should be measured using other

languages which the tools support. Research is predominantly been done on open source systems. Few commercial systems have been used. Clearly the use of open source systems is preferable from the viewpoint of repeatability of experiments and also allows tool comparisons to be easily extended to cover new or amended tools.

2.11 Achievements through Systematic Literature Review

We surveyed 213 articles out of a collection of 2039 and provided categorization and quantitative overview. Unlike previous surveys, we put an emphasis on clone management, model clones and semantic clones and classified the literature from different key areas. Existing surveys/ technical reports by Roy and Cordy [191] and Koschke [135] consider research findings till 2007. These surveys filled the initial void for useful text for budding researchers in this domain. The work by Roy et al. [195] focused on clone detection tools and techniques up to 2009. Pate et al. [182] presented systematic review of existing literature on clone evolution. The authors framed three research questions to investigate what methods have been used to study clone evolution, to study cloning patters and to discover the presence of consistent/ inconsistent changes to clones during evolution. The authors identified 30 primary studies regarding clone evolution. Our focus is broader than the earlier surveys and includes the latest research work related to software clones using the systematic literature review guidelines of Kitchenham and Charters [125]. In addition to clone detection tools and methods, we have addressed other issues related to software clone research such as clone analysis, clone evolution, impact of clones on software quality. We used a systematic method to develop a clone management map which identifies how clone management papers overlap with clone detection method papers and clone detection tool papers. We explored the model based and semantic clones in detail and compared the state of the art techniques. Moreover, major breakthroughs in model clone detection happened after 2007. We presented all the studies in different sections in chronological order which makes it easy to identify the latest research carried out after 2007 as done in earlier surveys.

This section discusses the principal findings of our systematic review, strengths and weaknesses of the evidence. We begin with the discussion of key sub areas followed by clone management. Implications for researchers and practitioners are summarized. Finally limitations of the review are given.

2.11.1 Key sub areas

We divided the literature into 6 different key areas realizing the importance from research perspective. We noticed that some of the areas are inter-related.

It is not an easy task to model clone evolution under a number of versions. The prediction accuracy of the approach depends upon parameters and their variation. Future research should consider using sound mathematical modeling approaches. Many clone genealogies are alive and long lived and clones are easy to manage in smaller systems as compared to large systems. The area is still open for research to study clone genealogies using state-of-the-art clone detection tools in different sample subject systems.

A large numbers of studies confirm the harmfulness of cloning in software systems. Less research was found in tracing useful patterns of cloning which help the programmer. We observed limited number of studies investigating cloning patters in different programming paradigms. More studies should be carried out to investigate the types of clones and their characteristics like persistence over time in different programming methodologies. In the initial years, we found few studies attempting to calculate the impact of code clones on maintenance cost. We noticed increase in the number of studies undertaken to compare the behavior of cloned and non-cloned code and their impact on system quality during last two years. Most of these studies use version control information. However, these systems detect minor changes like white spaces too, which should be ignored from the clones point of view. We found contradictions among the papers and the area is still open, since the number of subject systems is too low to arrive at any general conclusion. Future research lies in carrying out the same study with large number of comparison parameters, clone detection tools and subject systems. Recently, many studies presented findings that clones in general do not have adverse effect on quality. At the same time, we observed some contradicting studies. Such studies need to be extended using external quality attributes and a large subject base. The nature of the subject system and programmers' behavior has a profound effect on these studies. In one study, varied results were noted for the ArgoUML and Ant subject systems. We need more studies to accurately calculate the increase in maintenance cost of software due to presence of different types of clones.

We found two studies that checked the presence of clones in software requirement specifications. There is a need to conduct clone detection studies to investigate

redundancy throughout all documents of software development life cycle. Such repetitions when removed will lower the maintenance cost of the software in earlier phases of software development life cycle. We found only one study which applied clone detection to trace open source licensing violations. Such studies will be helpful to distinguish between reuse based and accidently produced clones

2.11.2 Clone Management – A cross cutting topic

Different clone management tools should be evaluated depending upon use cases and future tools can be developed catering to respective use cases.

The systematic map in Fig 2.4 helps in identifying which sub topics of clone management have been emphasized, which areas require further research, which clone detection tools and methods have high usage in clone management. In the map, we did not find any papers involving more than one tool in any aspect of clone management. This may indicate a lack of interoperability between different tools for clone management. We find a lack of work in clone classification. To assist the software maintainer, it is important to classify clones as "clones to be retained" and "clones to be removed". We found large number of studies regarding maintenance of code clones by identifying refactoring opportunities. There is dearth of studies validating the analytical results of clone detection, clone analysis, clone management, etc. with developers intent and behavior.

2.11.3 Implications for research and practice

The systematic review has implications for researchers who are looking for new concepts in the field of software clones, and for practitioners working in software companies who want to apply clone detection for software development.

As the research community is still arguing on the exact definition of the term clone, it is imperative to devise automatic oracles for all types of clones. Studies confirm that there is disagreement between human experts as to whether the candidate code is or is not a clone. It is a difficult and time-consuming task to manually classify the candidates as clones or not. Thus, we believe that experts from industry and academia related to diverse domains of clone detection should come together to create a verified reference corpus of clone candidates in standard subject systems. The study should be carried out differently for each type of clone and depending upon use case. Such benchmark suites would make

the results of empirical comparison consistent and reliable for use in research and industry.

For researcher and practitioners, a number of avenues are open. Clone management has emerged as a challenging area. Software developers in industry deal with large amounts of data. So clone management tools should be scalable and integrated in into development environments, to help programmers understand the behavior of cloning patterns. A comprehensive industrial strength clone management tool having integrated detection and developer friendly visualization of clones would help the developers observe clones as and when they proliferate during development.

The application of clone detection tool depends upon situations and objectives. There are different circumstances where clone detection is essential. Some of the areas are: finding cross cutting code [30], plagiarism detection [28], software product lines [204], clones in web sites [147], origin analysis [162], quality assessment [175] and detecting licensing violations [62]. Though these areas are independent research fields, yet these areas and clone detection can get benefited from each other. Usually cross cutting code is scattered in different implementations of the program. These implementations tend to be functionally similar, so semantic clone detection technique is helpful in finding cross cutting code. In plagiarism detection, the code is copied and disguised intentionally. By representing the code in an abstract representation like PDG, existing code clone detection tools may be customised to detect hidden changes in the code. Clone detection helps in detecting shared and common set of features in software product lines. Existing systems can be reengineered to obtain reusable assets with the help of clone detection. Origin analysis is the study of detecting the location of changes to the system from one version to the next. Clone detection may help in origin analysis with detection of similar function/class/file across versions. Code clone across systems may directly lead to licensing violations and copyright infringements. So clone detection technique can be applied to detect these violations.

2.11.4 Limitations of Systematic Literature Review

The main limitation of this study is multitude of meanings associated with the keyword ‘clone’. We tried to be extremely cautious in data extraction. Strings like code duplication, redundancy were manually searched in databases to increase the number of research articles in our study. However, manual searches may miss relevant articles.

Data extraction was carried independently by the researchers. But only one author reviewed the discarded articles. As far as the classification of studies is concerned, there were disagreements among researchers. Each researcher classified all papers individually before comparing the results. In cases where there was disagreement, the issue was discussed until consensus was reached. Thus the papers were classified in several different categories.

2.12 Summary

In the chapter, we presented the results of systematic literature review. We identified 213 studies from literature, of which 100 were found to be research studies of software clone detection. We have presented the results in different dimensions like classification of clone research, code clone management as cross cutting domain, types of clones, clone detection tools, clone detection approaches, internal representations, subject systems, semantic clones and model clones.

We noticed a great variation in the definition of the term “clones”. There are recent studies that show that clones can be often used as principled reengineering techniques, and can be beneficial in many ways. Also, it is not easy to refactor all the clones due to cost/ risk associated with refactoring. So it is suggested that instead of removing clones, we should have proper clone management facilities. In order to advance the state of the art in clone management, one needs to know the advances in clone research itself. We have attempted to do this by finding the relevant literature and summarizing it in the form of systematic map. So the results of this chapter are helpful in finding the research gaps in the area of software cloning in general and clone detection in particular.

Chapter 3

Investigation of Parameters for Semantic and Model Clone Detection

Semantic clone detection and model based clone detection techniques are challenging upcoming areas. Available techniques are explored in detail to identify the parameters which acts as the basis for our work in developing the proposed techniques of clone detection. Advantages and shortcomings of all the existing techniques are studied. Section 3.1 explores various semantic clone detection approaches. Various model clone detection approaches are detailed in sec 3.2. In section 3.3, various parameters affecting software clone detection are identified. Finally, in section 3.4, we list the motivation behind model based clone detection.

3.1 Semantic Clone Detection

Two program fragments differing in their concrete syntax may be semantically very close. Detecting semantic equivalence is very difficult. It needs deep semantic analysis. There are some studies which tried to detect semantic clones. They are mostly approximations to type-4 clones. Table 3.1 compares and details different semantic clone detection techniques.

Table 3.1 Semantic clone detection and comparative analysis

Author & Tool Name	**Normalizations/ Transformations**	**Source Code Representation**	**Clone Matching Technique**	**Advantages**	**Disadvantages**
Jens Krinke [137] ***Duplix***	PDG	Fine grained PDG	n-length patch matching (maximal similar subgraphs)	High precision and recall	Needs a PDG generator for different language, works for C language
Komondoor & Horwitz [130] ***PDG-DUP***	CodeSurfer to PDG	PDG	PDG Subgraph matching using program slicing	Mechanical refactoring can lead to procedure extraction	Needs a PDG generator, very slow for large code bases
Choi et al. [36]	programs to partite sets and functions to vertices	Birthmarks	maximum weighted bipartite matching	Efficient, highly resilient	Needs deobfuscation methods against attacks

Author & Tool Name	Normalizations/ Transformations	Source Code Representation	Clone Matching Technique	Advantages	Disadvantages
Marcus & Maletic [172]	Comment removal and token regularization	Text	Vector representation using LSI	Finds high level structural clones	Highly dependent on comments, low precision
Gabel et al. [60] ***Enhanced Deckard***	CodeSurfer to PDG to AST	PDG	Characteristic vectors in Euclidean Space	Highly scalable	Needs a PDG generator, slow for large code bases
Jiang & Su [97] ***EqMiner***	program text to intermediate language	C Intermediate Language	Automated Random Testing	Scalable	Works for C programs only
Kim et al. [123] ***MeCC***	Semantic based static program analysis tool	Abstract Memory States	Abstract Memory state comparison	Precise, can be used to identify bugs, inconsistencies, plagiarism	More false positives, semantic based static analyzer takes lot of time
Philipp Schugerl [205] ***DL_Clone***	AST to description logic	Description logic	Semantic web reasoner	Scalable, can be parallelized	Fails to detect smaller clones and clones across methods

PDG is a directed attributed graph representation for several program analyses and transformations. The edges in the PDG represent the control dependencies and flow dependencies. The clone detection lies in finding isomorphic subgraphs of PDG that represent clones. There are techniques by Krinke [137], Komondoor and Horwitz [130], Gabel et al. [60] who transformed the source code to PDG for clone detection. Krinke [137] used k-length patch matching to detect maximal induced subgraphs. The technique worked well with reasonable precision and recall on sample software systems. Komondoor and Horwitz [130] used program slicing to detect isomorphic subgraphs in the PDG. The technique uses backward and forward slicing and able to detect good clone candidates for procedure extraction. It can detect non-contiguous clones. Gabel et al. [60] presented a scalable technique to detect semantic clones from the PDG representation of the source code. The key element of the algorithm is to map the NP-complete graph isomorphism problem to tree similarity. The tree similarity is based upon comparing characteristic vectors. Upon empirical evaluation, the tool has good execution times on large code bases.

Marcus and Maletic [172] applied latent semantic indexing (LSI) on the textual representation of source code to identify semantic similarities across functions/ files/ programs. LSI is vector based statistical method to represent meanings of comments and identifiers of source code. They tried to detect similar high level concept clones e.g. abstract data types.

Software birthmarks have been used successfully to detect copied programs and software theft. Choi et al. [36] used a set of API calls to detect similar programs. The similarity technique depended upon maximum weighted bipartite matching. In this way, the method is useful in detecting semantic equivalences duplications in case of software thefts. However, the technique is vulnerable to deobfuscation attacks.

There is only one study by Jiang and Su [97] which identified functionally equivalent code fragments of arbitrary size depending on the input – output behavior of a piece of code. They detected two pieces of code that always produce same output on random inputs although they are syntactically different. They defined functional equivalence as a special case of semantic equivalence. The results were validated by applying random tests. The tool was scalable and able to work on million lines of code finding that 58% of functionally equivalent code was syntactically different. The technique worked for C programs only.

Kim et al. [123] proposed *MeCC*, a semantic clone detector based on a path-sensitive semantic-based static analyzer. The analyzer was used to estimate the memory states at each procedure's exit point; then memory states were compared to determine clones. The authors compared their findings with *CCFinder* and *Deckard*. *MeCC* is able to detect larger number of procedural clones, to trace inconsistencies, identify refactoring candidates and understand software evolution related to semantic clones.

Philipp Schugerl [205] presented a novel technique to detect global clones. An abstract syntax tree representation of the source code is normalized in the form of description logic. Then a semantic web reasoner is applied to trace similar source code based on control-blocks and used data types. The author compared the technique with state of the art clone detection tools but only on Java source code. The technique is highly scalable with the use of semantic web reasoner for match detection.

Several authors suggested areas for further research. Marcus and Maletic [172]

proposed the combination of multiple detection algorithms in future. A number of hybrid clone detection algorithms were developed as a result. We noticed that PDG based approaches of detecting semantic clones are quite slow. Future work [63] lies in developing the framework to help in fast generation of PDG from source code. So, approximate solutions of mapping PDG to trees as by Gabel et al. [60] by applying tree similarity technique are more scalable. Choi et al. [36] proposed both extending birthmarks with more information and making technique robust against attacks. Philip Schugerl [205]'s technique can be parallelized using a cluster of computers. Empirical evaluation across more subject systems would help identify future extensions of the tool which currently does not detect small clones and clones across method. Jiang and Su [97] proposed exploring future research on functionality-equivalent code refactoring and reuse. A general method for different programming languages should be developed to detect functionally equivalent but syntactically different code fragments. *MeCC* [123] can be extended by adapting a static analyzer to collect memory states for any arbitrary code blocks to make clone detection possible at finer granularity.

3.2 Model based Clone Detection

With the rise in abstraction, model driven development has turned to be an emerging area. Large models are developed using UML, Matlab/Simulink, domain specific modeling languages, etc. The presence of duplicated sub structures in different types of models cannot be ruled out. Thus, detecting clones in models is an emerging area. But, model based clone detection techniques are still in their infancy. So this classification includes studies and tools related to detection of duplication in different diagrams and models as summarized in Table 3.2.

Liu et al. [158] detected duplications in sequence diagrams by converting the 2-dimensional sequence diagram to a 1-dimensional array. Then, the 1-dimensional array is used to build suffix tree. Common prefixes are identified from the suffix tree in the form of reusable sequence diagram as refactoring candidates. The study confirmed the presence of 14% duplication in sequence diagrams of sample industrial projects.

The automatic detection of clones in models leads to identification of potential domain specific library elements [45]. Deissenboeck et al. [45] used *ConQAT* [100] as an integrated framework to detect clones in Matlab/Simulink models especially in

automotive domain. The tool *CloneDetective*, which is part of the *ConQAT* framework, works by representing the model as a normalised multi-graph where labels are assigned to relevant blocks. Similarity between blocks is checked by a heuristic which performs a depth first search based looking for matched pairs. After detection, clones are clustered based on set of nodes using union function.

Table 3.2 Model based clone detection and findings

	Liu et al. [158]	**Pham et al. [187]**	**Deissenboeck et al. [45]**	**Herald Storrle [216]**	**Hummel et al. [85]**
Preprocessing/ Normalization	Two dimensional sequence diagrams into one dimensional array	Transformation of models to graphs, assigning labels to relevant blocks	Transformation of models to graphs, assigning labels to relevant blocks	XMI files from UML domain models	Transformation of models to graphs, assigning labels to relevant blocks
Source Representation	One dimensional array	Sparse, labeled directed graph	Labeled multigraph	Prolog code	Directed, labeled multigraph
Clone Matching Technique	Suffix Tree	Canonical matching, vector based approach	Maximum Weighted Bipartite Matching	Model matching	Canonical matching, clone index based hashing
Advantages	High precision and recall	algorithm is able to detect model fragments with modifications, incremental	Scalable	Supports refactoring	Incremental, distributed , fast detection time
Disadvantages	Works only for sequence diagrams	Lower precision	Large number of false positives	Complex Java Implementation	Infeasible for large subgraphs
Application Area	Sequence Diagrams	Matlab/Simulink models	Matlab/ Simulink/ Targetlink models	UML Domain models	Matlab/ Simulink models
Model Clone Granularity	Extractable fragment of a sequence diagram	Number of blocks	Number of blocks	Sub models	Sub models
Tools	*Duplication Detector*	*ModelCD*	*CloneDetective*	MQ_{lone}	Integrated in *ConQAT*

Pham et al. [187] presented two algorithms namely *escan* and *ascan* to detect clones in models. These were incorporated in the tool *ModelCD*. Firstly, the model was pre-processed to be represented as a parsed, labeled directed graph. *escan* was used to detect exact matching using an advanced graph matching technique called canonical labeling and *ascan* was used to detect approximate matching by counting vector of sequence of nodes and edges' labels. The technique is incremental in the way it generates candidate cloned sub graphs. Their tool *ModelCD* was compared with *CloneDetective* which is included in the *ConQAT* framework [45]. For the same clone granularity and subject systems, both the tools were compared based on 4 parameters: Precision, completeness, scalability, incrementality. *ModelCD* performed better. In a subsequent paper, Deissenboeck et al. [47] discussed the presence of a large number of false positives in Matlab/Simulink models, pointing out that it is important to identify relevant clones. Model clones suffer from the problem of scalability, clone inspection and relevance. Their study provided useful insights in addressing these problems in real time industrial context. The authors proposed reducing the size of models to make clone detection speedier. Firstly, removal of obvious cloned sub-systems is carried out. After which, as proposed by Pham et al. [187], all nodes with high degree are removed. After the detection process is complete, the algorithm tries to connect smaller nodes that are connected to each other over high degree nodes. Deissenboeck et al. [47] then compared their enhanced version of *ConQAT* and *escan* showing ConQAT has a faster execution times than *escan*.

Störrle [216] pioneered the detection of clones in UML domain models. The technique was based on model querying. Using any of the UML case tools, XMI files are generated from UML domain models. These files are transformed into Prolog files. A model is input in the query and using model matching, the output is generated. We observed that the tool is still to be verified for heterogeneous subject systems. Hummel et al. [85] introduced an incremental algorithm for model clone detection. A Matlab/Simulink model is pre-processed by flattening the model into a directed multigraph. Then, relevant edges and blocks are labeled. A clone index is created for all subgraphs of same size. Canonical label of all subgraphs in the clone index is calculated and similar labels are hashed. Clone retrieval and index update are integrated for fast retrieval. It has not been verified on large models.

It is still to be verified whether canonical matching and vector based approach can be applied on other graph based models like UML models. We observed one study by Deissenboeck et al. [47] highlighting the practical issues to be resolved in model clone detection. The study pointed out that ranking of clones may help in improving scalability and relevance of model clones. A comprehensive model clone detection tool for UML models and other data flow languages is missing. Different forms of models have individual features which need to be exploited for clone detection [177]. We noticed vagueness in the definition of model clones which hinders understanding of the topic area [69].

3.3 Parameters for software semantic clone detection

From the comparative analysis of various semantic clone detection approaches, we concluded that the large number of techniques transform the source code to PDG. Krinke [137], Komondoor and Horwitz [130], Gabel et al. [60] are some of these techniques. We observed that these techniques are not scalable to run for large code bases primarily due to slow generation of clone pairs. After the transformation to PDG, source code is represented as a graph. Then, various researchers tend to apply graph mining techniques upon that to mine patterns. The application of subgraph isomorphism to this graphical representation of source code makes the technique computationally tough as subgraph isomorphism is NP-complete. Scalability of the PDG based technique is another challenge.

Due to the difficulty and cost associated in detection of semantic clones, we tend to orient towards methods of clone detection which are more close to implementation i.e. the use of object oriented models. Thus, we carried out a comprehensive analysis of sample forward designed and reverse engineered models. The main objective of this analysis is to identify key parameters for the proposed model clone detection technique. UML class models are created by reverse engineering using standard modeling tools. The UML class models are exported to intermediate representation i.e. *.mdl* and *.xml*. XMI file of sample open source subject system *eclipse-ant* is shown in Fig. 3.1. From the intermediate representation, we were able to identify key parameters which affect software clone detection and acts as the basis for our model clone detection approach.

We used following UML tools for comprehensive analysis of UML class models:

- Magic Draw
- Altova Spy
- MS Visual Studio
- Visual Paradigm for UML
- ConQAT
- Eclipse and various plug-ins

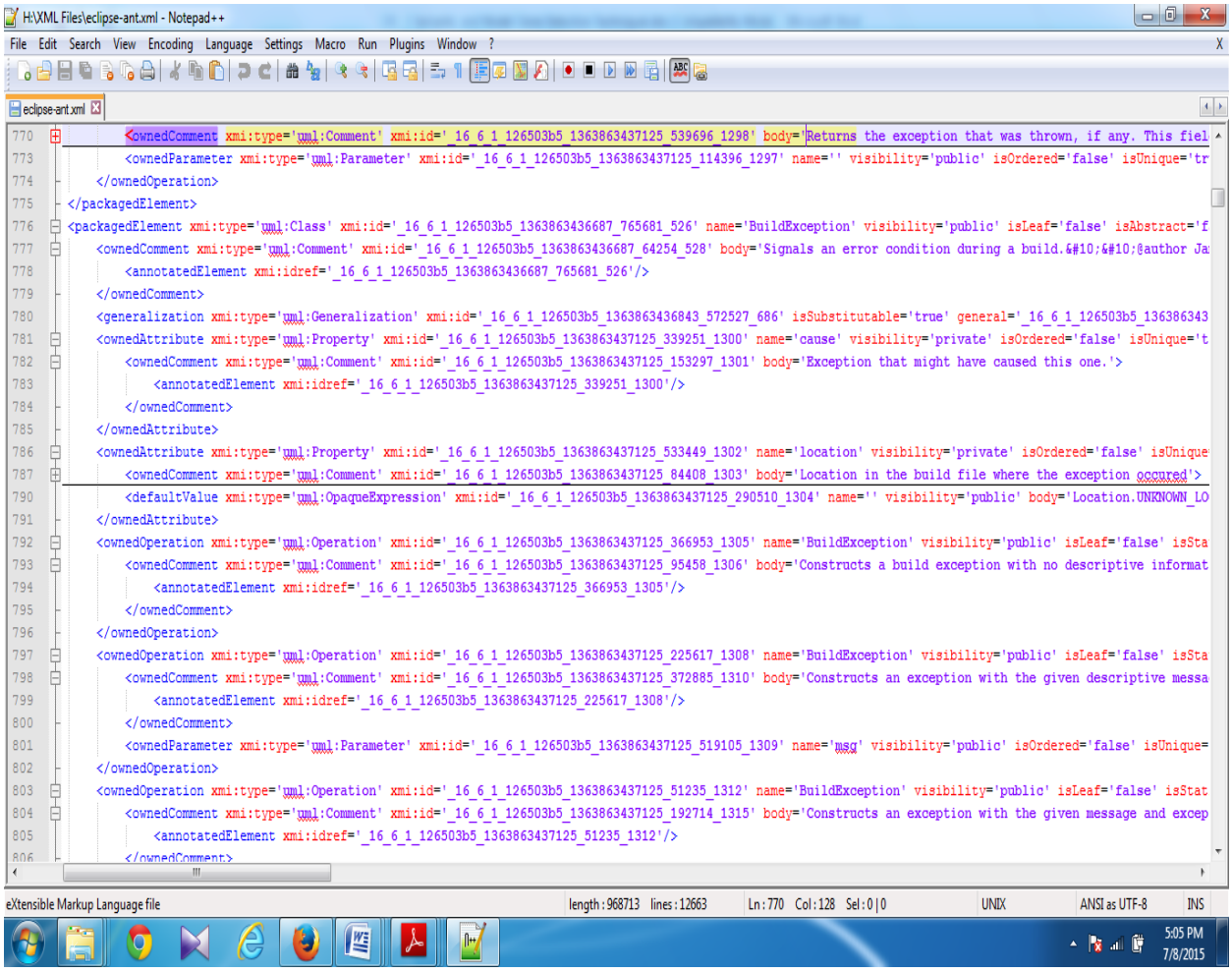


Fig. 3.1 XMI file for *eclipse-ant*

Our analysis of intermediate files of object oriented models revealed following parameters which affect software semantic clone detection:

- Name of the class
- Name of the fields, methods
- Return type of methods
- Number of arguments in methods

- Data type of arguments in methods
- Inheritance
- Association
- Dependency
- Realization

Thus our technique of clone detection for object oriented models is based on extracting the above said parameters from the models.

By studying the existing techniques of semantic and model clone detection, we identified following parameters to evaluate the effectiveness of software clone detection technique:

- Scalability
- Clone Coverage
- Frequency
- Precision
- Recall
- F-Measure

Above attributes are defined in chapter 4 and chapter 5 at the time of evaluation of proposed techniques.

3.4 Motivations behind Model Clone Detection

Models are being used as a way to express design. We are motivated by following reasons:

- Increasing size and complexity of software systems promotes the use of models for comprehensibility and better understanding.

- Due to the increase in use of Unified Modeling Language and Model Drive Architecture, models are replacing code as core artifacts.

- Models are independent of programming languages. Various software modeling tools permit automatic code generation in different languages.

- Models give architectural details which help in identifying parameters which are close to semantics [169].

- Model based system development simplifies design and offers better compatibility between subsystems, thereby promoting communication between individuals and teams.

3.5 Summary

In this chapter, we have discussed various semantic clone detection and model based clone detection techniques. These techniques are compared based upon different characteristics. Advantages and shortcomings of all the existing techniques are mentioned. We have mentioned the process of analysing the forward designed and reverse engineered models to identify the parameters which act as the basis for our work. Finally, we have listed various factors which acted as the motivation for our work to devise model clone detection techniques.

Chapter 4

Model based Clone Detection

4.1 Introduction and Motivation

Now-a-days modeling is playing a significant role in industries of different domains like automotive domain, web based applications and other complex systems [45]. Model driven software development using UML improves the quality of product delivery and analysis of the product [143]. UML models bring in extra advantages of automatic code generation, early verification and validation [143]. Since modeling has been accepted as best practice, UML class models may also be analyzed for the presence of clones.

Clones in UML class models refer to the presence of duplicate parts i.e. set of identical attributes, operations or both across different classes in the model. While its counterpart, code clone detection have been an active area of research for long. Evidences suggest that the presence of code clones may lead to bug propagation and maintenance problems [101]. For example, if one fragment of code is changed, that change has to be carried out in all duplicate instances which may lead to missing or erroneous changes [94]. As UML models provide an abstract view of the system thus detecting clones in models is equally important because similar challenges exist in the case of UML models [216], too.

Key Contributions

Primarily, the objective of the proposed technique is to detect model clones and present our classification and findings as observations of the characteristics of model clones. Our use case is to detect and analyze clones in forward designed UML class diagrams and open source reverse engineered class models. The detected exact and meaningful clones help in understanding the characteristics of UML models with regard to cloning. The major contributions of our approach are:

- Detection of model clones in UML class diagrams at different levels of granularity i.e. single attribute/operation, set of attributes/operations and recurring classes with their members.

- Detection and classification of model clones as:
    - Type-1 : model clones due to standard modeling/coding practice
    - Type-2 : model clones by purpose
    - Type-3 : model clones due to design practices

  Since UML modeling has got inherent object oriented features [232], thus our classification of model clones is inspired from these object oriented characteristics of UML class model where we view the model as a collection of logical entities defining data and behavior [198].

- Carry out the empirical evaluation on reverse engineered open source systems. Moreover, we are also considering forwarded designed models for evaluation to capture the essence of model driven development.

In this chapter, the background of our work is mentioned in section 4.2. Some elaborative examples to understand our classification of model clones is present in section 4.3 Section 4.4 explain our approach of model clone detection. The empirical evaluation and results of the tool are included in section 4.5. Discussion of the proposed work and threats to validity are mentioned in section 4.6. A brief survey related to work in model clone detection and comparison with existing work is done in section 4.7. Finally, Section 4.8 concludes the work.

4.2 Background

4.2.1 Model Clone Detection

To know the current state of the art in model clone detection, we came across techniques in the literature to detect clones in Matlab/Simulink models [6,45,47,85,187]. Most of these approaches are graph based and clone detection is carried out by applying graph matching techniques. But in case of UML models significant differences emerge after we transform a model into a graph. Störrle [216] has stated this difference in his study that "*UML models are not densely connected graphs of lightweight nodes, but rather loosely connected graphs of heavy nodes*". Therefore, a tree based approach is beneficial in exploring the heavy nodes of UML model i.e. classes, its members comprising of attributes and operations and relationships among classes. In our approach, we construct a labeled ranked tree from UML class model. The detail of our approach of model clone detection is present in section 4.3.

In a recent study by Saez et al. [57], a controlled experiment is carried out to compare the effectiveness of forward designed and reverse engineered models. The results of the study promoted the use of modeling as forward design models are easier to understand and maintain. Moving in the same direction [9], we evaluated our tool on forward designed and reversed engineered open-source UML class models.

4.2.2 Reasons for model clones

Here we mention some of the reasons that might lead to clones in UML models in brief.

- Accidental Cloning
- Lack of restructuring and programmers' limitation
- Reuse through copy and paste
- Language limitation

4.2.3 Definitions

We skip definitions of code clones, types and detection techniques in this chapter. The basic definitions of code clones are given in chapter 2. Definitions necessary for evaluation and understanding of the proposed work are:

Definition 1 (Model Element)
A model element refers to a class, an attribute declaration i.e. data type and name of the attribute or a method signature comprising of return type, name of the method and data type of parameters in a UML class model. It is abbreviated as *ME* throughout the paper. *#ME* refers to total number of model elements extracted from the XMI [239] file of the UML class model.

Definition 2 (Model Clone)
A model clone is a pair (A, B) where A & B are identical sets of model elements present in the model. A set of model elements may contain a single model element or a group of model elements.

Definition 3 (Frequency)
It is the total number of occurrences of a particular model clone across the model. It is calculated for all the model clones.

Definition 4 (Clone Cluster)

A group of model elements present in a model clones is referred to as a clone cluster.

It is represented as triplet (*uid, size, C*) where

uid is the unique identifier of the cluster

size is the total number of model elements in the group

C is the frequency of clone cluster

This definition is purposely included for classifying model clones as type-2 and type-3 as discussed in detail in section 4.4, where a detailed evaluation is given for different subject systems.

Definition 5 (Coverage)

It defines the number of duplicate model elements vs. total model elements (*#ME*). In other words, it determines the probability that a random model element is part of a clone. E.g. in next section in Fig. 4.1 on library management system, there are 34 model elements. Out of these, 9 are cloned *ME*. So the coverage % is 26.4%. It specifies the extent of duplication across classes in a UML class diagram.

Clone coverage has been successfully used in evolutionary analysis of software systems like Linux kernel [159]. By the same token, we believe that clone coverage be applied in the evolutionary study of model clones as well. Göde et al. [68] has mentioned clone coverage as an important cloning metric. The study highlight minimum clone length, normalization, structure and design of the subject system as key factors affecting clone coverage. In our paper, we have taken minimum clone size to be 1, a single model element and it is done uniformly for all the subject systems to make the results usable. We understand that when the minimum clone length is increased, clone coverage may decrease.

4.3 Model clone detection by example

The objective of our technique is to automatically detect exact and meaningful clones. Thus we propose following categories with explanation using an example.

Consider the sample UML class model of a library management system shown in Fig. 4.1. From the domain perspective, this model appears to be simple; however in the context of model clone detection, it will serve our purposes well.

Some fields/operations tend to repeat a lot in UML class model and in code. As shown in Fig. 4.1 of Library Management System, we notice that field *id: Integer* is present in most of the classes. It is a modeling practice to uniquely indentify an instance of an entity. We named such clones to be **type-1-Model clones due to standard modeling/coding practice**. We noticed such repetitions in reverse engineered class diagrams, too. One of them is *serialVersionUID*, which is used as a version control in Serializable class. The fact is that the serialization runtime associates with each serializable class a version number which is used during deserialization to verify the sender and receiver. If *serialVersionUID* is not explicitly declared by a serializable class, then the runtime will calculate a default *serialVersionUID* for that class.

Another example is *readObject* and *writeObject* which are used in Java's serialization to read and write byte stream in physical location. Most of these technical details surface in reverse engineered models and are unlikely to appear in realistic models. It may seem irrelevant in the first observance, but we intend to bring them into notice as well. Type-1 clones most of the time consists of a single attribute/operation. Thus the minimum clone length is set to 1 to identify such repeating field/method. Surely, presence of these fields in classes suggests application of programming/modeling practice.

Fig. 4.1 also shows a generalization relationship. The user generalization is shown as *Student* class and *Faculty* class to be the subclasses of *User* class. Both the subclasses override an operation named *details(): String* from the parent class. The *details(): String* operation must be implemented for each kind of user. This model clone is reported to occur in parent and all the subclasses because of the nature of the relationship. In other examples, there are several reasons why one may wish to override a feature. These types of clones are categorized as **Type-2-Model clones by purpose.** In nutshell, overriding is an important and useful language feature in object oriented programming.

Interfaces are an integral part of UML class diagrams. An interface [199] is a class like construct which shows a collection of operations that specify a service to a class. In a UML class model, realization relationship is used to show an abstraction and its

implementation. The figure shows an interface *catalog* with a number of abstract methods. *Catalog* aims to specify the behavior of classes *Manage* and *Search* using realization relationship. Both these classes implement different abstract methods specified in the interface. A number of programming languages support the concept of interfaces. E.g. In Java programming language *interface* keyword is used to specify an interface. This repetition of abstract operations in the interface and its implemented classes is also categorized as **Type-2-Model clones by purpose.** In the same diagram, we notice repetition of a cluster of size 3 consisting of attributes *id: Integer, name: String, address: String* in two classes namely *User* and *Publisher*. This type of repetition in different classes as reported by the proposed technique is categorized as **Type-3-Model clones due to design practices**.

We use the class diagram of Courier Management System shown in Fig. 4.2 as a running example. Among all the classes viz. *CompanyOwner*, *Customer*, *Franchisee* and *Employee*, we see repetition of group of fields and methods. In the figure, there is a large cluster of size 5 i.e. *address: String*, c*ompensation(): Integer*, *name: String*, *phNo: Integer*, *updateDetails(): void* which is present in 2 classes namely, *CompanyOwner* and *Franchisee.* The second cluster of size 4 is *id: Integer, name: String, address: String, phNo: Integer* which is present in 3 classes namely, *CompanyOwner*, *Customer* and *Employee*. The last cluster of size 3, viz. *name: String*, *address: String*, *and ph No: Integer* is present in all the classes. Table 1.1 shows all these clusters together with type of clone. The algorithms to find the clone clusters together with their size are mentioned later in this chapter in section 4.4. The presence of these clusters may be the result of unfinished design or due to any other possible reason. Thus, these clones may subject to further improvement in the design w. r. t. maintainability and extensibility.

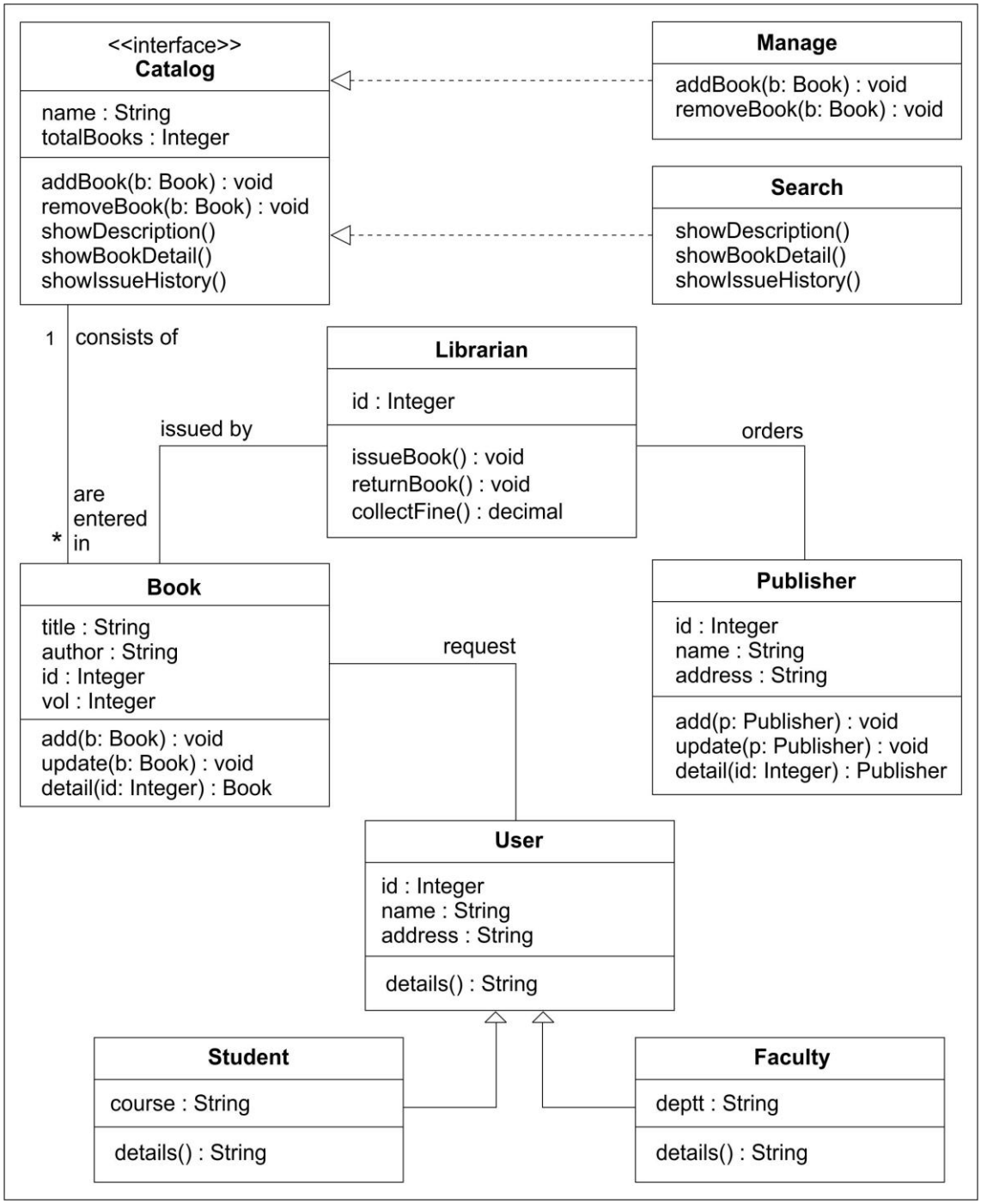


Fig. 4.1 Class diagram for Library Management System

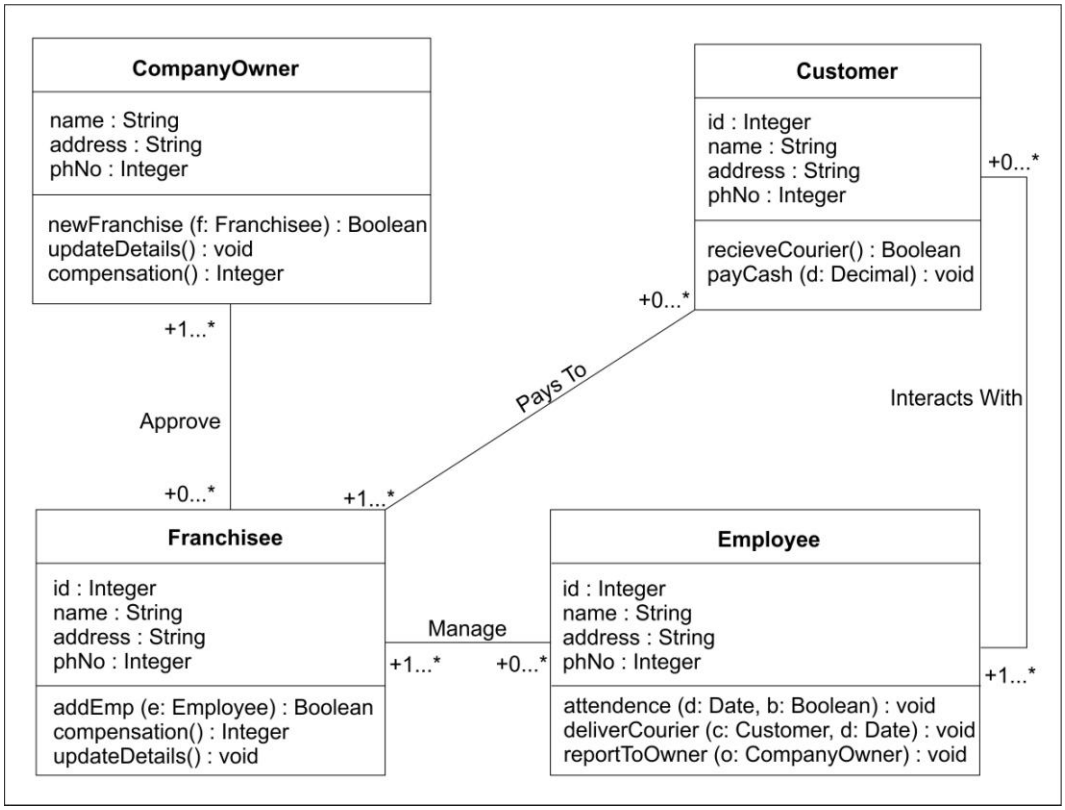


Fig. 4.2 A class diagram for courier management system

Table 4.1 Analysis of courier management system

Number of different model elements		Total = 17; Classes= 4; Methods= 9; Attributes= 4			
Sr. No.	**Name of attribute/method**	**Freq**	**Classes**		
1.	*compensation(): Integer, updateDetail(): void*	2	CompanyOwner, Franchisee		
2.	*id: Integer*	3	Customer, Employee, Franchisee		
3.	*Address: String, name: String, phNo: Integer*	4	CompanyOwner, Customer, Employee, Franchisee		
Coverage %		35% (6/17)			
Clone Cluster		UID	Size	C (Classes)	Type of clone
		1	3	4	Type-3
		2	4	3	Type-3
		3	5	2	Type-3

Table 4.1 shows the results after applying the proposed technique on the UML class model shown in fig. 4.2. In total we get 17 model elements comprising of 4 classes, 4 attributes and 9 methods. The list of model elements with their frequency and list of classes in which they appear are shown in table 4.1. The table lists the coverage percentage of the model. This is the number of cloned model elements vs. total number of elements. Among the total 17 model elements, 4 fields and 2 methods are repeating. It gives 35% coverage.

4.4 Proposed approach for detecting model clones

In this section, we formalize our approach of model clone detection that consists of three broad steps. Firstly, we export the UML class model to XMI file and parse the XMI file to construct a labeled, ranked tree. In the second step, we apply the algorithm to detect model clones in the constructed tree. In the final step we classify the detected model clones into different categories. Fig. 4.3 shows the block diagram of the proposed technique.

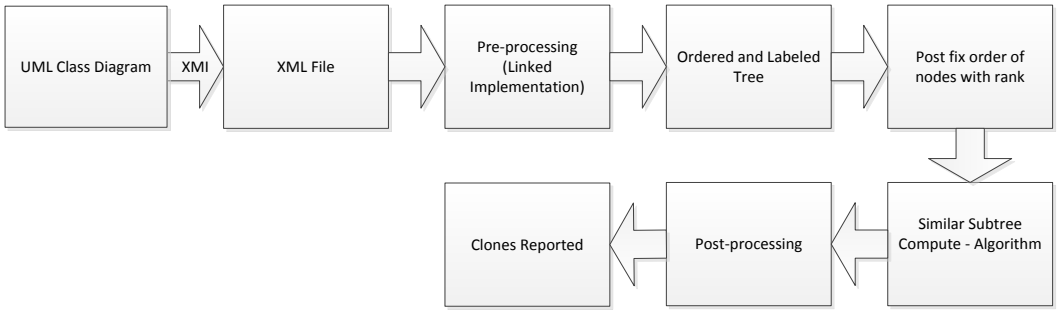


Fig. 4.3 Block diagram of the approach

4.4.1 Modeling and preprocessing

Modeling:

We use MagicDraw Enterprise 16.6 CASE tool for creating forward designed UML class models and for reverse engineering open source subject systems. The model is then exported to an XMI file using XMI export facility of the CASE tool. The exported XMI file is input to our tool to parse and extract information related to our clone detection process.

Preprocessing:

The exported XMI file contains a lot of tool specific information along with the class diagram/model data. Thus, the parsing of XMI file is a major step to extract data of interest i.e. model elements. The XMI elements/nodes are then realized into a tree structure equivalent to the model keeping in mind the constructs of language like nesting of packages, declaration of classes, fully qualified definition of attributes and operations. Generally, when the UML class model is created, the members of the class may be added in no particular order. So, during parsing of XMI file, these model elements are fetched and stored in lexicographic order in the tree.

The algorithm to construct the tree and related definitions are as follows.

Definition 4.4.1 (Element)

It is a common term that refers to a package or class or child class or field (attribute) or method (operation).

Definition 4.4.2 (PackageNode)

It stores information about a package element, its sub packages and classes.

Definition 4.4.3 (ClassNode)

It stores information about a class element, its inner-classes, fields and methods.

Definition 4.4.4 (ModelTree)

It is the tree constructed from model elements.

Definition 4.4.5 (Rank)

It is the out-degree of a node of the tree.

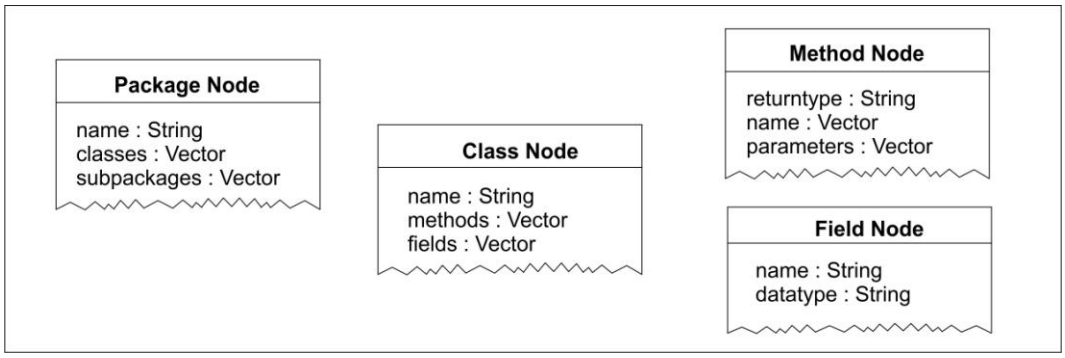


Fig. 4.4 Different nodes of Algorithm 1 as implemented

Fig. 4.4 shows different nodes defined above. These are used in Algorithm1 for storing data extracted from XMI file of the model for pre-processing.

Following is the algorithm applied to construct the ModelTree:

Algorithm 1: Tree Construction Algorithm

Input: An XMI file of UML class model exported by MagicDraw

Output: The post-fix traversal of the ModelTree with rank

find: The root package of UML model and store it in ModelTree

1) Read all the children of the root package node.
2) If the child is a package then store it in the ModelTree under its parent PackageNode and read all the children of this package recursively till all the packages are read.
3) If the child is a class then store it as ClassNode in the ModelTree under its parent PackageNode
4) Read all the children of the ClassNode recursively

   4.1) If the child is a field, then read its label and data type then add to the class's fields.

   4.2) Else if the child is a method, then read its label, return type and parameter types and add to the class's methods.

   4.3) Else if the child is an inner-class then store it in the ModelTree under its parent ClassNode from step 3.

5) Traverse the constructed ModelTree in postfix order and store it in a list

4.4.2 Match Detection

In the pre-processing phase, we constructed the ModelTree from the UML class model.

Fig. 4.5 shows the tree representation of UML class diagram shown in fig. 4.2 of Courier Management System. An attribute model element consists of data type and the name. This is shown as a subtree consisting of two nodes i.e. data type node with attribute name as its child node. Various attribute model elements are depicted in red color in fig. 4.5.

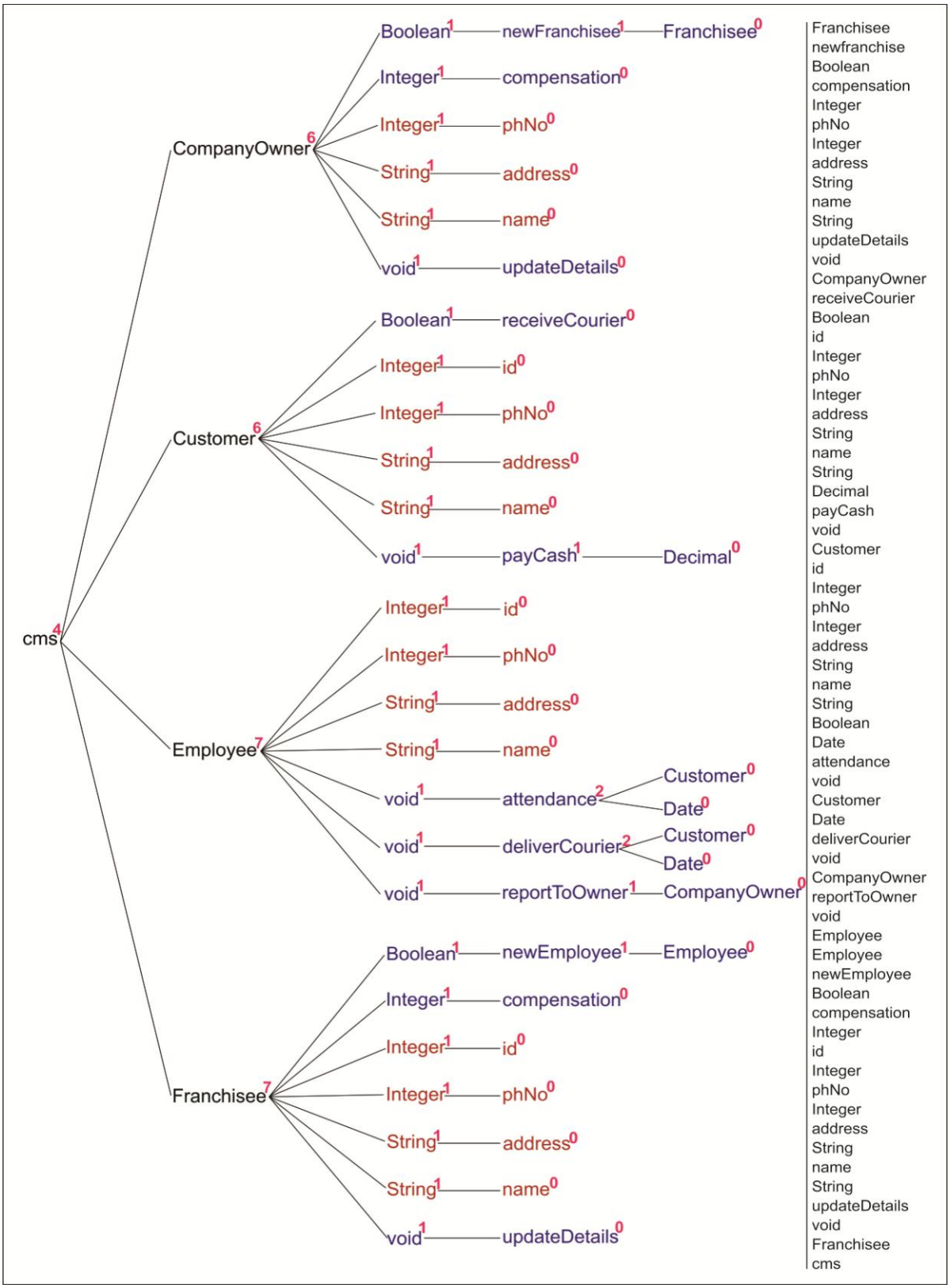


Fig. 4.5 Tree representation of the UML class model

Similarly, a method model element is also represented as a subtree with its return type, name and arguments located at different levels of the same subtree. Various method model elements are shown in blue color in the above figure. Therefore, detecting repeats of these subtrees in the model tree is essential to find model clones. To achieve this, we are using the approach by Christou et al. [37] to compute subtree repetitions. The technique is based on accepting the postfix string representation of tree and computing subtree repeats with varying sizes in a bottom-up manner. The algorithm has got linear time and space complexity.

The tree diagram in fig. 4.5 also shows the postfix string representation of model tree on the right hand side. The rank i.e. out-degree of a node in the tree is shown on the top of the respective node.

4.4.3 Post processing and clustering clones

The output of previous phase reports identical subtrees in the form: set of starting positions, length in corresponding to the input postfix string representation. We need to apply extensive post-processing techniques to make the results useful in a way we require to present the classification and granularity of our approach. To achieve this, the tool maps the output of algorithm to the model tree which is constructed in pre-processing phase for retrieving the results.

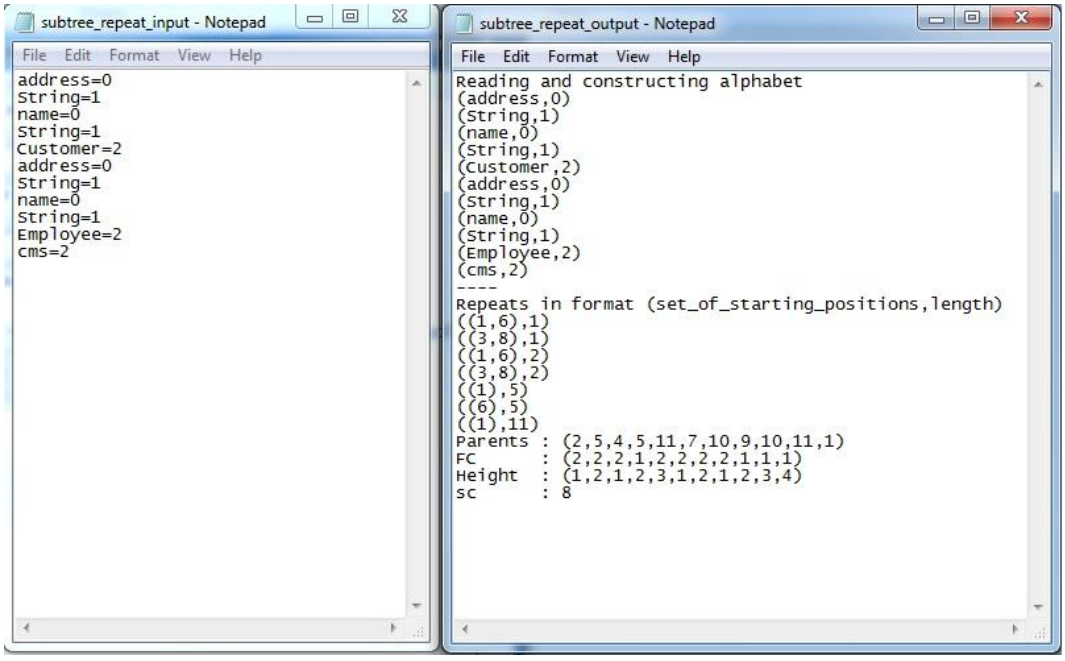


Fig. 4.6 The input and output of the subtree repeat algorithm [37] for a sample class model

We intend to demonstrate this with an example. A small class diagram having two classes with a pair of repeating attributes is created. Fig. 4.6 shows the actual output of the algorithm [37] in the form of sets of starting positions of repeating subtrees and their lengths. As this output cannot be directly applied to our approach, so the post-processing phase becomes an integral part of the whole process.

Following is the procedure to calculate frequency of cloned *ME,* grouping and clustering.

Note: OPL refers to ordered postfix list. *l* refers to label which is either a field or method. C_i indicates classes in ModelTree. CRM refers to class relationship matrix. A matrix element with non-zero value signifies relationship between classes or interfaces. Each relationship type is denoted with a unique identifier.

A.) **Frequency:**

Firstly, the tool applies following algorithm to distinctly report model clones with frequency 2 and model clones with frequency more than 2. The latter set of model clones is inspected to state type-1 clones.

Algorithm

1. For each repeating $l \in$ OPL

    1.1) [RepeatsList] $\leftarrow l$

2. For each $l \in$ [RepeatsList]

    2.1) Count the occurrences, *o* of *l* in OPL

    2.2) [NewList] $\leftarrow$ (*l*, *o*)

    2.3) *l* is categorized as field or method in [NewList]

B.) **Grouping:**

This step is the prerequisite for clustering algorithm.

Algorithm

1. For each $l \in$ [RepeatsList] && $C_i \in$ModelTree

    1.1) If $l \in C_i$

        1.1.1) [GroupingList] ← ($l, C_1, C_2, .., C_n$)

        1.1.2) l is categorized as field or method

2. For each $C_i \in$ModelTree && ($l \in$ [RepeatsList] && $l \in C_i$)

    2.1) [RepeatsinClass] ← l

    2.2) l is categorized as field/method

    [[RepeatsinClass] stores the repeating fields/methods for each class, with their count]

C.) **Clustering:**

To classify and report the clones as type -2 and type-3 model clones, the tool

1) Cluster the duplicate model elements
2) Find occurrence of such cluster across classes
3) Determine the relationship between those classes

Algorithm

1. For each $C_i \in$ ModelTree

    1.1) For each $l \in C_i$ && $l \in$ **[**RepeatsinList]

    1.2) Store repeating l for corresponding C_i

    [Traverse the RepeatsInClass and get a list of repeating members for that Class]

2. For each $C_i \in$ ModelTree

    2.1) Compare [RepeatsInClass] for C_i in Step-1 to the C_i in this step (Step-2)

3. [Cluster] ← C_i

   [Clusters of repetitive members (Fields/Methods) and their presence in Classes through-out the model]

4. Determine the nature of relationship from CRM among the classes from step-3.

4.4.4 Experimental Setup

The proposed technique has been implemented in Java and runs under Windows 7. It is capable to detect model clones in any object oriented class model. The tool receives XMI file of the class model as input and reports the clones as per the defined classification with the count. The tool uses a given algorithm by Christou et al. [37] to compute similar subtrees which has linear time and space complexity. Table 4.2 lists the hardware configuration and software tools needed in implementation.

Table 4.2 Hardware Configuration and Software Tools

Processor	Intel (R) Core (TM) i3-3110M CPU@ 2.40GHz 2.40 GHz
Installed Memory (RAM)	4.00 GB
System Type	32-bit Operating System
Windows Edition	Windows 7 Professional N Service Pack 1
Integrated Development Environment	• NetBeans IDE • Bloodshed Dev-C++
Software Development Kit	Java Development Kit 6
Software Modeling Tool	MagicDraw

4.4.5 Performance Analysis

Our tool can detect the model clones as proposed in the technique very efficiently. Since our tool is a multiphase detection approach, we tried to estimate the computational time of each phase using an average of five iterations for the same input. The execution time is

reported in milliseconds. It takes less than 1 minute to detect model clones in *eclipse-ant*, *eclipse-jdtcore* and *netbeans-javadoc* with more than 290 model elements.

Table 4.3 shows the actual measurement of time when the tool is executed on a PC with the configuration as given in table 4.2.

The UML class model is created using MagicDraw modeling tool. Then the class model is exported to XMI file which is given as input to the first phase of the tool.

Preprocessing and Input Transformation\: In this phase, the XMI file of the given UML class model is parsed and the relevant information is extracted from the XMI file which includes parameters like the name of the class, name of the attribute with their data type, methods with their return type and its arguments, etc. These parameters are then realized into a tree structure i.e. ModelTree equivalent to the model keeping in mind the constructs of language.

Computing Subtree Repeats: We used the algorithm [37] to computer similar subtrees repeating in the model tree constructed from the class model. The technique is based on accepting the postfix string representation of tree and computing subtree repeats with varying sizes in a bottom-up manner. The algorithm has got linear time and space complexity. It reports identical subtrees in the form: set of starting positions, length in corresponding to the input postfix string representation.

Analysis: In this phase, the tool reports model clones with frequency 2 and model clones with frequency more than 2. The latter set of model clones is inspected to state type-1 clones.

Grouping and Reporting: In this phase, the tool reports different field and methods repeating across classes with the corresponding classes, too. Finally, it clusters repeating field and methods which are present in more than one class together with the relationship between those classes.

The tool reports the clones in a way which require subjective assessment from developer’s viewpoint.

Table 4.3 Execution time

	eclipse-ant	**eclipse-jdtcore**	**netbeans-javadoc**
	Average of 5 runs	*Average of 5 runs*	*Average of 5 runs*
Preprocessing and Transform	209	322	217
Computing Subtree Repeats	15	11	13
Analysis	18	30	16
Grouping and Reporting	81	30	63
	323 ms	**393 ms**	**309 ms**

4.5 Empirical Evaluation

Empirical evaluation of the proposed technique is carried out on forward designed and reverse engineered UML class models. One of the UML models is forward designed i.e. created during normal development life cycle. Such model helps to check the practical relevance of the proposed approach. Moreover, no standard repository of UML models recognized by modeling community is available; therefore we chose class diagrams created by reverse engineering open source subject systems. It provides a platform for research community to compare and verify the results in future. We chose the subject systems from Bellon's experiment [21] for empirical evaluation as these systems are well known in code clone detection community.

Subject Systems: The forward designed model is a class model for Proxy Server. This class model is designed using MagicDraw Enterprise 16.6 tool using the standard modeling practices given by OMG [232]. Other subject systems of our study are reverse engineered using the same tool. The characteristics of these systems are listed in Table 4.4. All the systems are in Java with sizes varying from 35K SLOC to 148K SLOC and number of model elements varying from 174 to 292.

Table 4.4 Subject Systems

Subject System	Language	Program Size	#ME	XMI File Size (in KB)
eclipse-ant	Java	35K SLOC	292	947
eclipse-jdtcore	Java	148K SLOC	174	1445
netbeans-javadoc	Java	19K SLOC	267	798

Table 4.5 reports the number of cloned model elements. Clones (f=2) shows the total count of model clones with frequency 2. Clones (f>2) states model clones repeating more than two times.

Table 4.5 Clones in subject systems

Subject System		*#ME*	Cloned *ME*	Clones (f=2)	Clones (f>2)
eclipse-ant	Classes	30	00	00	00
	Attributes	106	17	13	04
	Methods	156	28	15	13
eclipse-jdtcore	Classes	08	00	00	00
	Attributes	85	00	00	00
	Methods	81	00	00	00
netbeans-javadoc	Classes	27	00	00	00
	Attributes	66	08	05	03
	Methods	174	21	17	04
Proxy-Server	Classes	10	00	00	00
	Attributes	33	2	2	0
	Methods	62	3	00	3

Next, the results of evaluation are presented for all the systems. The results are presented in a consistent manner reporting the model clones with high frequency. Next, clusters of attributes/methods present in different classes are reported. Evaluation is based on following three types of clones:

1. Type-1: Model clones due to standard modeling/coding practice
2. Type-2: Model clones by purpose
3. Type-3: Model clones due to design practices

Above categories of clones are explained in detail with examples in sec 4.2 – model clone detection by example. At last, for every class model coverage is mentioned to know the percentage of cloned model elements out of total model elements.

4.5.1 Clones in eclipse-ant

Eclipse-ant is a freely available Java based build tool. The total number of model elements extracted from the XMI file is 292. Table 4.5 shows that out of total 30 classes, no class is repeating. There are about 17 attribute clones out of 106 attributes and 28 method clones out of 156 repeating across the complete UML class model. Fig. 4.7 shows a bar chart showing repetition of attributes and methods across the model. There is one field *target: Target* and one method *startElement (String, attributeList): void* appearing seven times in the complete class model. The field *target: Target* is present in 7 classes viz. *BuildEvent*, *BuildSmallEvent*, *DataTypeHandler*, *NestedElementHandler*, *TargetHandler*, *Task* and *TaskHandler*. Similarly the method *startElement (String, AttributeList): void* is found in 7 classes namely, *AbstractHandler*, *DataTypeHandler*, *NestedElementHandler*, *ProjectHandler*, *RootHandler*, *TargetHandler* and *TaskHandler*. Except the class *RootHandler*, remaining 6 classes are in generalization hierarchy with root class *AbstractHandler*. Above such set of model clones detected by the tool need to be inspected to categorize them as Type-1 clone if these are the result of standard modeling/coding practice.

Clone clusters

The tool reported 18 clusters in *eclipse-ant* and categorized these clusters as type-2 and type-3 clones on the basis of relationship between the classes in which these clusters are present.

Table 4.6 Type-2 and Type-3 clones in eclipse-ant

Type of Clone	**Type-2 Clone**		**Type-3 Clone**
Nature of relationship across classes	Inheritance relationship	Realization relationship	No relationship
No. of clones	3	2	13

Type-2 clones: Model clones by purpose

Among type-2 clones, the tool reported 3 clusters repeating across generalization hierarchy in class diagram. The first cluster consists of 2 methods viz. *characters(char[], int, int): void*, *startElement(String, AttributeList): void* present in *AbstractHandler*, *DataTypeHandler*, *NestedElementHandler* and *TaskHandler* classes. In another case, there is a multilevel inheritance across *ProjectComponent*, *Task* and *UnknownElement* classes. The cluster consisting of three methods i.e. *execute(): void*, *getTaskName(): String*, *maybeConfigure(): void* is present in *Task* as well as *UnknownElement* class. Another cluster made up of 3 methods namely, *characters(char[], int, int): void*, *finished(): void, startElement(String, AttributeList): void* is reported in *AbstractHandler* and *TaskHandler* classes.

During analysis, we came across a couple of instances where a set of concrete classes realize the same interface. In one such case, a cluster of 7 methods viz. *buildFinished(BuildEvent): void*, *buildStarted(BuildEvent): void*, *messageLogged(BuildEvent): void*, *targetFinished(BuildEvent): void*, *targetStarted(BuildEvent): void*, *taskFinished(BuildEvent): void*, *taskStarted(BuildEvent): void* is repeating in *AntClassLoader*, *DefaultLogger*, *IntrospectionHelper* and *XmlLogger* classes. These 4 classes realize an interface named *BuildListener* made up of just mentioned 7 methods.

Type-3 clones: Model clones due to design practices

The tool reported 13 clusters of type-3 clones repeating across classes that do not have any relationship among them.

In first case, 1 method i.e. *log(String, int): void* and 1 field i.e. *project: Project* is present in 2 classes *AntClassLoader* and *ProjectComponent.* There is no relationship between these classes. Similarly, 2 methods *getLocation(): Location* and *setLocation(Location): void* and 1 field *location: Location* is present in *BuildException* and *Task* classes. Another large cluster of same type consisting of 5 fields viz. *exception: Throwable, message: String, priority: int, target: Target, task: Task* and 7 methods viz. *getException(): Throwable, getMessage(): String, getPriority(): int, getTarget(): Target, getTask(): Task, , setException(Throwable): void, setMessage(String, int): void* is found in *BuildEvent* and *BuildSmallEvent* classes. In total there are 7 such instances, but we have listed only some of them here.

In an interesting case, there are five classes viz. *DataTypeHandler, NestedElementHandler, ProjectHandler, TargetHandler, and TaskHandler* which are the child classes of *AbstractHandler* parent class in the UML class model for eclipse-ant. Different clusters of fields/methods repeat across these five classes but not in the parent class. E.g. One of the clusters consisting of 2 fields *target: Target, wrapper: RuntimeConfigurable* and three methods viz. *characters(char[], int, int): void, init(String, AttributeList): void, startElement(String, AttributeList): void* is present in *DataTypeHandler* and *TaskHandler* class. Another similar cluster comprising of two methods i.e. *init(String, AttributeList): void, startElement(String, AttributeList): void* and one field *target: Target* present in *DataTypeHandler*, *NestedElementHandler*, *TargetHandler* and *TaskHandler* classes. Similarly, among the same five subclasses, a cluster of 2 methods i.e. *init(String, AttributeList): void* and *startElement(String, AttributeList): void* is present.

Another similar cluster consists of 1 field and 3 methods existing in *DataTypeHandler, NestedElementHandler* and *TaskHandler* classes. In total we traced 6 such instances in which the duplication is prevalent across subclasses of the same parent class.

Coverage

In total of 292 model elements (*# ME*) in eclipse-ant, we detected that 45 out of it are clones. These may be attributes/operations in a class. So clone coverage is total cloned elements vs. total elements i.e. 15.4 %.

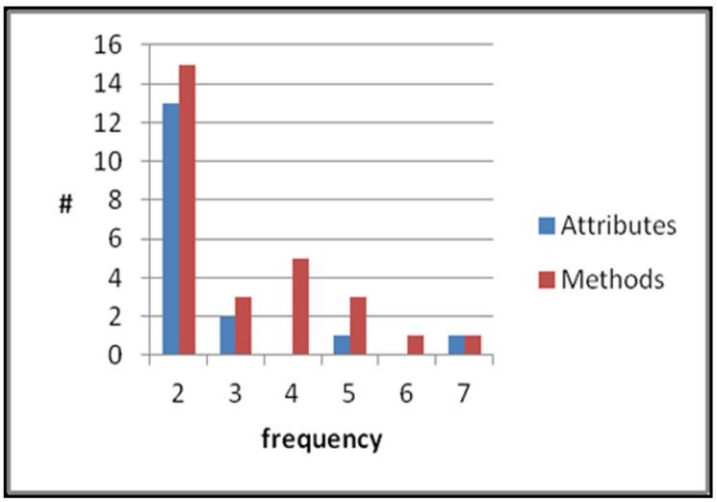


Fig. 4.7 eclipse-ant

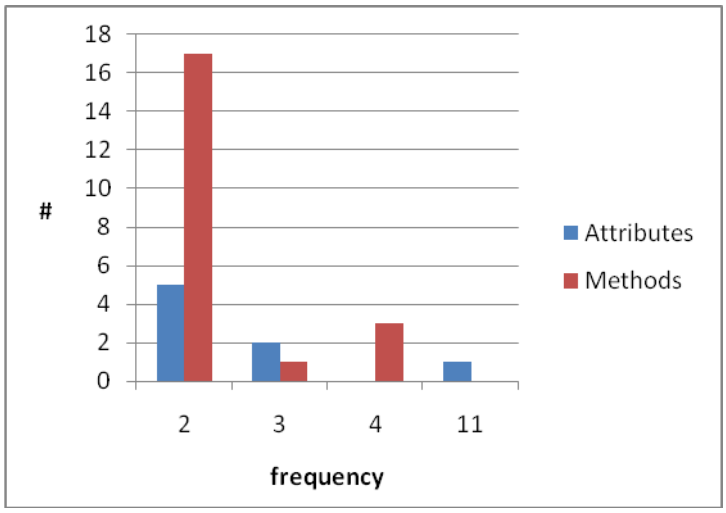


Fig. 4.8 netbeans-javadoc

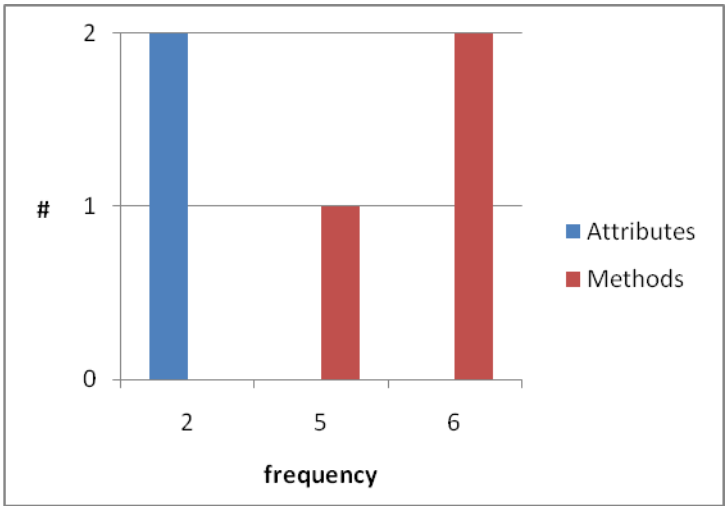


Fig. 4.9 proxy server

Figs. 4.7 – 4.9 show frequency of cloned *ME* in different subject systems.

4.5.2 Clones in netbeans-javadoc & eclipse-jdtcore

The technique has been applied to two systems namely, *netbeans-javadoc* and *eclipse-jdtcore* with 267 and 174 model elements. Primarily, we want to know the degree of cloning in the form of individual attributes/methods and clusters in UML class models. *netbeans-javadoc* has 29 cloned *ME* consisting of 8 attributes and 21 methods. *eclipse-jdtcore* has no attribute/method repeating out of 174 model elements as shown in Table 4.5. Fig. 4.8 shows frequency of cloned model elements for *netbeans-javadoc*. 75 % of model elements in *netbeans-javadoc* are repeating two times. There is one attribute *serialversionUID* which is repeating across 11 classes. Individual attributes/operations have relevance during forward engineering as we generate code from mode as well as during reverse engineering as we generate model from the code. Though these types of clones seem small in the first observance, these signify standard modeling/coding practice, thus classified as Type-1 clones.

Clone clusters

In total, 6 clone clusters are detected. As shown in Table 4.7, all these clusters are present across classes where there is no relationship between the classes. We list some of them here. One clone cluster has 3 methods namely, *getAdditionalBeanInfo(): BeanInfo, getBeanDescriptor(): BeanDescriptor*, *getIcon(int):Image* is present in 4 classes viz. *ExternalJavadocExecutorBeanInfo, GlobalLocalFileSystemBeanInfo, JavadocTypeBeanInfo* and *StdDocletTypeBeanInfo*. Another large cluster comprising of 2 fields namely, *docletS: StdDocletType* and *javadocS: ExternalJavadocSettingsService* and 5 methods *getDestinationDirectory(): String*, *isStyle1_1(): boolean, loadChoosenSetting(): void, setBooleanOption(Boolean, String, list): void, setStringOption(String, String, List): void* is repeating in classes *ExternalOptionListProducer* and *OptionListProducer.*

Table 4.7 Type-2 and Type-3 clones in netbeans-javadoc

Type of Clone	**Type-2 Clone**		**Type-3 Clone**
Nature of relationship across classes	Inheritance relationship	Realization relationship	No relationship
No. of clones	00	00	06

In two particular cases, we came across four classes namely, *ExternalJavadocExecutorBeanInfo, GlobalLocalFileSystemBeanInfo, JavadocTypeBeanInfo, StdDocletTypeBeanInfo* where all the methods *getAdditionalBeanInfo(): BeanInfo [], getBeanDescriptor(): BeanDescriptor, getIcon (int): Image* repeat. In another example, a group of 1 field *bundle: ResourceBundle* and 1 method *getBundledString(String): String* is present in 2 classes i.e. *CommonUtils* and *ResourceUtils*. We observed upon analysis of the results, 4 classes *ExternalJavadocExecutorBeanInfo, GlobalLocalFileSystemBeanInfo, JavadocTypeBeanInfo* and *StdDocletTypeBeanInfo* are exactly same. Each class is made up of same 3 methods as mentioned earlier. Another case is *CommonUtils* and *ResourceUtils* with similar contents.

Coverage

We get 11 % coverage in case of *netbeans-javadoc*. We detected 29 cloned *ME* in the total of 267 model elements.

4.5.3 Clones in proxy server class diagram

The proxy server class diagram is the forward designed class model. This class diagram is made up of 10 classes and 1 interface. In total we have 33 fields and 62 methods in the model. Fig. 4.9 shows a bar chart showing repetition of attributes and methods across the model. To analyze the type-1clones, we came across 2 fields viz. *SerialVersionUID: long* and *LOGGER: logger* which is repeating in 2 classes. As per our classification, these fields are the result of standard modeling practice.

Clone clusters

In this class diagram, the tool detected 2 clusters. First cluster is made up of three methods namely *onRequest(HttpServletRequest, HttpServletResponse, URL): void*, *onRemoteResponse (httpMethod): void* and *onFinish(): void* repeating in 5 classes *MimeTypeChecker*, *HostChecker*, *MethodsChecker*, *HostNameChecker*, *RequestTypeChecker* and 1 interface named *ProxyCallBack*. This is the result of classes implementing the same interfaces thus having similar set of methods. These types of clones show behaviors in model emerged from careful application of useful design paradigms. Thus classified as a type-2 clone.

As shown in Table 4.8, there is another cluster made up of 2 methods *onRemoteResponse (httpMethod): void* and *onFinish(): void* which is present in the class *HTTPProxy* in addition to all the five classes mentioned earlier in the first cluster. Since *HTTPProxy* class has no relationship with any of other class, thus categorized as type-3 clone.

Table 4.8 Type-2 and Type-3 clones in proxy server

Type of Clone	**Type-2 Clone**		**Type-3 Clone**
Nature of relationship across classes	Inheritance relationship	Realization relationship	No relationship
No. of clones	00	01	01

Coverage

We found that among 10 classes, 33 fields and 62 methods there is no class repeating, 2 fields and 3 methods are repeating. So there are 5 cloned *ME* out of 105 *ME*. We get around 5 % clone coverage.

4.6 Discussion

We start this section with discussion on the findings of the empirical evaluation.

The tool reports set of model clones with frequency more than 2. To find type-1 clones, an inspection is carried out by the authors in this set of model clones. We support our point of view using Stephan et al. [213] key parameter for model comparison i.e. "*ability*

to identify recurring patterns using a combination of manual inspection and model visualization".

As an application of the proposed technique in Java programming practice, if a class overrides equals method then it must override the hashcode method as well. One can confirm this by inspecting the output of the tool as follows - 1) Set of classes in which equals method repeat. 2) Set of classes in which hashcode method repeats. 3) Compare the two sets for difference.

Thus one can identify the classes which are not following the above standard programming practice.

We are keen to gain insights into type-3 clones reported by the tool as these will lead to improvement in the design of the system. For instance in *eclipse-ant*, we came across interesting clusters repeating in a set of classes. Though these classes inherit the same parent class but the repetitive clusters are absent in their parent class. As per our interpretation, one reason for the presence of such clusters may be the absence of support of multiple inheritance in Java. We believe that if a UML class model is designed first then such clones will be detected at an early stage before they appear in implementation.

Importantly, the tool is able to report model clones of different granularities across the model. For instance, during empirical evaluation of *netbeans-javadoc*, we came across instances where classes have identical members i.e. all attributes and operations but different class names. These type-3 clones of class level granularity are relevant to the modeling viewpoint and demand further investigation. Also, Stephan et al. [213, 214] highlighted relevance as one the parameters for comparison and evaluation. In another application, test cases for white box testing are generated from models and most of the logical errors are traced back to design. Therefore, if designs are free from clones then test cases can be designed effectively and efficiently. In nutshell, the classification and detection of model clones at different levels of granularity is the evidence of usefulness of the proposed approach.

4.6.1 Threats to Validity

There are only two potential threats to validity in our study. One potential threat is the definition of term 'model clone' and its types. In the field of UML models, the term clone

is still lacking standardization. We have defined the term model clones considering the elements of a class model. But we are of the view that model clones can be defined in other ways also considering different views of the UML.

An external threat to validity is the selection of subject systems for empirical evaluation. There is no standard repository of UML models and real world industrial systems are not available due to different proprietary reasons.

4.7 Comparative Analysis

An existing tool for clone detection in UML models is MQ_{lone} that presented model clone classification adapted from the work on source code clones [216]. Our technique of clone detection starts by exporting a UML class model to an XMI file using the inbuilt facilities of the CASE tool. The core of our technique is the construction of a labeled, ranked tree by carefully mapping the elements parsed from the XMI file to the tree representation such that attribute model elements and method model elements are represented as subtrees. The algorithm to detect duplicate subtrees is applied which yields the sets of starting positions of repeating subtrees and their lengths. To detect exact and meaningful clones as per the proposed classification, post-processing needs to be applied. The algorithms to calculate frequency of model clones, grouping and clustering are applied to classify the clones. In contrast to our approach, Störrle's MQ_{lone} [216] tool detects clones in UML domain models and is based on model querying. The UML model is exported to an XMI file using any of the UML modeling tools. XMI files are transformed to Prolog files in which clone detection is carried out based on model matching. The result is the prioritized list of matched model elements with their similarity.

In his study, the clones are categorized as Type A (exact model clone), Type B (modified model clone) and Type C (renamed model clone) adapted from code clone classification (Type-1/ Type-2/ Type-3). What's unique about our approach is the classification of model clones i.e. Type-1 (model clones due to standard modeling/ coding practice), Type-2 (model clones by purpose) and Type-3 (model clones due to design practices). The objective of his approach is to detect clones with varying degree of changes whereas our technique is aimed at identifying clones which are actually relevant for practical purposes. The evaluation in our work provides observations of the

characteristics of model clones which may help in gaining some insights from implementation point of view.

We understand that our approach will be first of its kind to identify exact and meaningful model clones for development and maintenance purposes. Moreover, we are performing the analysis of forward designed as well as reverse engineered UML class models. Reverse engineered UML class models are open source subject systems which provide the platform to research community to compare and evaluate the results in future. These systems have been extensively used in field of code clones to detect, compare and to know the impact of code clones on software quality. On the other hand, MQ_{lone} is evaluated on different UML models created by 16 Master's students. The best model with no natural clones is selected and seeded with artificial clones to emulate Type A, B, and C model clones. Recall and Precision is calculated based on the seeded clones. Different similarity heuristics based on element names and element index are compared on the basis of precision, recall and number of false positives from the subset of detected clones. His models are not publicly available.

Our subject systems are diverse, open source and widely used for development. It may help in understanding the nature of heterogeneous subject systems with respect to cloning. On the other hand, MQ_{lone} is evaluated on a set of homogeneous subject systems.

MQ_{lone} is applied to different kinds of UML domain model. But our work is focused on detection of exact and meaningful clones in the UML class models at different levels of granularity.

Indeed there is no standard definition of model clone in research community [69]. In code clone domain, the definition of clone is task oriented and dependent on detection technique [193]. Human judgment plays the key role in categorizing what is and is not a clone [234]. The findings of our work in terms of detected clones can be more closely inspected by researchers and practitioners to decide how to deal with them and improve the software design in the future.

4.8 Summary

In this chapter, we have presented a new technique to detect clones in UML class models based on a three stage process. In the first stage, the technique accepts the XMI file of a

UML class model as input. Secondly, a labeled, ranked tree is constructed by carefully mapping the elements parsed from the XMI file to the tree representation. The output of this phase is the list of repeating subtrees. In the third phase of the process, the duplicate subtrees are grouped and clustered with the aim to detect exact and meaningful clones. The tool is able to detect clones at different levels of granularity i.e. single attributes/methods of the class, clusters of attributes/methods and complete class in the UML model. We have proposed a novel classification of model clones keeping in mind the object oriented nature of UML class model such that the detected clones are meaningful for the developer. The technique has been empirically evaluated on open source reverse engineered and forward designed systems. We believe that the results of the tool are accurate but the detected clones need to be supported by the intent of software developers regarding relevance for practical purposes. We are currently extending our technique to efficiently refactor type-3 clones.

Chapter 5

Concept Clone Detection

5.1 Introduction and Motivation

Concept clones are defined as similar program structures within one software system or across versions reflecting similar domain concepts. Since UML [236] is increasingly replacing conventional programming languages for developing and modeling software systems. The presence of design level similarities in form of concept clones cannot be precluded in UML models [105]. In our work, we define **concept model clones** as two different class models representing the same conceptual/ domain model or design model. The class diagrams may be created by different people pertaining to same domain. We have developed a tool to detect concept clones.

It is true that similarities exist at higher level of abstraction in software. Marcus and Maletic [172] identified a scenario in which the software developer writes fresh solution to a problem knowingly or unknowingly for which the solution is already existing. This scenario leads to different solutions for the same problem. Identification of high level concept clones helps the software developer in understanding and comprehending the system at abstract level [18]. We realized the importance of similar domain concepts for sharing and interoperability [225]. During evolution, high level similarities emerge. The presence of structural similarities as the software system evolves helps in improving the understanding of the system. Detection of high level similarities throughout the history of the subject system helps in improving the different attributes of quality of the software like maintainability, reusability, extensibility [225].

Key Contributions

- To understand and analyze the concept clones, we detected clones at method level. A manual analysis of 219 sample methods within one version of software system is done to validate the technique.
- As an important application of the proposed technique, we applied it in coarse-grained multi-version program analysis to detect changes in methods across versions of

popular Java open source system *dnsjava* [48]. Concept clones are detected across 8 releases of *dnsjava.*

- The novelty of the proposed work goes beyond source code and lies in detection of concept model clones across UML models.

In this chapter, section 5.2 explains our approach of concept clone detection. The empirical evaluation and results of the tool are included in section 5.3. Section 5.4 discusses the proposed work. Comparison with existing work related to concept clone detection is done in section 5.5. Finally, Section 5.6 concludes the work.

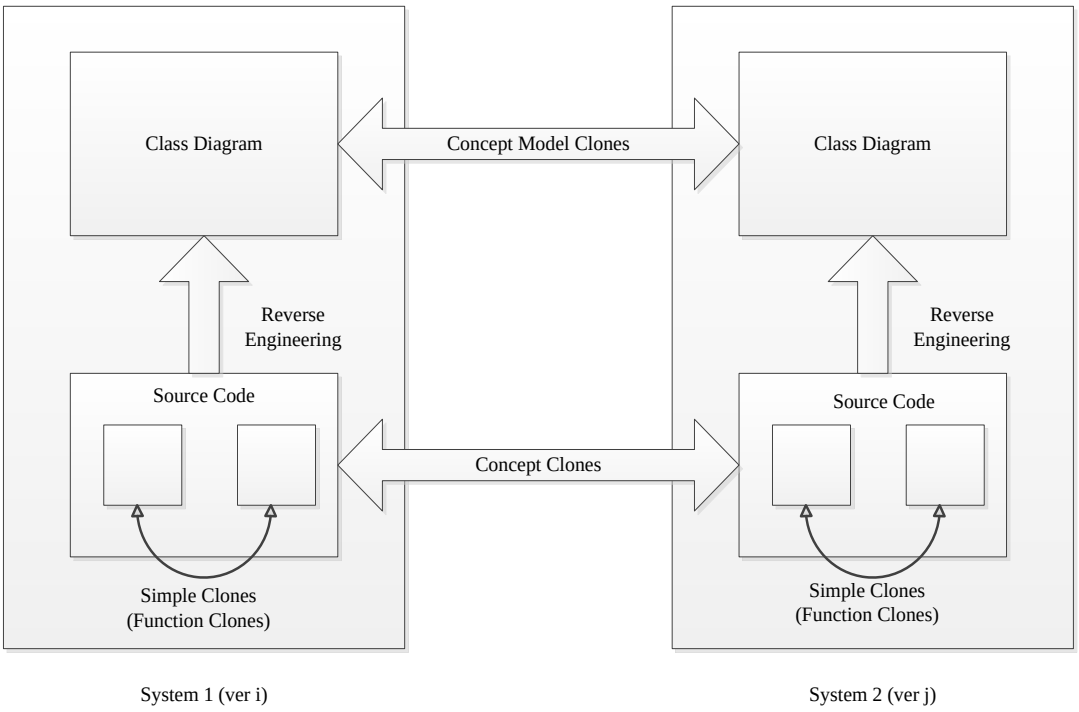


Fig 5.1 Different levels of clone representation

5.2 Proposed approach for concept clone detection

The proposed method compute the similarity between source code elements and UML class models at three levels. We detected high level concept clones in UML class models, similarity between source code software elements across versions, clones in software code fragments at method level as shown in fig. 5.1. We will refer to all the three types of input as source files. Currently the tool gives the similarity score between 0 and 1 at every level automatically once the file/files are given as input. Within a single class file as input, it compares every method with other method and generates the output. We detected

the high level similarities first and then explored the files deeper inside at method level to verify the results.

We applied the well known techniques which are widely applied in information retrieval field to search for similarities of concepts extracted from source code elements/ class model element. Further, we explore the details of our technique.

5.2.1 Vector space model

A vector space model [24] is a mathematical model having a consistent mathematical structure which advances conceptual understanding. In a data centric approach of vector space model, the data is typically represented as a matrix where vector is used to represent each item or document in a collection or database. Each component of the vector reflects a particular concept, key word, or term associated with the given document. The value assigned to that component reflects the importance of the term in representing the semantics of the document. Typically, the value is a function of the frequency with which the term occurs in the document or in the document collection as a whole. In our work, a database is a collection of two or more class diagrams/ package/ source code files. Terms are non-unique tokens extracted from different class models in the database or tokens extracted from methods in source code. So a database containing a total of d documents described by t terms is represented as a term-by-document matrix A.

$$A = t \times d$$

The d vectors representing the d documents form the columns of the matrix. Thus, the matrix element a_{ij} is the weighted frequency at which term i occurs in document j. In the parlance of the vector space model, the columns of A are the document vectors, and the rows of A are the term vectors. The semantic content of the database is wholly contained in the column space of A, meaning that the document vectors span that content. We try to exploit geometric relationships between document vectors to model similarities and differences in content. A variety of schemes are available for weighing the matrix elements. The elements a_{ij} of the term-by-document matrix A are often assigned two-part values $a_{ij} = l_{ij}\, g_i$. In this case, the factor g_i is a global weight that reflects the overall value of term i as an indexing term for the entire collection. The factor l_{ij} is a local weight that reflects the importance of term i within document j itself. Local weights range in complexity from simple binary values 0 or 1 to functions involving logarithms of term

frequencies. Global weighting schemes range from simple normalizations to advanced statistics-based approaches [70,146].

The existence of a vector space implies that we have a system with linear properties: the ability to add together any two elements of the system to obtain a new element of the system and the ability to multiply any element of the system by real number [202]. This is a model in which documents are mapped as real-valued vectors $d_i = (w_{i1}, w_{i2}. . . w_{im})$ where w_{ij} corresponds to the j^{th} key (j = 1, 2, . . .m) of the i^{th} document (i = 1, 2, . . ., n). Thus, the collection is represented as an n* m document-term matrix D of n documents and m keys. To weight the keys, we applied the well-known tf*idf scheme

$$W_{ij} = tf_{ij} \times \log \frac{N}{df_j}$$

Where the inverse of the document frequency df_j i.e. number of documents which include the jth key is multiplied by the key frequency tf_{ij} in a document. There are variations in the equation in addition to the former basic formula. Although tf.idf focuses on key frequencies in individual documents and the generality of keys in the collection, they are sensitive to the lengths of documents or numbers of words in them: the longer the document the more key occurrences. The normalization of document vectors adjusts the effect of both increasing word frequencies and increasing the numbers of word matches for increasing document lengths [73,202].

5.2.2 Latent semantic indexing

Latent semantic indexing (LSI) was introduced in 1990, and is able to predict some underlying latent semantic structure that is partially not visible due to randomness of selected tokens with respect to retrieval [70], [229]. LSI has been used to detect clone in source code [172]. Vector Space Model (VSM) forms the basis for LSI [44]. We are implementing LSI using singular value decomposition (SVD). Here, we take large matrices of diagram-token association data and generating a semantic space wherein tokens and diagrams that are closely related to each other are placed near one another. SVD is a method for transforming correlated variables into a set of uncorrelated ones that better exposes the major associative patterns in the data, and ignores the smaller and less important data elements. Although, it reduces the dimension of original matrix, but at the same moment it also reflects data points which exhibit the most variation. That is, to identify similar patterns of data, similarity measures can be applied to the reduced,

synthetic features rather than to original dataset. This approach has advantages in terms of speed, memory and even accuracy. One particularly nice application of this idea is to combine information retrieval with the use of principal components in dimension reduction.

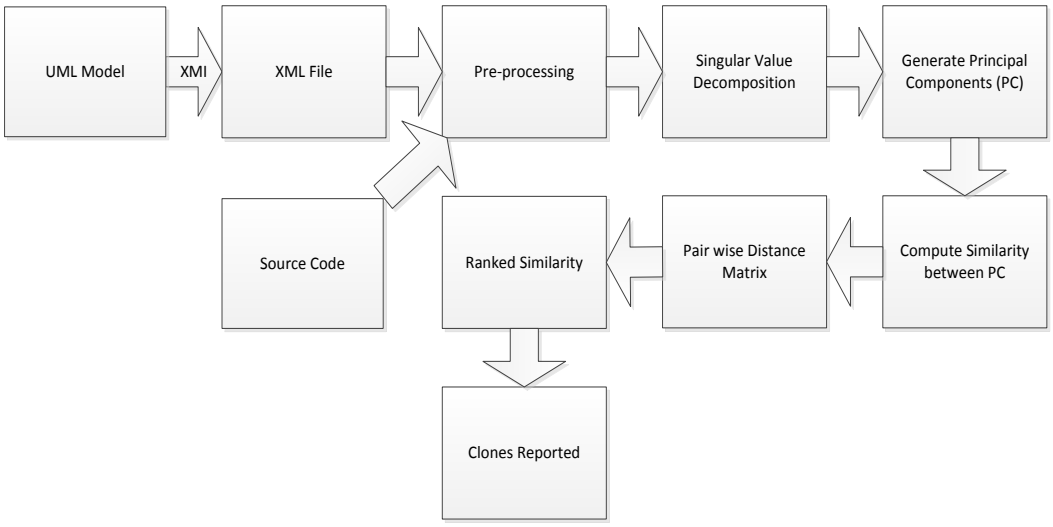


Fig 5.2 Overview of the process

5.2.3 Principal component analysis

The main idea of using principal component analysis (PCA) is to reduce the dimensionality of input data set in which there are a large number of interrelated variables, while maintaining as much as possible of the variation present in the data set. This variation and reduction in input data set is achieved by transforming it into a new set of variables, the principal components, which are uncorrelated, and which are ordered so that the first few retain most of the variation present in all of the original variables. It is a way of identifying patterns in data, and expressing the data in such a way as to highlight their similarities and differences. Since patterns in data can be hard to find in data of high dimension, where the luxury of graphical representation is not available, PCA is a powerful tool for analyzing data [39]. Firstly, data are typically mean corrected by subtracting its mean from each variable [71,157,229]. Secondly, if large differences in the variances of the original variables do not reflect their relative importance, the mean-corrected data may be standardized by dividing each variable by its standard deviation. The objective of linear PCA is to use a linear mapping to find a set of d orthogonal basis vectors that maximally captures the relationship between original dimensions. The process of determining most influential features having maximum eigen values is called

SVD. SVD is a decomposition of a matrix. It helps in finding principal components. Then we find covariance matrix from the normalized matrix. Then we apply SVD on covariance matrix of both data sets, which picks out structures in one data set which are best correlated with the structures in the other data set.

5.2.4 Similarity metric

When documents are represented as term vectors, the similarity of two documents corresponds to the correlation between the vectors. This is known as the cosine of the angle between vectors, that is, cosine similarity. Cosine similarity is one of the most popular similarity measure applied to text documents, such as in numerous information retrieval applications and clustering too. In this experiment documents are classes or class diagrams. Similarity of two document vectors will be calculated as:

$$Cosine\ Similarity\ (d_1, d_2) = \frac{d_1 \times d_2}{|d_1| . |d_2|}$$

Where d_1 and d_2 represents their own term weights and $|d_1|$ and $|d_2|$ represents length of vectors.

5.2.5 Experimental Setup

The proposed technique has been implemented in Java and runs under Windows 7. It is able to compare two files at different levels of granularity and reports the result as a number between 0 and 1. *Dnsjava* is an object oriented software system which is developed in Java. It is used for evaluation. The tool has three variants:

- The first variant receives XMI files of two class models as input and reports the result. This variant extracts name of the class, name of the attribute and name of the operations of a class from the XMI file which is used to estimate the similarity between class models.

- The second variant receives two class files from the source code as input and reports the result. This variant is based on extracting non-unique tokens from the class file which acts as the basis for comparison.

- The third variant receives a single source code class file as input and automatically extracts all the methods from the class file and reports the result as

the similarity between 0 and 1 for all the methods. It is again based on extracting non-unique tokens from all the methods.

Table 5.1 lists the hardware configuration and software tools needed in implementation.

Table 5.1 Hardware configuration and software tools

Processor	Intel (R) Core (TM) i3-3110M CPU@ 2.40GHz 2.40 GHz
Installed Memory (RAM)	4.00 GB
System Type	32-bit Operating System
Windows Edition	Windows 7 Professional N Service Pack 1
Integrated Development Environment	NetBeans IDE
Software Development Kit	Java Development Kit 6
Software Modeling Tool	MagicDraw

JAMA 1.0.2: It is a basic linear algebra package for Java. It is standard matrix class for Java. JAMA is comprised of six Java classes: Matrix, CholeskyDecomposition, LUDecomposition, QRDecomposition, SingularValueDecomposition and EigenvalueDecomposition.

Java Parser 1.0.8: It is a lightweight and easy to use parser. It supports AST generation. It is possible to create AST from scratch or change the AST nodes.

5.2.6 An example

Fig. 5.2 shows the overview of the process. In this technique, firstly, input source code will be searched for tokens using a java API i.e. javaparser. The technique works in three phases as mentioned below:

1) **Parsing and identification of non-unique tokens**

2) **Computation - involves SVD and PCA**

3) **Comparison and reporting - involves computing cosine similarity between principal component vectors**

Three classes are made for this purpose:-

(1) FileParserManager (2) ClassFileParser (3)ClassFileComputation

1) Parsing and identification of non-unique tokens:

We used Java Parser, a freely available tool for parsing. It consists of a class FileParserManager. Once we drag and drop the target file to the tool, this class will immediately call its subordinate class i.e. ClassFileParser that will further request java parser to find methods and tokens file contains. It will return counted methods and tokens to class FileParserManager. Here we are considering only those tokens that at least consists of 3 letters and have a frequency count >2.

2) Computation:

It involves calculating local and global weights of individual tokens within a given target file and computing singular value decomposition of given vector space. SVD helps in removing insignificant tokens. At last, principal component vectors will be identified.

3) Comparison and Reporting

This is the last step of the process in order to detect clones. As in second step, we have identified principal component vectors. Similarity value will be computed within file and in-between files.

5.3 Empirical Evaluation

Software developers may be interested to know the similarity between software elements at different levels depending upon the need. The developed tool works at different levels in a top down fashion. Firstly we detected larger clones at class model level in UML models. Then we checked the similarity across classes in the source code to check identical concepts. At the lower level, we checked simple clones at method level for source code file.

Once the similarities between source code elements have been computed at every level, software developer may cluster input class models/ files/ methods depending on score. One cluster at the abstract level of model or class will represent similar concepts implemented [92]. Similar clusters of methods may also be grouped into different classifications depending upon the interval in which the score lies. The files which are

very close with score between 0.9 and 1 significantly describe similar high level concept at all levels.

5.3.1 Real world concepts

It has been planned to identify examples from related concepts stemming from real world scenarios as concepts are the building blocks of any information system. In one example, we took small class diagrams for library management system modeled by different developers. Fig 5.3 shows both the diagrams. We understand that UML class model is more like a graph but in our work we are not considering two dimensions of models. Our approach is motivated by following key points:

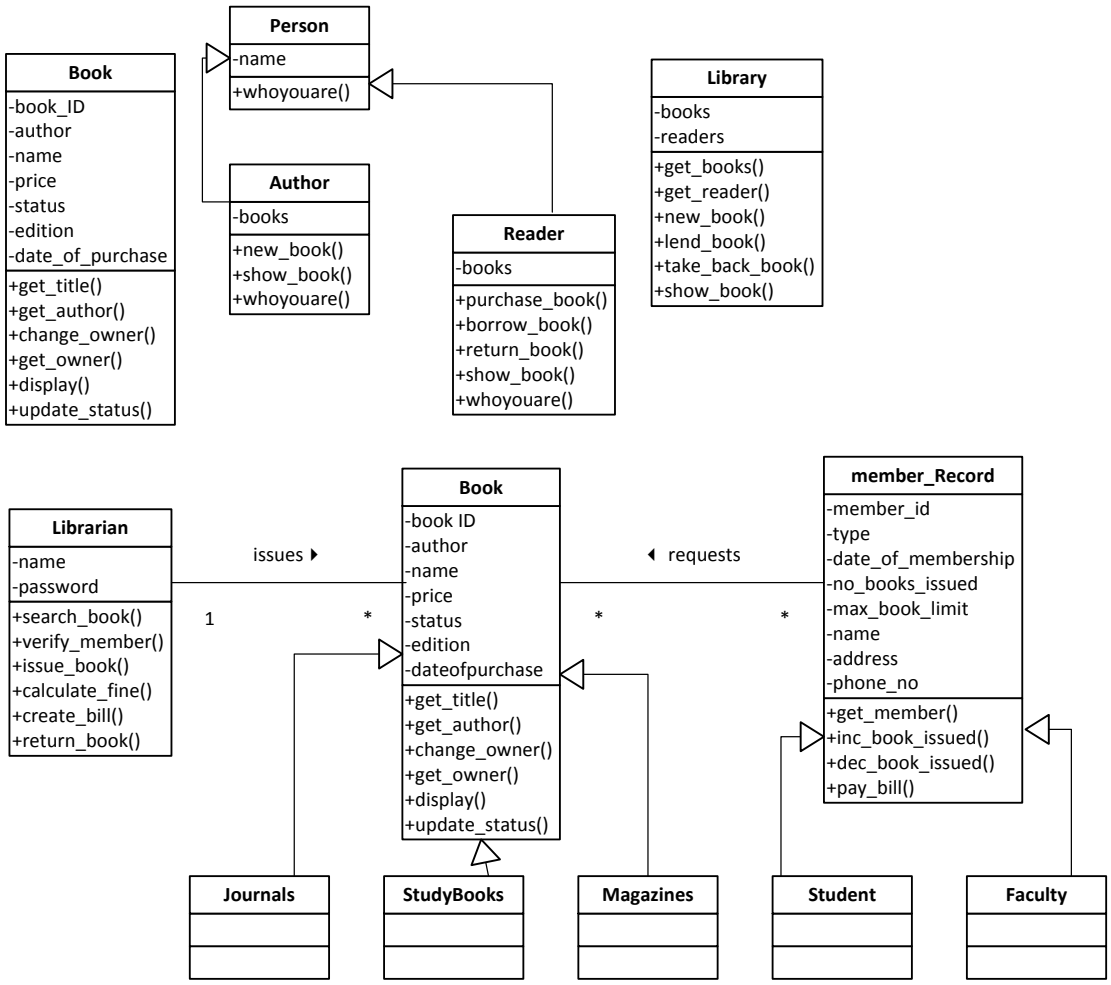


Fig 5.3 Class diagrams showing library management system

- UML class model is a graph but source code is a graph too considering control and data flow.
- Source code is written linearly but model is code too [74]. In Mark Harman's [74] keynote address at 10th International Working Conference on Source Code Analysis and Manipulation 2010, he articulated the idea of high level code for UML class models. We have textual representation for UML class models in the form of XMI files [119]. We used the XMI file to extract terms i.e. name of class, name of the attribute and name of the operation from the class diagram.
- Conceptual models are usually developed independently by different persons. Identification of structural patterns i.e. name of class, name of attribute/ method in UML class diagrams and tokens in source code helps in comparison of conceptual models. Further it may aid model integration and merging. Such a relationship between conceptual models promotes interoperability.

Applying our approach, we came to the similarity score of 0.7365 for library management system. The similarity score is a clear indication of proximity of both conceptual models.

5.3.2 Conceptual clones across version control systems

Comparison of class models/ source code across versions helps in identifying conceptual clones. We may have several related models/ source code files describing the same concept. The related files may be representing different variants of the same software. In another application of the technique, in *copy-modify-merge* model of version control system, many users simultaneously modify their private copies. We can apply this technique to identify similar concepts across projects being developed by users working on same software. Whether one intends to merge/differ two models/ source code files, our tool is helpful in checking the proximity of models/ source code files. To demonstrate the usefulness of our approach, we have taken the example of open source system in Java. We used the version information to track high level clones across versions of the software system. We are working at class model/ source code class level of granularity across versions. Similar files across versions represent similar intentions.

Multi-version program analysis/ software evolution/ origin analysis

Understanding the history of a software system helps the software developer in quick program analysis. As new versions of software appear, the software elements tend to

change to keep it more useful as per the current requirements. We have tried to learn the evolution history by using techniques of high level concept clone detection. Using multi-version program analysis, we compared the number of functions between subsequent versions of the software. We have mined important change patterns from the evolution history of software system.

We have performed a case study on *dnsjava* [48]. It is open-source software system and an implementation of Domain Name Server in Java. The *dnsjava* project started in September 1998. It is still an active project and continues to evolve. For our case study, we selected 8 releases of *dnsjava* starting from 2.0.6 (Jan. 2008) to 2.1.4 (Jan. 2013). We decided to study the class files in org/xbill/dns sub package which is carrying out most of the DNS functions.

Table 5.2 summarizes the releases of *dnsjava* and its growth. The number of files is the number of Java class files in org/xbill/dns sub package. It has increased from 107 to 117 in the five years.

Table 5.2 Growth of dnsjava

Sr. No.	**Release**	**Date**	**# files**	**# LOC**	**# SLOC**
1.	dnsjava-2.0.6	24 Jan, 2008	107	18536	9918
2.	dnsjava-2.0.7	25 Sep, 2009	112	19503	10325
3.	dnsjava-2.0.8	21 Nov, 2009	112	19514	10328
4.	dnsjava-2.1.0	07 Sep, 2010	112	20308	10865
5.	dnsjava-2.1.1	09 Feb, 2011	112	20387	10914
6.	dnsjava-2.1.2	24 July, 2011	116	20991	11178
7.	dnsjava-2.1.3	24 Oct, 2011	116	21085	11226
8.	dnsjava-2.1.4	04 Jan, 2013	117	21454	11441

We applied the tool across subsequent versions of *dnsjava*. We compared class files to get the score between 0 and 1, where 1 denotes a perfect match. Value close to 1 indicates high proximity than the value close to 0. As shown in the table 5.3 most of the class files

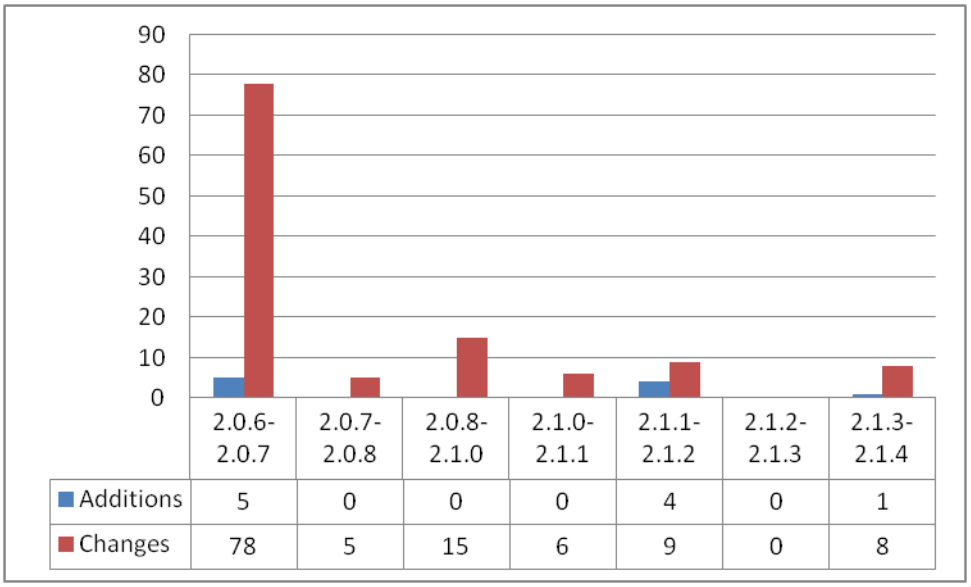
Table 5.3 Concept clone detection across versions of dnsjava

Sr. No.	Vold → Vnew	Name of class file	Change in number of methods	Addition/ Deletion	Cosine similarity result
1.	2.0.6 → 2.0.7	APLRecord	11 → 12	+1	1.00
2.		LOCRecord	16 → 18	+2	0.61
3.		Name	35 → 34	-1	0.62
4.		NSECRecord	10 → 8	-2	0.48
5.		ResolverConfig	16 → 17	+1	0.62
6.		Tokenizer	32 → 34	+2	0.63
7.		UDPClient	6 → 7	+1	0.61
8.	2.0.8 →2.1.0	DNSSEC	2 → 31	+26	0.48
9.		KEYBase	8 → 5	-3	0.74
10.		ResolverConfig	17 → 18	+1	1.00
11.		SIGBase	13 → 14	+1	0.74
12.		UDPClient	7 → 8	+1	0.66
13.	2.1.0 →2.1.1	Lookup	10 → 11	+1	0.62
14.		ResolverConfig	18 → 21	+3	0.58
15.	2.1.1 →2.1.2	Address	17 → 18	+1	0.60
16		DNSOutput	13 → 14	+1	0.62
17.		DNSSEC	31 → 32	+1	0.68
18.		OPTRecord	12 → 11	-1	0.65
19.		Record	40 → 41	+1	0.50
20.		WireParseException	2 → 3	+1	0.00
21.	2.1.2 →2.1.3	ZoneTransferIn	27 → 34	+7	0.67
22.	2.1.3 →2.1.4	DNSSEC	32 → 36	+4	0.63

are unchanged as they are exact duplicates i.e. value 1. We picked up all those class files where the score is less than 1. Those class files in both the versions are analyzed to detect

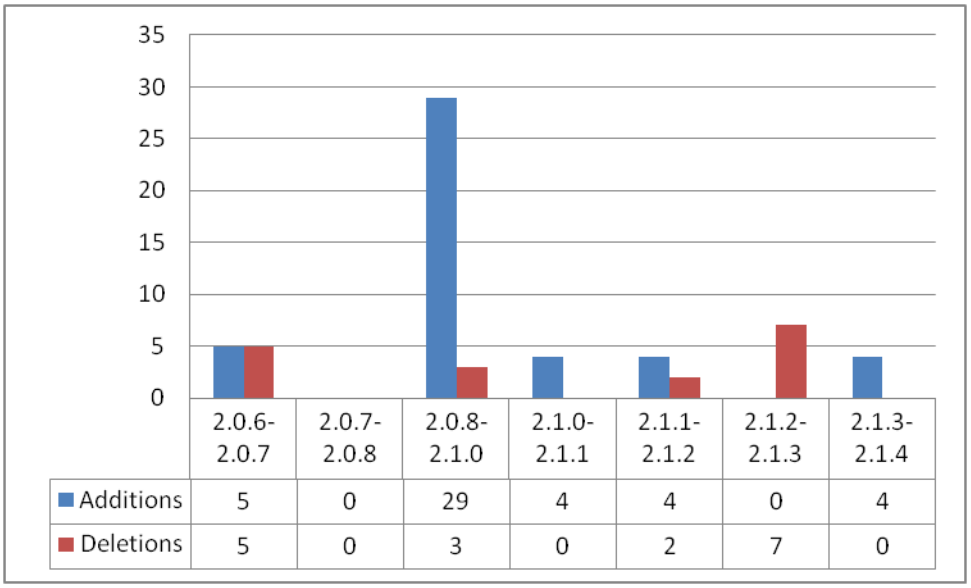
method duplicates. It helps in knowing whether the change across versions in induced by cloning or not. We have chosen program element matching at file level of granularity to check for the number of methods to aid program understanding for the programmer. Our technique is robust against file renaming and method renaming. This is due to extraction of terms from files and methods and similarity computation based on non-unique tokens.

We evaluated the proposed tool for all the class files in which there is change in number of methods. Fig. 5.4 shows the number of class files added or removed across versions of *dnsjava*.



	2.0.6-2.0.7	2.0.7-2.0.8	2.0.8-2.1.0	2.1.0-2.1.1	2.1.1-2.1.2	2.1.2-2.1.3	2.1.3-2.1.4
Additions	5	0	0	0	4	0	1
Changes	78	5	15	6	9	0	8

Fig 5.4 Number of class files added/ removed across versions



	2.0.6-2.0.7	2.0.7-2.0.8	2.0.8-2.1.0	2.1.0-2.1.1	2.1.1-2.1.2	2.1.2-2.1.3	2.1.3-2.1.4
Additions	5	0	29	4	4	0	4
Deletions	5	0	3	0	2	7	0

Fig 5.5 Number of methods added/ removed across versions

To measure the performance of the proposed technique based on all the comparisons between Java class files for two versions, we manually verified the change in number of methods for every pair of comparison. Fig. 5.5 shows the number of methods added or removed across versions of *dnsjava*. The total number of comparisons is given by the number of files which are to be compared for both the versions in the sub package of *dnsjava* we have taken for study as mentioned in Table 5.3.We chose to measure precision, recall and F-measure for the proposed technique. Is the technique working for all possible pair of files is described by recall measure? Are the generated pairs actually reflected evolutionary change is described by precision measure? Let A be the set of actual file pairs to be compared. Let D be the set of generated pairs by the technique. Let C is the set of possible candidate pairs. The

Precision = D/C

Recall = D/A

F-measure is the harmonic mean of precision and recall. It describes overall performance of the proposed technique and is calculated as:

F-measure = 1/ ((1/precision) + (1/recall))

We have 22 class files in which the number of methods has been changed. Upon manual analysis by checking software change management repositories, it is confirmed that there are 22 class files across different releases which have undergone change. So actual file pairs which are compared and possible candidate pairs is same i.e. A=22 and C=22. The technique fails to work in one case i.e. wireparseexception.java between 2.1.1 and 2.1.2 releases as there is single non-unique token only.

So Recall = 21/22=95.45

As far as precision is concerned, there are three comparisons in which the technique outputs wrong result. In two class files viz. APLRecord.java across 2.0.6 and 2.0.7 releases and ResolverConfig.java between 2.0.8 and 2.1.0 releases, result is 1 but there are changes in both the files as one new method is added in both the class files. So the tool should output less than 1. The third class file is mentioned above in which technique fails to deliver.

So Precision=19/22=86.36

F-measure = 2/ ((22/20) + (22/21))=2/2.15=0.93

In this paper, we have not extended our technique to detect changes at high level of abstraction i.e. class diagram reverse engineered from the corresponding code for *dnsjava*. This is due to large time involved in generating class diagrams for all class files using any UML modeling tool supporting reverse engineering. In future, we may wish to extend current multi-version program analysis at class diagram level representation.

5.3.3 Clones within the file at method level

We also applied the technique to trace proximity across methods in class files changed over time. It is done to detect method clones i.e. whether cloning has led to change in files over versions. Simple clones are part of conceptual clones at file level in source code. This is verified by applying the tool to detect code clones at function level of granularity within one version. We took dnsjava-2.1.3 to evaluate the effectiveness of our approach to detect function clones. We applied the technique on 30 java class files in subfolder dnsjava-2.1.3/org/xbill/dns randomly having 219 functions. Within every class file, first method is compared with the second. Undoubtedly, methods represent a coherent piece of code performing a particular function. As the technique is based on extracting common tokens within two methods, we observed the methods for which the comparison lies between 0.9 and 1.0. We checked such methods manually to verify if the score between 0.9 and 1.0 is really a clone.

We verified the output manually for a sample set of 219 functions in single version of *dnsjava*. We validated the technique by measuring the precision and recall for percentile values lying between 0.9 and 1.0. Upon comparison, we get a ranked list of scores between methods. We observed the methods for which the score lies between 0.9 and 1.0. While it is true that the applied technique doesn't compare all the methods across files with each other, yet the proximity value within first method and the next clearly depicts viability of the approach.

5.4 Discussion

- We view our system as an application in software product line analysis. Software product line is based on reusing similar concepts by applying common means of

production. Some of the studies [204] have applied clone detection in software product lines. We may apply our technique to identify similar concepts belonging to one family, thereby promoting product line design for that domain.

- The technique has got an added advantage that it is not defined for any specific modeling language but will work for all languages supporting XMI standard [239]. It also overcomes the tight integration of the UML with any modeling tool. XMI is a well known exchange representation by OMG for UML models.

- We may extend the tool to detect plagiarism in UML class models. In one typical scenario, the professor assigned the task to students to create UML class models related to same domain. Our technique can be used to measure the degree of similarity among all the class models created by students to detect the degree of plagiarism. The tool helps in exporting the *.mdl* files of all the students to XMI representation and comparing all the models with each other in one go. The degree of similarity is a clear indication of level of plagiarism.

5.5 Comparative Analysis

Clone detection in source code is an active research area and a number of techniques have been developed. But, the focus of the proposed work is to detect high level similarities in source code and UML class models. Therefore, in table 5.4, we compare the proposed technique with those existing techniques which aim to detect high level similarities/ concept clones in source code.

We understand that the goal of all the studies mentioned in table 5.4 is to detect higher level similarities but the techniques and methods used are different.

Marcus and Maletic [172]'s approach is based on examining identifiers and comments to find similar high level concepts but our technique is based on extracting non-unique tokens from input file and ignores identifiers and comments. Another work closest to ours is possibly that of Grant and Cordy [70]. Their technique applied independent component analysis and evaluated C programs at method level of granularity using raw distance metric. Our technique uses principal component analysis and works for object oriented programs at three different levels of granularity by calculating distance using cosine similarity.

Table 5.4 Comparison of proposed work with existing methods

	Marcus & Maletic [172]	**Basit & Jarzabek [18]**	**Grant & Cordy [70]**	**Proposed Method**
Intermediate representation/ transformation technique	comment removal and token regularization	Tokens	vector space representation	vector space representation
Match detection algorithm	vector representation using LSI	frequent itemset mining	independent component analysis	principal component analysis and cosine similarity
Clone granularity level	code segments, files	files, methods	methods, blocks	class diagram, class file and methods
Subject system(s)/ Language	Mosaic 2.7/ C	Eclipse Graphical Editing Framework/ Java, Eclipse Visual Editor/ Java, OpenJGraph/ Java J2ME Wireless Toolkit 2.2/Java Java Pet Store 1.3.2/ Java	Linux/ C	dnsjava/ Java

Basit and Jarzabek [18] gave the concept of structural clones which are made up of simple clones that co-exist and relate to each other. We have successfully validated our approach in detecting concept clones across 8 releases of *dnsjava* by calculating precision and recall. Detecting high level concept model clones across UML models is another novelty of our approach.

5.6 Summary

In this chapter, we have presented a new tool to detect high level similarities in source code. The tool is also capable of estimating the similarity between two UML class models. From the source code, non-unique tokens are extracted. Similarly, from the XMI

file of the UML class model, parameters like name of the class, name of attribute, name of the operation are extracted. To the best of our knowledge, this is the first attempt to apply principal component analysis to identify concept clones within single software system or across versions. Another highlight of the proposed technique is its ability to detect concept clones at different levels of granularity.

In calculating the similarity between two UML class models, the proposed technique does not take the structure of the class diagram into account. Calculation of similarity between two graphs is a lot more computationally expensive than between two vectors. We understand that considering structures, relations and node types of UML entities, we are able to fully address the challenge of model clone detection, but subgraph isomorphism is computationally demanding. Also, the difference between model and other kind of source code is that the latter is written linearly while a model is organized in two dimensions. Our approach is based on token extraction from the model and source code in vector space. Moreover, the applied technique searches for similar concepts based on extracted token matching.

Chapter 6

Conclusions and Future Work

In this thesis, we have carried out a thorough systematic literature review of software clones in general and software clone detection in particular. Systematic literature reviews are carried out in accordance with predefined search strategy and provides a thorough and complete review of existing literature. The thesis has presented a technique for clone detection in UML models. We have also devised a tool to detect high level similarities in source code and between UML class models. Section 6.1 concludes the thesis and section 6.2 discusses future scope of work.

6.1 Conclusions

We believe that the results of our systematic literature review will be useful for any researcher who wants to carry forward the research in any domain pertaining to software clones such as clone management, clone detection, clone analysis, impact of software clones on software quality, etc. Further model and semantic clone detection techniques have been investigated. We noticed that the series of the International Workshops on Detection of Software Clones have made a significant contribution towards promotion of research in the field of software clones.

Software developers in industry deal with large amounts of process and project data. So clone management tools should be scalable and integrated into development environments, to help programmers understand the behavior of cloning patterns. A clone management tool having integrated detection and developer friendly visualization of clones would help the developers observe clones as and when they proliferate during development. The tool should be able to detect real clones i.e. clones which are really interesting and useful to the developer and should not report non-useful and uninteresting clones.

There are many language specific issues which hinder code clone classification. Subjective studies carried out by several human experts vary a lot on creating reference data for creating different benchmark suits. There is disagreement between human experts as to whether the candidate code is or is not a clone. It is a definite challenge as it is a difficult and time-consuming task to manually classify the candidates as clones or not. Thus, we believe that experts from industry and academia related to diverse domains of

clone detection should come together to create a verified reference corpus of clone candidates in standard subject systems. The study should be carried out differently for each type of clone and depending upon use case. Such benchmark suites would make the results of empirical comparison consistent and reliable for use in research and industry.

There is a lack of research in cloning beyond source code. There are different software artifacts where cloning may occur. Clones do occur in requirement specifications, models and test cases too. There is an urgent need to explore the reasons for clones and efficient clone detection techniques in these software artifacts. Different artifacts have inherent characteristics which have to be exploited to apply the clone detection algorithm for that artifact. Empirical studies need to be carried out to understand the patterns in clone evolution for various artifacts. We realised the importance by proposing the techniques for clone detection for UML models. Clones detection and removal in earlier phases of software development life cycle will reduce the maintenance costs.

Software developers tend to copy the existing code into other sections of code. They may intentionally modify it to bring the desired behavior. But, if there is any bug detected in one section of code that has been copied at other places, the same bug will propagate to other sections. Software clones affect system quality. Model driven development is a standard practice in industry. Since, models are usually created before the source code, so these are closer to real world concepts. Hence, the presence of clones in models like UML models will affect the quality of the system, further.

The objective of our work is to explore UML models for the presence of clones. In our thesis, we have devised a strategy in which UML models are created using any standard modeling tool. These models are then exported to XMI representation. The XMI file is parsed to extract different attributes of the UML class model like name of the class, name of the field with data type, name of the method and return type, etc. The core of our technique is the construction of labeled, ranked tree such that its subtrees represent field and its data type and method signatures. An existing algorithm identifies various repeating subtrees of different sizes from the constructed tree. By grouping and clustering of these repeating subtrees, the proposed technique is able to detect model clones of different granularity. The novel classification of model clones provide useful insights into the software modeling practices. Another highlight of the proposed technique is its ability

to detect clones at three different levels of granularity in a UML class model i.e. complete class, attributes with their data types and methods with their signatures and cluster of such repeating attributes and methods.

It has been observed that there is clear need to address the lack of empirical studies to examine the effects of cloning in real world models. There is no open source repository of UML models of real world industrial applications due to different proprietary reasons. No UML based model clone detection tool is available for open source subject systems. So, we carried out the empirical evaluation of the proposed technique using open source and forward designed models. We firmly believe that our results will help the researchers and practitioners to compare and evaluate the model clone detection techniques in choosing the right technique based on application.

A large number of clone detection tools in the literature focus on fragments of duplicated code with smaller granularity (Chapter 2). To comprehend and understand large systems, one needs to see the bigger picture. Thus the second technique of our thesis is focused on identifying the similarity at higher levels in source code and UML models. We termed these clones of high granularity as concept clones. Since, users may choose different implementations in the form of class models for the same real world concept. Thus, we extended our technique to find similarity between two different UML class models. Detection of concept clones finds its applications in different domains like software product lines, clone evolution, software plagiarism detection, etc.

The proposed technique is able to identify concept clones by applying principal component analysis and latent semantic indexing. The distance between two fragments of source code or UML models is calculated by applying cosine similarity. It is more important to identify concept clones at abstract model level than at lower levels. So we evaluated the tool by applying it to sample UML class models down to method level within same source code file. It is important to understand code clone evolution to know whether clones are harmful or not and to know the impact of code clones on maintenance. Thus our tool is also able to estimate the extent of similarity between class files across versions of an open source software system. The results will assist the software engineer to understand as to why two different domain concepts are so close.

The major contributions of our work are:

- A systematic literature review was carried out to show the current status of clone detection, various techniques of clone detection and all allied areas.
- Proposed the classification of model clones as type-1 model clones due to standard modeling practice, type-2 model clones due to purpose and type-3 model clones due to design practices.
- Developed two model based clone detection techniques and their validation on open source subject systems.
- Both the proposed techniques have successfully detected model clones of different granularity level.

6.2 Future Scope of Work

- More studies need to be undertaken to know how harmful actually clones are. Automatic tools for investigation and visualization of clone genealogies across different versions of the software will be helpful for developers.
- The behavior and impact due to code clones is still not known precisely. The study of changeability of cloned vs non-cloned system depends on the choice of application as some applications are continuously restructured. Such studies should be empirically carried out using different types of clone detectors on large subject system base to conclude general remarks. Code fragments do not appear consistently in all revisions of the software. The code fragment skips one version and again reappears in the next revision. This non-continuous code appearing across revisions of the software need to be validated with developers' intent as to why code has been copied from the old revisions.
- The analytical results of any clone detection tool need to be complemented with empirical behavioral studies of the practitioners. The use of self learning technique and history based log techniques can be explore in the field of software clone detection.
- The reliable and scalable detection of behaviorally similar code is an open research area. In software reuse we need to identify the most relevant component for given

context. From existing software repository, developer may find many candidate components for given context, the components that are semantically same but may differ in one or other criteria such as cyclomatic complexity, algorithmic complexity, time/ space trade-off, known bugs and many more. With the help of semantic clone detection, developer will be able to find out most suitable and efficient component out of available candidate components.

- Different forms of models have individual features which need to be exploited for clone detection. We noticed vagueness in the definition of model clones. There is apparent need to classify the comparative and empirical studies differently for Matlab/Simulink models, UML models and data flow models as the application area is different. We foresee the use of multiple threads or parallel machines to speed up and distribute the task of retrieval and detection in future.
- Currently, the tool detects clones in UML class models only. In future, work can be carried out to know how type-3 clones can be used to improve the design? UML model refactoring has emerged as an allied area similar to code refactoring. Detection of patterns for model refactoring with the help of our model clone detection approach together with semantic preservation may help in improving the design and structure of the UML class model. Further, the proposed technique can be extended for other UML models.

- A mapping of UML constructs from syntactical to semantic domain may help in detecting clones having same behavior but different syntactical structure. Our work can be extended further to detect semantic similarities by using formal methods such as object constraint language, Object–Z, etc.

References

[1] E. Adar, M. Kim, SoftGUESS: Visualization and exploration of code clones in context, in: Proceedings of 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 2007, pp. 762-766.

[2] K. K. Aggarwal, Y. Singh, J. K. Chhabra, An integrated measure of software maintainability, in: Proceedings of Annual Reliability and Maintainability Symposium, Seattle, WA, 2002, pp. 235-241.

[3] K. K. Aggarwal, Y. Singh, A. Kaur, R. Malhotra, Empirical study of object-oriented metrics, Journal of Object Technology 5 (8) (2006) 149-173.

[4] K. K. Aggarwal, Y. Singh, A. Kaur, R. Malhotra, Empirical analysis for investigating the effect of object oriented metrics on fault proneness: a replicated case study. Software Process Improvement and Practice 14 (1) (2009) 39-62.

[5] R. Al-Ekram, C. Kapser, R. Holt, M. Godfrey, Cloning by accident: An empirical study of source code cloning across software systems, in: Proceedings of International Symposium on Empirical Software Engineering (ISESE'05), Noosa Heads, Australia, 2005, pp. 376-385.

[6] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, A. Stevenson, Model are Code too: Near Miss Clone Detection for Simulink Models, in: Proceedings of International Conference on Software Engineering (ICSE' 12), Zurich, Switzerland, 2012, pp. 295-304.

[7] G. Antoniol, G. Cassaza, M. Di Penta, E. Merlo, Modeling clones evolution through time series, in: Proceedings of the 17th International Conference on Software Maintenance (ICSM '01), 2001, pp. 273-280.

[8] G. Antoniol, U. Villano, E. Merlo, M. Di Penta, Analyzing cloning evolution in the Linux kernel, Information and Software Technology 44 (13) (2002) 755-765.

[9] E. P. Antony, M. H. Alalfi, J. R. Cordy, An approach to clone detection in behavioural models, in: Proceedings of 20th Working Conference on Reverse Engineering (WCRE'13), Koblenz-Landau, Germany, 2013, pp. 472-476.

[10] L. Aversano, G. Canfora, A. De Lucia, P. Gallucci, Web site reuse: Cloning and adapting, in: Proceedings of the 3rd International Workshop on Web Site Evolution (WSE ’01), Florence, Italy, 2001, pp. 107-111.

[11] L. Aversano, L. Cerulo, M. Di Penta, How clones are maintained: An empirical study, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, Amsterdam, The Netherlands, 2007, pp. 81-90.

[12] B. Baker, On finding duplication and near-duplication in large software systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE'95), Toronto, Ontario, Canada, 1995, pp. 86-95.

[13] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, K. Kontogiannis, Measuring clone based reengineering opportunities, in: Proceedings of the 6th International Software Metrics Symposium (METRICS'99), Boca Raton, Florida, USA, 1999, pp. 292-303.

[14] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, K. Kontogiannis, Advanced clone-analysis to support object-oriented system refactoring, in: Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00), Brisbane, Queensland, Australia, 2000, pp. 98-107.

[15] M. Balint, T. Gîrba, R. Marinescu, How developers copy, in: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC ’06), Athens, Greece, 2006, pp. 56-68.

[16] L. Barbour, H. Yuan, Y. Zou, A technique for just-in time clone detection, in: Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC ’10), Washington DC, USA, 2010, pp. 76-79.

[17] H. Basit, S. Puglisi, W. Smyth, A. Turpin, S. Jarzabek, Efficient token based clone detection with flexible tokenization, in: Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '07), Dubrovnik, Croatia, 2007, pp. 513-515.

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

[19] H. Basit, U. Ali, S. Jarzabek, Viewing simple clones from a structural clones' perspective, in: Proceedings of 5th International Workshop on Software Clones, Honolulu ,USA, 2011, pp.1-8

[20] Project Bauhaus. Last Accessed April 2012. URL http://www.bauhaus-stuggart.de

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

[22] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, A. E. Hassan, An empirical study on inconsistent changes to code clones at the release level, Science of Computer Programming 74 (7) (2010) 1 - 17.

[23] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the 14th International Conference on Software Maintenance (ICSM '98), Bethesda, Maryland, USA, 1998, pp. 368-378.

[24] M. W. Berry, Z. Drmac, E. R. Jessup, Matrices, Vector Spaces, and Information Retrieval, Society for Industrial and Applied Mathematics Review 41 (2) (1999) 335-362.

[25] Y. Bian, G. Koru, X. Su, P. Ma, SPAPE: A semantic-preserving amorphous procedure extraction method for near-miss clones, Journal of Systems and Software 86(8) (2013) 2077-2093.

[26] B. Biegel, S. Diehl, Highly configurable and extensible code clone detection, in: Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10), Beverly, MA, USA, 2010, pp. 237–241.

[27] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, M. Khalil, Lessons from applying the systematic literature review process within the software engineering domain, The Journal of Systems and Software 80 (4) (2007) 571-583.

[28] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, R. Robbes, Language independent clone detection applied to plagiarism detection, in: Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '10), Timisoara, Romania, 2010, pp. 77-86.

[29] C. Brown, S. Thompson, Clone detection and elimination for Haskell, in: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM ’10), Madrid, Spain, 2010, pp. 111-120.

[30] M. Bruntink, A. van Deursen, R. van Engelen, T. Tourwe, On the use of clone detection for identifying crosscutting concern code, IEEE Transactions on Software Engineering 31 (10) (2005) 804-818.

[31] P. Bulychev, M. Minea, Duplicate code detection using anti-unification, in: Proceedings of Spring/Summer Young Researchers' Colloquium on Software Engineering, St. Petersburg, Russia, 2008, pp. 51-54.

[32] E. Burd, J. Bailey, Evaluating clone detection tools for use during preventative maintenance, in: Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02), Montreal, Canada, 2002, pp. 36-43.

[33] G. Cassaza, G. Antoniol, U. Villano, E. Merlo, M. Di Penta, Identifying clones in the Linux Kernel, in: Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'01), Florence, Italy, 2001, pp. 90-97.

[34] M. Chilowicz, É. Duris, G. Roussel, Finding similarities in source code through factorization, Electronic Notes in Theoretical Computer Science 238 (5) (2009) 47-62.

[35] M. Chilowicz, É. Duris, G. Roussel, Syntax tree fingerprinting for source code similarity detection, in: Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC ’09), Vancouver, British Columbia, Canada, 2009, pp. 243-247.

[36] S. Choi, H. Park, H. Lim, T. Han, A static API birthmark for Windows binary executables, The Journal of Systems and software 82 (5) (2009) 862-873.

[37] M. Christou, M. Crochemore, T. Flouri, C. S. Iliopoulos, J. Janoušek, B. Melicha, S. P. Pissis, Computing all Subtree Repeats in Ordered Trees, Information Processing Letters 112 (24) (2012) 958-962.

[38] P. Ciancarini, G. P. Favini, Detecting clones in game playing software, Entertainment Computing 1 (2009) 9-15.

[39] G. D. Clifford, Singular Value Decomposition & Independent Component Analysis for Blind Source Separation, Biomedical Signal and Image Processing (2005) pp. 49.

[40] A. Corazza, S. D. Martino, V. Maggio, G. Scanniello, A tree kernel based approach for clone detection, in: Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10), Timisoara, Romania, 2010, pp. 1-5.

[41] J. R. Cordy, T. R. Dean, N. Synytskyy, Practical language-independent detection of near-miss clones, in: Proceedings of the 14th IBM Centre for Advanced Studies Conference (CASCON'04), Toronto, Ontario, Canada, 2004, pp. 1–12.

[42] M. Dagenais, E. Merlo, B. Laguë, D. Proulx, Clones occurrence in large object oriented software packages, in: Proceedings of the 8th IBM Centre for Advanced Studies Conference (CASCON'98), Toronto, Ontario, Canada, 1998, pp. 192-200.

[43] I. J. Davis, M. W. Godfrey, From whence it came: Detecting source code clones by analyzing assembler, in: Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10), Beverly, MA, USA, 2010, pp. 242-246.

[44] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, R. Harshman, Indexing by latent semantic analysis, Journal of the American Society for Information Science 41(6) (1990) 391–407.

[45] F. Deissenboeck, B. Hummel, E. Juergens, B. Schätz, S. Wagner, J. Girard, S. Teuchert, Clone detection in automotive model-based development, in: Proceedings of 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, 2008, pp. 603-612.

[46] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, P. Mas, M. Pizka, Tool support for continuous quality control, IEEE Software 25 (5) (2008) 60-67.

[47] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, B. Schaetz, Model clone detection in practice, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 37-44.

[48] DNS Java < http://www.xbill.org/dnsjava/> (2014).

[49] C. Domann, E. Juergens, J. Streit, The curse of copy & paste- cloning in requirements specifications, in: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, Florida, USA, 2009, pp. 443-446.

[50] E. Duala-Ekoko, M. Robillard, CloneTracker: Tool support for code clone management, in: Proceedings of 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, 2008, pp. 843-846.

[51] E. Duala-Ekoko, M. Robillard, Clone Region Descriptors: Representing and tracking duplication in source code, ACM Transactions on Software Engineering and Methodology, 20 (1) (2010) 1-31.

[52] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance (ICSM '99), Oxford, England, UK, 1999, pp. 109-119.

[53] S. Ducasse, O. Nierstrasz, M. Rieger, On the effectiveness of clone detection by string matching, Journal on Software Maintenance and Evolution: Research and Practice 18 (1) (2006) 37-58.

[54] W. S. Evans, C. W. Fraser, F. Ma, Clone detection via structural abstraction, Software Quality Journal 17 (4) (2009) 309-330.

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

[56] R. Fanta, V. Rajlich, Removing Clones from the Code, Journal of Software Maintenance: Research and Practice 11 (4) (1999) 223-243.

[57] A. M. Fernández-Sáez, M. R. V. Chaudron, M. Genero, I. Ramos, Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?: a controlled experiment, in: Proceedings of International Conference on Evaluation and Assessment in Software Engineering (EASE' 13), Porto de Galinhas, Brazil, 2013, 60-71.

[58] M. Fowler, K. Beck, T. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the design of existing code, Addison –Wesley Longman, 1999.

[59] Y. Fukushima, R. Kula, S. Kawaguchi, K. Fushida, M. Nagura, H. Iida, Code clone graph metrics for detecting diffused code clones, in: Proceedings of the 16th Asia Pacific Software Engineering Conference (APSEC'09), Penang, Malaysia, 2009, pp. 373-380.

[60] M. Gabel, L. Jiang, Z. Su, Scalable detection of semantic clones, in: Proceedings of 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, 2008, pp. 321-330.

[61] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, Z. Su, Scalable and systematic detection of buggy inconsistencies in source code, in: Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications, Nevada, USA, 2010, pp. 175-190.

[62] D. M. German, M. Di Penta, Y.-G Gueheneuc, G. Antoniol, Code siblings: technical and legal implications of copying code between applications, in: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR'09), Vancouver, BC, Canada, 2009, pp. 81-90.

[63] D. Gitchell, N. Tran, Sim: a utility for detecting similarity in computer programs, ACM SIGCSE Bulletin 31 (1) (1999) 266-270.

[64] N. Göde, R. Koschke, Incremental clone detection, in: Proceedings of the 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, 2009, pp. 219-228.

[65] N. Göde, Evolution of Type-1 clones, in: Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09), Edmonton, Canada, 2009, pp. 77-86.

[66] N. Göde, Clone Removal: Fact or Fiction, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 22-40.

[67] N. Göde, J. Harder, Clone Stability, in: Proceedings of 15th European Conference on Software Maintenance and Reengineering, Oldenburg, Germany, 2011, pp. 65-74.

[68] N. Göde, B. Hummel, E. Juergens, What Clone Coverage Can Tell, in: Proceedings of International Workshop on Software Clones (IWSC'12), Zurich, Switzerland, 2012, pp. 90-91.

[69] N. Gold, J. Krinke, M. Harman, D. Binkley, Issues in Clone Classification for Data flow Languages, in: Proceedings of 4th International Workshop on Software Clones (IWSC'10), Cape Town, South Africa, 2010, pp. 83-84.

[70] S. Grant, J. R. Cordy, Vector space analysis of software clones, in: Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC '09), Vancouver, BC, Canada, 2009, pp. 233-237.

[71] G. Guerrini, M. Mesiti, I. Sanz, An overview of similarity measures for clustering XML documents, in: Proceedings of Web Data Management Practices: Emerging Techniques and Technologies, A. Vakali and G. Pallis, Eds., PA, USA, IGI Global, 2007, pp. 56-78.

[72] J. Guo Y. Zou, Detecting clones in business applications, in: Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08), Antwerp, Belgium, 2008, pp. 91-100.

[73] J. Han, M. Kamber, Data Mining: Concepts and Techniques, 2nd edition, Morgan Kaufmann, (2006).

[74] M. Harman, Why source code analysis and manipulation will always be important, Keynote address at 10th International Working Conference on Source Code Analysis and Manipulation (SCAM'10), Timisoara, Romania, 2010.

[75] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, On software maintenance process improvement based on code clone analysis, in: Proceedings of the 4th International Conference on Product Focused Software Process Improvement (PROFES '02), Rovaniemi, Finland, 2002, pp. 185-197.

[76] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, ARIES: Refactoring support environment based on code clone analysis, in: Proceedings of the 8th IASTED International Conference on Software Engineering and Applications, MA, USA, 2004, pp. 222-229.

[77] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, Method and implementation for investigating code clones in a software system, Information and Software Technology 49 (9-10) (2007) 985-998.

[78] Y. Higo, S. Kusumoto, K. Inoue, A metric based approach to identifying refactoring opportunities for merging code clones in a Java software system, Journal of Software Maintenance and Evolution: Research and Practice 20 (6) (2008) 435-461.

[79] Y. Higo, K. Sawa, S. Kusumoto, Problematic code clones identification using multiple detection results, in: Proceedings of the 16th Asia Pacific Software Engineering Conference (APSEC'09), Penang, Malaysia, 2009, pp. 365-372.

[80] Y. Higo, S. Kusumoto, Code clone detection on specialized PDG's with heuristics, in: Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11), Oldenburg, Germany, 2011, pp. 75-84.

[81] S. Horwitz, Identifying the semantic and textual differences between two versions of a program, in: Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI'90), White Plains, New York, 1990, pp. 234-245.

[82] D. Hou, F. Jacob, P. Jablonski, Exploring the design space of proactive tool support for copy-and-paste programming, in: Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON '09), Ontario, Canada, 2009, pp. 188-202.

[83] D. Hou, P. Jablonski, F. Jacob, CnP: Towards an environment for the proactive management of copy-and-paste programming, in: Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC '09), Vancouver, BC, Canada, 2009, pp. 238-242.

[84] B. Hummel, E. Juergens, L. Heinemann, M. Conradt, Index-based code clone detection: Incremental, distributed, scalable, in: Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10), Timisoara, Romania, 2010, pp. 1-9.

[85] B. Hummel, E. Juergens, D. Steidl, Index-based model clone detection, in: Proceedings of 5th International Workshop on Software Clones, Honolulu ,USA, 2011, pp. 21-27.

[86] T. Imai, Y. Kataoka, T. Fukaya, Evaluating software maintenance cost using functional redundancy metrics, in: Proceedings of 26th Annual International Computer Software and Applications (COMPSAC ’02), Oxford, England, 2002, pp. 299-306.

[87] P. Jablonski, D. Hou, CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE, in: Proceedings of Eclipse Technology Exchange Workshop at OOPSLA 2007 (ETX'07), Montreal, Quebec, Canada, 2007, p. 5.

[88] P. Jablonski, D. Hou, Aiding software maintenance with copy and paste clone awareness, in: Proceedings of the 18th IEEE International Conference on Program Comprehension (ICPC ’10), Washington DC, USA, 2010, pp. 170-179.

[89] F. Jacob, D. Hou, P. Jablonski, Actively comparing clones inside the code editor, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 1-8.

[90] K. Jalbert, J. S. Bradbury, Using clone detection to identify bugs in concurrent software, in: Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10), Timisoara, Romania, 2010, pp. 1-5.

[91] S. Jarzabek, S. Li, Unifying clones with a generative programming technique: A case study, Journal of Software Maintenance and Evolution: Research and Practice, John Wiley & Sons 18 (4) (2006) 267-292.

[92] M. Jehanzeb, G. Bin Sulong, I. Siddiqi, Improving codebook-based writer recognition, International Journal of Pattern Recognition and Artificial Intelligence 27 (6) (2013), 1353003 pp. 18.

[93] Z. M. Jiang, A. E. Hassan, R. C. Holt, Visualizing clone cohesion and coupling, in: Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC'06), Bangalore, India, 2006, pp. 467-476.

[94] L. Jiang, Z. Su, E. Chiu, Context-based detection of clone-related bugs, in: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '07), Dubrovnik, Croatia, 2007, pp. 55-64.

[95] L. Jiang, G. Misherghi, Z. Su, S. Glondu, DECKARD: Scalable and accurate tree-based detection of code clones, in: Proceedings of 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 2007, pp. 96-105.

[96] Z. M. Jiang, A. E. Hassan, A framework for studying clones in large software systems, in: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07), Paris, France, 2007, pp. 203-212.

[97] L. Jiang, Z. Su, Automatic mining of functionally equivalent code fragments via random testing, in: Proceedings of 18th International Symposium on Software Testing and Analysis (ISSTA'09), Chicago, Illinois, USA, 2009, pp. 81-92.

[98] Jian-lin Huang, Fei-peng Li, Quick similarity measurement of source code based on suffix array, in: Proceedings of International Conference on Computational Intelligence and Security, Beijing, China, 2009, pp. 308-311.

[99] J. H. Johnson, Substring matching for clone detection and change tracking, in: Proceedings of the 10th International Conference on Software Maintenance, Victoria, British Columbia, Canada, 1994, pp. 120-126.

[100] E. Juergens, F. Deissenboeck, B. Hummel, CloneDetective – A workbench for clone detection research, in: Proceedings of 31st International Conference on Software Engineering (ICSE'09), Vancouver, Canada, 2009, pp. 603-606.

[101] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter? In: Proceedings of 31st International Conference on Software Engineering (ICSE'09), Vancouver, Canada, 2009, pp. 485-495.

[102] E. Juergens, N. Göde, Achieving accurate clone detection results, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 1-8.

[103]E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, J, Streit, Can clone detection support quality assessments of requirement specifications? in: Proceedings of 32nd International Conference on Software Engineering (ICSE'10), Cape Town, South Africa, 2010, pp. 79-88.

[104]E. Juergens, F. Deissenboeck, B. Hummel, Code similarities beyond copy & paste, in: Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR ’10), Madrid, Spain, 2010, pp. 78-87.

[105]E. Juergens, Research in cloning beyond code: a first roadmap, in: Proceedings of 5th International Workshop on Software Clones, Honolulu, USA, 2011, pp. 67-68.

[106]W. Jung, C. Wu, E. Lee, WSIM: Detecting clone pages based on 3-levels of similarity clues, in: Proceedings of 9th IEEE/ACIS International Conference on Computer and Information Sciences, Yamagata, Japan, 2010, pp. 702-707.

[107]T. Kamiya, F. Ohata, K. Kundou, S. Kusumoto, K. Inoue, Maintenance support tools for JAVA Programs: CCFinder and JAAT, in: Proceedings of 23rd International Conference on Software Engineering (ICSE’01), Toronto, Ontario, Canada, 2001, pp. 837-838.

[108]T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: A multi-linguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654-670.

[109]T. Kamiya, The Official CCFinderX website. Last Accessed April 2012. URL http://www.ccfinder.net

[110]T. Kamiya, C. Ghezzi, How code skips over revisions, in: Proceedings of 5th International Workshop on Software Clones, Honolulu, USA, 2011, pp. 69-70.

[111]C. J. Kapser, M. W. Godfrey, Aiding comprehension of cloning through categorization, in: Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE'04), Kyoto, Japan, 2004, pp. 85-94.

[112]C. J. Kapser, M. W. Godfrey, Improved tool support for the investigation of duplication in software, in: Proceedings of the 21st International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, 2005, pp. 305-314.

[113] C. J. Kapser, M. W. Godfrey, Supporting the analysis of clones in software systems: A case study, Journal of Software Maintenance and Evolution: Research and Practice 18 (2) (2006) 61-82.

[114] C. J. Kapser, P. Anderson, M. Godfrey, R. Koschke, M. Rieger, F. van Rysselberghe, P. Weisgerber, Subjectivity in clone judgment: Can we ever agree? In Duplication, Redundancy, and Similarity in Software, Dagstuhl Seminar Proceedings, 2007.

[115] C. J. Kapser, M. W. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software, Empirical Software Engineering 13 (6) (2008) 645-692.

[116] S. Kawaguchi, P. K. Garg, M. Matsushita, K. Inoue, Automatic categorization algorithm for evolvable software archive, in: Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE'03), Helsinki, Finland, 2003, pp. 195-200.

[117] I. Keivanloo, C. K. Roy, J. Rilling, SeByte: Scalable clone and similarity search for bytecode, Science of Computer Programming 95(4) (2014) 426-444.

[118] I. Keivanloo, J. Rilling, Software trustworthiness 2.0: A semantic web enabled global source code analysis approach, The Journal of Systems and Software 89 (2014) 33-50.

[119] U. Kelte, J. Wehren, J. Niere, A generic difference algorithm for UML models, in: Proceedings of Software Engineering 2005 (SE' 05), Innsbruck, Austria, 2005, pp. 105-116.

[120] M. Kim, L. Bergman, T. Lau, D. Notkin, An Ethnographic study of copy and paste programming practices in OOPL, in: Proceedings of 3rd International ACM-IEEE Symposium on Empirical Software Engineering (ISESE '04), Redondo Beach, CA, USA, 2004, pp. 83- 92.

[121] M. Kim, D. Notkin, Using a clone genealogy extractor for understanding and supporting evolution of code clones, in: Proceedings of the 2nd International

Workshop on Mining Software Repositories (MSR'05), Saint Louis, Missouri, USA, 2005, pp. 1-5.

[122] M. Kim, V. Sazawal, D. Notkin, G. C. Murphy, An Empirical study of code clone genealogies, in: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/SIGSOFT FSE 2005 '05), Lisbon, Portugal, 2005, pp. 187-196.

[123] H. Kim, Y. Jung, S. Kim, and K. Yi, MeCC: Memory comparison-based clone detector, in: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), Honolulu, Hawaii, 2011, pp. 301-310.

[124] M. Kim, Understanding and aiding code evolution by inferring change patterns, in: Proceedings of 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 2007, pp. 101-102.

[125] B. A. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, School of Computer Science and Mathematics, Keele University, Keele and Department of Computer Science, University of Durham, Durham, UK, 2007, p. 65.

[126] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman, Systematic literature reviews in software engineering – A systematic literature review, Information and Software Technology 51 (1) (2009) 7-15.

[127] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, B. V. Saranya, Detection of Type-1 and Type-2 code clones using textual analysis and metrics, in: Proceedings of 2010 International Conference on Recent Trends in Information, Telecommunication and Computing, Kochi, Kerala, India, 2010, pp. 241-243.

[128] E. Kodhai, S. Kanmani, Method-level code clone detection through LWH (Light Weight Hybrid) approach, Software Engineering Research and Development 2(12) (2012) 1- 29.

[129] S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, P. Van Gorp, Evaluation of model transformation approaches for model refactoring, Science of Computer Programming 85 (2014) 5- 40.

[130] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis (SAS' 01), Vol. LNCS 2126, Paris, France, 2001, pp. 40-56.

[131] K. Kontogiannis, Partial design recovery using dynamic programming, in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON ’94), Toronto, Ontario, Canada, 2004, pp. 34.

[132] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, M. Bernstein, Pattern matching for clone and concept detection, Automated Software Engineering 3 (1-2) (1996) 77-108.

[133] K. Kontogiannis, Evaluation experiments on the detection of programming patterns using software metrics, in: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'97), Amsterdam, The Netherlands, 1997, pp. 44-54.

[134] R. Koschke, R. Falke, P. Frenzel, Clone detection using abstract syntax suffix trees, in: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06), Benevento, Italy, 2006, pp. 253-262.

[135] R. Koschke, Survey of research on software clones, in: Duplication, Redundancy, and Similarity in Software, Dagstuhl Seminar Proceedings, 2007, p. 24.

[136] D. Kozlov, J. Koskinen, M. Sakkinen, J. Markkula, Exploratory analysis of the relations between code cloning and open source software quality, in: Proceedings of 7th International Conference on the Quality of Information and Communications Technology, Porto, Portugal, 2010, pp. 358-363.

[137] J. Krinke, Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01), Stuttgart, Germany, 2001, pp. 301-309.

[138] J. Krinke, A study of consistent and inconsistent changes to code clones, in: Proceedings of the 14th Working Conference on Reverse Engineering (WCRE'07), Vancouver, BC, Canada, 2007, pp. 170-178.

[139] J. Krinke, N. Gold, Y. Jia, D. Binkley, Distinguishing copies from originals in software clones, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 41-48.

[140] J. Krinke, N. Gold, Y. Jia, D. Binkley, Cloning and Copying between GNOME projects, in: Proceedings of 7th IEEE International Conference on Mining Software Repositories, Cape Town, SA, 2010, pp. 98-101.

[141] J. Krinke, Is cloned code more stable than non-cloned code, in: Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '08), Beijing, China, 2008, pp. 57-66.

[142] J. Krinke, Is cloned code older than non-cloned code? in: Proceedings of 5th International Workshop on Software Clones, Honolulu ,USA, 2011, pp. 28-33.

[143] V. Kulkarni, S. Reddy, A. Rajbojh, Scaling up Model Driven Engineering – experience and lessons learnt, in: Proceedings of 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10), Oslo, Norway, pp. 331-345.

[144] B. Laguë, D. Proulx, J. Mayrand, E. Merlo, J. Hudepohl, Assessing the benefits of incorporating function clone detection in a development process, In: Proceedings of the 13th International Conference on Software Maintenance (ICSM '97), Bari, Italy, 1997, pp. 314-321.

[145] A. Lakhotia, J. Li, A. Walenstein, Y. Yang, Towards a clone detection benchmark suite and results archive, in: Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, Oregon, USA, 2003, 285-286.

[146] T. K. Landauer, P. W. Foltz, D. Laham, Introduction to latent semantic analysis, Discourse Processes 25 (1998) 259-284.

[147]F. Lanubile, T. Mallardo, Finding function clones in web applications, in: Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR ’03), Benevento, Italy, 2003, pp. 379-386.

[148]T. Lavoie, M. Eilers-Smith, E. Merlo, Challenging cloning related problems with GPU-based algorithms, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 25-32.

[149]T. Lavoie, E. Merlo, Automated type-3 clone oracle using Levenshtein metric, in: Proceedings of 5th International Workshop on Software Clones, Honolulu, USA, 2011, pp. 34-40.

[150]S. Lee, I. Jeong, SDD: High performance code clone detection system for large scale source code, in: Proceedings of the Object Oriented Programming Systems Languages and Applications Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA Companion '05), San Diego, CA, USA, 2005, pp. 140-141.

[151]H. Lee, K. Doh, Tree-pattern-based duplicate code detection, in: Proceedings of International Workshop on Data-intensive Software Management and Mining, Philadelphia, PA, USA, 2009, pp. 7-12.

[152] S. Lee, G. Bae, H. S. Chae, D-H Bae, Y. R. Kwon, Automated scheduling for clone-based refactoring using a competent GA, Software – Practice and Experience 41(5) (2011) 521-550.

[153]Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, S. Kim, Instant code clone search, in: Proceedings of 18th International Symposium on Foundations of Software Engineering, NY, USA, 2010, 167-176.

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

[155]H. Li, S. Thompson, Clone detection and removal for Erlang/OPT within a refactoring environment, in: Proceedings of ACM SIGPLAN Workshop on Partial

Evaluation and Program Manipulation (PEPM ’09), Savannah, GA, USA, 2009, pp. 169-178.

[156] Z. O. Li, J. Sun, A metric space based software clone detection approach, in: Proceedings of 2nd International Conference on Software Engineering and Data Mining, Chengdu, China, 2010, pp. 111-116.

[157] J. Liu, J. T. L. Wang, W. Hsu, K. G. Herbert, XML clustering by principal component analysis, in: Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence, Boca Raton, FL, USA, 2004, pp. 658-662.

[158] H. Liu, Z. Ma, L. Zhang, W. Shao, Detecting duplications in sequence diagrams based on suffix trees, in: Proceedings 13th Asia-Pacific Software Engineering Conference (APSEC'06), Bangalore, India, 2006, pp. 269-276.

[159] S. Livieri, Y. Higo, M. Matsushita, K. Inoue, Analysis of the Linux kernel evolution using code clone coverage, in: Proceedings of the 4th International Workshop on Mining Software Repositories (MSR'07), Minneapolis, MN, USA, 2007, pp. 22.

[160] S. Livieri, Y. Higo, M. Matsushita, K. Inoue, Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder, in: Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 2007, pp. 106-115.

[161] A. Lozano, M. Wermelinger, B. Nuseibeh, Evaluating the harmfulness of cloning: A change based experiment, in: Proceedings of the 4th International Workshop on Mining Software Repositories (MSR'07), Minneapolis, MN, USA, 2007, pp. 4.

[162] A. Lozano, M. Wermelinger, Assessing the effects of clones on changeability, in: Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM'08), Beijing, China, 2008, pp. 227-236.

[163] A. Lozano, M. Wermelinger, Tracking clones’ imprint, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 65-72.

[164] G. A. Lucca, M. Di Penta, A. R. Fasolino, An approach to identify duplicated web pages, in: Proceedings of the 26th International Computer Software and Applications Conference (COMPSAC'02), Oxford, England, 2002, pp. 481-486.

[165] A. Lucia, R. Francese, G. Scanniello, G. Tortora, Reengineering web applications based on cloned pattern analysis, in: Proceedings of 12th International Workshop on Program Comprehension (IWPC'04), Bari, Italy, 2004, pp. 132-141.

[166] A. Lucia, R. Francese, G. Scanniello, G. Tortora, Understanding cloned patterns in web applications, in: Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05), St. Louis, MO, USA, 2005, pp. 333-336.

[167] A. Lucia, M. Risi, G. Tortora, G. Scanniello, Clustering Algorithms and latent semantic indexing to indentify similar pages in web applications, in: Proceedings of the 9th International Workshop on Web Site Evolution (WSE ’07), Paris, France, 2007, pp. 65-72.

[168] Lucia, D. Lo, L. Jiang, A. Budi, Active Refinement of Clone Anomaly Reports, in: Proceedings of International Conference on Software Engineering (ICSE’12), Zurich, Switzerland, 2012, pp. 397-407.

[169] D. C. Luckham, J. Vera, S. Meldal, Key Concepts in Architecture Definition Languages, Invited paper in G. T. Leavens, and M. Sitaraman, editors, *Foundations of Component-Based Systems*, Cambridge University Press, 2000, pp. 23-45.

[170] K. Maeda, Syntax sensitive and language independent detection of code clones. World Academy of Science, Engineering and Technology 60 (2009) 350-354.

[171] A. Y. Mao, J. R. Cordy, T. R. Dean, Automated conversion of table-based websites to structured stylesheets using table recognition and clone detection, in: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research (CASCON ’07), Richmond Hill, Ontario, Canada, 2007, pp. 12-26.

[172] A. Marcus, J. I. Maletic, Identification of high-level concept clones in source code, in: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01), San Diego, CA, USA, 2001, pp. 107-114.

[173] D. Martin, J. R. Cordy, Analyzing web service similarities using contextual clones, in: Proceedings of 5th International Workshop on Software Clones, Honolulu, USA, 2011, pp. 41-46.

[174] J. Mayrand, C. Leblanc, E. M. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in: Proceedings of the 12th International Conference on Software Maintenance (ICSM'96), Monterey, CA, USA, 1996, pp. 244-253.

[175] A. Monden, D. Nakae, T. Kamiya, S. Sato, K. Matsumoto, Software quality analysis by code clones in industrial legacy software, in: Proceedings of 8th IEEE International Symposium on Software Metrics (METRICS'02), Ottawa, Canada, 2002, pp. 87-94.

[176] A. Monden, S. Okahara, Y. Manabe, K. Matsumoto, Guilty or not guilty: Using clone metrics to determine open source licensing violations, IEEE Software 28 (2) (2011) 42-47.

[177] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, P. Zave, Matching and merging of statecharts specifications, in: Proceedings of International Conference on Software Engineering (ICSE' 07), Minneapolis, MN, 2007, pp. 54–64.

[178] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, T. N. Nguyen, Cleman: Comprehensive clone group evolution management, in: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, Italy, 2008, pp. 451-454.

[179] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, T. N. Nguyen, Clone-aware configuration management, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, 2009, pp. 123-134.

[180] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, T. N. Nguyen, ClemanX: Incremental clone detection tool for evolving software, in: Proceedings of 31st International Conference on Software Engineering (ICSE'09), Vancouver, Canada, 2009, pp. 437-438.

[181] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, T. N. Nguyen, Scalable and incremental clone detection for evolving software, in: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09), Edmonton, AB, 2009, pp. 491-494.

[182]J. R. Pate, R. Tairas, N. A. Craft, Clone evolution: a systematic review, Technical Report SERG-2010-01, University of Alabama, Alabama, USA, 2010, p. 20.

[183]J. –F. Patenaude, E. Merlo, M. Dagenais, B. Laguë, Extending software quality assessment techniques to Java systems, in: Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99), Pittsburgh, PA, 1999, pp. 49-56.

[184] M. Di Penta, Evolution Doctor: A framework to control software system evolution, in: Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR ’05), 2005, pp. 280-283.

[185]M. Di Penta, M. Neteler, G. Antoniol, E. Merlo, A language independent software renovation framework, The Journal of Systems and Software 77 (3) (2005) 225-240.

[186]A. Perumal, S. Kanmani, E. Kodhai, Extracting the similarity in detected software clones using metrics, in: Proceedings of International Conference on Computer and Communication Technology, 2010, Allahabad, Uttar Pradesh, India, pp. 575-579.

[187]N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, T. N. Nguyen, Complete and accurate clone detection in graph based models, in: Proceedings of 31st International Conference on Software Engineering (ICSE'09), Vancouver, Canada, 2009, pp. 276-286.

[188]W. Qu, Y. Jia, M. Jiang, Pattern mining of cloned codes in software systems, Information Sciences, 180 (2010) 1 -11.

[189]F. Rahman, C. Bird, P. Devanbu, Clones: What is that smell, in: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories, Cape Town, South Africa, 2010, pp. 72-81.

[190]D. Rajapakse, S. Jarzabek, Using server pages to unify clones in web applications: A trade-off analysis, in: Proceedings of the 29th International Conference of Software Engineering (ICSE'07), Minneapolis, USA, 2007, pp. 116-126.

[191]C. K. Roy, J. R. Cordy, A survey on software clone detection research, Technical Report 2007-541, Queen's University at Kingston Ontario, Canada, 2007, p. 115.

[192]C. K. Roy, J. R. Cordy, Scenario-based comparison of clone detection techniques, in: Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC ’08), Amsterdam, The Netherlands, 2008, pp. 153-162.

[193]C. K. Roy, J. R. Cordy, A Mutation / Injection-based automatic framework for evaluating code clone detection tools, in: Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshops, Denver, Colorado, USA, 2009, pp. 157-166.

[194]C. K. Roy, Detection and analysis of near miss software clones, in: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09), Edmonton, AB, 2009, pp. 447-450.

[195]C. K. Roy, J. R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming 74 (7) (2009) 470-495.

[196]C. K. Roy, J. R. Cordy, Near miss function clones in open source software: an empirical study, Journal of Software Maintenance and Evolution: Research and Practice 22 (3) (2010) 165-189.

[197]C. K. Roy, J. R. Cordy, Are scripting languages really different, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 17-24.

[198] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design. Addison Wesley, 1991.

[199] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual. Addison Wesley, 1999.

[200]A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan, Z. Su, Detecting code clones in binary executable, in: Proceedings of International Symposium on Software Testing and Analysis, Chicago, Illinois, USA, 2009, pp. 117-127.

[201]R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, K. A. Schneider, Evaluating code clone genealogies at release level: An empirical study, in: Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM ’10), Timisoara, Romania, 2010, pp. 87-96.

[202] G. Salton, M. J. McGill, Introduction to Modern Information Retrieval, McGraw Hill, (1986).

[203] Y. Sasaki, T. Yamamoto, Y. Hayase, K. Inoue, Finding file clones in FreeBSD ports collection, in: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories, Cape Town, South Africa, 2010, pp. 102-105.

[204] S. Schulze, S. Apel, C. Kastner, Code clones in feature oriented software product lines, in: Proceedings of Generative Programming and Component Engineering (GPCE'10), Eindhoven, The Netherlands, 2010, pp. 103-112.

[205] P. Schugerl, Scalable clone detection using description logic, in: Proceedings of 5th International Workshop on Software Clones, Honolulu, USA, 2011, pp. 47-53.

[206] Gehan M. K. Selim, K. C. Foo, Y. Zou, Enhancing source based clone detection using intermediate representation, in: Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10), Beverly, MA, USA, 2010, pp. 227-236.

[207] Gehan M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, Y. Zou, Studying the impact of clones on software defects, in: Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10), Beverly, MA, USA, 2010, pp. 13-21.

[208] D. M. Shawky, A. F. Ali, An Approach for assessing similarity metrics used in metric based clone detection techniques, in: Proceedings on 3rd IEEE International Conference on Computer Science and Information Technology, Chengdu, China, 2010, pp. 580-584.

[209] D. M. Shawky, A. F. Ali, Modeling clones evolution in open source systems through chaos theory, in: Proceedings on 2nd International Conference on Software Technology and Engineering, San Juan, Puerto Rico, 2010, pp. VI-159 – VI-164.

[210] Tool Simian, Last Accessed April 2012. URL http://www.harukizaemon.com/simian/index.html.

[211] Tool SimScan, Last Accessed April 2012. URL http://www.blue-edge.bg/download.html.

[212] M. Stephan, J. R. Cordy, A Survey of methods and applications of model comparison, Tech. Rep. #2011-582, Queen's University, School of Computing, 2011, pp. 43.

[213] M. Stephan, M. H.Alalfi, A. Stevenson, J. R. Cordy, Towards Qualitative Comparison of Simulink Model Clone Detection Approaches, in: Proceedings of 6th International Workshop on Software Clones (IWSC'12), Zurich, Switzerland, 2012, pp. 84-85.

[214] M. Stephan, M. H. Alalfi, A. Stevenson, J. R. Cordy, Using Mutation Analysis for a Model-Clone Detector Comparison Framework, in: Proceedings of International Conference on Software Engineering (ICSE'13), San Francisco, CA, 2013, pp. 1261-1264.

[215] H. Störrle, Effective and Efficient Model Clone Detection, in: Proceedings of European Conference on Software Architecture (ECSA'10), Copenhagen, Denmark, 2015, pp. 440-457.

[216] H. Störrle, Towards Clone Detection in UML Domain Models, Software and Systems Modeling 12(2) (2013) 307-329.

[217] A. Sutton, H. Kagdi, J. I. Maletic, G. Volkert, Hybridizing evolutionary algorithms and clustering algorithms to find source-code clones, in: Proceedings of Genetic and Evolutionary Computation Conference (GECCO'05), Washington DC, USA, 2005, pp. 1079-1080.

[218] N. Synytskyy, J. R. Cordy, T. Dean, Resolution of static clones in dynamic web pages, in: Proceedings of 5th IEEE International Workshop on Web Site Evolution (WSE'03), Amsterdam, The Netherlands, 2003, pp. 49-56.

[219] R. Tairas, J. Gray, I. D. Baxter, Visualization of clone detection results, in: Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange, Portland, Oregon, USA, 2006, pp. 50-54.

[220] R. Tairas, J. Gray, Phoenix-based clone detection using suffix trees, in: Proceedings of the 44th annual Southeast regional conference (ACM-SE'06), Melbourne, Florida, USA, 2006, pp. 679-684.

[221] R. Tairas, Clone maintenance through analysis and refactoring, in: Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium, Atlanta, GA, USA, 2008, pp. 29-32.

[222] R. Tairas, Centralizing clone group representation and maintenance, in: Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications companion to the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, Orlando, Florida, USA, 2009, pp. 781-782

[223] R. Tairas, J. Gray, An Information Retrieval process to aid in the analysis of code clones, Empirical Software Engineering 14 (1) (2009) 33-56.

[224] R. Tairas, F. Jacob, J. Gray, Representing clones in a localized manner, in: Proceedings of 5th International Workshop on Software Clones, Honolulu ,USA, 2011, pp. 54-60.

[225] U. Tekin, F. Buzluca, A graph mining approach for detecting identical design structures in object-oriented design models, Science of Computer Programming 95(4) (2014) 406-425.

[226] R. Tiarks, R. Koschke, R. Falke, An assessment of type -3 clones as detected by state-of-the-art tools, in: Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09), Edmonton, Canada, 2009, pp. 67-76.

[227] S. Thummalapenta, L. Cerulo, L. Aversano, M. Di Penta, An empirical study on the maintenance of source code clones, Empirical Software Engineering 15 (1) (2010) 1-34.

[228] A. K. Tripathi, R. Gupta, M. Gupta, Some observations on software processes for CBSE, Software Process: Improvement and Practice 13 (5) (2008) 411-419.

[229] T. Tran, R. Nayak, P. Bruza, Combining structure and content similarities for XML document clustering, in: Proceedings of 7th Australasian Data Mining Conference (AuSDM 2008), Glenelg, South Australia, 2008, pp. 219-226.

[230] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, On detection of gapped code clones using gap locations, in: Proceedings 9th Asia-Pacific Software Engineering Conference (APSEC'02), Gold Coast, Queensland, Australia, 2002, pp. 327-336.

[231] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, Gemini: Maintenance support environment based on code clone analysis, in: Proceedings of the 8th IEEE Symposium on Software Metrics (METRICS'02), Ottawa, Canada, 2002, pp. 67-76. OMG: OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1 (formal/2011-08-05). Tech. rep., Object Management Group, Feb 2011.

[232] OMG UML: OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1 (formal/2011-08-05). Tech. rep., Object Management Group, Feb 2011.

[233] V. Wahler, D. Seipel, J. W. Gudenberg, G. Fischer, Clone detection in source code by frequent itemset techniques, in: Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation (SCAM'04), Chicago, IL, USA, 2004, pp. 128-135.

[234] A. Walenstein, N. Jyoti, J. Li, Y. Yang, A Lakhotia, Problems creating task-relevant clone detection reference data, in: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03), Victoria, BC, Canada, 2003, pp. 285-295.

[235] R. Wettel, R. Marinescu, Archeology of code duplication: Recovering duplication chains from small duplication fragments, in: Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2005, p. 8

[236] T. Weilkiens, Systems Engineering with SysML/ UML, Morgan Kaufmann (2007).

[237] J. Whaley, M. S. Lam, Cloning- based context-sensitive pointer alias analysis using binary decision diagrams, in: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'04), Washington DC, USA, 2004, pp. 131-144.

[238] M. D. Wit, A. Zaidman, A. van Deursen, Managing code clones using dynamic change tracking and resolution, in: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09), Edmonton, AB, 2009, pp. 169-178.

[239] XMI Guide Version 2.4.1. Tech. rep., Object Management Group. http://www.omg.org/spec/XMI, document number formal/ 2011-08-09, 2011.

[240] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, H. Iida, SHINOBI: A tool for automatic code clone detection in the IDE, in: Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09), Lille, France, 2009, pp. 313-314.

[241] W. Yang, Identifying syntactic differences between two programs, Software Practice and Experience 21 (7) (1991) 739-755.

[242] R. Yokomori, H. Siy, N. Yoshida, M. Noro, K. Inoue, Measuring the effects of aspect-oriented refactoring on component relationships: Two case studies, in: Proceedings of 10th Aspect-Oriented Software Development Conference (AOSD'11), Pernambuco, Brazil, 2011, pp. 215-226.

[243] F. Van Rysselberghe, S. Demeyer, Evaluating clone detection techniques from a refactoring perspective, in: Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04), Linz, Austria, 2004, pp. 336-339.

[244] N. Yoshida, T. Ishio, M. Matsushita, K. Inoue, Retrieving similar code fragments based on identifier similarity for defect detection, in: Proceedings of the 2008 workshop on Defects in large software systems in companion to International Symposium on Software Testing and Analysis (DEFECTS '08), Seattle, Washington, 2008, pp. 41-42.

[245] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, On refactoring support based on code clone dependency relation, in: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05), Como, Italy, 2005, pp. 16-25.

[246] N. Yoshida, T. Hattori, K. Inoue, Finding similar defects using synonymous identifier retrieval, in: Proceedings of 4th International Workshop on Software Clones, Cape Town, SA, 2010, pp. 49-56.

[247]Y. Zhang, H. A. Basit, S. Jarzabek, D. Anh, M. Low, Query-based filtering and graphical view generation for clone analysis, in: Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM'08), Beijing, China, 2008, pp. 376-385.

LIST OF PUBLICATIONS

PAPERS PUBLISHED IN JOURNALS

D. Rattan, R. Bhatia, M. Singh, Software clone detection: a systematic review. Information and Software Technology 55 (7) (2013) 1165-1199. (SCI Indexed)

D. Rattan, R. Bhatia, M. Singh, Detecting high level similarities in source code and beyond. International Journal of Energy, Information and Communications 6 (2) (2015) 1-16.

D. Rattan, R. Bhatia, M. Singh, Detection and Analysis of Clones in UML Class Models. International Journal of Software Engineering (Accepted for publication with minor changes)

D. Rattan, R. Bhatia, M. Singh, An empirical study of clone detection in MATLAB/Simulink models. International Journal of Information and Communication Technology (Scopus Indexed). (Accepted for publication with minor changes)

M. Kaur, D. Rattan, R. Bhatia, M. Singh, Comparison and evaluation of clone detection tools: an experimental approach. CSI Journal of Computing 1 (4) (2012) 44-54.

PAPERS COMMUNICATED IN JOURNALS

D. Rattan, R. Bhatia, M. Singh, Systematic mapping study of metrics based clone detection techniques. Journal of Engineering Research, Submitted on May 13, 2015. (SCI Indexed) ID: 864

PAPERS PUBLISHED IN CONFERENCES

D. Rattan, R. Bhatia, M. Singh, Model clone detection based on tree comparison, in: Proceedings of Annual IEEE India Conference (INDICON ’12), Kochi, India, 2012, pp. 1041-1046.

M. Kaur, D. Rattan, R. Bhatia, M. Singh, Clone detection in models: An empirical study, in: Proceedings of the 3rd IBM Collaborative Academia Research Exchange (I-CARE), New Delhi, India, 2011.