DEVELOPMENT OF AN EFFICIENT SEMANTIC CODE CLONE DETECTION TECHNIQUE

A Thesis submitted in fulfillment of the requirement for the award of the degree of

DOCTOR OF PHILOSOPHY
IN
COMPUTER SCIENCE AND ENGINEERING

Submitted By

Rajkumar Tekchandani
(Registration Number: 951003002)

Under the guidance of

Dr. Rajesh Bhatia
Professor
Computer Science and Engineering Department
PEC University of Technology

Chandigarh, India

Dr. Maninder Singh
Professor
Computer Science and Engineering Department
Thapar Institute of Engineering & Technology
Patiala (Punjab), India

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
PATIALA — 147004

AUGUST 2018

CERTIFICATE

I, Rajkumar Tekchandani, Regn.No. 951003002, hereby declares that the thesis entitled **"Development of an Efficient Semantic Code Clone Detection Technique"** submitted to the Department of Computer Science & Engineering at Thapar Institute of Engineering & Technology, Patiala, Punjab, India is an authenticated record of my own work for the award of degree of "Doctor of Philosophy" under the supervision of Dr. Rajesh Bhatia and Dr. Maninder Singh. This report has not been submitted to any other institution for award of any degree.

Rajkumar Tekchandani

951003002

Place: Patiala
Date: 4-1-2018

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.

Approved by:

Dr. Rajesh Bhatia
Professor
Computer Science and Engineering Department
PEC University of Technology

Chandigarh, India

Dr. Maninder Singh
Professor
Computer Science and Engineering Department
Thapar Institute of Engineering & Technology
Patiala (Punjab), India

ABSTRACT

Over the last few years, code clones have emerged as an active area of research because of their wide range of applications in different domains of software engineering. Code clones are the result of copy paste activities. Similar code fragments that exist at different locations are called code clones. Code clones are reported in the form of clone pairs. Clone pairs are further clustered to form code clone groups. Code clones are broadly categorized into four types from Type 1 to 4. In literature, numerous code clone detection techniques exist to find different types of code clones. Knowledge extraction from existing software resources for maintenance, re-engineering and bug removal through code clone detection is an integral part of software systems. Code clone detection techniques are mainly classified into text based, token based, tree based, metric based and semantic code clone detection techniques.

Most of the existing semantic code clone detection techniques in literature are based on the comparison of program dependence graphs through sub graph isomorphism, which is NP-Complete. Moreover, these techniques for semantic code clone detection are unable to provide heuristic solution for problems such as statement reordering, inversion of control predicates and insertion of irrelevant statements which may cause a performance bottleneck. To address these issues, we proposed a novel approach that finds semantic code clones between code fragments using data flow analysis on the basis of reaching definition and liveness analysis. The algorithm based on reaching definition and liveness analysis is designed to find similar code fragments which are structurally divergent, but semantically equivalent. The results obtained demonstrate that the pro-

posed approach using reaching definition and liveness analysis is effective in detection of semantic code clones for various applications. Results obtained on subject systems taken from DeCapo Benchmark confirms the effectiveness of the proposed approach.

Further, code clone groups are extracted among different versions of the program file distributed over thousands of commit hashes in distributed version control system (DVCS). Code clone group extraction has many software applications that help in refactoring and maintenance of code in open source software systems. The evolution of code clone groups across the history of a software system is termed as code clone genealogy. Most of the existing solutions for code clone group extraction are based on text similarity among different versions of program files stored in centralized version control system (CVS). However, existing proposals in literature for code clone group extraction fail to extract code clone groups among different versions of program files stored in distributed version control system.

To address these issues, we presented a novel Git code clone group extraction model based on transitive closure computation on directed acyclic graphs using Big Data Technologies. Our insight is to extract clone pairs from thousands of commits on a software system in Git by transitive closure computation and mapping of clone pair parameters in genealogy to extract code clone evolution patterns in graph database (Neo4j). We efficiently detected code clone genealogies on Git based e-health care system and created a scalable solution. We performed evaluations on OpenMRS, an open source e-health system on Git and presented interesting code clone evolution relationships in code clone genealogy. The performance of the proposed approach is evaluated using parameters such as transitive depth, ratio of similarity and count of clones.

Acknowledgments

First and foremost, thanks to almighty God for all his blessings without which nothing of my work would have been possible. I am highly indebted to Prof. (Dr.) Prakash Gopalan, Director, Thapar Institute of Engineering & Technology, Patiala and Prof. (Dr.) R. S. Kaler, Deputy Director, Thapar Institute of Engineering & Technology, Patiala, for providing me the opportunity to pursue my course work and research. My deep regards to Dean of Research and Sponsored Projects for his enormous help in completion of my research work. This thesis would have not been possible without the enabled guidance and keen interest of my supervisor's Dr. Rajesh Bhatia, Professor, Department of Computer Science and Engineering, PEC University of Technology, Chandigarh and Dr. Maninder Singh, Professor and Head, Department of Computer Science and Engineering, Thapar Institute of Engineering & Technology, Patiala.

The extensive discussions with them have always made me to stay on the right track. They have always been patient and cooperative whenever their guidance and expertise was needed. They helped me not only by sparing their valuable time, but also analytically reviewing my experimental setup, publications, reports and presentation from time to time. The constructive criticism and motivation by my supervisor's has groomed me to my present shape. I would also like to express my gratitude to Dr. Maninder Singh, Head, Computer Science and Engineering Department, Thapar Institute of Engineering & Technology, Patiala for providing me the continuous feedback and motivation throughout my work. I am highly thankful to my doctoral committee members Dr. Seema Bawa, Dr. Inderveer Chana and Dr. M.D. Singh for their constructive suggestions and ensuring the progress of my research work at correct pace. Their critical analysis and encouraging support brought me to this stage of my course. The help rendered by them is greatly accredited. My deep regards to Dean of Research and Sponsored Projects for his enormous help in completion of my course work. I will go amiss if I forget to thank Dr. Prateek Bhatia for his valuable suggestions throughout the program.

I wish to thank all the faculty and staff members of Computer Science and Engineering Department of Thapar Institute of Engineering & Technology, Patiala for their enormous support. Lastly, I would like to pay my gratitude to my parents, wife, son, daughter, brother and to all friends and colleagues for love and encouragement all through my life. They have always helped me at every twirl of my life when I am in a need. Words cannot truly express my deepest gratitude and appreciation to all of the above. I am sorry, if I have forgotten someone and I cannot thank everyone enough for the involvement they have shown and the willingness they have expressed to take on the completion of tasks beyond their comfort zones.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

During recent decades, rapid advancements have been observed in the field of code cloning. Copying code segments and then reuse by pasting with or without minor modifications or adaptations are common activities in software development. This type of reuse approach of existing code is called code cloning and the pasted code segment is called a clone of the original [3]. A code segment is labeled as a code clone if it is identical or highly similar to another code segment. Two similar code segments form a clone pair. Groups of clone pairs are known as clone classes or clone groups [4][5]. Clones can be introduced into systems deliberately (e.g., copy and paste actions) or inadvertently by a developer during development and maintenance activities. Like all code segments, code clones are not resistant to change. It is believed that the code clones have lot of impact on software maintenance, therefore studying code clones and their evolution is an interesting topic of research. Code clones are often the result of copy paste activities. Such activities reduces programming effort and time as they reuse the existing code rather than rewriting it again. Code clones are broadly classified into four types as: Type 1, Type 2, Type 3 and Type 4 clones.

1.1 Types of Code Clone

- **Type 1 Clones**: Code segments which are same except for variations in white space and comments are Type 1 clones. Type 1 clones are also known as exact

clones. For example:

Original Code Segment:

if (a<b) {

c = d + b;

} and exact copy of the original code can be as follows

if (a<b)

{

c=d+b;

}

- **Type 2 Clones**: Code segments which are structurally or syntactically equivalent except for changes in identifiers, literals, types, layout and comments [1] are Type 2 clones. For example:

  Original Code Segment:

  if (a<b) {

  c = d + b;

  e = a + 4;

  } and Type 2 clone of this code segment can be as follows

  if (m<n) {

  x = y + n;

  w = m + 9;

  }

- **Type 3 Clones**: Code fragments that have been copied with further modifications like statement insertions or deletions in addition to the variation in identifier names, literals, types, layouts and comments are Type 3 clones. For example:

  Original Code Segment:

  if (a<b) {

c = d + b;

e = a + 4;

} and Type 3 clone of this code segment can be as follows

if (m<n) {

x = y + n;

f = 6;

w = m + 9;

}

- **Type 4 Clones**: If two or more code fragments performs the same computation, but implemented with different syntactic structures are Type 4 clones. For example:

  Original Code Segment:

  if (a<b) {

  c = d + b;

  e = a + 4;

  }

  else {

  g = t - r;

  } and Type 4 clone of this code segment can be as follows

  if (a>b) {

  g = t - r;

  }

  else {

  e = a + 4;

  c = d + b;

  }

1.2 Reasons for Code Cloning

Code clones do not take place in software systems by itself. There are a number of factors that might compel software developers in creation of cloned code in the system. Some of the factors that leads to code clones are discussed below:

- Code can be reused because of copy paste activity. If there exists a solution to a problem, then the same function can be reused.
- It has been observed that there is a risk in developing the new code, as well as to keep software architecture clean and understandable [6][4]. Cloned fragments speeds up maintenance activities.
- Code clones can be introduced due to underlying language limitation and programmers limitation. Like difficulty in understanding large system, time limit assigned to the developers, lack of developer's knowledge in particular domain. To write reusable code might be fault prone. It is preferred to copy the code and reuse it by pasting with or without modification rather that writing it from scratch [7].
- Code clones may be introduced inadvertently by implementing the same logic by different developers across different versions of a program file during evolution of a software system.

1.3 Limitations of Code Cloning

Code clones do take place in large software systems. While it is beneficial to carry out code cloning, code clones can have bad impact on the software quality, reusability and maintainability. Some of the limitations of code clones are discussed as follows:

- If a code fragment holds a bug and that code fragment is used frequently in the source code, then that bug also spreads all over the program where that code segment is reused [8][9]. To rectify that bug at all places may increase the maintenance cost and delay.

- It is developer's responsibility to adapt the code fragment that is already tested and bug free. If that copied fragment chosen by developer contains bug, then that new bug is also introduced in the system where that fragment is reused [10][11].

- Code cloning may increase the probability of bad design. It increases the maintenance efforts of a software system [12].

- Cloning may introduce new resources to handle the higher growth rate of system size.

1.4 Advantages and Applications of Code Cloning

Cloned code fragments are more stable and reliable as compared to non cloned fragment. It has great impact on software quality. It is easier to refactor cloned code as compared to non cloned code. Some of the benefits of code cloning are discussed as follows:

- If the functionality of cloned code can be understood, then it may help in understanding the program up to some extent or fully.

- Clones can be used as macros or functions and can be called directly according to usage.

- It helps in finding the malicious code, that if a malicious code can be found in one code fragment then it is also propagated in the rest of code fragments and it can

be rectified at all places by just find and replace action [13].

- It has been observed that the code fragments that have been copied and used multiple times can be incorporated in the library [14][15].
- Code clones in a software system may be useful in finding plagiarism disguises and copyright infringement [13][16][1].
- Code clone detection also helps in software evolution analysis by extracting code clone genealogy across different versions of software system. It helps in analysis of code clone evolution patterns during software evolution [17][18].
- Code compaction can be achieved by using code clone detection techniques [19].

1.5 Code Clone Classification

Although there are four major types of code clones as discussed earlier as Type 1, 2, 3 and 4. Apart from these researchers used different terms in literature to identify code clones.

- **Exact Clones:** Two or more code segments are called as exact or Type 1 clones if they are exactly similar to each other. There may be some differences in layouts, comments and whitespace [10].
- **Parameterized Clones:** A parameterized clone is a renamed clone. These are created by identifier renaming [20]. These are detected by the comparison of tokens. CCFinder [1] is one of the leading token based code clone detection tool that can efficiently find parameterized clones. They belong to Type 2 code clones.
- **Near Miss Clones:** These types of clones are achieved by incorporating small changes as identifier renaming, change in literal values and insertion or deletion of

statements from original code fragments. Near miss clones are mainly categorised in Type 3 clones [21][11][22].

- **Non Contiguous Clones:** Non contiguous or gapped clones are kind of near miss clones that contain gaps between the code fragments. These kind of code clones are classified as Type 4 code clones. Gaps between consecutive statements are created by inserting irrelevant statements. Gapped code clone fragments in source code are detected by carrying out control and data flow analysis [23][11].

- **Reordered Clones:** Reordered clones are possible by reordering the statements in copied fragment. Reordered statements in copied fragment do not change control and data flow dependencies. These kind of clones are structurally different but semantically equivalent. On the basis of semantic similarity these are classified as Type 4 code clones [4].

- **Structural Clones:** Structural clones represents design level similar program structures. These clones are larger than simple clones. Structural clones can belong to any type based on their level of similarity. These are very good candidates for refactoring. Various types of structural clones can be extracted using Clone Miner [24][4].

- **Intertwined Clones:** In intertwined code clones, two similar code segments are interweaved together in a single code segment. The similar code fragments are implemented in a single function by twisting different statements. Intertwined Clones are considered as Type 4 clones. The backward slicing approach is used by Komondoor and Horwitz [23][4] to find such clones.

- **Short-lived Clones:** These types of code clones evolve during software evolution and are easily refactorable for maintenance activities. They live for a short period

of time in a software system. Any kind of clones from Type 1 to Type 4 can be classified into the category of short lived clones [4][25].

- **Long-lived Clones:** These type of code clones belong to version control systems. They are generally unrefactorable code clones and lives for long time across different versions of a software system [4][26].

1.6 Code Clone Relationships

Code clones are reported in the form of clone pairs or clone groups or clone class families. Clone pairs are defined in terms of similarity relationships. The similarity

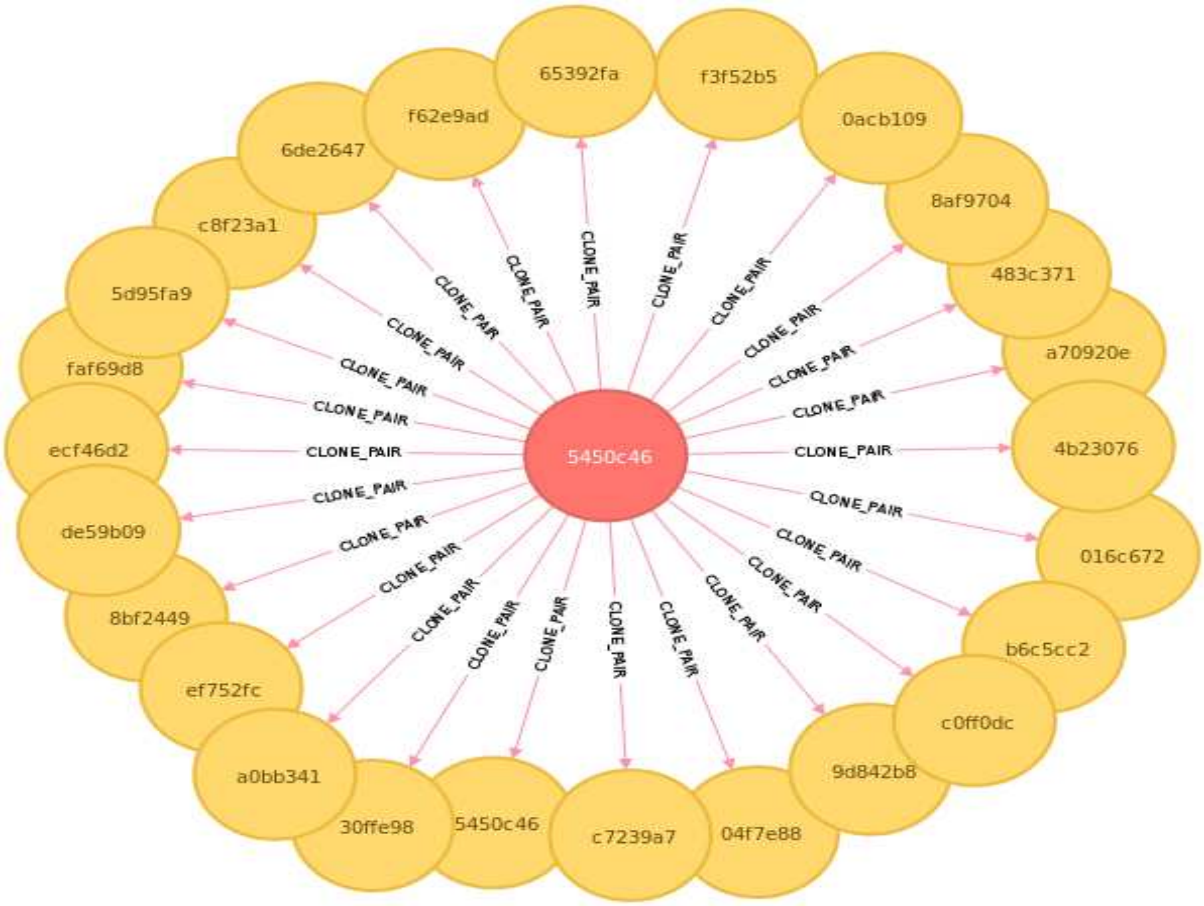


Figure 1.1: Code clone pairs and clone group.

relationship between code fragments is an equivalence relation (i.e, reflexive, symmetric and transitive). Code clone pairs are the output of copy-paste activities in a software system. The collection of clone pairs form a clone group. The collection of clone groups leads to clone class family [4][3]. Figure 1.1 shows clone pairs between commit hashes. Each commit hash represents the version of a program file. The collection of all such clone pairs between commit hashes is termed as a code clone group or clone class. The granularity of code clone group is more as compared to a clone pair.

1.7 Code Clone Detection Process

A code clone detector try to detect all the possible code clones at different locations in a software system. The detector compares every possible code fragment with every other possible code fragment. It follows six phases as preprocessing, transformation, matching, formatting, filtering and aggregation during code clone detection process [3].

- **Preprocessing:** In this phase of code clone detection process, uninterested code fragments are filtered to determine the source code units that are suitable for comparison.

- **Transformation:** The output of first phase is passed into transformation. In this phase, transformation techniques are applied to obtain intermediate code representation that is suitable for comparison of code fragments. The transformation approaches may involve pretty printing, ignore/remove comments, whitespace removal, tokenization, parsing, program dependence graph generation, metric value calculation and normalization of identifiers.

- **Matching:** The transformed code acts as an input to suitable comparison technique where transformed code fragments are compared to find code clones. Some

of the popular matching techniques are token-based [1][20], tree-based [27][11], PDG based [28][29], metric based [30][31], pattern matching [32][33] and hash value comparison [11][34]. The output of this phase is code clones either in form of clone pairs or clone groups.

- **Formatting:** In this phase of code clone detection process, code clone pair locations are mapped to original source code with respect to line numbers.
- **Filtering:** In this phase false positive code clones are removed with the help of visualization tools. A visualization tool is helpful to filter false positive clones by speeding up the process of manual analysis.
- **Aggregation:** In order to conduct some analysis, code clones are grouped in the form of classes, clusters, clone groups or clone class families.

1.8 Code Clone Detection Techniques

Code clone detection techniques are broadly classified into three categories as:

- Syntactic Code Clone Detection
- Semantic Code Clone Detection
- Code Clone Genealogy Detection

1.8.1 Syntactic Code Clone Detection

Techniques for detection of code clones up to Type 3 are syntactic code clone detection techniques. The approaches for code clone detection are classified as text based approaches [8], token based approaches [1][9], tree based approaches [11][35][36], metric

based approaches [32][34][37][38][30], program dependence graph based [23][39][40][28] and hybrid approaches [27][22][41].

In text based approaches, code clones are detected on the basis of string matching algorithms. In this the source code is considered as sequence of lines. In token based approach the complete code is parsed into the sequence of tokens. This lexeme of tokens is scanned to find duplicate subsequences of tokens that are termed as clones. In tree based approach, a program is parsed with the help of parser to build an abstract syntax tree [42]. Similar sub trees are searched in the tree with tree matching techniques and the corresponding source codes of similar sub trees are returned as clone pairs.
In metric based clone detection approach, different metrics are collected for code fragments and then the values of collected metrics are compared instead of comparing the source code directly. The code clones up to Type 3 can be detected using text, token, tree and metric based comparisons. Code clones are represented with the help of metrics graph, scatter plots and Hasse Diagram's. [43].

1.8.2 Semantic Code Clone Detection

Program dependence graph (PDG) based code clone detection approaches go one step further than other approaches by considering the semantic information of the program. PDG contains control and data flow information of the programs and hence, carries the semantic information. Once the set of PDG's are obtained for code fragments, isomorphic sub graph matching algorithm is applied to find similar sub graphs. These similar sub graphs are reported as semantic code clones. PDG based approach can identify non contiguous or gapped clones [23]. The PDG based approach can also be used to find reordered clones. The major limitation of PDG based approach is to generate and handle large PDG's. Although, semantic code clones reported using sub graph isomorphism is NP-complete problem. Another approach for detection of

functionally equivalent code fragments is on the basis of input-output behavior through random testing [44].

1.8.3 Code Clone Genealogy Detection

Moreover, as code clones are modified, a change evolution history, known as a clone genealogy [26][45], is generated. Code clone evolution in an important aspect of software development. Code clone genealogies are analyzed on the basis of clone lineages. A Clone Lineage is a directed acyclic graph that describes the evolution history of a sink node. In a clone lineage, a clone group in $(k + 1)^{th}$ version is connected by k^{th} version. A set of clone lineages constructs code clone genealogy [26][45][46]. Code clone genealogies are classified as alive, dead, short lived and long lived genealogies [26][45]. Code clone genealogies are evolved on the basis of code clone groups. A code clone group is the collection of clone pairs. Code clone evolution depicts the development of code clones across different versions of program file. Code clone evolution patterns are classified as consistent evolution (CO), late propagation (LP), delayed propagation (L2) and independent evolution (IE) [47].

1.9 Evaluation of Code Clone Detection Techniques

As number of clone detection techniques are evolved with time, therefore a comparison of these code clone detection techniques is essential. Some of the parameters for comparing different code clone detection techniques are as follows.

- **Portability:** A clone detection tool is easily portable and configurable in multiple platforms. It should be portable in terms of programming languages.

- **Precision:** The clone detector tool should be efficient to find code clones with less number of false positives. It should find code clone pairs or groups with high precision.

- **Recall:** The clone detector tool should be robust to find code clones that have hidden clone pairs. Programmers often try to hide code clones by statement reordering, inversion of control predicates and insertion of irrelevant statements. The clone detector should be enough capable to find all types of clones in a software system.

- **Scalability:** The tool should be able to find code clones in optimum time from large software system that is distributed over several versions.

- **Robustness:** A good code clone detector should be robust enough to find similar code fragments that are modified using editing activities such as layout variation, identifier renaming, statement reordering, inversion of control structures, insertion of irrelevant statements and code structure variation.

1.10 Research Gaps

Code clones carry domain knowledge for better program understanding, code quality analysis, software evolution analysis and bug detection. This requires the extraction of syntactically or semantically similar code fragments and makes code clone detection an important and valuable part of software domain. Besides the tremendous benefits of code clone detection, many challenges still need to be addressed. Among these the biggest challenge is to find semantic similarity which requires deep semantic analysis of source code programs [5]. The literature available for software clones mainly focuses on Type 1 to Type 3 clones. There is enough room for the work to be carried out for Type

4 clones. Code cloning has been investigated by the research community from different perspectives in many applications. Lot of research has been carried out by researchers in this direction but still there is scope for further improvements in the existing solutions. Hence, based on above discussions, following gaps are identified for code clone detection in software systems.

- Hybrid techniques are usually derived from one or more clone detection techniques. There exists no hybrid technique that can detect contiguous and non contiguous clones simultaneously.

- There is still no special treatment for identifiers and literal values for detecting code clones in Abstract Syntax Trees.

- Not enough attempts have been made to implement parsing actions such as LR, LALR and SLR to detect code clones by building parsing tables.

- As PDG based techniques are based on comparison of PDG's of code fragments using sub graph isomorphism. The generated PDG's are returned as semantically similar code fragments, if they are isomorphic to each other. It is difficult to handle the large PDG's and further isomorphism returns approximate solution.

- As per literature survey, most of the techniques for detection of Type 4 clones are PDG based. Type 4 clone detection techniques have not been explored using formal methods of program analysis.

- Most of the semantic code clone detection techniques are unable to provide a heuristic solution to solve the issues like statement reordering, inversion of control predicates and insertion of irrelevant statements. Therefore, a new type of semantic code clone detection technique needs to be designed that can address these issues.

- Till now, as per literature survey, most of the existing approaches for code clone group extraction are focused on centralized version control system. These approaches are not much efficient to find code clone groups on distributed version control system (DVCS) in optimum time.

- Git (a distributed version control system) is the most broadly used version control system and is used as the modern standard of software development. It provides exceptional support to branching, merging and rewriting history that provides many innovative and powerful workflows.

- Large software systems on Git go through lots of commits over their life cycles. Code clone genealogy is evolved over time with each commit made by user in a software system on Git.

- There exists an enough room to carry out research on code clone group extraction on distributed version control system.

1.11 Research Objectives

From the above discussions and gaps identified, following are the objectives of the thesis.

- To study and analyze the existing code clone detection techniques and to carry out their comparative analysis.

- To design and develop an improved code clone detection technique for detection of semantic clones.

- To verify and validate the proposed technique.

1.12 Thesis Organization

The thesis is organized as follows:

- The second chapter provides summary of various code clone detection techniques. A taxonomy of existing code clone detection techniques is provided in the text. Some important key concerns related to code cloning as code clone analysis, clone stability, code clone detection in aspect oriented programming and clone detection in websites are discussed. A comparison table with detailed comparison of existing semantic code clone detection techniques with respect to various parameters are provided in the text. Some of the relevant code clone genealogy extraction techniques are also discussed. This chapter helps in understanding the basic gaps between existing techniques and the current proposal. At the end, we have prepared the objectives of the proposed scheme.

- In third chapter, we have mentioned the data flow analysis techniques used in the proposed technique. This chapter explains proposed approach along with algorithms for semantic code clone detection that addresses the issues as statement reordering, inversion of control predicates and insertion of dead or irrelevant statements.

- In fourth chapter, we have mentioned the infrastructure and tools used in proposed code clone group extractor model. This chapter explains the proposed Git code clone group extractor model using transitive closure computation on directed acyclic graphs (DAG) in Hadoop Distributed File System (HDFS). Moreover, an efficient algorithm is also presented to extract code clone groups from genealogy in optimum time.

- In fifth chapter results are divided into two sections. In first section result analysis

of semantic code clone detection is conducted. The performance of proposed approach is evaluated on standard benchmark. In second section of this chapter code clone groups are extracted on subject system. In experimental study, we have identified the relationships of transitive depth between different versions of the program file for extraction of code clone evolution patterns on e-health system.

- Sixth chapter concludes the findings of the proposed work. The results obtained have concluded that the proposed approach performs better than existing solutions. Lastly, we have also provided the directions for the future advancements of the proposed work.

Chapter 2

Literature Review

Significant work has been done by number of researchers on software code clone detection and evolution of code clones [4][5][48][49]. The whole domain of software code clone detection consists of numerous techniques based on textual, lexical, syntactic, semantic, metric, hybrid and model based code clone detection. Moreover, from last decade lot of work has been carried out on code clone evolution and visualization. Some other important key concerns are also discussed in this chapter.

2.1 Challenges in Code Cloning

Code clone detection has been used in various applications with a goal to provide maintenance and refactoring benefits to developers. Software maintenance is one of the most expensive phases of software development [50]. Hence, there is a requirement of an optimized solution to code cloning. However, there are number of challenges that needs to be addressed. Some of these challenges are to identify the insertion of irrelevant statements, inversion of control predicates, statement reordering and extraction of clone groups in code clone genealogy.

In existing study, most of the semantic code clones are detected by comparing program dependence graphs (PDG's) using sub graph isomorphism. It is difficult to handle large PDG's and moreover, sub graph isomorphism is NP-Complete problem. Hence, there is a need to present an alternative approach that can find semantic code clones in

real time. Moreover, in existing study another challenge is to extract code clone groups in distributed version control systems. Currently in existing study code clone groups are extracted on centralized version control system using text and location overlapping.

In today's era, distributed version control systems are widely used by software developers for version management due to its powerful workflow features. Moreover, code clone stability has gained a lot of attention from research community. The biggest challenge in code clone stability is to find *'whether cloned code is stable or not'*. With the increase in duplication of source code, the proper visualization of code clones for easier maintenance and refactoring has become a challenge.

2.2 Taxonomy of Code Clone Detection Approaches

Code clone detection is an active research area due to its refactoring and maintenance benefits. In this direction, many researchers have proposed different code clone detection approaches to find code clones. These approaches consists of text based, token based, tree based, metric based and graph based comparisons to find code clones. Moreover, significant research has been done in the area of code clone evolution, visualization, stability and refactoring. A detailed taxonomy of various code clone detection techniques is described in Figure 2.1.

2.2.1 Text Based Code Clone Detection Approaches

In text based code clone detection techniques similar texts are compared line by line and similar code fragments are reported as code clones.

Johnson *et al.* [51] presented the code clone detection approach based on fingerprinting to detect text based clones on a substring of the source code. In this approach,

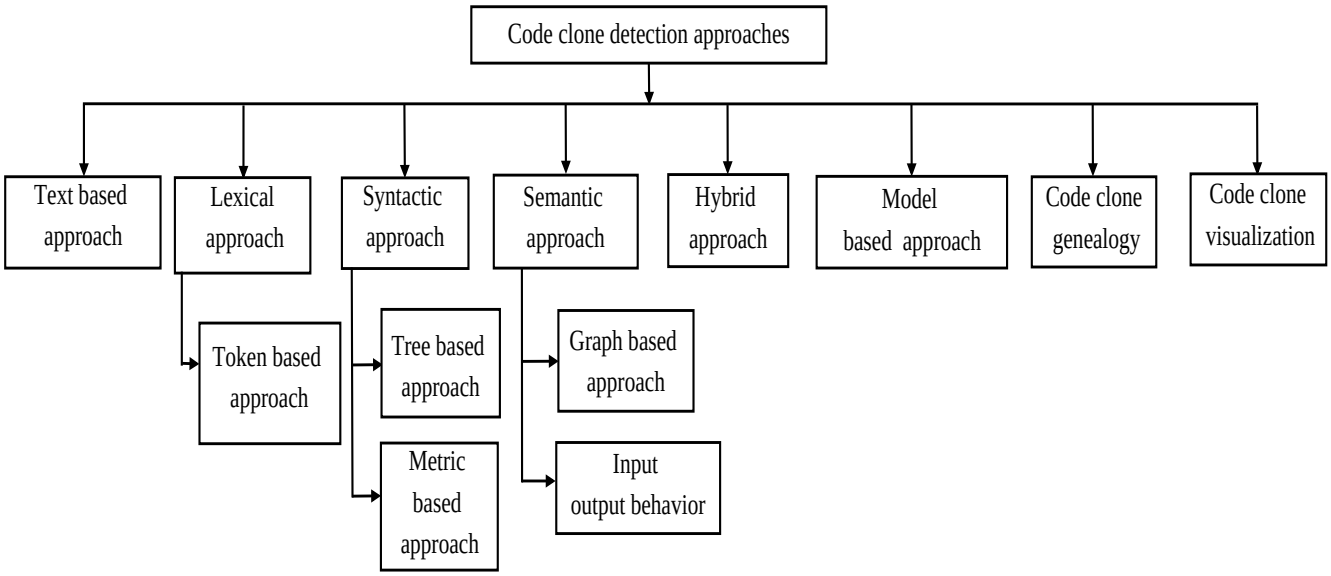


Figure 2.1: Taxonomy of existing code clone detection approaches.

signatures calculated per line are compared in order to identify matched substrings. Karp Rabin fingerprinting algorithm [52] is used for calculating the fingerprints of substrings. First, a text to text transformation is performed on the considered source file for discarding uninteresting characters. Following this the entire text is subdivided into a set of substrings so that every character of the text appears in atleast one substring. Finally, the matching substrings are identified as code clones [8].

Ducasse *et al.* [53] presented another text based approach that is language independent, because their approach does not rely upon parsers and parameterized strings, that needs a lexer. This approach reads the source file, make sequence of lines, removes whitespace and comments in the lines and detects match by string based pattern matching algorithm. The result of string comparison is stored in the comparison matrix using reflexive and symmetric nature of matrices. The output of this approach is the form of line numbers of clone pairs, possibly with gapped lines between them.

2.2.2 Lexical Approaches

Lexical approaches for code clone detection are based on comparison of sequence of tokens (lexemes). These approaches are generally more vigorous over minor code changes as compared to text based code clone detection approaches.

2.2.2.1 Token Based Code Clone Detection Approaches

In token-based code clone detection approach, scanner is used to create tokens of the source code that skips whitespace and comments. Then similar token sequences (lexemes) are compared to find code clones. If the pair of lexemes is similar then these are reported as token-based code clones. This approach generally returns Type 1 and 2 code clones.

Baker *et al.* [54][16] developed a code clone detection tool named as Dup that is purely line based as well as token based in the sense that a lexer is used to tokenize the source code. Then tokens of each line are compared using suffix tree algorithm to detect exact and near miss clones.

Kamiya *et al.* [1] developed one of the leading token based technique named as CCFinder. In this a lexer is used to tokenize the source code and then tokens of all source files are concatenated into a single token sequence or lexeme. Transformation rules on the input source string are applied and comparison is done token by token. The output of CCFinder is in the form of code clone pairs or code clone classes. Visualization of code clones is done by visualizing diagonal entries on the scatter plots. Most of the plagiarism detection systems are based on the comparison of lexical structure of programs [55][56].

Li *et al.* [9] introduced another token based code clone detection technique where a frequent subsequence mining technique [57] is used for identifying a similar sequence

of tokenized statements. Due to sequential analysis of source code in CCFinder [1] and Dup [16], they are generally fragile to statement reordering and code insertion. A reordered or inserted statement can break a token sequence which may otherwise be regarded as duplicate of another sequence. These limitations are eliminated by using a frequent subsequence mining technique where a frequent subsequence can be interleaved in its supporting sequences. Moreover, the existing tools do not detect copy paste related bugs. In their work they proposed a tool as CP-Miner, which uses data mining technique to identify copy paste related bugs in large software systems.

Basit *et al.* [20] developed an efficient token based code clone detection technique with flexible tokenization. It finds similar token sequences by applying the transformation rules as removal of tokens like access specifiers and operators. Later on, suffix trees are replaced by suffix arrays to save memory space [58]. Another multi-input open source code clone detection tool as *CloneDetective* is built on ConQAT [59] infrastructure for code clone detection research. It detects code clones using suffix trees. It searches for similar subsequences in a program.

Yamashina *et al.* [60] presented an automatic code clone detection tool as SHINOBI that is implemented as plug-in of Visual Studio. It detects code clones with the source code being edited. SHINOBI is implemented as client server architecture. It uses CCFinderX to tokenize sequence of program code. Finally, a suffix array index is created to search similar token sequences.

Li *et al.* [61] designed CCLearner, a token based clone detection approach using deep learning. A classifier is trained using deep learning to detect code clones in a given codebase. In this study, a classifier model is trained on the basis of training data (clones and non clones) and other part is tested to find code clones. They evaluated CCLearner on BigCloneBench benchmark for empirical evaluation. In their finding, they found that CCLearner has high precision and recall as compared to existing token and tree

based code clone detection techniques. To extract the features that describe code clone relationships, CClearner uses ANTLR lexer and EclipseAST parser for tokenization and parsing.

2.2.3 Syntactic Approaches

Syntactic approach uses a parser to convert source code program into abstract syntax tree (AST's) or parse tree which can be processed using comparison of features of AST's.

2.2.3.1 Tree Based Code Clone Detection Approaches

In tree based code clone detection approach both scanner and parser are used to create an AST. Further AST's are compared to find similar code fragments.

Yang [62] developed a comparison algorithm that exploits syntactic structure of programs described by grammar of programming languages. The algorithm is based on dynamic programming that can point out the differences between two programs more accurately as compared to text based approaches.

Baxter *et al.* [11] developed tree based code clone detection tool CloneDR, in which a compiler generator is used to generate an AST. It compares the sub-trees by characterization metrics based on a hash function through tree matching. Sub trees that are identical to each other are inserted into the same bucket using hashing and rest is inserted into different buckets. Source codes of similar sub trees are then returned as code clones. This technique is useful for detection of code clones having the consecutive code elements in two code fragments. It is not useful to detect code clones in which there is an insertion of the new code in the middle of the code clone, forming a split or non contiguous clones. It detects the preceding and following fragments as clones. Non contiguous clones or split clones are kind of Type 4 clones where gaps are allowed

between the code fragments and therefore, non contiguous clones are called as gapped clones [4].

Gitchell *et al.* [63] designed a program called as *'sim'* to find similarity between two C codes. It measures structural similarity of source code by reducing them in corresponding parse tree structure and then aligns them by inserting spaces between strings of parse trees to obtain a maximal common lexeme.

Wahler *et al.* [36] described a tree based approach for detection of code clones in source code, which is inspired by the concept of frequent item sets from data mining. In data mining, frequent item sets are used to illustrate relationship between large amounts of data in the form of association rules. The classic example is the analysis of buying behavior of customers. They found exact and parameterized clones in a more abstract level than AST, where the AST of a program is built currently in Java, C++ and Prolog. Further, generated AST's are converted into XML representations and then frequent item set mining technique is applied on the XML representations of the AST's for finding code clones. This approach is very flexible as it can be configured to work with multiple programming languages.

Koschke *et al.* [27] extended tree based code clone detection with the help of abstract syntax suffix trees. A Suffix tree is the representation of a string as a trie or prefix tree where every suffix is presented through a path from root to leaf. Instead of comparing the AST nodes, their approach compares the tokens of the AST nodes using suffix tree based algorithm, and therefore this approach can find code clones in linear time and space, a significant improvement over the usual AST based techniques.

Jiang *et al.* [22] deduced a novel approach of detecting similar trees and a practical implementation as DECKARD, for code clone detection, where certain characteristic vectors are computed to approximate the structural information within the AST's in the Euclidean space R_n. A Locality Sensitive Hashing (LSH) [64] scheme is used to

cluster similar vectors with respect to Euclidean distance metric and thus similar code fragments are returned as code clones. A pair wise tree comparison could be used to detect such clones, but this is expensive for the large programs because of the large number of sub trees. For this they demonstrated a novel technique based on characteristic vector and vector clustering to handle large number of characteristic vectors. However, there is still no special treatment for identifiers and literal values for detecting clones in AST's. The AST based approach disregards the information about identifiers. It ignores data and control flows, and therefore fragile to statement reordering, insertion and replacement, that leads to evolution of PDG based technique.

Bulychev *et al.* [65] presented a new algorithm for code clone detection. It is independent of source code and works at AST level. The algorithm considers that two sequences of statements are considered as code clones if one code segment is achieved by replacing some of the sub trees of other code segments. They implemented the algorithm in CloneDigger.

Evans *et al.* [35] introduced further abstraction (known as structural abstraction) of a program AST represented with XML for finding exact and near miss clones with gaps. While AST's are build from lexical abstraction of the program by parameterizing (substitution) only AST leaves (abstracting identifiers and literal values), structural abstraction is obtained by further parameterizing the arbitrary sub trees of AST's. Structural abstraction is more general than AST representation.

Nguyen *et al.* [66] introduced an incremental clone detection tool, ClemanX for management of code clones. Code clones are detected by generating the potential clones along with their corresponding characteristic vectors. Characteristic vectors are constructed on the basis of feature extraction approach. The similar code fragments are clustered as a clone pair on the basis of minimum editing distance between characteristic vectors. Further, similar pairs are hashed into buckets using locality sensitive hashing

(LSH) technique [64].

Kamiya [67] presented a tree based code clone detection technique that find near miss or Type 3 code clones in python source code. It detects similar sequences during the execution of program by applying apriori algorithm [68] (a frequent item set mining technique) on call tree generated through execution sequence of source code.

2.2.3.2 Metric Based Code Clone Detection Approaches

Software metric is defined by computing some property of a part of software. It presents quantitative techniques which are helpful in evaluating the software quality. The aim of software metrics is to manage and recognize the necessary parameters that influence software development. Numerous techniques are presented to rank software metrics. Garg *et al.* [69] presented a framework for ranking of software metrics on the basis of fuzzy based matrix methodology. Metric based approaches collect a number of metric values for code segments and then compare those metric values instead of AST or code based comparisons.

Kontogiannis *et al.* [32] developed an abstract pattern matching tool to identify probable matches using Markov models. It measures the similarity between code fragments. They have presented three pattern matching techniques as direct comparison of metric values, dynamic programming approach and statistical matching algorithm to detect code clones. In the first approach, metrics taken into consideration are fan out, ratio of input output variables to the fan out and McCabe's cyclomatic complexity. On the other hand in dynamic programming approach, the distance between the pair of begin-end blocks is defined as the least costly sequence of insert, delete and edit steps required to make one block identically equivalent to another block. Finally, statistical matching algorithm is applied to detect similar code fragments.

Mayrand *et al.* [34] developed the tool *CLAN* that returns code clones by finding

values of several metrics (e.g, no of lines of source code, number of function calls contained, number of CFG edges etc.) for each function unit of the program. Units with similar metric values are considered as code clones. Partly similar units are not detected as code clones.

Patenaude *et al.* [70] detected method level metrics such as number of calls from a method, number of statements, McCabes cyclomatic complexity and number of local variables. These metric values are compared to detect similar methods.

Balazinska *et al.* [38] presented an algorithm for automatic classification of clones using metrics extracted from AST representation of source code. The purpose of this research is to investigate the use of clones as a basis for those re-engineering actions which are useful for maintenance of software systems.

Lanubile *et al.* [37] deduced a semi automated approach to detect function clones in web applications. Code clones are detected by comparing metric values. Perumal *et al.* [71] presented a metric based approach based on fingerprinting technique. They calculated the similarity score for a pair of code fragments. They found clusters of code clones that are mutually similar.

Kodhai *et al.* [72] presented code clones on the basis of textual comparison of code fragments. They compared several method level metrics as number of lines, arguments passed, function calls, local variables declared, conditional statements, loop control statements and return statements for detection of Type 1 and Type 2 code clone methods.

Li and Sun [73] proposed a metric space based code clone detection approach. It uses distance metric to measure the similarity level of code. A proximity query based technique based on *Range Query* and *Nearest Neighbor* is applied to find the closeness between a pair of code fragments.

Lavoie *et al.* [74] presented a technique based on graphic processing unit (GPU)

algorithms to compute many instances of the longest common subsequences on GPU architecture using DP matching algorithm. This approach is useful to address the problem of filtering false positives produced by metrics based clone detection methods.

Bansal *et al.* [75] proposed an approach which evaluates the set of independent metrics on the basis of precision and recall values in clone detection. They have further increased the metric combinations to choose the appropriate set of metrics for code clone detection.

Salwa [31] presented an efficient metric based data mining approach for code clone detection. Metrics are collected for all functions in the software system and fractal clustering is applied to find relatively small number of clusters. These clusters represent similar code fragments.

Joshi *et al.* [76] investigated Type 1 and Type 2 function clones using data mining technique. They have created dataset by collecting metrics for all functions in a software system and applied Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clustering algorithm on the dataset to cluster Type 1 and Type 2 function clones.

2.2.4 Semantic Approaches

Semantic approaches have been proposed to provide more exact information as compared to syntactic approaches. Semantic approaches for code clone detection are based on comparison of semantics of a program. Semantic code clones are detected using comparison of program dependence graphs (PDG's) and by evaluating input output behavior of code fragments.

2.2.4.1 Graph Based Code Clone Detection Approaches

In graph based approach, a program dependence graph is created for control and data flow dependence. Further, these control and data flows are matched through graph isomorphism. It is capable of finding non contiguous code clones which were separated through irrelevant code statements.

Horwitz [77] identified textual and semantic differences between two versions of a program. Program components are partitioned on the basis of equivalent behaviors using partitioning algorithm. Program representation graphs are used for graph representation of program.

Krinke *et al.* [39] identified the code clones by taking the concept of parse trees and found that these trees do not contain any control and data dependence information with it. By introducing control and data dependencies along the edges and nodes of the AST, a new dependence graph as program dependence graph (PDG) is generated. Further, an iterative approach (K-length path matching) has been used for detection of maximal similar sub graphs.

Komondoor *et al.* [23] developed one of the leading PDG based code clone detection technique which finds isomorphic PDG sub graphs using program slicing [78]. They proposed an approach to convert non contiguous clones into contiguous. They preserved the semantics of original code for automatic procedure extraction to support abstraction [28].

Liu *et al.* [40] developed a PDG based code clone detection tool, GPLAG for plagiarism detection. Plagiarism disguises are the most important part of this technique. Plagiarism disguises discussed in this work are identifier renaming, format alteration, code insertion, statement reordering and control replacement. Implementation of GPLAG is done with the help of PDG's of the two source codes. These PDG's are passed to lossless and lossy filters for pruning plagiarism search space. Finally, isomorphic sub

graphs are detected between two PDG's to find semantically equivalent code fragments. Gabel *et al.* [79] worked upon the scalable detection of semantic code clones. They considered the base as PDG and found that complex PDG's are difficult to analyze. They removed level of difficulty to some extent by converting the PDG's into AST's that are easier to analyze. The AST's are then analyzed with the help of characteristic vectors [22].

Higo *et al.* [29] worked on the new area of clone detection which uses PDG's for contiguous clones. The paper presented by them proposes PDG specializations for contiguous code clone detection. The specialization is to introduce an extra link known as execution next link in the PDG that expands the range of program slicing, so that the ability to detect contiguous clones is improved. An execution next link is an edge that represents the order of execution of program elements. There exists an execution next link between the two nodes if the program element represented by one of the nodes may only be executed after the program element represented by the other node. Another specialization is to merge directly connected equivalence nodes that reduces the computational cost of code clone detection.

As most of the graph based approaches for code clone detection are static. Neubauer [88] presented a dynamic approach for semantic code clone detection. He developed a tool named as KAMINO that tracks control and data flow dependencies at method level during runtime by using Java byte code. Finally, similar flows are matched using longest common substring algorithm to find semantic code clones.

Sheneamer and Kalita [89] detected Type 3 and Type 4 code clones using machine learning by extracting the features of abstract syntax trees and program dependence graphs. They presented the pair of code fragments as vector and used classification algorithms to train the model. The classification models are trained and tested using fifteen different machine learning algorithms. Moreover, these classification algorithms

Table 2.1: Comparative analysis of semantic code clone detection techniques.

Author	**Year**	**Source Code Representation**	**Code Clone detection Technique**	**Advantages**	**Scope/Limitations**
Baxter *et al.* [11]	1998	AST	Hashing	Near miss clones	Could not find split clones
Marcus *et al.* [80]	2001	Text	Latent Semantic Indexing	Structural clones	Includes whitespace and comments
Krinke *et al.* [39]	2001	PDG	N-length path	High precision and recall	Slow rate of PDG Generation, unable to detect reordered statements
Komondoor *et al.*[23]	2001	PDG	Sub-graph isomorphism	Procedure Extraction	NP-complete
Gabel *et al.* [79]	2008	PDG and AST	Mapping of PDG on AST	Scalable	Slow generation of PDG's for huge source code
Choi *et al.* [81]	2009	Birthmarks	Max weighted bipartite matching	Efficient	Deobfuscation Attacks
Jiang *et al.* [44]	2009	C Intermediate Language	Random testing	Scalable	Does not take care about structural information of code
Kim *et al.* [82]	2011	Path sensitive Semantic-based static analyzer	Abstract memory State comparison	Precise	More false positives
Schugerl *et al.* [83]	2011	AST	Semantic web reasoning	Scalable	Only find bigger code fragments
Higo *et al.*[29]	2011	PDG	PDG mapped with execution links	More precise with contiguous code	Heavier in structure
Elva *et al.* [84]	2012	Java methods	IOE(Input, output and effect)Behavior	Higher recall	Works only for Java methods
Kamiya et al. [85]	2013	Java Byte code	n-grams	Scalable	Needs refinement to permit more code modifications during refactoring
Tekchandani *et al.*[86]	2013	AST	Grammar Recovery	Semantic clones	More false positives
Wang *et al.* [87]	2014	SDG (System Dependence Graph)	Hybrid (metrics + Graph)	Semantic clones	False negatives and false positives
Proposed Work	2016	AST	Reaching definition and liveness analysis	Less number of false positives, considers statement reordering and dead code detection	Works only for Java code

are compared to measure their ability to detect code clones. In their findings, they concluded that Xgboost [90] is an exceptional classification algorithm for detection of

syntactic and semantic code clones.

2.2.4.2 Input Output Behavior Based Clone Detection Approaches

Jiang *et al.* [44] proposed an algorithm based on input output behavior that presents functional equivalence between a set of code fragments. They developed their algorithm using automated random testing. Their experimental result shows that there exist many functionally equivalent code fragments that are syntactically different from each other. In this study, they covered more than 624 KLOC of Linux kernel and found that 58% of the functionally equivalent code fragments are syntactically different.

Elva *et al.* [84] presented a semantic code clone detection approach based on input output behavior. Two code fragments are semantically equivalent if their input output behavior is identical. Input is determined by the parameters passed to the function and state of the heap when function is called. The output behavior is determined by function return values and its effects.

Kamiya [85] developed an execution based semantic code clone detection tool, *Agec*. It detects code clones by extracting n-grams from execution sequences and further these n-grams are compared to detect semantically equivalent code fragments. In his experimental study, the n-gram size is taken as six to find code clones in open source subject system as ArgoUML.

2.2.5 Hybrid Code Clone Detection Approaches

Hybrid techniques for code clone detection are the combination of one or more techniques discussed from section 2.2.1 to 2.2.4. For instance, token based and PDG based techniques can be merged together to find contiguous as well as non contiguous code clones.

Maeda *et al.* [91] worked on code clone detection using parsing actions in which parsers are used for parsing the source code. It has been suggested that any kind of parsers (LR, LALR, SLR etc.) can be used to parse the source code. They built the parsing tables for a given set of rules consisting of terminals and non terminals. Action and goto tables are created for state transitions. Finally, suffix trees and suffix arrays are used to detect code clones.

Gode *et al.* [92] worked on the incremental detection of code clones. In incremental detection of code clones they preserve the current code clones and then use them further to detect code clones in other source codes. Their empirical results reveal that incremental code clone detection takes less time as compared to non incremental code clone detection if the changes in the source code do not exceed a certain fraction of source code. The incremental clone detection algorithm developed by them is based upon token based approach and uses the nodes of suffix trees for reusability.

Roy *et al.* [93][21] proposed a multi pass hybrid approach as NICAD that detects both exact and near miss clones using flexible pretty printing and code normalization techniques. In the first phase potential clone pairs are identified, pretty printed and extracted from subject code. Further, NICAD compares pretty printed potential clones using longest common subsequence algorithm.

Basit *et al.* [94] developed the tool Clone Miner that uses frequent item-set mining approach which works on the output of Repeated Token Finder (RTF) [20]. Clone Miner is used to find some specific type of structural clone sets such as simple clone, method clone, file clone and directory clone sets.

Chilowicz *et al.* [95] presented a new hybrid approach on the basis of code factorization and pattern matching algorithms using suffix arrays. It detects function clones in source code that are represented in the form of call graphs. In call graphs vertices represents functions and edges represents the function calls.

Li and Thompson [96] presented a token and tree based hybrid technique to detect code clones in Erlang programs. The fundamental property of Erlang language is built-in support for light weight processes. The proposed approach is able to find code clones in Erlang programs that are syntactically similar.

Selim *et al.* [97] enhanced source based code clone detection using intermediate representation. The source code representation (e.g. Java and C files) of a software system has been traditionally used for clone detection. They proposed a technique that transforms the source code into an intermediate representation and then reuses established source based clone detection techniques to detect clones in intermediate source code representation. The extracted code clones are mapped back to the source code and are used to augment the results reported by source based clone detection. This approach can detect upto Type 3 code clones.

Hummel *et al.* [98] presented a novel, index based code clone detection algorithm for detection of Type 1 and 2 clones. They performed code clone detection in incremental and scalable manner. They detected code clones in 73 MLOC and demonstrated that incremental updates are useful for code clone management.

2.2.6 Model Based Code Clone Detection Approaches

Model driven development has been an active area of research with the rise in abstraction. Large models are developed using various modeling languages like UML, Matlab, Z specification languages etc. The presence of duplicate sub structures in different types of model based specification languages cannot be ignored. Many researchers worked in this area of research.

Sequence diagrams are widely used in system modeling. Duplication in sequence diagrams reduces maintainability and reusability. As a result, maintainability of models becomes an important concern for software fertility. Liu *et al.* [99] detected code clones

in sequence diagrams by transforming 2-D sequence diagrams into 1-D array. Then a suffix trees are build using 1-D arrays for similarity detection in sequence diagrams.

As discussed in section 2.1, that the problem of finding largest clone pair using sub graph matching is NP-complete, *i.e.*, no polynomial time algorithm is expected to find maximal clone pairs which one can use for model based code clone detection. Deissenboeck *et al.* [100] extracted model based code clones by using maximum weighted bipartite matching technique on labeled multigraphs, which can be solved in polynomial time. They represented Matlab/Simulink target link models as graphs. Model code clones are affected by the issues of scalability, code inspection and relevance. Their study provided useful finding in addressing these issues in real time.

Pham *et al.* [101] presented a novel clone detection tool as *ModelCD* for Matlab/Simulink models that detects exact and near miss model clones. They transformed models into directed graphs by assigning labels to relevant blocks. They proposed two algorithms as *escan* and *ascan* to detect code clones in models. Exact model clones are detected by *escan* using canonical labeling (an advanced isomorphic graph matching technique). Moreover, approximate model clones are detected by *ascan* that finds non-overlapping weakly connected sub graphs with same structure.

The lack of incremental model based clone detection obstructs an efficient clone management. It requires latest cloning information. Hummel *et al.* [102] proposed an incremental and distributed approach for model based code clone detection. A UML model is pre-processed by converting the model into labeled directed graph. Model code clones are detected by applying canonical matching on labeled nodes and edges. The sub graphs are indexed using a clone index and then hashed into a hash table to retrieve isomorphic sub graphs.

Storrle [103] initiated the detection of code clones in UML domain models by defining a labeled graph structure that is suitable for representing models. UML case tools

are used to generate XMI files from UML domain models. These XMI files are further converted into Prolog files. The output is generated using model matching technique on some input model. The classification for model based clones as exact, modified, renamed and semantic model clones is also proposed in this work.

2.2.7 Code Clone Evolution Approaches

Software system needs to grow in order to be used for longer period of time. It should be updated across different versions to deal with ever changing requirements of users. The various software metrics get affected during software evolution [104]. This section describes the research closely related to code clone evolution or code clone genealogy detection. Code clone evolution depicts the development of code clones throughout the history of a software system. This study consists of code clone evolution patterns and evolution of code clone groups in a software system with time.

Lague *et al.* [105] initiated evolution of code clones over time using source code metrics for large telecommunication software having history of three years. They applied two changes as *Preventive Control* (new clone is added for good reason) and *Problem Mining* (to deal with existing software clones) for effective clone management and tracking.

Antoniol *et al.* [18] presented a method to predict and monitor code clone evolution across various versions of a software system. In this average number of clones per function are identified using metrics based approach and then these extracted clones are modeled in terms of time series to predict code clone evolution patterns. A time series is a stochastic process in which changes are collected over time for future predictions. An experimental study is performed on 27 successive versions of *MiniSQL* (a relational database system written in C language). Later on, different versions of Linux Kernel (versions 2.4.0 to 2.4.18) are analyzed to check changes in code clone evolution patterns

[17]. The similarity between the Linux subsystems is measured through metric based code clone detection technique proposed by Mayrand *et al.* [34].

Kim *et al.* [106] carried an ethnographic study of code clones to understand copy and paste behavior of programmers in object oriented systems. They developed a logger in Eclipse IDE that accounts key strokes and editing operations. Edit logs captured by logger are replayed to infer change patterns in code clone evolution. Their study revealed that programming language limitations and structural templates results in code duplication. These findings results in code clone genealogy study. In code clone genealogy, a clone group is traced to its base clone group in previous versions of source code projects [107].

Di Penta *et al.* [108] presented an evolution framework as *Evolution Doctor* [109] to monitor software evolution. It also helps in renewal of the software system by improving the directory structure of source files. It deals with refactoring, removal and reorganization of source code repositories during software evolution.

Kim *et al.* [26][25] were the first to analyze code clone genealogies. They analyzed code clone genealogies on the basis of clone lineages. Sink node of clone lineage represents a clone group. A clone lineage is a directed acyclic graph that describes the evolution history of a sink node. In a clone lineage a clone group in $(k + 1)^{th}$ version is connected by k^{th} version. A set of clone lineages constructs a code clone genealogy [26] [45] [46]. Further, they have classified code clone genealogies as alive, dead, short lived and long lived genealogies [26] [45]. They have build a clone genealogy extractor (CGE) based on text similarity and location overlapping function that automatically extracts versions of program file from source code repositories stored in centralized version control system (CVS).

Balint *et al.* [110] developed a visualization tool named as *Clone Evolution View* to analyze the various changing evolution patterns. It analyzes the number of lines cre-

ated by a software developer, time of modification, location of code clones in a software system. In their approach duplicate text fragments are aggregated into sets called as clone class families. Clone class families describe the relationship of cloning between multiple versions of the text [111]. Clone evolution view is visualized with the help of three developer activities as line cloning, block cloning and line fixing. In their empirical observations, two types of attributes as *consistent* and *inconsistent* are discussed from the viewpoint of maintenance.

Aversano *et al.* [112] extended the findings of Kim *et al.* [26] and carried an empirical study on ArgoUML and DNSJava CVS repositories. They studied code clone maintenance during evolution activity. However, they manually investigated code clone genealogies for finding change patterns and focused on exact code clones.

Gode [113] presented a study on evolution of Type 1 (exact) clones and proposed an approach that models code clone evolution based on the changes in the source code that were made between consecutive versions of the program in various open source software systems. Livieri *et al.* [114] conducted an evolution analysis of Linux Kernel using code clone coverage (CVR) metrics. They checked 136 versions of Linux Kernel using *D-CCFinder* [115] (a distributed extension of *CC-Finder*). In their study, results are visualized using a heat map graph.

Bakota *et al.* [116] proposed an approach based on similarity measurement to map code clones from one specific version of the software to another. In their study, they worked on dynamic behavior of code clones and defined an evolution mapping between two code fragments from different versions. The evolution mapping is partial one to one mapping of clone instances between consecutive versions of the subject software system. Moreover, they introduced the concept of code smells that refers to the detection of harmful code fragments among different versions of the software system. In this approach, code clones are detected using AST based code clone detection tool, *clones*

[27].

Thummalapenta *et al.* [47] extended the work of Aversano *et al.* [112] and conducted a similar study to understand the extent of code clone propagation in genealogy. They considered four evolution patterns as consistent evolution (CO), late propagation (LP), delayed propagation (L2) and independent evolution (IE).

Shawky *et al.* [117] modeled evolution of code clones in open source systems using chaos theory for prediction of code clones in new versions of software system. Code clones are identified using CloneDR [11]. Extracted code clones from CloneDR in each version are represented in the form of time series data. The Maximal Lyapunov exponent [118] is calculated for the identification of chaos in systems. Chaos points to the set of systems that are at some intermediate points between entirely predicted and random systems.

Saha *et al.* [45] carried the study of Kim *et al.* [26]. Their study differs from the study of Kim *et al.* [26] in the aspect of delivery. They evaluated code clone genealogies at release level, while previous study on code clone genealogy is at revision level. Further they have conducted an in-depth empirical study for evaluation of code clone genealogies on 17 open source systems covering programming languages as Java, C++, C and C#. They have replaced location overlapping function with snippet matching algorithm that works well with statement modification and reordering. They have classified one add on genealogy as syntactically similar genealogy (SSG). However, they focus at the release level in open source software systems. Moreover, Saha *et al.* [119] carried their previous study for extracting near miss code clone genealogies. They presented a framework for extraction of exact and near miss code clone genealogies across multiple versions of program.

Barbour *et al.* [120] studied late propagation in software clones to find inconsistent changes during software evolution. *Simian* and *CCFinder* [1] clone detection tools are

used to examine the characteristics of late propagation in two long-lived open source software systems as ArgoUML and Apache-ANT. They identified eight types of late propagation from *LP1* to *LP8* and studied them to identify the type of late propagation that leads to most of the faults in a software system.

Gode *et al.* [49] presented an incremental clone detection approach that detects and maps code clones for multiple versions of a program based on token based clone detection using suffix trees. They have evaluated that most of the clones remain unchanged during their lifetime and were mostly changed inconsistently.

Pate *et al.* [48] conducted a systematic review on code clone evolution which indicates that there are contradictions regarding lifetime of the clone lineages and the consistency of code clones during evolution of software system.

Saha *et al.* [46] performed an exploratory study on the evolution of Type 1 (exact), Type 2 (parameterized) and Type 3 (near miss) clones in six open source software systems and found that Type 3 code clones are more likely to change inconsistently in a software system. They compared the results of Type 3 code clones with that of exact clones to understand the behavior of Type 3 code clones.

Xie *et al.* [121] estimated the possibility of faults by studying the migration of code clones during code clone evolution. They performed their empirical study on fault proneness on three open source systems as ArgoUML, JBOSS and Apache-ANT. The code clones in this study are detected using NICAD [122]. They found that the presence of clone mutation makes clone migration more unsafe.

2.2.8 Code Clone Visualization Approaches

Code clone visualization is an active area of research. It deals with the visualization of code clones after detection. It helps in speedy analysis of results. Johnson *et al.* [43] visualized code clones using Hasse diagram. Hasse diagram is the graphical representa-

tion of partially ordered sets. A relation 'R' on set 'A' is said to be a partially ordered relation if it is reflexive, anti-symmetric and transitive [123].

Code clones are visualized using scatter plots and metrics graph in Gemini [2]. It reads the output of CCFinder [1]. It detects code clones by comparing the lexemes of tokens and outputs the location of code clone pairs. It consists of four steps as lexical analysis, transformation, match detection and formatting.

Kim *et al.* [26] represented code clone genealogies in terms of clone lineages. A clone lineage is a directed acyclic graph where sink node represents a clone group. Tairas and Gray [124] developed an Eclipse plug-in that shows the results of CloneDR [11]. This is an extension of AspectJ Development Tool (AJDT) that presents graphical view of code clones detected in the source file. In this code clones are considered similar to aspects and AJDT visualiser plug-in is used to display the cross cutting concerns of aspect oriented programs.

Adar *et al.* [125] introduced code clone exploration system as *SoftGUESS* for code clone genealogy visualization. They represented *SoftGUESS* through number of applications to analyze code clone evolution patterns in software systems. *SoftGUESS* is based on a graph exploration system GUESS [126]. It consists of encapsulation and genealogy browsers for visualization of single and multiple versions of code clones.

Jiang and Hassan [127] introduced the concept of clone mining. It is the process of mining interesting clone patterns from huge amount of potential code clone candidates. They used data mining framework to uncover code clones. Moreover, a visualization graph as clone system hierarchical graph is also presented in this work. It uses a tree structure that copies the directory structure of the file system.

Zhang *et al.* [128] presented a visualization tool as *Clone Visualizer* in Eclipse framework. It filters and visualizes the output of Clone Miner. *Clone Miner* is a token based tool that uses frequent subsequence mining technique to filter contiguous and non con-

tiguous code clones [24]. They developed a clone query system that specifies filters to extract abstract views from large cloning data. The output of clone query system is stored in a relational database.

Fukushima *et al.* [129] proposed a code clone visualization method as code clone graph. It gives graphical representation of clone sets. It reads the output of CCFinder [1]. In code clone graph, if code clone sets are included in the same program file, then each code clone set is represented as node of the graph and nodes are connected through an edge. Further, metrics are calculated to find diffused code clones.

Murakami *et al.* [130] developed an Eclipse plug-in named as *ClonePacker*. It visualizes code clones using Circle Packing. It reports location of code clones in source code much faster as compared to an existing tool, Libra [131]. Circle Packing represents enclosure diagrams for visualizing detected clones with file hierarchies.

2.2.9 Other Key Concerns in Code Cloning

Apart from the techniques discussed above there exist some other key concerns in code cloning that are gaining lot of attention from research community. These key concerns includes code clone analysis, refactoring, impact of clones on software systems, code clone detection in websites and code cloning in Aspect Oriented Programming (AOP).

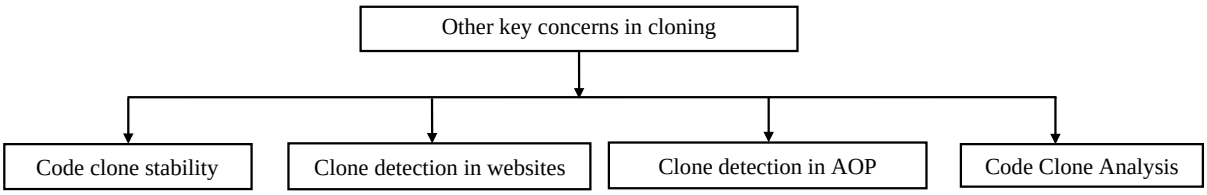


Figure 2.2: Other key concerns in code cloning.

2.2.9.1 Code Clone Stability

This study summarizes the studies on whether cloned code is more stable and reliable as compared to non-cloned code. It also analyzes the impact of code clones on software quality.

Monden *et al.* [12] conducted a module based analysis to find relation between software quality and code clones. Their study reveals that program files containing code clones are more reliable than other program files. They found that modules with code clones are easy to maintain than non clone modules in a software system. The study reveals that large code clones are less reliable and difficult to maintain as compared to non clone modules.

Lozano *et al.* [132] designed a prototype tool as *CloneTracker* to collect proofs to verify the belief that *'Code clones are generally harmful'*. Software prototyping is a process of developing prototypes of software systems [133]. It analyzes clones at method level and count the number of times the method was changed in different versions of program file with in a particular time frame. It uses CCFinder [1] to detect code clones. In their analysis they found that code clones are harmful and have adverse impact on maintenance.

Krinke [134] presented a study on stability of code clones. To understand the stability of cloned code he conducted two separate but related studies on evolution of exact clones. He tried to answer the question as *'if cloned code is more stable than non-cloned code during software evolution'* by finding change evolution patterns. Source code changes are distinguished by addition, deletion and updation of existing source code. He found that deletions of code clones act as dominating factor to evaluate the stability of code clones. Further, this study is carried by Gode and Harder [135] by considering different parameters for code clones to validate the findings of Krinke [134]. In continuation, Krinke [136] evaluated the '*age*' parameter to check stability of cloned

code in centralized version control system. In his study, he proposed that if a fragment of cloned code is changed less as compared to non cloned code during its lifetime, then it is more stable as compared to non-cloned code. Changes are tracked line by line using CVS *'annotate'* command.

Juergens *et al.* [137] introduced CloneDetective framework to find inconsistent code clones. Their study presented a strong evidence regarding code clones, that major source of defects in a software system are contributed by inconsistent code clones. Inconsistent code clones are detected by measuring the edit distances on suffix trees to find significant overlap among code fragments.

Selim *et al.* [138] conducted a study on survival analysis to understand the effect of code clones on software defects. The risk of experiencing a defect in software system is modeled by Cox hazard function. The study used two types of predictors as *Control Predictors* and *Cloning Characteristic Predictors* to classify code clones as harmful or safe. In their study they concluded that code clones are not always be more harmful than non cloned clones.

Rahman *et al.* [139] tried to validate the general belief that *'code clones are bad smells'*. They conducted an analysis between code cloning and defect proneness. In their findings, they found that code clones are always not the bad smells. Moreover, they found that the cloned program fragments are *less* inclined towards defects as compared to non cloned code fragments. In this study, code clone detection tool DECKARD [22] is used to obtain code clone information.

2.2.9.2 Clone Detection in Web Applications

Website consists of lot of code and usually suffers from short development life cycles due to client site requirements. It includes content, structure and usage analysis. The rapid evolution of web pages leads to introduction of code clones in websites. Lot of

studies confirms the presence of code clones in websites at different levels. Due to expansion of web and e-commerce sites the demand of new websites and web applications increases manifolds. Moreover, it is easier to maintain the web data in cloud environment, instead of managing that data in the form of web pages [140].

Aversano *et al.* [141] proposed an approach to reuse the existing sites by means of duplication. They used UML to support design and implementation of websites based on relational database management systems. The conceptual views of the repository of the website are represented through UML diagrams.

Lucca *et al.* [142] proposed an approach to detect duplicate web pages in websites on the basis of similarity metrics. Code clones are detected for the web pages that are designed using HTML and ASP technologies. The levenshtein distance metric is used to determine the degree of similarity between the pair of web pages. If the values of levenshtein distance metric is zero then the two HTML strings are considered as code clones, otherwise these strings are evaluated on some threshold value to specify the extent of code clones in web pages.

Lanubile *et al.* [37] proposed an approach for speedy selection of function clones in web applications that can be applied to prevent clone spreading. In this study function clones embedded in HTML scripts of web applications are identified.

Lucia *et al.* [143] presented a tool to understand the cloning patterns in websites through analysis of code clones and clustering. Similar pages are analyzed and clustered on the basis of levenshtein edit distance. Finally, uninteresting parts of clustered navigational schema are filtered to improve understanding of code clone patterns in web applications.

Rajapakse *et al.* [144] conducted a quantitative study of code cloning on 17 web applications of varying sizes, developed using different scripting languages. In their study, they found 17% to 63% of clones in web applications. They developed a code clone

analyzer to control and analyze code clones detected by CCFinder [1].

Lucia *et al.* [145] presented an approach based on Latent Semantic Indexing (LSI) [146] that computes the dissimilarity between web pages. The similar pages are clustered using Hierarchical, Partitional and Artificial Neural Network (ANN) based clustering algorithms. In this work single link, complete link and average link methods are used to compute distances between two clusters.

2.2.9.3 Clone Detection in Aspect Oriented Programming

The domination of the decomposition states that, no matter how well a software system is decomposed into modules, some concerns crosscuts the decomposition. Code will get spread throughout other modules. Code spreading will lead to code duplication in cross cutting concerns. Cross cutting concerns are the parts of the program that depend on other modules in a system.

Bruntink [147] discussed an approach to relate code clone detection to aspect mining. They developed a method to filter clone classes on the basis of clone metrics. The relevance of clone class for aspect mining is shown by assigning a grade to clone class. Clone classes that scores less than a predefined threshold value are filtered out and only relevant clone classes are collected.

Bruntink *et al.* [148] evaluated the code clone detection techniques for identification of cross cutting concerns. The evaluation considers token based *CCFinder* [1], AST based *ccdiml* [27] and PDG based *PDG-DUP* [23] clone detection techniques. The objective of this study is to evaluate the relationship among five cross cutting concerns (error handling, tracing, NULL-value checking, range checking and memory error handling) and code clones in CC component written in C language. In this study, they found that lot of time is spent in dealing with the concerns as error handling and tracing. Their study showed that token based and abstract syntax tree based clone detection

techniques are best suited for error handling and NULL-value checking, while abstract syntax tree based techniques are also appropriate for the range checking aspect. The PDG based techniques are suitable to deal with tracing and memory error handling concerns.

Schulze *et al.* [149] presented a code clone classification approach that makes a decision to use aspect oriented or object oriented refactoring for removing bad smells from code. In the proposed classification approach, semantic information as *type* and *location* is added to code clones. The metric *DIST* is proposed in their study for classification. The range of DIST metric is categorized into three categories varying from '0' to '1'. The value of DIST < 0.3 specifies that clone classes are equivalent to each other, $0.6 \leq$ DIST < 1.0 indicates scattered code clones and $0.3 \leq$ DIST < 0.6 indicates refactoring candidates.

Yokomori *et al.* [150] investigated the impact of refactoring activities on component relationships of two large projects. In their findings they indicated that aspect oriented refactoring is successful for modularity of code. They measured two kinds of relationships as *use relation* and *clone relation* between components. The *use relation* component graph is shown as a digraph while *clone relation* is represented as an undirected graph. Code clone relations are calculated using *CCFinder* [1].

2.2.9.4 Code Clone Analysis

Code clone analysis is the extraction of code clone fragments which can be refactored easily. In some studies cloning patterns are also evaluated in this domain.

Higo *et al.* [151] proposed the tool *Aries* to support refactoring of code clones. Metric graph view is used to identify and filter clone sets. The metric view graph uses the metrics as average length, number of code fragments, number of tokens removed, number of referred variables and number of assigned variables. Clone pairs are detected

using CCFinder [1]. In another study [131] simultaneous modification support method is proposed to maintain consistency among code clones. They proposed the tool as *Libra* that finds number of code clones suitable for refactoring in open source software systems.

Basit *et al.* [152] conducted study of code clones in STL. The Standard Template Library (STL) is the collection of containers, algorithms, iterators, functions and adapters. CCFinder [1] is used to analyze patterns of simple code clones in STL. Experimental studies revealed that extensive amount of clones are found in the associative containers and adapters, while few numbers of clones are found in algorithms. They unified code clones in STL using a clone free representation that is supported by XML-based variant configuration language.

Jarzabek and Li [153] presented mixed-strategy approach that analyzes similar patterns in Java Buffer Library JDK 1.5. In this study, buffer classes are build and maintained using generative programming technique of XML-based variant configuration language (XVCL).

Tairas *et al.* [154][155][156] developed *CeDAR* (clone detection, analysis and refactoring) tool as an Eclipse plug-in that represents the location of code clones and also extend the refactoring capabilities of Eclipse, that allows simultaneous refactoring of code clone groups. It can parse the output of many clone detection tools at a single platform. The refactoring capabilities of CeDAR is extended by allowing more types of parameterized clones in a centralized manner.

Tairas and Gray [157] detected associations between clone classes for maintenance of code clones. An information retrieval technique as Latent Semantic Indexing (LSI) is used to cluster clone classes that have been detected initially by code clone detection tool. In their empirical evaluation, code clones are analyzed in Microsoft Windows NT kernel source code.

Tiarks *et al.* [158] suggested an improvement to be made in Type 3 clone detectors. In their findings they found that only 25% of Type 3 clones are detected correctly by code clone detectors. In their work, decision tree based algorithm is used to differentiate real clones from false positives.

Li *et al.* [96] proposed a hybrid approach to detect code clones in Erlang programs. They integrated clone detector and refactoring in Wrangler by removing code clones from Erlang programs. Wrangler is a tool that supports code refactoring in Erlang programs. Further, in another study code clones are detected and refactored for Haskell programs [159]. A framework as Haskell Refactorer (HaRe) is presented to detect and refactor Haskell programs. It works over an abstract syntax tree representation of Haskell program.

Shawky and Ali [160] proposed an approach for evaluating the similarity metrics used in metric based code clone detection. They assessed the set of metrics that are best suited for similarity prediction in context of code clone detection. In their findings, they found that in metric based clone detection approach precision can be increased by choosing an optimal sequence of metric vectors.

Krinke *et al.* [161] presented an approach to classify code clones of a clone pair into original and copied by extracting information from centralized version control system (CVS). Code clones are classified into three classes as identical, copied and unclassifiable. The levenshtein edit distances are calculated between different versions of a source code file for classification. In this study, small tolerances like few number of lines of code clones may be ignored for the purpose of classification.

Tairas *et al.* [162] represented code clone groups in a localized way such that the information about each clone in a clone group can be viewed at a single location to support refactoring. It is implemented as a part of CeDAR (clone detection, analysis and refactoring) tool [154]. An abstract syntax tree (AST) representation is used to

trace similarities and parameterized differences between clones in a clone group.

Choi *et al.* [163] proposed a filtering approach to extract code clones from source code for refactoring activity. Gemini provides quantitative information on clone sets through clone metrics to software developers. Clone sets are extracted using combination of clone metrics as average length of token sequences, ratio of non repeated token sequences and population.

Basit *et al.* [24][94][33] analyzed higher level similarity patterns in program files using structural clones. Structural clone shows bigger picture of simple clones and are created by finding recurring patterns of simple clones using frequent item set mining technique. In this study they analyzed the benefits of simple clones from structural clone perspective. *Clone Miner* is used to find simple code clones. In their finding they analyzed that nearly 50% of simple clones were the part of structural clones in 11 open source subject systems. The analysis of high level similarities through structural clones helps in refactoring, recovery and understanding of software systems.

Kanwal *et al.* [164] analyzed refactoring patterns in code clones during software evolution. They carried their empirical study on different versions of five open source Java systems. In this study they focused on the frequency of code clone refactoring in software. CloneMiner [94] is used to detect code clones among different versions of software system and Ref-finder [165] is used to extract useful refactoring patterns. In their findings they found that 40% of code clones that belongs to same clone class are refactored consistently. Moreover, it is found in their study that software developers are worried about the quality of a software system during software evolution.

2.3 Summary

Code clone detection is an active area of research and used for maintenance of code in a software system. This chapter provides the complete taxonomy of existing code clone detection approaches along with the concerns of code clone management. Code clone management is an umbrella activity that covers all concerns related to code cloning. Also, a detailed discussion with comparative analysis of semantic code clone detection techniques is provided with respect to various parameters. In this chapter, we have also covered other key concerns related to code cloning as code clone analysis, code clone visualization, code clones in websites, code clones in cross cutting concerns, impact of code clones on software quality and code clone evolution.

Summarizing the existing studies on code clone evolution, there are two basic studies based on revision level and at release level. These studies focus on detecting code clones in multiple versions of a program. It is found in both the studies that code clone genealogy exists in a software system. Moreover, each section is described with various code clone detection techniques with their advantages and disadvantages over the others. The analysis provided for various presented proposals also provides parameters to select one of the schemes with respect to its qualities over the others.

Chapter 3

Semantic Code Clones

Over the last few years, code clone detection is an integral part of software maintenance. Lot of advances in information and communication technologies have led to evolution of Internet of Things. The Internet of Things (IoT) is the network of objects embedded with software, sensors and network connectivity that enable these objects to collect and exchange data [166]. The software engineering community is not an exception, and the challenge is even more with the evolution of Internet of Things. Knowledge extraction from sensed raw data and analysis of source code from existing software resources that are embedded in Internet of Things will further enrich IoT with lot of information [167].

During this era, numerous code clone detection techniques have been presented. Copying code segments and then reuse these segments by pasting with or without updations are common activities in software development [4]. As evident from literature survey, most of the existing techniques for finding semantically equivalent code fragments require PDG. Semantic code clones are detected using sub graph isomorphism on PDG's that is NP-complete and returns approximate solutions. Semantic code clones are functionally and logically equivalent code fragments. These code clone fragments can be used to re-engineer the software objects.

In this chapter, a semantic oriented code clone detection approach is proposed. This approach eliminates the effect of statement reordering, inversion of control statements and insertion of irrelevant statements in a program by using reaching definition and

liveness analysis. Reaching definition and liveness analysis are formal methods of program analysis and are used to carry data flow analysis. Reaching definition analysis outputs set of definitions that remain live at the start or end of the statement and liveness analysis is used to eliminate irrelevant insertions form source code fragments.[1] An efficient semantic code clone detection algorithm based on reaching definition and liveness analysis is proposed in this chapter for extraction of semantically equivalent clone pairs. Moreover, code clone pairs are clustered into clone groups in next chapter.

3.1 Semantic Code Clone Detection

This section introduces the problem of semantic code cloning and techniques used in proposed semantic code clone detection approach. As per literature review, there are numerous approaches to detect code clones in a software system. Most of the semantic code clone detection techniques are unable to provide a heuristic solution to solve the issues like statements reordering, inversion of control predicates and insertion of irrelevant statements. To address these issues, the proposed research work includes design and implementation of semantic code clone detection technique in this chapter.

3.1.1 Tokenization

Tokenization or lexical analysis is the process of converting a string or sequence of characters into a sequence of tokens or lexeme. JastAdd [168] specification is used for tokenization of Java source code. Tokenization consists of different types of tokens as

[1]The contents of this chapter are published as per following details:

- Rajkumar Tekchandani, Rajesh Bhatia, Maninder Singh, "Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis", The Journal of Supercomputing, pp. 1-28, Springer, IF 1.326, August 2016.

Keywords, Numerals or literals, Separators, Identifiers and Errors. A 'Numeral' starts with one or more digits followed by a point and one or more digits. An 'Operator' consists of the set of operator tokens.

A 'Separator' consists of set of separator tokens. An 'Identifier' consisting of one or more letters. The last pattern 'Error' is used to match any characters not appearing with in the brackets. This will prevent the scanner from throwing an exception for unrecognized tokens and makes the parser simpler. The 'Skip' specification just lists the characters that will be skipped during scanning and tokens will not be generated for them. It ignores the characters like whitespace, tab (\t), newline (\n) and carriage return (\r) in the given source code.

3.1.2 Parsing

Parsing or syntactic analysis is the process of analyzing the string of symbols in the form of abstract syntax trees. The abstract syntax tree (AST) of the given source code is generated with the help of parser. It consists of non-terminals and terminals. Root node and intermediate nodes are termed as non-terminals and leaf nodes are the terminal nodes of the AST. When AST is traversed in preorder traversal the statements of the source code are ordered in the same sequence as in the original source code [27]. With the rules defined in JastAdd grammar [168] source code of the program is parsed and AST's are generated for given source code.

3.1.3 Control Flow Analysis

Control flow graphs (CFG's) are used to obtain the control dependency between the nodes of the graph that is useful for generating the semantic information about the source code.

Definition 1 (Control Dependency) [169]: There exists a control dependency between 'V_1' to 'V_2', if the following conditions get satisfied

- V_1 is a conditional predicate, and the result of 'V_1' directly influences the execution of 'V_2' i.e., whether 'V_2' will execute or not depends upon 'V_1'.

Control flow analysis is done on CFG's that are generated corresponding to the source code. We have used Eclipse IDE plug-in as control flow graph factory [169] for the generation of CFG's.

3.1.4 Data Flow Analysis

Data Flow Analysis derives the information about the flow of data along program execution paths. The execution path from point 'p_1' to 'p_n' is a sequence of points p_1, p_2, ... p_n such that for each i = 1, 2, 3, ... n − 1, 'p_i' is the point immediately preceding a statement and 'p_{i+1}' is the point immediately following that same statement, or 'p_i' is the end of some block and 'p_{i+1}' is the beginning of a successor block.

Definition 2 (Data Dependency) [170]: There is a data dependency from element 'V_1' to 'V_2', via variable 'X', if the following conditions get satisfied

- 'V_1' defines 'X' (IdDecl of X), 'V_2' references 'X' (IdUse of X), and there is at least one execution path from 'V_1' to 'V_2' without redefining 'X'.

Semantic analysis typically consists of data flow analysis, to bind uses of identifiers to their declarations. During data flow analysis two AST classes as 'IdDecl' and 'Iduse' are used, where 'IdDecl' is an identifier that names a declaration and 'IdUse' refers to a declaration.

3.2 Semantic Code Clone Detection Approach

In our approach, we have considered semantically equivalent source codes as set S = $\{S_1, S_2, S_3, S_4\}$ in Table 3.1, where $S_1 \rightarrow$ Original Source Code, $S_2 \rightarrow$ Code with statement reordering, $S_3 \rightarrow$ Code with inversion of control statements and $S_4 \rightarrow$ Code with insertion of irrelevant statements. Figure 3.1 shows proposed semantic code clone detection approach.

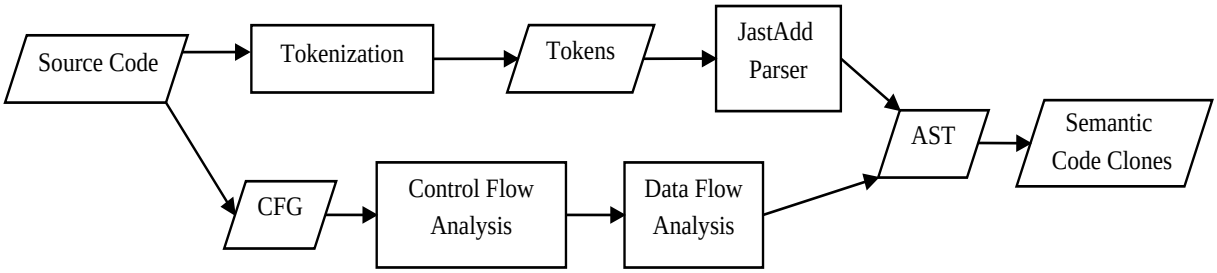


Figure 3.1: Proposed semantic code clone detection approach.

- Token sequences have been generated for each source code S_1, S_2, S_3 and S_4 for input to parser.
- Parsing is done to create AST's as AST_{S_1}, AST_{S_2}, AST_{S_3} and AST_{S_4}.
- Control flow graph (CFG's) are generated as CFG_{S_1}, CFG_{S_2}, CFG_{S_3} and CFG_{S_4} for control flow analysis.
- Data flow analysis at AST level is done using reaching definition and liveness analysis on CFG's for S_1, S_2, S_3 and S_4.
- Finally, after data flow analysis semantic code clones are detected.

In Table 3.1:

- S_2 differs from S_1 by reversing the order of statements in line number 8 and 9 in block B2 of S_2.

Table 3.1: Source code S_1,S_2,S_3,S_4 with different syntactic structures.

(a) Original Source Code (S_1)

```
1.public class S1{
2.public static void main(String[]args){
3.d1: int i=1; |
4.d2: int j=16;|Block B1
5.d3: int k=0; |
6.while (i < 8)
7.{
8.d4:    k = k + 1;| Block B2
9.d5:    j = j - 1;|
10.if (j <= 16)
11.d6:       k = 1;| Block B3
12.else
13.d7:       j = 6;| Block B4
14.d8:i = i + 1;   | Block B5
15.}
16.System.out.println("i:"+i);
17.System.out.println("j:"+j);
18.System.out.println("k:"+k);
19.}
20.}
Output of S1 : i = 8, j = 9, k = 1
```

(b)Code with Statement Reordering(S_2)

```
1.public class S2{
2.public static void main(String[]args){
3.d1:int k=0;   |
4.d2:int j=16;  |    Block B1
5.d3:int i=1;   |
6.while (i < 8)
7.{
8.d4:j = j - 1; | Block B2
9.d5:k = k + 1; |
10.if (j <= 16)
11.d6:k = 1;     | Block B3
12.else
13.d7:j = 6;     | Block B4
14.d8:i = i + 1;| Block B5
15.}
16.System.out.println("i:"+i);
17.System.out.println("j:"+j);
18.System.out.println("k:"+k);
19.}
20.}
Output of S2 : i = 8, j = 9, k = 1
```

(c) Code with Inversion of Control Predicate (S_3)

```
1.public class S3{
2.public static void main(String[]args){
3.d1: int j=16;|
4.d2: int k=0; | Block B1
5.d3: int i=1; |
6.while (i < 8)
7.{
8.d4:k = k + 1;| Block B2
9.d5:j = j - 1;|
10.if (j >= 16)
11.d6:j = 6; | Block B3
12.else
13.d7:k = 1; | Block B4
14.d8:i = i + 1; | Block B5
15.}
16.System.out.println("i:"+i);
17.System.out.println("j:"+j);
18.System.out.println("k:"+k);
19.}
20.}
Output of S3 : i = 8, j = 9, k = 1
```

(d) Code with Insertion of Irrelevant Statement (S_4)

```
1.public class S4{
2.public static void main(String[]args){
3.d1: int i=1; |
4.d2: int j=16;|  Block B1
5.d3: int k=0; |
6.d4: int l=5; |
7.while (i < 8)
8.{
9. d5:k = k + 1;|  Block B2
10.d6:j = j - 1;|
11.if (j <= 16)
12.d7:k = 1; | Block B3
13.else
14.d8:j = 6;     | Block B4
15.d9:i = i + 1;| Block B5
16.}
17.System.out.println("i:"+i);
18.System.out.println("j:"+j);
19.System.out.println("k:"+k);
20.}
21.}
Output of S4 : i = 8,j = 9,k = 1
```

- S_3 differs from S_1 by inversion of control or branch predicate in line number 10 in S_3.
- S_4 differs from S_1 by insertion of irrelevant statement at line number 6 in block B1 of S_4 .
- All differs in their syntactic structures but having the same semantics.

3.3 Algorithm Description

In this section, detailed description of proposed algorithm (section 3.3.1) is presented for detection of semantic code clone pairs. In algorithm 1, semantic code clone pairs are found by calling reaching definition analysis (section 3.3.2) and liveness analysis (section 3.3.3).

3.3.1 Semantic Code Clone Detection

In algorithm 1, reaching definitions have been calculated for all the sub-blocks (Bj) of program S_i. The values of reaching definition for S_i and $S_{(i+1)}$ are compared for finding reordered clone pairs. If both the values of reaching definition are equivalent then these sub-blocks in source code S_i and $S_{(i+1)}$ are semantically equivalent code fragments. Further, semantic clone pairs ($ClonePair_1$,$ClonePair_2$, ...,$ClonePair_m$) are detected using liveness analysis. If ($\exists$ k $\notin$ (in[k] $\parallel$ out[k])), then statement 'k' is an irrelevant statement. A refactored program (S_i') is retrieved after eliminating irrelevant statements from original source code S_i. Finally, the output of original program S_i is compared with refactored program S_i'. If the output of S_i and S_i' generates the same semantic meaning, then both have same data flow paths and is returned as semantic clone pair (S_i,S_i').

Algorithm 1: Semantic code clone detection

Input : Program(S_i), where {i := 1, 2, 3...,m}, Block B $\in S_i$, in[B] $\in S_i$, B := {Bj | j := 1, 2, 3...,n} and statement k $\in S_i$

Output: Semantic Code Clone Fragments: $ClonePair_1$, $ClonePair_2$, ... ,$ClonePair_m$

```
for i := 1 to m do
    for ∀ B ∈ S_i do
        for ∀ Bj ∈ B do
            call reaching definition analysis(kill[Bj],gen[Bj]);
            out[Bj.S_i];
        end
    end
end
for i := 1 to m do
    for ∀ B ∈ S_i do
        for ∀ Bj ∈ B do
            if out[Bj.S_i] ≡ out[Bj.S_(i+1)] then
                ClonePair_i(Bj.S_i, Bj.S_(i+1));
            end
            for ∀ statement k ∈ Bj do
                call liveness analysis(succ[k],use[k],def[k]);
                fetch line number of statement k of Bj such that k ∉ (in[k] || out[k]);
                if ∃ k ∉ (in[k] || out[k]) then
                    statement k is a dead code;
                    after elimination of k;
                    S_i' := S_i − k;
                end
            end
        end
    end
end
for i := 1 to m do
    for ∀ S_i do
        if output[S_i'] == output[S_i] then
            ClonePair_i(S_i',S_i);
        end
    end
end
```

3.3.2 Reaching Definition Analysis

Reaching definition analysis outputs the set of definitions that remain live at the start or end of statement or set of statements included inside the block. We define four sets as: **In**, **Out**, **Gen** and **Kill** for reaching definition analysis. **In[s|B]** and **Out[s|B]** are the data flow values before and after each statement 's' or set of statements enclosed inside the block 'B'. **Gen[s|B]** and **Kill[s|B]** are constants. These are the attributes that are created at a node for a particular statement 's' or set of statements enclosed inside the block 'B' and are computed before data flow analysis. Gen[s|B] is the set of definitions "generated by s|B", that are visible immediately after 's|B'. Kill[s|B] is the union of the definitions in all 's|B' of the flow graph that are killed by individual statements in 's|B' [170].

Definition 3 Let D_z is the set of definitions for variable 'z' and 'd' is the only definition of variable 'z' that will reach at the end of 's|B' i.e., Gen[s|B] = d, then

$$kill[s|B] = D_z - \{d\} \; or \; kill[s|B] = D_z \bigcap \{d\}^c \tag{3.1}$$

Definition 4 The set in[s|B] is defined as the set of definitions reaching at the beginning of 's|B', provided the flow of control passes through the entire program and similarly, out[s|B] is the set of definitions reaching at the end of the 's|B'.

$$in[s|B] = \bigcup_{p \in predecessor of [s|B]} out[p] \tag{3.2}$$

$$out[s|B] = gen[s|B] \cup (in[s|B] - kill[s|B]) \tag{3.3}$$

or

$$out[s|B] = gen[s|B] \cup (in[s|B] \cap kill[s|B]^c) \tag{3.4}$$

Definition 5 Let B is the block containing 'x1' and 'x2' as sequential statements then [170]

$$gen[B] = gen[x2] \cup (gen[x1] - kill[x2]) \tag{3.5}$$

$$kill[B] = kill[x2] \cup (kill[x1] - gen[x2]) \tag{3.6}$$

Figure 3.2 shows the control flow graph for the source code shown in S_1 (Table 3.1(a)),

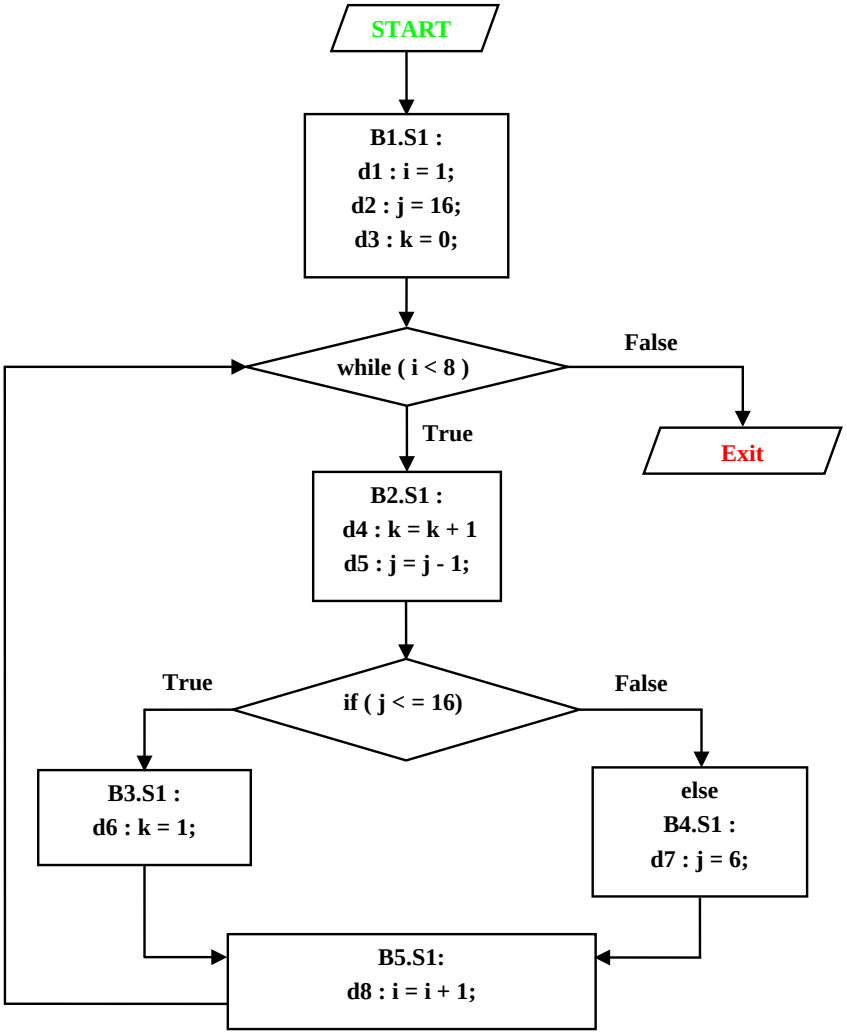


Figure 3.2: Control flow graph for S_1, Table 3.1(a).

which shows execution of while looping construct. The generation and kill sets for source code S_1 are represented with bit values as '0' and '1' for definitions 'd1' to 'd8' in Figure

3.3. The advantage of using bit vector representation for set of definitions is that the set operations on bit vectors can be implemented efficiently. In Figure 3.3, gen[d1,d2] and kill[d1,d2] are calculated for definitions 'd1' and 'd2' (Table 3.1(a)) of S_1 in following sequence of steps using definition 5.

$$gen[d1] = 10000000 \tag{3.7}$$

$$gen[d2] = 01000000 \tag{3.8}$$

kill [d1] = all possible definitions of variable 'i' in Figure 3.2 − current definition of 'i'

$\Rightarrow$ kill [d1] = 1000 0001 − 1000 0000

$\Rightarrow$ 1000 0001 $\cap$ 1000 0000^c, where 'c' is binary complement

$\Rightarrow$ 1000 0001 $\cap$ 0111 1111

$\Rightarrow$ 0000 0001.

$$kill[d1] = 00000001 \tag{3.9}$$

kill [d2] = all possible definitions of variable 'j' in Figure 3.2 − current definition of 'j'

kill [d2] = 0100 1010 − 0100 0000

$\Rightarrow$ 0100 1010 $\cap$ 0100 0000^c, where 'c' is binary complement

$\Rightarrow$ 0100 1010 $\cap$ 1011 1111

$\Rightarrow$ 0000 1010.

$$kill[d2] = 00001010 \tag{3.10}$$

$$gen[d1, d2] = gen[d2] \cup (gen[d1] - kill[d2]) \tag{3.11}$$

$\Rightarrow$ gen[d1,d2] = gen[d2] $\cup$ (gen[d1] $\cap$ kill[d2]^c), where 'c' is binary complement

Substituting (3.7), (3.8) and (3.10) in (3.11)

$\Rightarrow$ 0100 0000 $\cup$ (1000 0000 − 0000 1010)

$\Rightarrow$ 0100 0000 $\cup$ (1000 0000 $\cap$ 0000 1010^c), where 'c' is binary complement

Algorithm 2: Reaching Definition and Liveness Analysis

```
Input  : A program[p], control flow graph having kill[B], gen[B], succ[i], use[i]
         and def[i], i ∈ statement, Block B := {b_k | k = 1, 2, 3..., n}
Output: in[B] and out[B]
initially in[B] := ∅;
for ∀ block B do
    reaching definition analysis(kill[B], gen[B])
    out[B] := gen[B], flag = true;
    while flag do
        for ∀ block B do
            in[B] := ∪_{p∈pred of i} out[p];
            previous out := out[B];
            out[B] := gen[B] ∪(in[B] ∩ kill[B]^c);
            if out[B] ≠ previous out then
                flag := true;
            else
                flag := false;
            end
        end
    end
    liveness analysis(succ[i], use[i], def[i]) for ∀ statement i ∈ B do
        initially out[i] := in[i] := ∅, flag := true;
        while flag do
            flag := false;
            for ∀ statement i ∈ B do
                oldout := out[i];
                out[i] := ∪{in[j] | j ∈ succ[i]};
                oldin := in[i];
                in[i] := use[i] ∪(out[i] − def[i]);
                if out[i] ≠ oldout && in[i] ≠ oldin then
                    flag := true;
                else
                    flag := false;
                end
            end
        end
    end
end
```

$\Rightarrow$ 0100 0000 $\cup$ (1000 0000 $\cap$ 1111 0101)

$\Rightarrow$ 0100 0000 $\cup$ 1000 0000 $\Rightarrow$ 1100 0000.

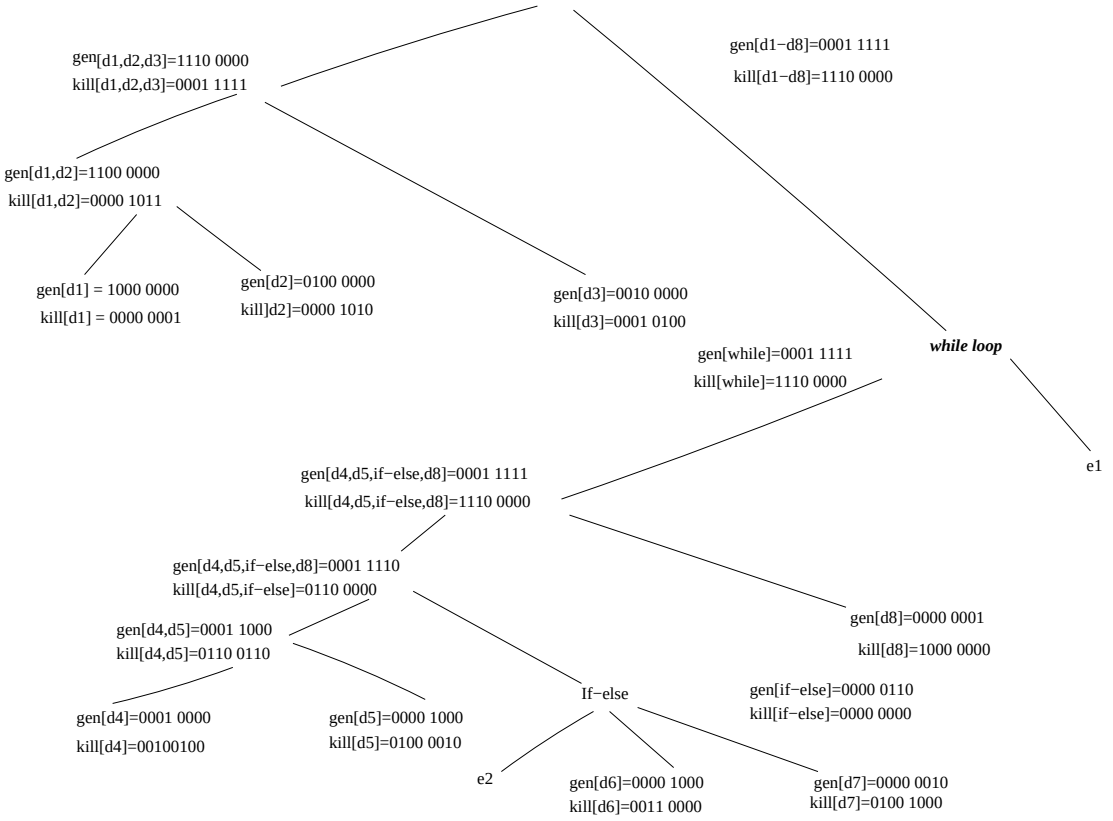


Figure 3.3: The gen and kill sets for definitions d1-d8 at nodes in abstract syntax tree of S_1 (Table 3.1(a)).

Table 3.2: Computation of **in** and **out** for B.S_1.

Statement	**Initial**		**Pass 1**		**Pass 2**	
Block B.S_1	**in**[**B.**S_1]	**out**[**B.**S_1]	**in**[**B.**S_1]	**out**[**B.**S_1]	**in**[**B.**S_1]	**out**[**B.**S_1]
B1.S_1	0000 0000	1110 0000	0000 0000	1110 0000	0000 0000	1110 0000
B2.S_1	0000 0000	0001 1000	1110 0111	1001 1001	1111 1111	1001 1001
B3.S_1	0000 0000	0000 0100	1001 1001	1000 1101	1001 1001	1000 1101
B4.S_1	0000 0000	0000 0010	1001 1001	1001 0011	1001 1001	1001 0011
B5.S_1	0000 0000	0000 0001	1001 1111	0001 1111	1001 1111	0001 1111

$\Rightarrow$ gen[d1,d2] = 1100 0000.

It represents that the definitions ‘d1’ and ‘d2’ are generated in gen[d1,d2] as shown in

Figure 3.3. Further,

$$kill[d1, d2] = kill[d2] \cup (kill[d1] - gen[d2]) \tag{3.12}$$

Substituting (3.8), (3.9) and (3.10) in (3.12)

$\Rightarrow$ 0000 1010 $\cup$ (0000 0001 $-$ 0100 0000)

$\Rightarrow$ 0000 1010 $\cup$ (0000 0001 $\cap$ 0100 0000^c), where 'c' is binary complement

$\Rightarrow$ 0000 1010 $\cup$ (0000 0001 $\cap$ 1011 1111)

$\Rightarrow$ 0000 1010 $\cup$ 0000 0001 $\Rightarrow$ 0000 1011.

$\Rightarrow$ kill[d1,d2] = 0000 1011.

In Figure 3.3, value of kill[d1,d2] = 0000 1011, represents that the definitions 'd5', 'd7' and 'd8' have not been defined at this program point, because 'd1' and 'd2' have been already generated at this program point. Similarly, all the other definitions are generated in AST representation for S_1 (Table 3.1(a)) as shown in Figure 3.3.

Definition 6 Let *if* block is represented by 'x1' and *else* block by 'x2' then

$$gen[\textit{if-else}] = gen[x1] \cup gen[x2] \tag{3.13}$$

$$kill[\textit{if-else}] = kill[x1] \cap kill[x2] \tag{3.14}$$

The gen [*if-else*] in Figure 3.3 is calculated using Eq.3.13 of definition 6.

$\Rightarrow$ gen[d6] = gen[if] and gen[d7] = gen[else]

$\Rightarrow$ gen[if] = 0000 0100 and gen[else] = 0000 0010

$\Rightarrow$ gen[*if-else*] = gen[*if*] $\cup$ gen[*else*]

$\Rightarrow$ gen[if-else] = gen[if] $\cup$ gen[else]

$\Rightarrow$ 0000 0100 $\cup$ 0000 0010

$\Rightarrow$ 0000 0110.

The gen[*if-else*] = 0000 0110 represents that definitions 'd6' and 'd7' are generated as

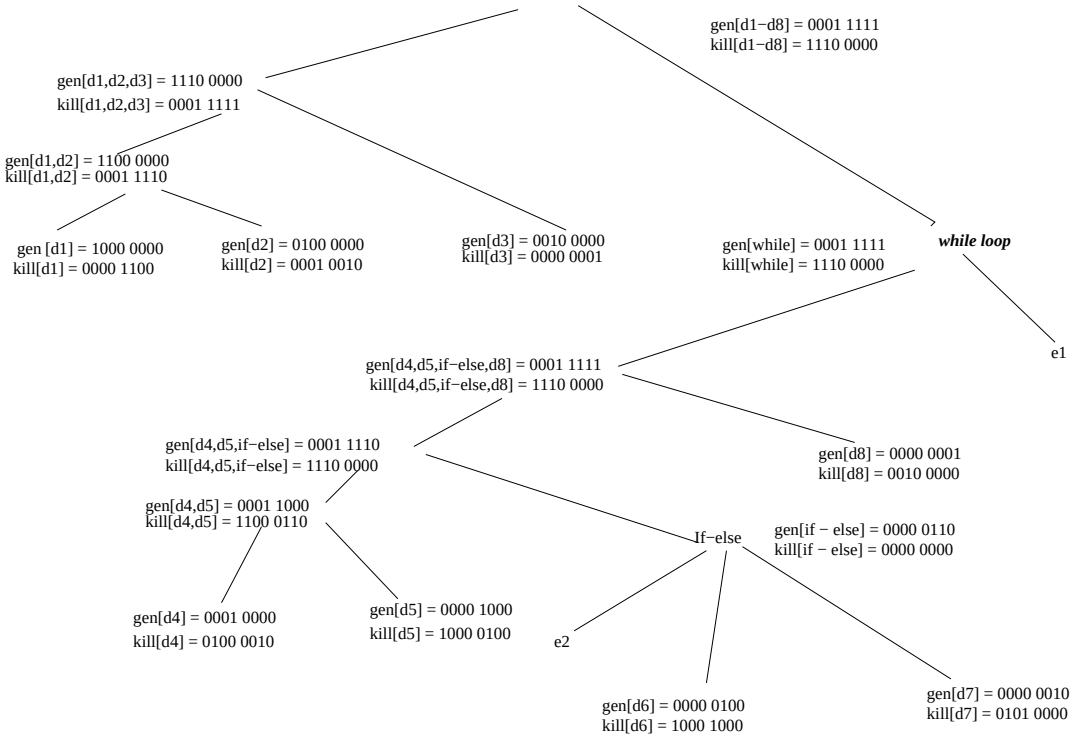


Figure 3.4: The gen and kill sets for definitions d1-d8 at nodes in abstract syntax tree of S_2 (Table 3.1(b)).

Table 3.3: Computation of **in** and **out** of B.S_2.

Statement	**Initial**		**Pass 1**		**Pass 2**	
Block B.S_2	**in**[**B.**S_2]	**out**[**B.**S_2]	**in**[**B.**S_2]	**out**[**B.**S_2]	**in**[**B.**S_2]	**out**[**B.**S_2]
B1.S_2	0000 0000	1110 0000	0000 0000	1110 0000	0000 0000	1110 0000
B2.S_2	0000 0000	0001 1000	1110 0111	0011 1001	1111 1111	0011 1001
B3.S_2	0000 0000	0000 0100	0011 1001	0011 0101	0011 1001	0011 0101
B4.S_2	0000 0000	0000 0010	0011 1001	0010 1011	0011 1001	0010 1011
B5.S_2	0000 0000	0000 0001	0011 1111	0001 1111	0011 1111	0001 1111

shown in AST of Figure 3.3.

Similarly, The kill[*if-else*] is calculated using Eq.3.14 of definition 6.

$\Rightarrow$ kill[d6] = kill[if] and kill[d7] = kill[else]

$\Rightarrow$ kill[if] = 0011 0000 and kill[else] = 0100 1000

$\Rightarrow$ kill[if-else] = kill[if] $\cap$ kill[else]

$\Rightarrow$ 0011 0000 $\cap$ 0100 1000

$\Rightarrow$ 0000 0000.

The input to algorithm 2, for reaching definition is source code S_i, CFG having kill[B] and gen[B]. Reaching definition analysis function in algorithm 2, computes the two equations as in[B] and out[B]. Initially values of in[B] is equal to 'ϕ', out[B] is equal to gen[B] and flag is set to true. The value of flag remains true until the old value of out[B] is not equal to new value of out[B]. The while loop in algorithm 2, for reaching definition analysis will terminate when the value of flag becomes false.

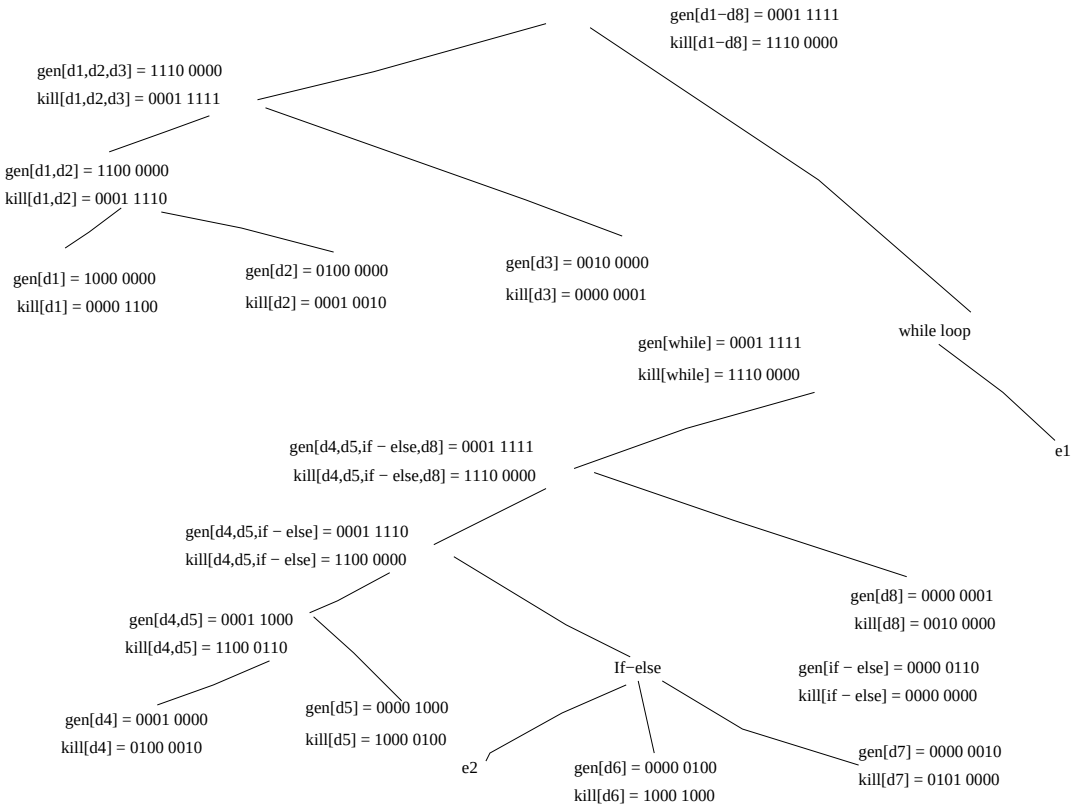


Figure 3.5: The gen and kill sets for definitions d1-d8 at nodes in abstract syntax tree of S_3 (Table 3.1(c)).

The generation and kill sets for S_1, S_2, S_3, and S_4 are calculated in Figure 3.3, 3.4, 3.5 and 3.6 respectively. Further, after calculating the values of all the nodes of AST with their corresponding generation and kill sets, values of **in** and **out** sets are calculated

Table 3.4: Computation of **in** and **out** of B.S_3.

Statement	**Initial**		**Pass 1**		**Pass 2**	
Block B.S_3	**in**[**B.**S_3]	**out**[**B.**S_3]	**in**[**B.**S_3]	**out**[**B.**S_3]	**in**[**B.**S_3]	**out**[**B.**S_3]
B1.S_3	0000 0000	1110 0000	0000 0000	1110 0000	0000 0000	1110 0000
B2.S_3	0000 0000	0001 1000	1110 0111	0011 1001	1111 1111	0011 1001
B3.S_3	0000 0000	0000 0100	0011 1001	0011 0101	0011 1001	0011 0101
B4.S_3	0000 0000	0000 0010	0011 1001	0010 1011	0011 1001	0010 1011
B5.S_3	0000 0000	0000 0001	0011 1111	0001 1111	0011 1111	0001 1111

using definition 4, for Bj.S_1, Bj.S_2, Bj.S_3, and Bj.S_4 where j $\in$ {1,2...,5} as shown in Table 3.2, 3.3, 3.4 and 3.5 respectively. For example, **in** and **out** sets of B2.S_1 (Figure 3.2) are calculated using definition 4 as follows,

$$out[B2.S_1] = gen[B2.S_1] \cup (in[B2.S_1] - kill[B2.S_1]) \tag{3.15}$$

In Figure 3.2, Block B1.S_1 = {d1,d2,d3}, B2.S_1 = {d4,d5}, B3.S_1 = {d6}, B4.S_1 = {d7} and B5.S_1 = {d8}. Initially, in[Bj.S_1] = ϕ and out[Bj.S_1] = gen[Bj.S_1], where j $\in$ {1,2...,5}. To find out[B2.S_1], i.e., the set of definitions reaching at the end of block B2.S_1, we have to find the set of definitions reaching at the beginning of B2.S_1, i.e., in[B2.S_1].

Using Eq. 3.2, we get in[B2.S_1] in pass 1 of Table 3.2 as follows.

$$in[B2.S_1]_{Pass1} = out[B1.S_1] \cup out[B3.S_1] \cup out[B4.S_1] \cup out[B5.S_1] \tag{3.16}$$

Initially in Table 3.2, out[B1.S_1] = gen[B1.S_1], out[B3.S_1] = gen[B3.S_1], out[B4.S_1] = gen[B4.S_1] and out[B5.S_1] = gen[B5.S_1]

$\Rightarrow$ in[B2.S_1] = gen[B1.S_1] $\cup$ gen[B3.S_1] $\cup$ gen[B4.S_1] $\cup$ gen[B5.S_1]

$\Rightarrow$ gen[d1,d2,d3] $\cup$ gen[d6] $\cup$ gen[d7] $\cup$ gen[d8]

Taking the values of gen[d1,d2,d3], gen[d6], gen[d7] and gen[d8] from AST in Figure 3.3 and then substituting these in Eq.(3.16), we get $in[B2.S_1]_{Pass1}$ as:

$\Rightarrow$ 1110 0000 $\cup$ 0000 0100 $\cup$ 0000 0010 $\cup$ 0000 0001

$\Rightarrow$ 1110 0111.

$\Rightarrow$ $in[B2.S_1]_{Pass1}$ = 1110 0111 represents 'd1', 'd2', 'd3', 'd6', 'd7' and 'd8' are the set of definitions that are reached at the start of block B2.S_1. The set of definitions 'd4' and 'd5' have not reached at the beginning of B2.S_1, because these are generated in block B2.S_1.

From AST in Figure 3.3, we get

gen[B2.S_1] = gen[d4,d5] = 0001 1000 and kill[B2.S_1] = kill[d4,d5] = 0110 0110.

Substituting the values of gen[B2.S_1], kill[B2.S_1] and in[B2.S_1] in Eq. 3.15, $out[B2.S_1]_{Pass1}$ is obtained as follows.

$\Rightarrow$ 0001 1000 $\cup$ (1110 0111 $-$ 0110 0110)

$\Rightarrow$ 0001 1000 $\cup$ (1110 0111 $\cap$ 0110 0110^c), where 'c' is binary complement

$\Rightarrow$ 0001 1000 $\cup$ (1110 0111 $\cap$ 1001 1001)

$\Rightarrow$ 0001 1000 $\cup$ 1000 0001

$\Rightarrow$ 1001 1001.

$\Rightarrow$ $out[B2.S_1]_{Pass1}$ = 1001 1001.

Further, in Pass 2 of Table 3.2:

Again **in** and **out** sets for block B2.S_1 are calculated iteratively, until the same value of out[B2.S_1] is found. If the value of out[B2.S_1] in pass 1 is equal to out[B2.S_1] in pass 2, then algorithm 2 terminates. In pass 2, values generated from pass 1 for **in** and **out** sets are used, which can be retrieved from pass 1 in Table 3.2.

$in[B2.S_1]_{Pass2} \Rightarrow out[B1.S_1]_{Pass1} \cup out[B3.S_1]_{Pass1} \cup out[B4.S_1]_{Pass1} \cup out[B5.S_1]_{Pass1}$

$\Rightarrow$ 1110 0000 $\cup$ 1000 1101 $\cup$ 1001 0011 $\cup$ 0001 1111

$\Rightarrow$ 1111 1111.

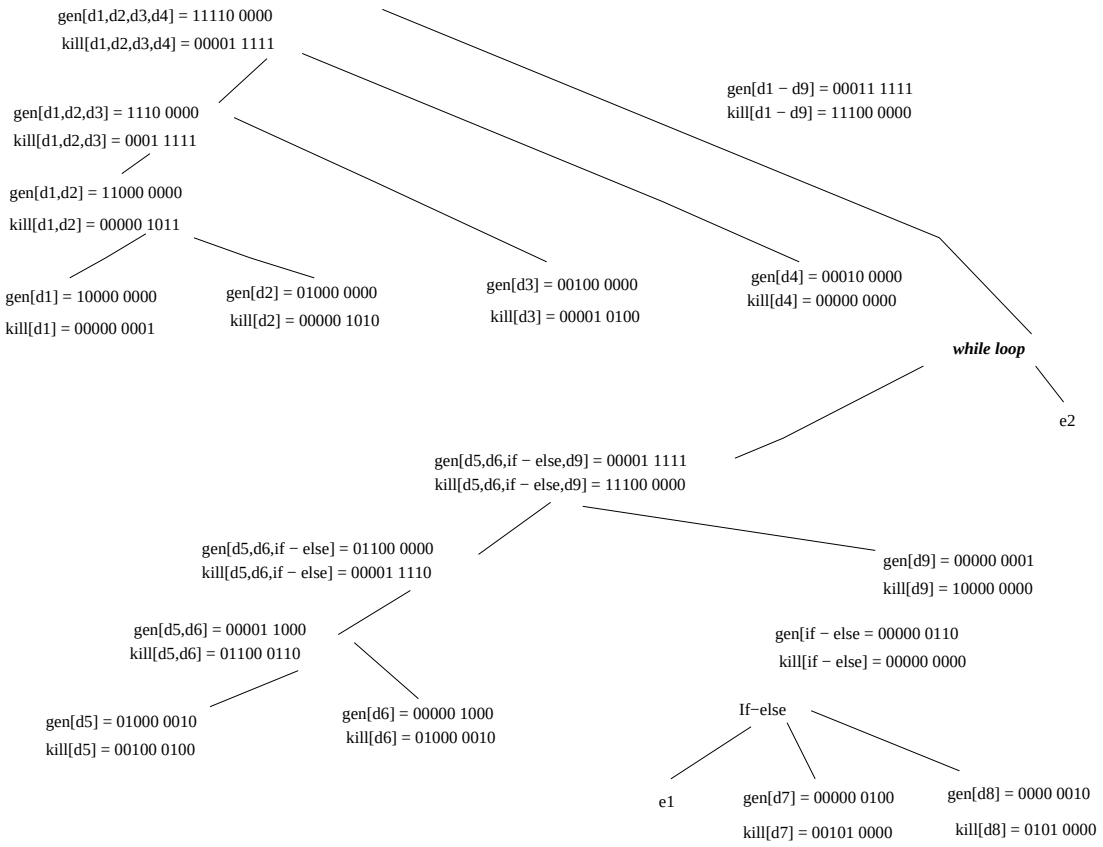


Figure 3.6: The gen and kill sets for definitions d1-d9 at nodes in abstract syntax tree of S_4 (Table 3.1(d)).

Table 3.5: Computation of **in** and **out** of B.S_4.

Statement	**Initial**		**Pass 1**		**Pass 2**	
Block B.S_4	**in**[**B.**S_4]	**out**[**B.**S_4]	**in**[**B.**S_4]	**out**[**B.**S_4]	**in**[**B.**S_4]	**out**[**B.**S_4]
B1.S_4	00000 0000	11110 0000	00000 0000	11110 0000	00000 0000	11110 0000
B2.S_4	00000 0000	00001 1000	11110 0111	10011 1001	11111 1111	10011 1001
B3.S_4	00000 0000	00000 0100	10011 1001	10010 1101	10011 1001	10010 1101
B4.S_4	00000 0000	00000 0010	10011 1001	10011 0011	10011 1001	10011 0011
B5.S_4	00000 0000	00000 0001	10011 1111	00011 1111	10011 1111	00011 1111

$in[B2.S_1]_{Pass2}$ = 1111 1111.

$out[B2.S_1]_{Pass2}$ = gen[d4,d5] $\cup$ ($in[B2.S_1]_{Pass2}$−kill[d4,d5])

$\Rightarrow$ 0001 1000 $\cup$ (1111 1111 $-$ 0110 0110)

$\Rightarrow$ 0001 1000 $\cup$ (1111 1111 $\cap$ 01100110^c), where 'c' is binary complement

$\Rightarrow$ 0001 1000 $\cup$ (1111 1111 $\cap$ 1001 1001) $\Rightarrow$ 0001 1000 $\cup$ 1001 1001

$\Rightarrow$ 1001 1001.

$\Rightarrow$ $out[B2.S_1]_{Pass2} \equiv$ 1001 1001 $\equiv out[B2.S_1]_{Pass1}$

$\Rightarrow$ $out[B2.S_1]_{Pass2} \equiv out[B2.S_1]_{Pass1}$.

out[B2.S_1] $\Rightarrow$ d1 = 1, d2 = 0, d3 = 0, d4 = 1, d5 = 1, d6 = 0, d7 = 0 and d8 = 1. In **out[B2.**S_1] value of bit vector is 1001 1001 represents that definitions 'd4' and 'd5' have been generated and definitions 'd1' and 'd8' are reached at the end of block B2.S_1. Similarly, in Table 3.2 the values of **in** and **out** sets for rest of the blocks of source code S_1 are generated in $Pass_2$. The values of out[B1.S_1], out[B3.S_1], out[B4.S_1] and out[B5.S_1] are represented using bit vector notations as follows.

- **out[B1.**S_1] $\Rightarrow$ d1 = 1, d2 = 1, d3 = 1, d4 = 0, d5 = 0, d6 = 0, d7 = 0 and d8 = 0. In **out[B1.**S_1], the value of bit vector represents that definitions 'd1', 'd2' and 'd3' have been generated at the end of Block B1.S_1.

- **out[B3.**S_1] $\Rightarrow$ d1 = 1, d2 = 0, d3 = 0, d4 = 0, d5 = 1, d6 = 1, d7 = 0 and d8 = 1. In **out[B3.**S_1], the value of bit vector represents that definition 'd6' is generated and definitions 'd1', 'd5' and 'd8' have been reached at the end of Block B3.S_1.

- **out[B4.**S_1] $\Rightarrow$ d1 = 1, d2 = 0, d3 = 0, d4 = 1, d5 = 0, d6 = 0, d7 = 1 and d8 = 1. In **out[B4.**S_1], the value of bit vector represents that definition 'd7' is generated and definitions 'd1', 'd4' and 'd8' have been reached at the end of Block B4.S_1.

- **out[B5.**S_1] $\Rightarrow$ d1 = 0, d2 = 0, d3 = 0, d4 = 1, d5 = 1, d6 = 1, d7 = 1 and d8 = 1. In **out[B5.**S_1], the value of bit vector represents that definition 'd8' is generated and definitions 'd4', 'd5', 'd6' and 'd7' have been reached at the end of Block B5.S_1.

Similarly, the set of reaching set definitions have been generated for S_2, S_3 and S_4 because basic constructs will remain same for any length of program and their control structure will decide structure of the program. We have analyzed looping conditions with statement reordering, code inversion and insertion of irrelevant statements. In Table 3.1, S_2 represents statement reordering in block **B1** between definitions {d1, d2, d3} and block **B2** between definitions {d4, d5} with respect to original source code represented in S_1. Source code S_3 represents the change in the blocks **B3** and **B4** by inverting the condition of if-else predicate with respect to original source code S_1. In source code S_4, a statement as (d4: l = 5) is inserted in block **B1** that is not redefined again in the program, but it may be used at some program point in the source code.

As given in the future scope of Baxter [11], for achieving deep semantic orientation we have to represent the program with explicit control and data flow dependencies. By achieving control and data flows explicitly we can find semantically equivalent code fragments that would be insensitive to change in identifier names, statement reordering and insertion to irrelevant statements. As discussed in related work irrelevant statements, statement reordering and inverted branch statements are just used to hide code clones in source code. The data dependencies between various program points in a program are not affected by these issues.

In algorithm 2, data flow analysis is achieved by the combination of reaching definition and liveness analysis. Using algorithm 1, we have derived the results of semantic code clone analysis in section 5.1. The abstract syntax trees with generation and kill sets for definitions 'd1' to 'd8' for S_2 and S_3 have been generated in Figure 3.4 and 3.5. Similarly, in Figure 3.6 the AST with generation and kill sets for definitions 'd1' to 'd9' for S_4 is generated. It represents the insertion of irrelevant statement as (d4: l = 5), that is further eliminated from source code using liveness analysis.

3.3.3 Liveness Analysis

Definition 7 A variable 'x' is live at some point 'p' in the program, if the value of 'x' at 'p' could be used along some path starting at 'p' in the flow graph, otherwise variable 'x' is dead at program point 'p'. For each statement 'i' we use two sets to carry the liveness information as : in[i] and out[i]. The in[i] holds the set of variables that are live at the start of statement 'i' and out[i] holds the set of variables that are live at the end of statement 'i'. Let 'i' be a statement and succ[i] is the successor node of statement 'i', then

$$out[i] = \bigcup_{j \in succ[i]} in[j] \tag{3.17}$$

$$in[i] = use[i] \cup (out[i] - def[i]) \tag{3.18}$$

where use[i] is the set of variables whose values may be used in statement 'i' prior to any definition of variable. The def[i] is the set of variables that are defined at statement 'i', prior to any use of that variable in statement 'i' and succ[i] represents successor of statement 'i'. Liveness analysis function in algorithm 2, is used for finding the values of

```
1.  i = 0;
2.  j = 16;
3.  k = 0;
4.  l = 5;
5.  label loop
6.  k = k + 1;
7.  j = j - 1;
8.  if j <= 16 then body else body 1
9.  label body
10. k = 1;
11. label body1
12. j = 6;
13. i = i + 1;
14. if i < 8 goto label loop else end
15. label end
```

Figure 3.7: Intermediate code representation of S_4.

Table 3.6: Liveness analysis of S_4 using fixed point iteration.

Statement	**Initialization**		**Iteration 1**		**Iteration 2**		**Iteration 3**	
i	**out[i]**	**in[i]**	**out[i]**	**in[i]**	**out[i]**	**in[i]**	**out[i]**	**in[i]**
15.	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
14.	ϕ	ϕ	ϕ	ϕ	k, j, i	k, j, i	k, j, i	k, j, i
13.	ϕ	ϕ	i	i	k, j, i	k, j, i	k, j, i	k, j, i
12.	ϕ	ϕ	i	i	k, i	k, j, i	k, j, i	k, i
11.	ϕ	ϕ	i	i	k, i	k, i	k, i	k, i
10.	ϕ	ϕ	i	i	k, j, i	j, i	k, j, i	j, i
9.	ϕ	ϕ	i	i	j, i	j, i	j, i	j, i
8.	ϕ	ϕ	j, i	j, i	k, j, i	k, j, i	k, j, i	k, j, i
7.	ϕ	ϕ	j, i	j, i	k, j, i	k, j, i	k, j,i	k, j, i
6.	ϕ	ϕ	k, j, i	k, j, i	k, j, i	k, j, i	k, j, i	k, j, i
5.	ϕ	ϕ	k, j, i	k, j, i	k, j, i	k, j, i	k, j, i	k, j, i
4.	ϕ	ϕ	k, j, i	k, j, i	k, j, i	k, j, i	k, j, i	k, j, i
3.	ϕ	ϕ	k, j, i	k, j, i	k, j, i	j, i	k, j, i	j, i
2.	ϕ	ϕ	j, i	j, i	j, i	i	j, i	i
1.	ϕ	ϕ	i	i	i	ϕ	i	ϕ

in[i] and out[i] sets for intermediate code representation of S_4 (Table 3.1(d)), as shown in Figure 3.7, representing an irrelevant statement at line number 4. In algorithm 2, input to liveness function is succ[i], use[i] and def[i]. Initially the value of in[i], out[i] is 'ϕ' and flag is set to true. The while loop in liveness analysis function is executed for in[i] and out[i] sets until the value of flag is true.

We have deduced use[i], def[i] and succ[i] using data flow analysis and calculated liveness analysis in Table 3.6 from bottom to top using fixed point iteration for statements 'i = 15' to 'i = 1' for intermediate source code of S_4 shown in Figure 3.7. It is evident from Table 3.6, that the values of in[i] and out[i] sets in second iteration are equal to the values of in[i] and out[i] sets in third iteration, so the fixed point iteration is attained for source code S_4.

From the results of liveness analysis in Table 3.6, we have deduced that the declaration (d4: l = 5) is not used in any of the future program computations and therefore it is marked as an irrelevant statement in source code S_4. It should be eliminated from

S_4. Finally, a refactored program with same semantic meaning is achieved by removing the irrelevant statement (d4: l = 5) from the source code S_4.

3.4 Summary

In this chapter, a semantic code clone detection approach based on reaching definition and liveness analysis is proposed. The proposed approach addresses the key issues such as statement reordering, inversion of control predicates and insertion of irrelevant statements. Most of the existing methods of semantic code clone detection are based on the comparison of program dependence graphs. Further, semantic similarity between program dependence graphs is achieved through subgraph isomorphism, which is NP-complete problem. We have computed code clone pairs by carrying out control and data flow analysis on abstract syntax trees. In this work, we have also designed an algorithm to carry out semantic analysis of program code fragments using reaching definition analysis and liveness analysis. We have also detected refactored program fragments that are syntactically different but semantically equivalent to original program fragments.

Chapter 4

Code Clone Groups

Code clones are not constrained to the single version of program file. Code clones between two versions of a program file form a code clone pair. The collection of code clone pairs leads to code clone groups. The development of code clone groups with time generates history of code clone group, called as code clone genealogy. Code clone genealogy is to find code clone groups on different versions of program file in a software system. Code clone genealogies are analyzed on the basis of clone lineages. A clone lineage is a directed acyclic graph that describes evolution history of code clones in a program file [26].

Existing code clone genealogy detection approaches are based on centralized version control system. These approaches are not scalable enough to find code clone groups on distributed version control system (DVCS). Hence, a new approach is required to detect code clone groups on distributed version control system. To address these issues, this chapter describes a new approach to extract code clone groups on distributed version control system. Efficient code clone group extraction algorithm based on transitive closure computation is proposed in this chapter.[1] The results show that there exists potential clone pairs between different versions of program file on DVCS. In recent years, there has been lot of attention devoted to research in the area of code clone

[1]The contents of this chapter are published as per following details:

- Rajkumar Tekchandani, Rajesh Bhatia, Maninder Singh, "Code clone genealogy detection on e-health system using Hadoop", Computers and Electrical Engineering, Vol. 61, pp 15-30, Elsevier, IF 1.570, July 2017.

genealogy detection [26][48][45]. The focus in this research work is moved towards finding code clones among different versions of the program file stored in distributed version control system like Git. Large software systems on Git (a distributed version control system) undergo thousands of commits over their life cycles. Each commit can involve modifications to code clones. Code clone genealogy is evolved over time along with each commit made by user in a software system on Git. Commit history of Git is stored in the form of directed acyclic graph (DAG). Each commit in Git adds a new node in existing DAG. To study evolution of code clone groups, knowledge needs to be extracted from DAG. With thousands of commits on a software system in Git, knowledge extraction from DAG to study evolution of code clone groups requires high computation with addition of new parameters. To deal with intensive computations normal solutions are unsuitable and cause an unnecessary overhead in runtime.

4.1 Code Clone Group Extraction

This section introduces code clone group extraction, along with tools and infrastructure used in the proposed Git code clone group extractor model.

4.1.1 Infrastructure and Tools used

To implement our proposed Git code clone group extractor model on subject system the following infrastructure and tools are used.

4.1.1.1 Git Repository

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency [171]. When a

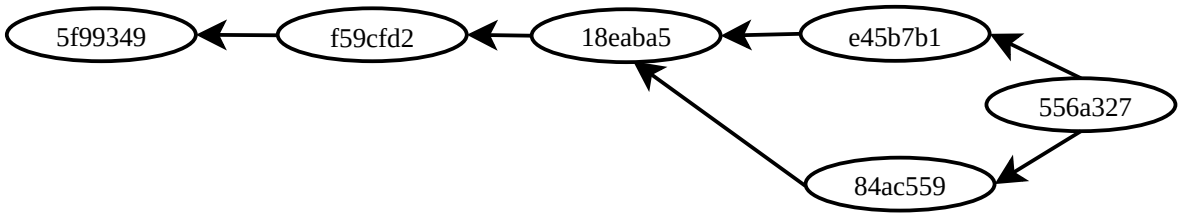


Figure 4.1: DAG data structure in Git.

commit is made, Git stores a commit object that contains a pointer to the snapshot of the content staged. This object also contains author's name and email, message and pointers to the commit or commits that directly came before this commit (its parent or parents). Zero parents for the initial commit, one parent for a normal commit and multiple parents for a commit that results from the merger of two or more branches. Git uses directed acyclic graph (DAG) data structure with each commit as a node. The term acyclic indicates that there are no cycles in the graph. The history of everything in the Git repository is modeled as a DAG. A DAG is a data structure that can be used to model a wide variety of problems.

The DAG consists of the following elements:

- Nodes: Each node represents some object or piece of data. In Git, each node represents a version of program file.
- Directed edges: A directed edge from one node to another represents child to parent relationship, from the new revision to the revision from which it was derived.
- Root node : At least one of the nodes will have no parents. This is the root of the DAG.
- Leaf nodes: One or more of the nodes will have no children. These are called leaves or leaf nodes.

For example, in Figure 4.1, 5f99349, f59cfd2, 18eaba5, e45b7b1, 84ac559 and 556a327 are respective commit nodes of the Git repository represented in DAG. Commit node 5f99349 is the root and 556a327 is leaf node in the commit history.

4.1.1.2 Neo4j

Neo4j is an open source NoSQL graph database implemented in Java and Scala. Neo4j implements the Labeled Property Graph Model efficiently down to the storage level [172][173].
A labeled property graph has the following characteristics:

- It consists of nodes and relationships.
- Nodes contains properties.
- Nodes can be labeled with one or more labels.
- Relationships are named and directed, and always have a start and end node.
- Relationships can also contain properties.

These databases are navigated by following the relationships. Neo4j has an ability to hold billions of nodes and relationships using Cypher. Cypher is Neo4j graph query language. Cypher syntax provides a familiar way to match patterns of nodes and relationships in the graph. Cypher query language (CQL) is a declarative, SQL-inspired language for describing patterns in graphs [174]. A graph is represented as G = (V,E), where set 'V' represents collection of vertices and set 'E' represents collection of edges [175].

4.1.1.3 Perl

Perl (Practical Extraction and Report Language) is a stable, cross platform programming language [176]. It supports both procedural and object-oriented programming. We created Perl scripts to automate various stages in our clone group extraction model.

4.1.1.4 Hive

To carry out operations like querying and analyzing huge amount of data, Hadoop framework offers an open source data warehouse system called as Hive [177]. Hadoop is an open-source framework that allows to store and process Big Data in a distributed environment across clusters of computers using simple programming models [178]. Hive structures warehouses in HDFS and other input sources, such as Amazon S3. Hive is a sub-platform in the Hadoop ecosystem and produces its own query language (HiveQL). This language is compiled by Map-Reduce and enables user-defined functions (UDFs). The Hive platform is primarily based on three related data structures as tables, partitions and buckets [179].

4.2 Clone Group Extraction Model

In this section, we describe our proposed Git code clone group extraction model. The components used in our model are shown in Figure 4.2. The input to our code clone group extraction model is the DAG of commit nodes extracted from Git commit history of a program file distributed across all commits. Each commit node in DAG has ten attributes as sha1, hash, parents, abbreviated parents, author email, author name, refs, subject, timestamp and date time. A commit node in DAG represents a version of the program file in Git commit history. The DAG of commit nodes is extracted from

logs using Git command as git log −−reverse −−format = 'format: %H, %h, %P, %p, %an, %ae, %d, %s, %at, %ai' and stored in a comma separated value (CSV) file as CommitHistory.csv.

The format of CommitHistory.csv file includes the attributes of commit node as '%H, %h, %P, %p, %an, %ae, %d, %s, %at, %ai' as described in Table 4.1. The CommitHistory.csv file is loaded into Neo4j using LOAD CSV utility (Figure 4.2) for graphical (DAG) representation of relationships among commit nodes. Load CSV is used to import data from (CSV) file into Neo4j. A commit node as Commit(c) is created and connected with its author(u), day(d), month(m) and year(y) nodes representing its version. Parent nodes are stored in a list and connected to commit node using CQL commands. In Figure 4.3, the DAG of commit nodes are represented in Neo4j graph

Table 4.1: Description of attributes in CommitHistory.csv file.

S.No.	Attribute	Name	Description
1.	%H	Commit hash / Sha1	Unique 40 character string
2.	%h	Abbreviated commit hash / hash	First seven characters of Commit Hash
3.	%P	Parent hashes	Commit Hash of parent nodes
4.	%p	Abbreviated parent hashes	First seven characters of Commit Hash of parent nodes
5.	%an	Author name	Name of author who authored that Commit Hash
6.	%ae	Author email	Email of author who authored that Commit Hash
7.	%d	Ref names	Names of Branches
8.	%s	Subject	Message on Commit hash
9.	%at	Author date, UNIX timestamp	Unique UNIX Time stamp
10.	%ai	Author date, ISO 8601-like format	Date and time of Commit hash

database for e-health system program file. The nodes Commit(c), Author(u), User(u), Parent(p), Day(d), Month(m) and Year (y) are connected through following relation-

ships as AUTHORED, ON_DAY, IN_MONTH, IN_YEAR and PARENT in Figure 4.3 using the following CQL commands in Neo4j.

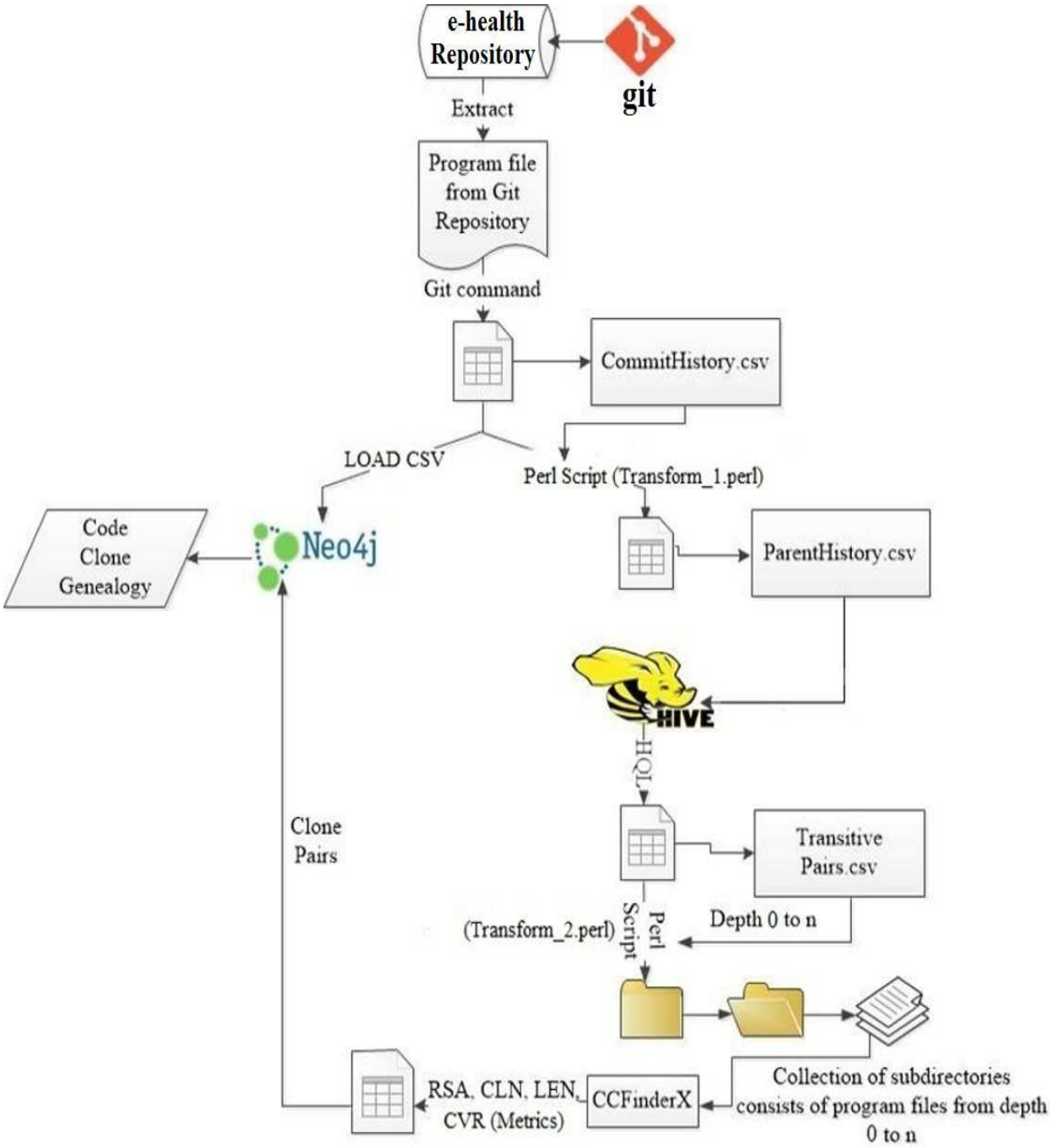


Figure 4.2: Code clone group extraction model on e-health system.

- CREATE (u) - [:AUTHORED] - > (c)

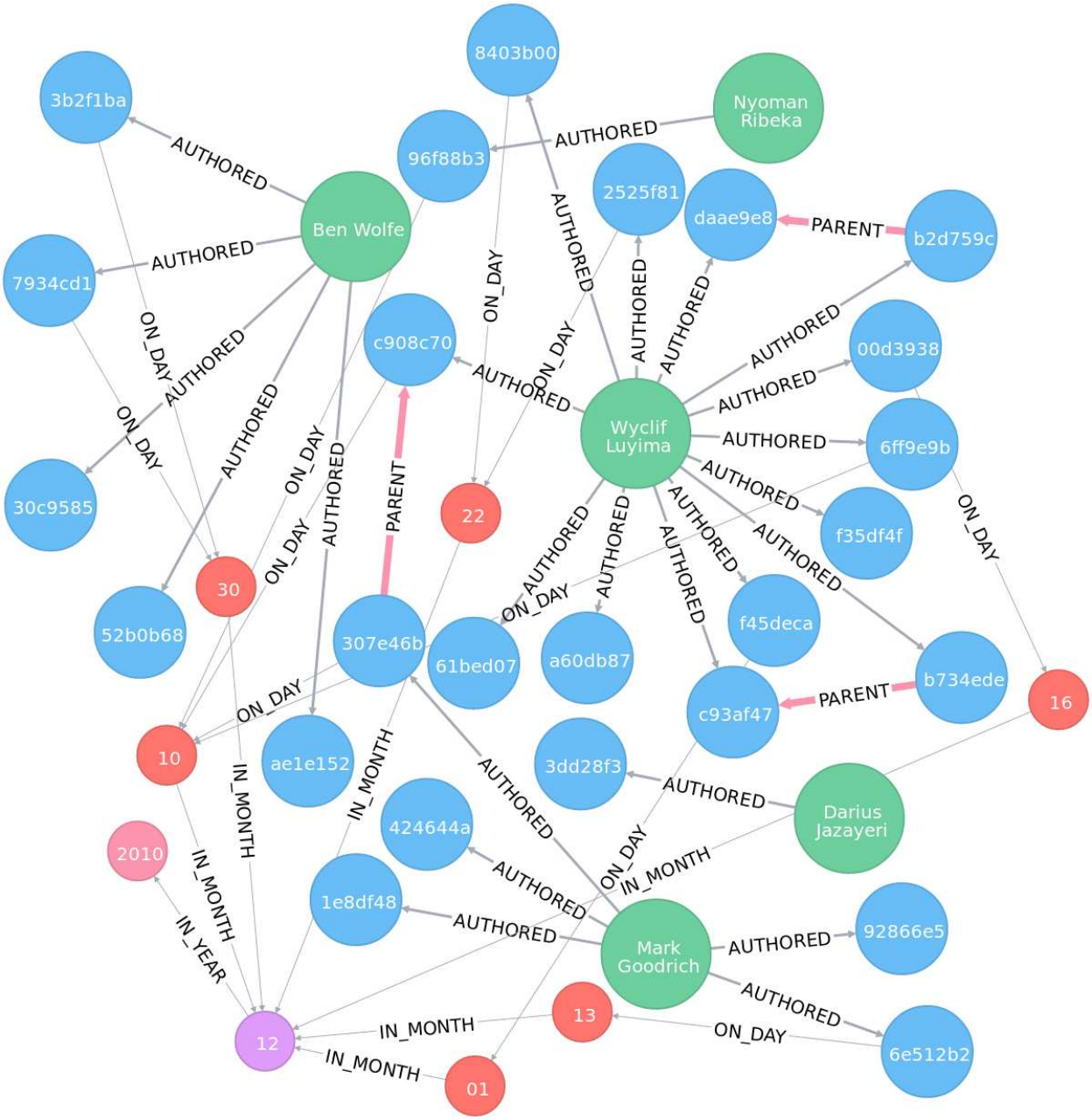


Figure 4.3: Graphical representation of relationships among various commit hashes of HibernateConceptDAO.java in Neo4j.

- CREATE (c) - [:ON_DAY] - > (d)
- CREATE (d) - [:IN_MONTH] - > (m)
- CREATE (m) - [:IN_YEAR] - > (y)

- CREATE (c) - [:PARENT] - > (p)

These relationships connects nodes as Author(u) authored commit(c), Commit(c) is done on day(d), Day(d) falls in month(m), Month(m) falls in year(y) and Parent(p) is the parent of commit(c). To extract code clone groups from commit hashes, clone pairs needs to be extracted on the basis of code clone parameters as transitive depth, ratio of similarity and clone coverage using relationships in Neo4j. A clone group is the collection of clone pairs between a sink commit hash and the other parent commit hashes at different transitive depth values (Figure 4.5). To extract clone pairs, CommitHistory.csv file acts as an input to the Transform_1.perl (Perl Script) which splits the parents of each commit node and output two attributes as commit hash and its parent hash. This pair of commit hash and its parent hash provides us the relationship at depth value equals to one. The output of Transform_1.perl is stored in a comma separated value

Table 4.2: File set metrics extracted from CCFinderX [1][2].

S.No.	File set metrics	Description
1.	LEN	Length of file in tokens
2.	CLN	Count of clones included in the file
3.	RSA	Ratio (percentage) of similarity with another file
4.	CVR	Ratio (percentage) of tokens that are covered by any code clone

file as ParentHistory.csv (Figure 4.2). The ParentHistory.csv file is loaded into Hive Data warehouse for transitive closure computation. Transitive closure computes depth relationship between commit nodes for each depth value from '0' to 'n' using algorithm 3. A relation 'R' on a set 'A' is called as transitive if whenever (a,b) $\in$ R and (b,c) $\in$ R, then (a,c) $\in$ R, for all a,b,c $\in$ A [123].

Similarly, we have defined transitive pairs in context to commit hashes, as if there exists a relation from '*commit_hash_1* $\rightarrow$ *commit_hash_2*' with depth value 'x' and a

relation from '*commit_hash_2* → *commit_hash_3*' with depth value 'y', it implies that there exists a transitive relation from '*commit_hash_1* (from) → *commit_hash_3* (to)' with depth value equals to 'x + y'. Hive returns all transitive pairs between commit hashes

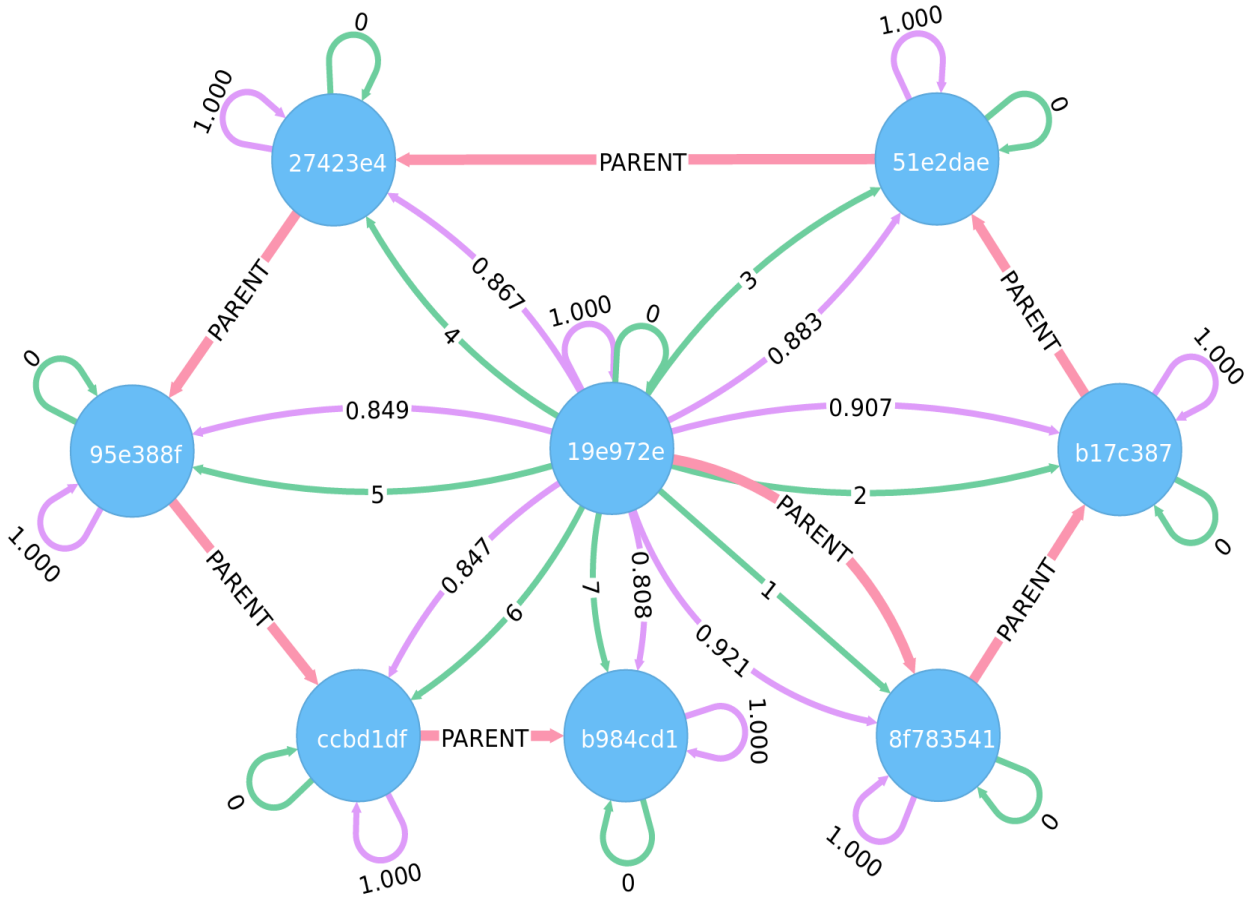


Figure 4.4: Depth {0, 1, 2, ..., 7}, RSA {1.000, 0.921, 0.907, ..., 0.808} and PARENT relationships among different versions {19e972e, 8f783541, ..., b984cd1} of HibernateConceptDAO.java in Neo4j.

for a program file in a comma separated value file as TransitivePairs.csv (Figure 4.2) that returns different versions of the program file at different transitive depth values. These different versions of the program file are stored in a directory that is created using Transform_2.perl (Perl Script) in which TransitivePairs.csv file acts as an input (Figure 4.2). The code clone pairs are detected between different versions of the program file at different transitive depth values using CCFinderX [1][2] (a token based code clone

detection tool) and the following file set metrics, as described in Table 4.2 are retrieved to extract clone evolution patterns on Git. Each clone pair with extracted file set metrics are exported into Neo4j for code clone genealogy relationship detection. The code clone genealogy relationship for depth and RSA is build using Query-1 in Neo4j graph database (Figure 4.4).

Query-1:

1. LOAD CSV WITH HEADERS FROM 'file:///TransitivePairs.csv' AS line,
2. MATCH (a : Commit { hash : line.from }) , (b : Commit { hash : line.to }),
3. CREATE (a) − [depth : depth { depth : line.depth }] - > (b),
4. CREATE (a) − [rsa : RSA { RSA : line.rsa }] − > (b),
5. RETURN a.hash as from_Commit_hash, b.hash as to_Commit_hash, rsa, depth;

The LOAD CSV command is used to import bulk data from external system to graph. The first line of cypher script in Query-1 loads TransitivePairs.csv file into Neo4j graph database. WITH HEADERS tells the database that first line of (.csv) file contains named HEADERS as from(commit_id), to(commit_id) and transitive depth. AS line assigns the input file to the variable line. The rest of script will be executed for each row of TransitivePairs.csv file. In line 2 of query-1, (a) and (b) are identifiers, Commit is node label and hash is the property of node.

The MATCH command fetches data from database and binds it with identifier (a) and (b), where identifier (a) consists of form_commit_id and (b) consists of to_commit_id. Relationships as depth and RSA are created in line 3 and 4 between (a) and (b) using CREATE clause. In line 5, nodes and relationships bound to identifier (a) and (b) are returned using RETURN clause. Moreover, code clone groups are represented in genealogy using Query-2 in Neo4j (Figure 4.5).

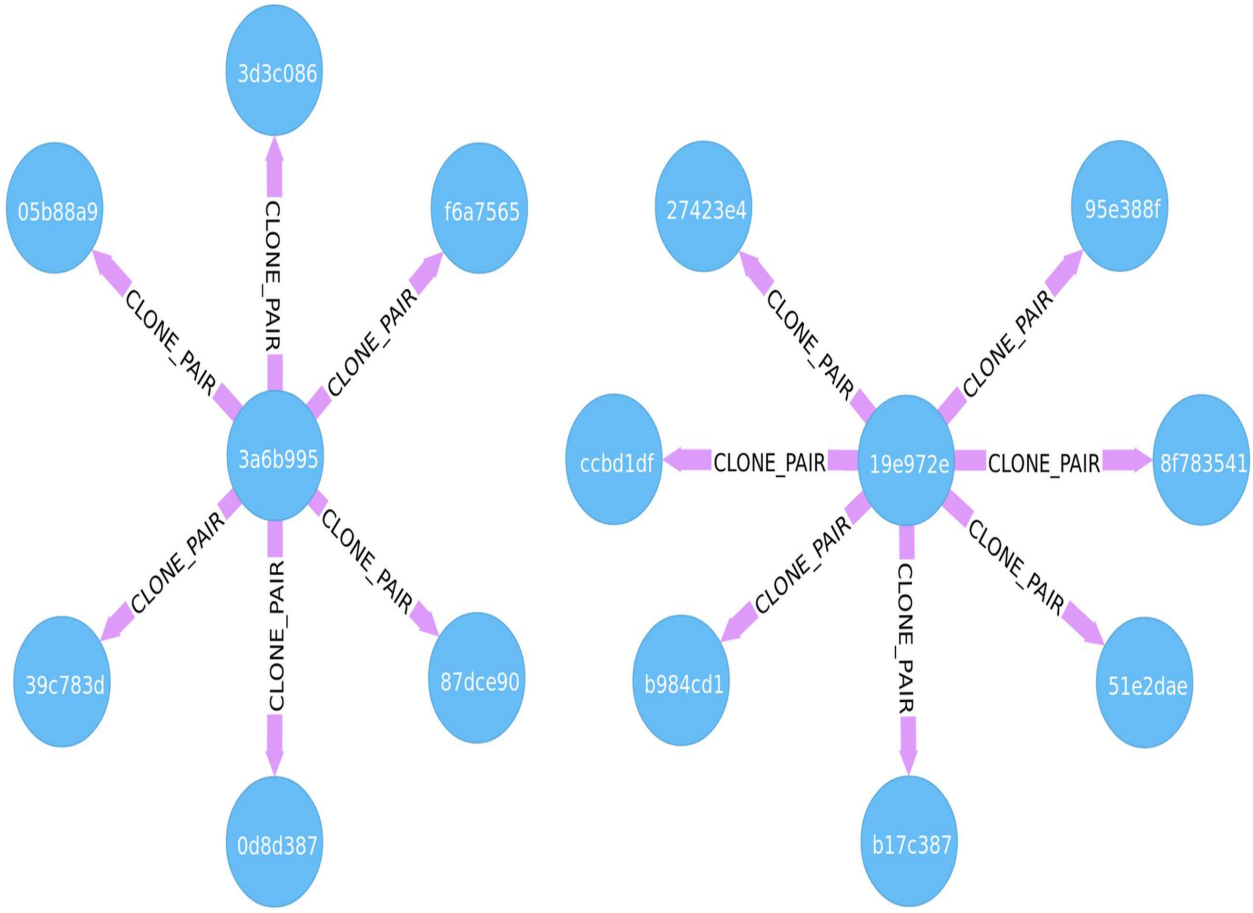


Figure 4.5: Graphical Representation of Clone groups in genealogy for sink commit nodes {3a6b995 and 19e972e} of e-health system in Neo4j.

Query-2:

1. LOAD CSV WITH HEADERS FROM 'file:///TransitivePairs.csv' AS line,
2. MATCH (from : Commit {hash : line.from}),
3. MERGE (to : Commit {hash : line.to, depth : line.depth}),
4. CREATE (from) − [: CLONE_PAIR] − > (to),
5. RETURN from as From_Commit_Hash, to as To_Commit_Hash_Depth;

In query-2 data is imported into graph database from TransitivePairs.csv file using LOAD CSV utility. In second line of CQL Query, MATCH command fetches the data

from database and binds it with identifier/node (from). In line 3 of Query-2, MERGE (MATCH + CREATE) command searches for the given pattern in the graph, if it exists, the results will be returned , else it will create a new node/relationship and returns the results. Here we have matched the pattern as to_commit_id, depth and merged it into node/identifier (to). In line 4, a relationship as CLONE_PAIR is built between 'from' and 'to' nodes. In line 5, values of identifier (from) and (to) are returned using RETURN clause. As the CQL script in query-2, will be executed for each row of TransitivePairs.csv file, it returns clone pair between (from) and (to) commit nodes for each iteration of CQL script and collection of all such clone pairs will form a clone group (see Figure 4.5).

4.3 Algorithm Description

In algorithm 3, transitive clone pairs between different versions of program file are found to extract code clone groups in genealogy. The procedures used in algorithm 3, to calculate transitive closure between commit hashes are *createHiveTables, tcDepth0, tcDepth1, tcDepth(n), truncateTransition, loadTransition, mergeTransitionToTc and count.*

- *createHiveTables*: This procedure creates base, reporting and transition tables.
  - Base Table : A Hive Managed Table that stores the source data obtained from ParentHistory.csv file.
    - Schema of Base Table : tableName : base :: fieldSchemas (base_ fromCommitID : STRING, base_toCommitID : STRING, depth : INT)
  - Reporting Table : A Hive Managed Table partitioned by depth that holds the output of transitive closure.

Algorithm 3: Transitive closure based code clone computation

```
Algorithm Transitive Clone Pairs()
1   createHiveTables()
2   tcDepth0()
3   loadTransition() ← Load output of tcDepth0()
4   tcDepth1()
5   loadTransition() ← Load output of tcDepth1()
6   mergeTransitionToTc()
7   truncateTransition()
8   d = 2                              ▷ d is depth parameter
9   do
10      rowCountPrevious = count()
11      tcDepth(d)
12      loadTransition() ← Load output of tcDepth(d)
13      mergeTransitionToTc()
14      truncateTransition()
15      rowCountCurrent = count()
16      increment d
    while rowCountPrevious != rowCountcurrent
17  return
```

    - ∗ Schema of Reporting Table : tableName : tc :: fieldSchemas (tc_from CommitID : STRING, tc_toCommitID : STRING, depth : INT)

  - Transition Table : A Hive Managed Table that stores the results of transitive closure at each depth before merging into Reporting table. At the start of each processing cycle it is overwritten by transitive closure of processing depth.

    - ∗ Schema of Transition Table : tableName : transition :: fieldSchema (transition_fromCommitID : STRING, transition_to CommitID : STRING, depth : INT)

- *tcDepth0* : Compute transitive clone pairs at depth = '0'.
- *tcDepth1* : Compute transitive clone pairs at depth = '1'.
  - Transitive clone pairs in base table are at depth '1'.

- *tcDepth(n)* : Requires a single input parameter 'n', ('n' is between '2' to maximum depth value) and computes the transitive clone pairs at depth = 'n'.
- *truncateTransition* : Truncates the transition table.
- *loadTransition* : Loads the transition table.
- *mergeTransitionToTc* : The transitive clone pairs generated at each depth are merged from transition table to reporting table.
- *count* : Computes the number of records in reporting table.

4.3.1 Transitive Closure Computation

Algorithm 3 finds transitive clone pairs at each depth through transitive closure computation in Hive. The *createHiveTables* (line1) procedure creates base, transition and reporting tables. Input to algorithm 3, is ParentHistory.csv file which contains transitive clone pairs at depth '1' and stored in base table. The *tcDepth0* (line 2) procedure calculates the transitive clone pairs at depth '0'. Depth '0' transitive clone pairs are self commits. In Figure 4.4, it is represented by self loop at each commit hash. Depth '0' transitive clone pairs are computed by selecting distinct base_fromCommitID, base_toCommitID and assigning depth value as '0'.

The *tcDepth1* (line 4) procedure finds transitive clone pairs at depth '1' which are retrieved from base table. Transitive clone pairs obtained at depth '0' and depth '1' are stored into transition table using *loadTransition* (line 3 and 5) and finally merged into reporting table using *mergeTransitionToTc* (line 6). To calculate transitive clone pairs from depth two onwards, do-while loop is executed from 'd = 2' to the maximum depth (line 10-16). For each iteration in loop, transitive clone pairs are calculated using *tcDepth(d)* (line 11), stored in transition table (line 12) and merged to reporting table

(line 13).

The *tcDepth(d)* procedure is a self join between reporting table on tc_fromCommitID, tc_toCommitID and filtered by 'tc1.depth + tc2.depth = d', where 'tc1' and 'tc2' are alias of reporting table. Depth parameter 'd' is incremented until maximum depth is reached which is tracked by count of number of rows in reporting table. When count is same for successive iterations it indicates that maximum depth is reached and transitive clone pairs for depth = '0' to maximum depth are obtained.

4.4 Summary

In this chapter, a novel Git code clone group extraction model based on transitive closure computation using Big Data technologies is proposed. Efficient code clone genealogy detection gives developers useful feedback about various change patterns of code clones in a software system. To extract code clone groups from distributed repository in Git, transitive pairs between different versions of program files have been computed using Hadoop ecosystem. The computed transitive pairs are evaluated and investigated structurally and semantically using Neo4j graph database for code clone evolution. The code clone genealogy relationships are calculated on the basis of code clone parameters as transitive depth, ratio of similarity and clone coverage ratio.

Chapter 5

Results and Discussions

The experimental results in this chapter are carried into two parts. In the first part, semantic code clone analysis is carried out using reaching definition and liveness analysis on Java benchmark systems. In second part of code clone analysis, extracted code clones pairs are clustered into code clone groups during software evolution in distributed version control system. Code clone groups are extracted using Hadoop ecosystem on OpenMRS (an open source Git repository). Earlier methods of semantic code clone detection are based on comparison of program dependence graphs through graph isomorphism, which returns approximate solutions. Programmers often try to infer semantic differencing between different versions of program files by inserting irrelevant statements, statement reordering and inversion of control predicates. Existing semantic code clone detection techniques are unable to provide a heuristic solution to these issues.

In result analysis (section 5.1.1), the performance of the proposed semantic code clone detection approach is evaluated on Java based applications of varying sizes taken from Decapo Benchmark [180]. Code clone pairs are extracted between different versions of program files in a subject system using the proposed code clone group extraction model. In the case study (section 5.2.2), three research questions are framed to describe code clone evolutions patterns. An adequate amount of result analysis is provided in section 5.2.3 to address the research questions framed in case study.

5.1 Semantic Code Clone Analysis

To carry out experimental study on semantic code clone detection[1], we analyzed the results obtained from Table 3.2, 3.3, 3.4 and validated the results on large scale Java based subject systems. We have deduced results in the terms of definitions 'd1' to 'd8' for program code fragments S_1, S_2 and S_3. Similarly, from Table 3.5 results were obtained from definitions 'd1' to 'd9' for source code S_4.

5.1.1 Results Analysis

We find that after reordering the definitions as shown in Table 3.1(b), the same set of definitions have been obtained for each sub-block as shown below.

Results From Table 3.2 (Original source code, S_1)

out[B1.S_1] = d1, d2, d3

out[B2.S_1] = d1, d4, d5, d8

out[B3.S_1] = d1, d5, d6, d8

out[B4.S_1] = d1, d4, d7, d8

out[B5.S_1] = d4, d5, d6, d7, d8

Results From Table 3.3 (Reordered statements, S_2)

out[B1.S_2] = d1, d2, d3

out[B2.S_2] = d3, d4, d5, d8

out[B3.S_2] = d3, d4, d6, d8

out[B4.S_2] = d3, d5, d7, d8

out[B5.S_2] = d4, d5, d6, d7, d8

[1]The contents of this section are published as per following details:

- Rajkumar Tekchandani, Rajesh Bhatia, Maninder Singh, "Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis", The Journal of Supercomputing, pp. 1-28, Springer, IF 1.326, August 2016.

Results From Table 3.4 (Inversion of branch statements, S_3)

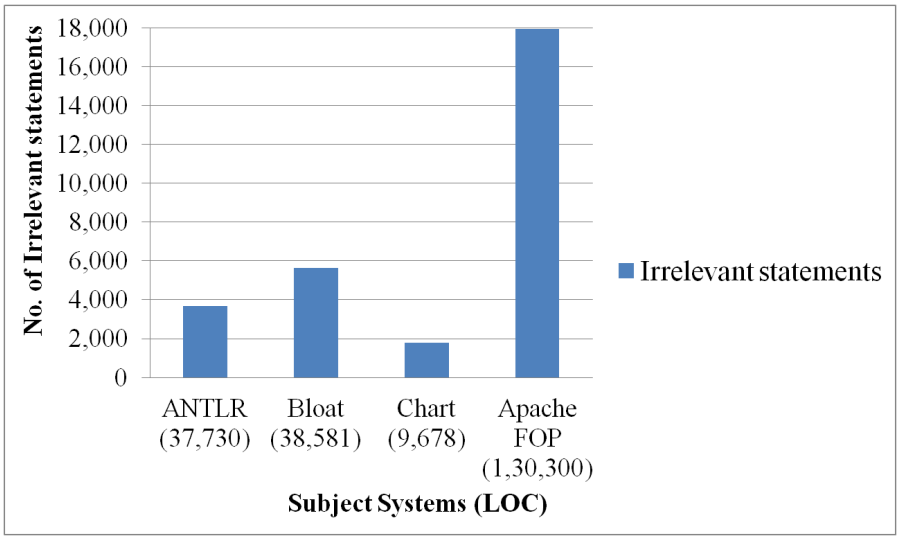


Figure 5.1: Number of irrelevant statements in subject systems.

out[B1.S_3] = d1, d2, d3

out[B2.S_3] = d3, d4, d5, d8

out[B3.S_3] = d3, d4, d6, d8

out[B4.S_3] = d3, d5, d7, d8

out[B5.S_3] = d4, d5, d6, d7, d8

Similarly, the set of definitions has been obtained for the program code represented in S_4 (Table 3.1(d)), representing the insertion of an irrelevant statement as (d4: l = 5). It is not redefined at any future program point as the value of kill set for (d4: l = 5) $\in$ S_4 is 'ϕ' in Figure 3.6.

Results From Table 3.5 (Insertion of irrelevant statement, S_4)

out[B1.S_4] = d1, d2, d3, d4

out[B2.S_4] = d1, d4, d5, d6, d9

out[B3.S_4] = d1, d4, d6, d7, d9

out[B4.S_4] = d1, d4, d5, d8, d9

out[B5.S_4] = d4, d5, d6, d7, d8, d9.

From Table 3.5, we have analyzed that the definition (d4: l = 5) remains defined in all the blocks of 'S_4' and it may used at some program point 'p' in the program, that can be checked by carrying out liveness analysis. By assimilating the results, we found

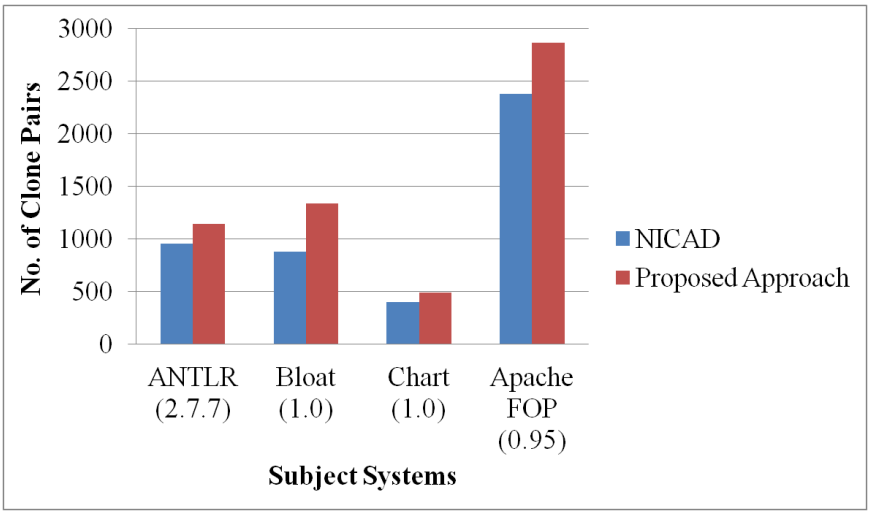


Figure 5.2: Comparison of clone pairs with NICAD.

that reaching definition equations that are live at the end of each block is equivalent to the reaching definition equations for the program blocks having reordered statements. We have achieved the solution of statement reordering as mentioned by Baxter [11] using explicit data flow analysis through reaching definition analysis. Similarly, reaching set definitions are calculated for the source code with inversion of branch or control statements. Another area of interest for carrying semantic analysis is the insertion of irrelevant statements. Programmers often try to insert irrelevant declarations or statements in the source code to hide code clones. These irrelevant statements cannot be detected using token or tree based techniques from source code, so to extract these irrelevant statements, data flow analysis is performed using liveness analysis.

Using reaching definition analysis of source code 'S_4' in Figure 3.6, we deduced that the definition (d4: l = 5) in 'S_4' is not killed anywhere in the program. The value of the attribute **kill[d4]** $= \phi$ (Figure 3.6) signifies that definition **'d4'** is not redefined in the program. The definition (d4: l = 5) may be used at any of the program point in

Table 5.1: Java benchmark subject systems.

Subject System	**Version**	**LOC**	**No. of dead assignments**	**NICAD (Clone Pairs)**	**Clone Pairs detected using proposed approach**
ANTLR	2.7.7	37,730	3,676	953	1,140
Bloat	1.0	38,581	5,635	875	1,337
Chart	1.0	9,968	1,788	402	488
Apache FOP	0.95	1,30,300	l7,930	2,380	2,866

the program. Liveness analysis is performed to find that whether definition (d4: l = 5) is used at any of the program point in the source code. From second and third iteration of Table 3.6, we found that the definition (d4: l = 5) is not live at any of the program point of S_4 in **out[i]** and **in[i]** sets of liveness analysis. It indicates that the definition (d4: l = 5) is not used as well as not redefined (kill[d4] = ϕ) at any of the program point in S_4 (Figure 3.6). This leads to the conclusion that statement induced as (d4: l = 5) in program code S_4 is an irrelevant statement and it should be removed from the program.

Further, the change in value of any variable that is live at any program point inside the program will not affect the structure of program because its structural representation will remain same. Moreover, its runtime behavior will also remain same regardless of number of iterations if the variable is used in some looping construct. To evaluate the performance of the proposed approach, Java based applications of varying sizes as ANTLR, Bloat, Chart and Apache FOP from DaCapo Benchmark suite [180] are selected. ANTLR is parser generator, Bloat is an optimization and analysis tool, Chart is a charting utility tool and Apache FOP is print formatting tool as shown in Table 5.1.

Figure 5.1, represent number of irrelevant statements that exists in the subject systems taken from DeCapo Benchmark. Further in Table 5.1, we represented the number of clone pairs (CP's) detected by the proposed approach on subject systems as ANTLR

(1,140 CP's), Bloat (1,337 CP's), Chart (488 CP's) and Apache FOP (2,866 CP's) and compared our results with NICAD (a near miss clone detection tool) [21][122]. Figure 5.2 shows comparative analysis of results with NICAD.

5.2 Code Clone Group Analysis

In second part of code clone analysis[2], code clone pairs are extracted from OpenMRS (an open source Git repository) using Hadoop ecosystem. The extracted code clone pairs are clustered to form code clone groups. Code clone groups are the collection of code clone pairs and result of copy paste activities in a software system. Code clone pairs are extracted between different versions of program file on distributed version control system (Git) using transitive closure computation on directed acyclic graphs (DAG). Further, code clone genealogy is detected among different versions of the program file. Detection of code clone genealogy gives immense benefits to developers with respect to maintenance and refactoring perspectives.

In experimental study, code clone pairs at different transitive depth values are found. In result analysis (section 5.2.3), changing evolution patterns, code clone pairs and code clone evolution patterns are discussed. The results data set show, that as the transitive depth increases between different versions of program file the similarity decreases due to more numbers of insertions and deletions of program statements. These inserted program statements may be irrelevant statements in version control system, which can be eliminated from program files using formal methods of program analysis.

[2]The contents of this section are published as per following details:

- Rajkumar Tekchandani, Rajesh Bhatia, Maninder Singh, "Code clone genealogy detection on e-health system using Hadoop", Computers and Electrical Engineering, Vol. 61, pp 15-30, Elsevier, IF 1.570, July 2017.

5.2.1 Experimental Setup

To carry out experimental study on code clone group extraction, we used CDH 5 [181] (Cloudera Distribution Including Apache Hadoop) for Hadoop ecosystem. For our experiment, we installed CDH in local hardware, which includes a 3.6 GHz quad core processor, 16 GB memory, Ubuntu Precise (12.04) operating system, JDK Version 1.7.0_80 and Neo4j 2.3.2 graph database.

5.2.2 Case Study

Our code clone group extraction model on Git extracts code clone evolution patterns and allows us to explore various research questions as

RQ1: How often change occurs between different versions of a program file in Git?

RQ2: How clone pairs are extracted between different versions of a program file?

RQ3: How code clone genealogy parameters evolve at different transitive depth values of clone pairs in a clone group?

To answer these research questions, we choose a subject system on e-health from Git repository for code clone genealogy detection with significant commit history. We selected the subject system as OpenMRS [182] from open source Git repository as candidate repository for code clone genealogy detection having the commit history of approximately ten years from September 2005 to August 2015. OpenMRS is a patient centric, e-healthcare system that records the details of interactions between health care providers and patients. Information is stored in a way that makes it easy to summarize and analyze, minimizing the use of free text and maximizing the use of coded information. The software collects a patient's treatment details into a single patient chart.

Table 5.2: OpenMRS program files with commit history and execution time.

S.No.	Program files from e-health (OpenMRS) system	Maximum transitive depth	Execution time for genealogy extraction	Commit History	
				From (Date)	To (Date)
1.	HibernateConceptDAO.java	7	2 min 43 sec	17-08-2010	23-06-2014
2.	OrderServiceImpl.java	6	2 min 26 sec	17-08-2010	05-06-2014
3.	OrderServiceTest.java	6	2 min 24 sec	17-08-2010	04-06-2014
4.	ConceptServiceImpl.java	4	1 min 42 sec	17-08-2010	14-07-2014
5.	ModuleFactory.java	4	1 min 37 sec	17-08-2010	29-10-2013

Having this complete patient history available at one place allows clinicians to make better decisions about care. It also enables a deeper analysis of patient health in order to draw more meaningful conclusions [183]. Code clone genealogy is extracted for program files (Table 5.2) of OpenMRS e-health care system on Git. The commit history graph of HibernateConceptDAO.java is extracted using CQL and stored in Neo4j (Figure 4.3).

Figure 4.4 gives the graphical representation of extracted code clone genealogy of HibernateConceptDAO.java using Neo4j graph database from depth '0' to '7'. For our code clone genealogy analysis on Git, we focused on finding the transitive closure between different versions of the program file distributed over several commits in commit history of program file. Every commit in commit history represents different version of the program file having an increase or decrease in 'LEN' (Number of tokens) parameter (see Table 5.3). The 'LEN' parameter is calculated using CCFinderX [1][2]. As per the study of Kim *et al.* [26] there exists some false positives and false negatives in CCFind-

erX [1][2] clone detection tool but the previous comparison of clone detector tools [3] suggests that CCFinderX has much higher recall value as compared to CloneDr [11]. Further, while configuring CCFinderX [1][2], we have taken the minimum token length as '30' (a default setting) and TKS (minimum number of token types) equals to '12', as very short code snippets are not considered as code clones by programmers.

5.2.3 Results Analysis

This section presents the evolution patterns of code clones in OpenMRS e-healthcare system and answers the research questions raised in section 5.2.2.

5.2.3.1 Changing Evolution Patterns

To find how often the change occurs between different versions of the program file across the whole commit history in Git, we measured the change in token length between different versions of the program file at different transitive depth values. In Table 5.3, we extracted transitive pairs between '*from_commit_hash* → *to_commit_hash*' in increasing order of depth values for program files shown in Table 5.2. We further measured the parameters as '*LEN_from_commit_hash*' and '*LEN_to_commit_hash*' from CCFinderX [1][2] that gives the token length of different versions of program file at '*from_commit_hash*' and '*to_commit_hash*'.

In OpenMRS, we found that there is an increase or decrease in the number of tokens at various commits between different versions of the program file. In Table 5.3, '*LEN_to_Commit_hash*' parameter changes between different versions of program file at different transitive depth values. This change in 'LEN' parameter suggests that programmers have done a substantial amount of change among different versions of the program file on different commits at various transitive depth values.

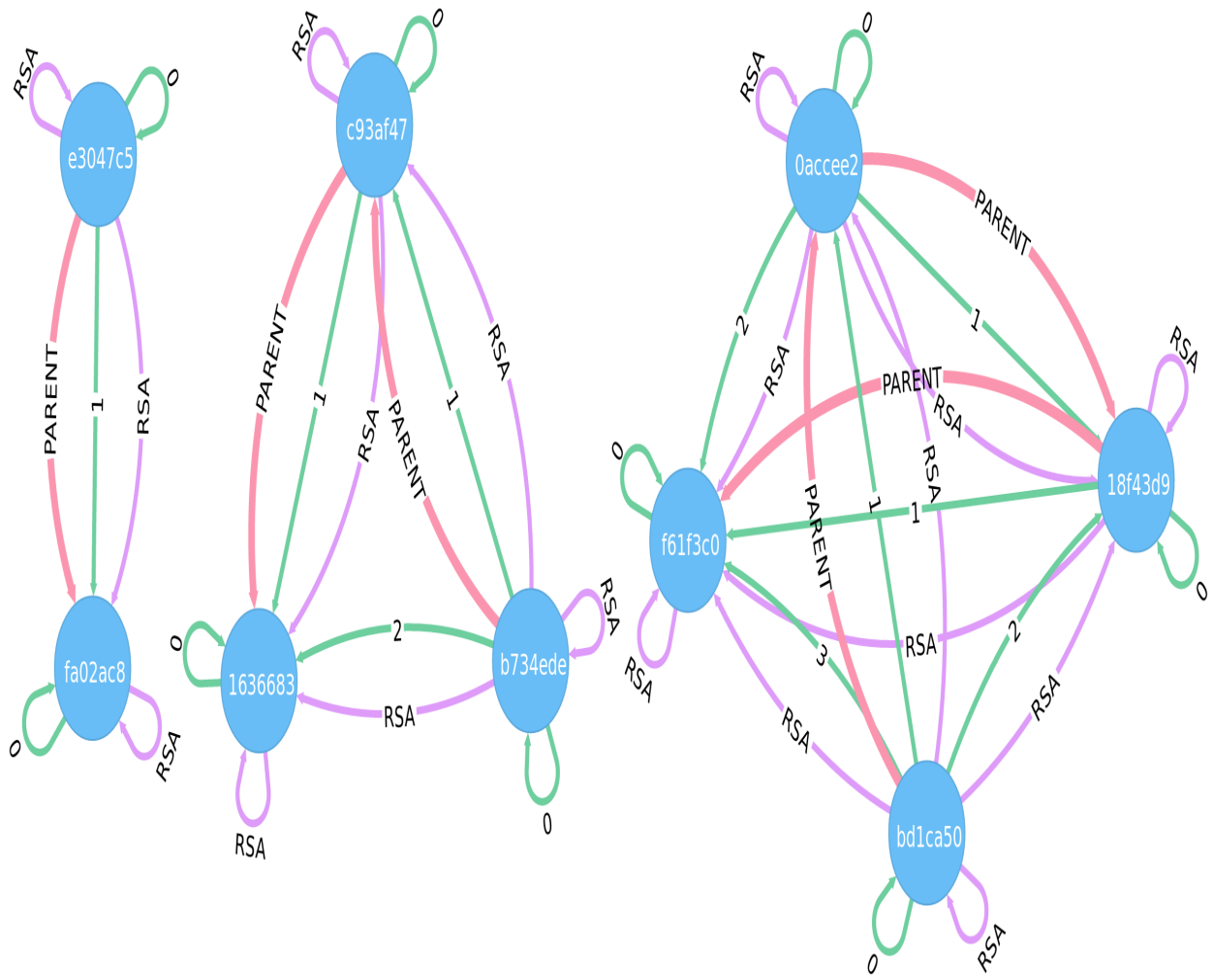


Figure 5.3: Clone pair relationships among different versions {fa02ac8, e3047c5, c93af47, ..., bd1ca50} of HibernateConceptDAO.java in Neo4j.

5.2.3.2 Code Clone Pairs

To find clone pairs, we input transitive pairs in CCFinderX [1][2] and measured the ratio of similarity parameter between different versions of the program file at a particular transitive depth value. The Ratio of similarity between two versions of the program file is calculated using the following RSA function [2] as RSA(f) = 1/LOC(f) $\sum_{c \in CF(f)}$ LOC(c) where, LOC(c) : number of lines of code 'c'. Here, CF(f) is a set of code fragments included in program file (f) and have a clone relation with some code

fragments in other program file, and 'c' is the element of CF(f).

Table 5.3: Clone pairs in OpenMRS.

S.No	from_ Commit_ hash	to _Commit _hash	Transitive depth	LEN_from_ Com- mit_hash	LEN_to_ Com- mit_hash	RSA
(a) HibernateConceptDAO.java						
1.	19e972e	19e972e	depth : 0	9917	9917	1.000
2.	19e972e	8f783541	depth : 1	9917	9165	0.921
3.	19e972e	b17c387	depth : 2	9917	9147	0.907
4.	19e972e	51e2dae	depth : 3	9917	8994	0.883
5.	19e972e	27423e4	depth : 4	9917	8933	0.867
6.	19e972e	95e388f	depth : 5	9917	8662	0.849
7.	19e972e	ccbd1df	depth : 6	9917	8660	0.847
8.	19e972e	b984cd1	depth : 7	9917	8727	0.808
(b) OrderServiceImpl.java						
1.	3a6b995	3a6b995	depth : 0	1332	1332	1.000
2.	3a6b995	3d3c086	depth : 1	1332	1332	0.933
3.	3a6b995	87dce90	depth : 2	1332	1310	0.933
4.	3a6b995	05b88a9	depth : 3	1332	1110	0.797
5.	3a6b995	f6a7565	depth : 4	1332	1081	0.664
6.	3a6b995	0d8d387	depth : 5	1332	1077	0.631
7.	3a6b995	39c783d	depth : 6	1332	1059	0.621
(c) OrderServiceTest.java						
1.	3a6b995	3a6b995	depth : 0	3919	3919	1.000
2.	3a6b995	3d3c086	depth : 1	3919	3942	0.888

contd..

Table 5.3: Clone pairs in OpenMRS. (continued..)

S.No	from_ Commit_ hash	to_ Commit _hash	Transitive depth	LEN_from_ Com- mit_hash	LEN_to_ Com- mit_hash	RSA
3.	3a6b995	87dce90	depth : 2	3919	3942	0.888
4.	3a6b995	05b88a9	depth : 3	3919	3370	0.755
5.	3a6b995	f6a7565	depth : 4	3919	3191	0.703
6.	3a6b995	0d8d387	depth : 5	3919	3191	0.703
7.	3a6b995	39c783d	depth : 6	3919	3159	0.685
(d) ConceptServiceImpl.java						
1.	604f245	604f245	depth : 0	5442	5442	1.000
2.	604f245	e1604d9	depth : 1	5442	5442	1.000
3.	604f245	0293e5b	depth : 2	5442	5442	1.000
4.	604f245	a1adcf6	depth : 3	5442	5217	0.777
5.	604f245	ab3b9b1	depth : 4	5442	6652	0.595
(e) ModuleFactory.java						
1.	90fe416	90fe416	depth : 0	5837	5837	1.000
2.	90fe416	9af5f48	depth : 1	5837	5843	0.973
3.	90fe416	bf79c32	depth : 2	5837	5843	0.973
4.	90fe416	364372d	depth : 3	5837	5686	0.899
5.	90fe416	1941264	depth : 4	5837	5679	0.899

In Table 5.3(a), RSA value varies from 1.000 to 0.808 for transitive pairs from depth '0' to '7' for HibernateConceptDAO.java. Similarly, Table 5.3(b)-(e) represents 'RSA' at different transitive depth values for OrderServiceImpl.java, OrderServiceTest.java, ConceptServiceImpl.java and ModuleFactory.java. This shows that transitive pairs at

each depth are potential clone pairs and ratio of similarity changes between different versions of program file at various commits.

Table 5.4: Clone groups in genealogy of HibernateConceptDAO.java at minimum and maximum depth values.

S.No	Path (FromCommitHash_ToCommitHash_Depth)	RSA
(a) Depth : 1 to 7		
1.	$19e972e\_8f783541\_1$	0.921
	$19e972e\_b984cd1\_7$	0.808
(b) Depth : 1 to 6		
2.	$8f783541\_b17c387\_1$	0.983
	$8f783541\_b984cd1\_6$	0.876
(c) Depth : 1 to 5		
3.	$c6d616c\_91d0f7f\_1$	1.000
	$c6d616c\_ae0fa70\_5$	0.975
4.	$b17c387\_51e2dae\_1$	0.959
	$b17c387\_b984cd1\_5$	0.878
5.	$0293e5b\_a1adcf6\_1$	0.965
	$0293e5b\_accf5c0\_5$	0.930
(d) Depth : 1 to 4		
6.	$a1adcf6\_a3be8ee\_1$	0.976
	$a1adcf6\_accf5c0\_4$	0.963
7.	$91d0f7f\_3e44129\_1$	1.000
	$91d0f7f\_ae0fa70\_4$	0.975
8.	$51e2dae\_27423e4\_1$	1.000
	$51e2dae\_b984cd1\_4$	0.910

contd..

Table 5.4: Clone groups in genealogy of HibernateConceptDAO.java at minimum and maximum depth values. (continued..)

S.No	Path (FromCommitHash_ToCommitHash_Depth)	RSA
9.	0225*f*7*e*_*f*1018*a*1_1	0.949
	0225*f*7*e*_8818*dec*_4	0.806
(e) Depth : 1 to 3		
10.	*f*1018*a*1_*e*6*ead*75_1	0.920
	*f*1018*a*1_8818*dec*_3	0.772
11.	*bd*1*ca*50_0*accee*2_1	1.000
	*bd*1*ca*50_*f*61*f*3*c*0_3	0.988
12.	*a*3*be*8*ee*_8834817_1	1.000
	*a*3*be*8*ee*_*accf*5*c*0_3	0.987
13.	3*e*44129_*de*32940_1	0.994
	3*e*44129_*ae*0*fa*70_3	0.975
14.	04*f*8*aa*7_*ca*627*b*5_1	1.000
	04*f*8*aa*7_1617851_3	0.985
(f) Depth : 1 to 2		
15.	0*accee*2_18*f*43*d*9_1	0.995
	0*accee*2_*f*61*f*3*c*0_2	0.988
16.	1*f*19*afc*_*e*873*e*4*f*_1	1.000
	1*f*19*afc*_986*f*236_2	0.997
17.	32*ad*9*b*4_06309*fa*_1	1.000
	32*ad*9*b*4_89*f*096*c*_2	0.999
18.	307*e*46*b*_*c*908*c*70_1	0.974
	307*e*46*b*_719*dc*3*f*_2	0.933

contd..

Table 5.4: Clone groups in genealogy of HibernateConceptDAO.java at minimum and maximum depth values. (continued..)

S.No	Path (FromCommitHash_ToCommitHash_Depth)	RSA
19.	$b734ede\_c93af47\_1$	0.908
	$b734ede\_1636683\_2$	0.908
20.	$ca627b5\_2bbd26a\_1$	0.995
	$ca627b5\_1617851\_2$	0.985
21.	$d7a152d\_9de8c13\_1$	1.000
	$d7a152d\_c0958e4\_2$	0.982
22.	$fc5c212\_cd27f7a\_1$	0.978
	$fc5c212\_95b5d02\_2$	0.964

5.2.3.3 Code Clone Evolution Patterns

For evolution of code clone genealogy parameters at different transitive depth values of clone pairs in a clone group, we found 520 clone pairs in HibernateConceptDAO.java lifetime from transitive depth '0' to '7'. These clone pairs are stored in Neo4j graph

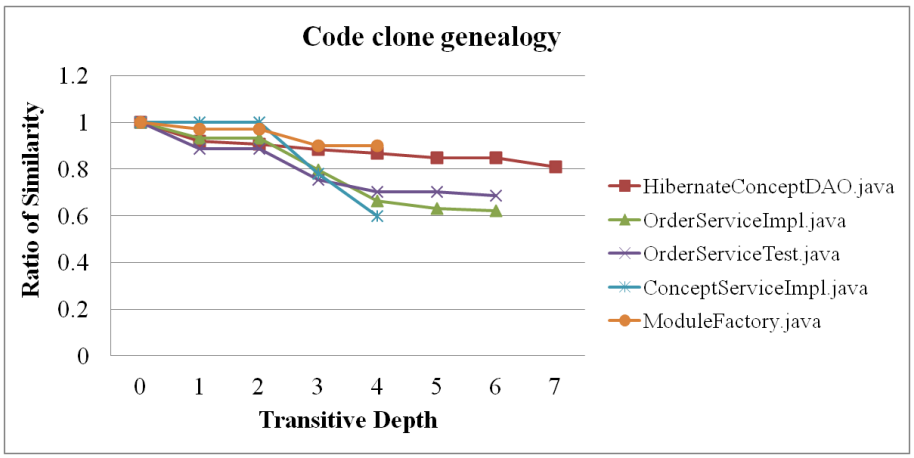
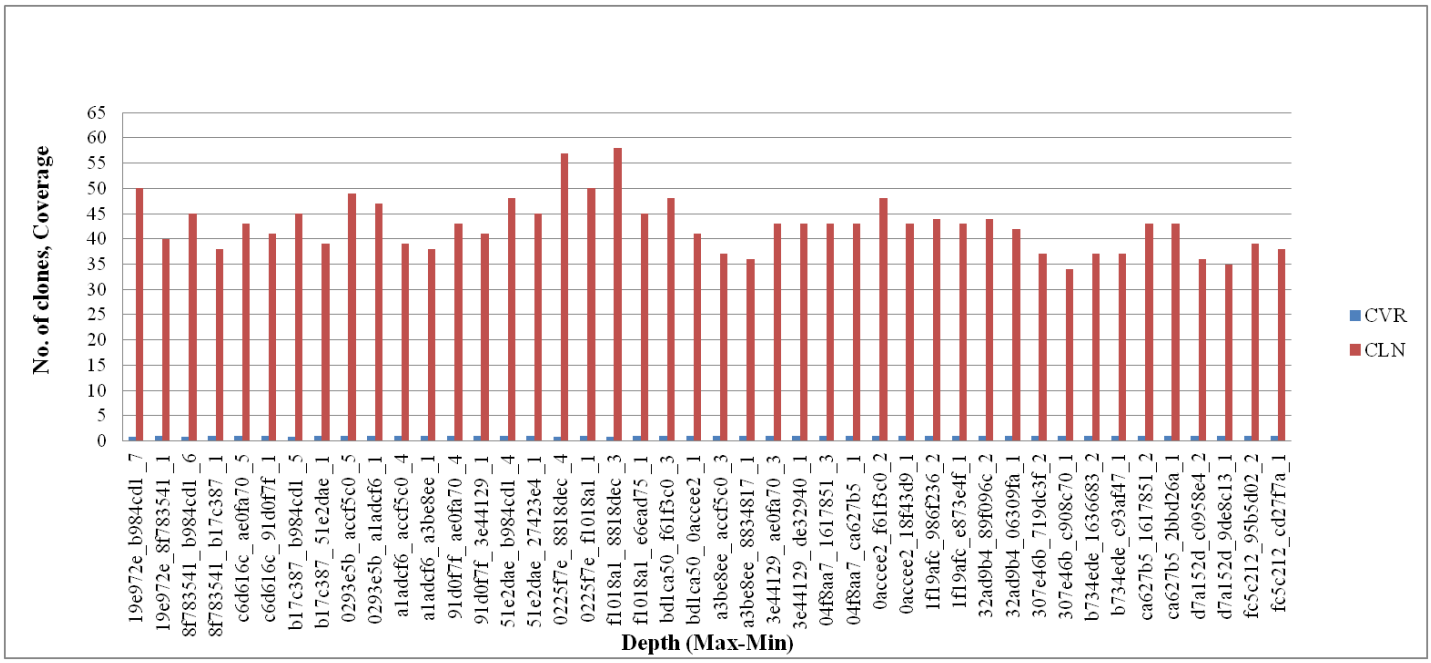
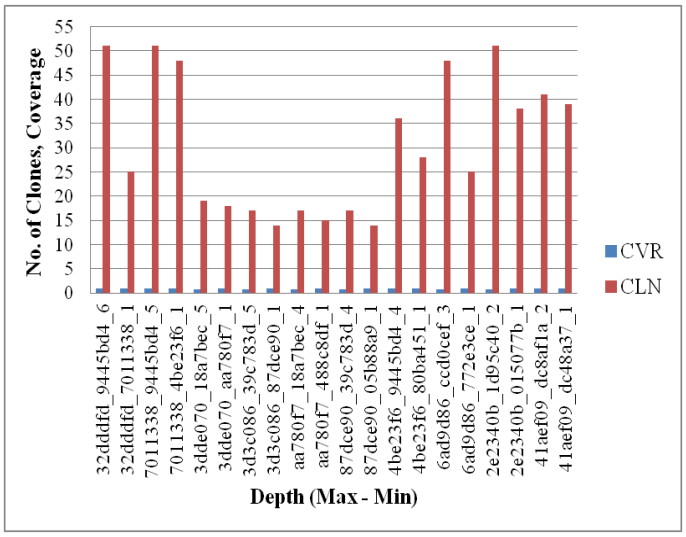


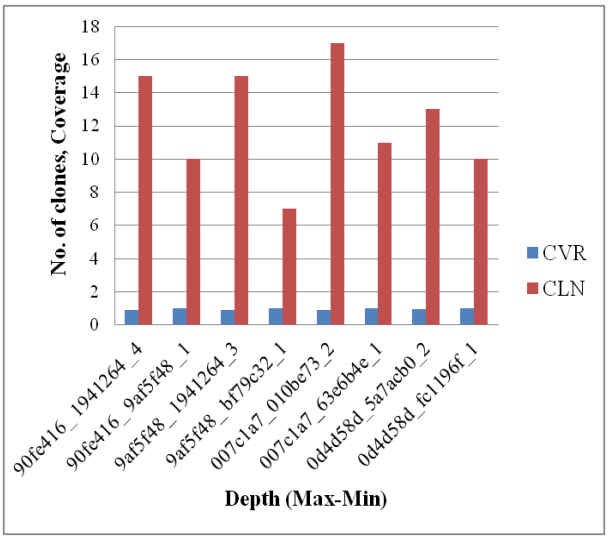
Figure 5.4: Clone evolution patterns of clone groups in genealogy extraction of program files.



(a) Clones in genealogy of HibernateConceptDAO.java.



(b) Clones in genealogy of OrderServiceTest.java.



(c) Clones in genealogy of ModuleFactory.java.

Figure 5.5: CLN and CVR at Maximum and Minimum transitive depth of clone groups in genealogy of program files in openMRS.

database with depth and RSA properties. Figure 5.3 shows that these clone pairs are mapped in clone evolution pattern for genealogy extraction of HibernateConceptDAO.java. Clone groups for HibernateConceptDAO.java are shown in Table 5.4. Similarly, code clone genealogy is extracted and mapped for OrderServiceImpl.java, OrderServiceTest.java, ConceptServiceImpl.java and ModuleFactory.java as shown in Figure 5.4. We identified the following evolution patterns:

1. *Clone Group Evolution* : We extracted 22 clone groups from genealogy of HibernateConceptDAO.java and calculated the range of RSA for each clone group from minimum to maximum transitive depth (Table 5.4). Figure 5.4 represents the clone evolution pattern of clone groups in genealogy extraction of program files of OpenMRS. This shows that as transitive depth increases in genealogy, the ratio of similarity decreases between different versions of the program file.

2. *Evolution of Clones in Genealogy* : Figure 5.5 presents file set metrics as CLN and CVR, that gives the count of clones and clone coverage ratio on different versions of program file in e-health system. As the transitive depth decreases in a clone group the count of clones also decreases in genealogy due to more number of insertions and deletions in the program file. The versions of a program file towards higher transitive depth values have less clone coverage ratio (CVR) as compared to lower depth values in genealogy of code clones.

5.3 Summary

In this chapter, the proposed approach was evaluated on large scale Java benchmark systems in section 5.1. In empirical study, the performance of proposed approach is compared with a near miss clone detection tool, NICAD. In section 5.2, code clone groups

were extracted on distributed version control system in optimum time. An empirical study on code clone group evolution for OpenMRS, (an open source Git repository) is presented. Our study identifies relationships of transitive depth between different versions of the program file for extraction of code clone evolution patterns on OpenMRS. The result data set shows that transitive pairs at each depth are potential clone pairs and as transitive depth increases in clone group the ratio of similarity decreases between different versions of program file.

Further, during evolution of code clones in genealogy, the count of clones decreases as transitive depth decreases in genealogy due to more number of statement insertions or deletions. Moreover, the change in length parameter suggests that significant amount of change have been done by programmers among different versions of program file at various transitive depth values. To reduce the effort of maintenance of code and to identify the amount of semantic variation during code clone evolution across different versions of program files, it would be valuable to apply reaching definition and liveness analysis based proposed approach on different versions of program files.

Chapter 6

Conclusion and Future Scope

This chapter concludes the research work on semantic code clone detection and code clone group extraction model as presented in the thesis. It also discusses open research problems along with future research scope.

6.1 Conclusion

Code clone detection techniques have been widely used in various software applications with a goal to provide software maintenance and refactoring benefits to programmers. The output of code clone detection techniques is reported in the form of code clone pairs. Cluster of clone pairs leads to code clone groups. The evolution of code clone groups across the history of software system is termed as code clone genealogy. Earlier methods of semantic code clone detection uses program dependence graphs, but comparison of program dependence graphs to find semantic similarity between code fragments is done with the aid of graph isomorphism, which is NP-complete problem. The existing code clone group extraction techniques are not good and scalable enough to find code clone groups across different versions of software system stored in distributed version control system (DVCS). The following objectives are discussed and analyzed in the thesis.

First, we discussed the classification of code clones. After reviewing a wide range of available literature which helped in preparing a detailed analysis from the various

research proposals, few gaps in the existing literature were identified which required further investigation. After presenting a review of code clone detection techniques, we presented a semantic code clone detection technique and code clone group extraction model, which fulfills most of the requirements for an effective code clone detection algorithm. We have proposed a new approach based on reaching definition and liveness analysis for semantic code clone detection.

We have computed semantic code clones between program fragments by carrying out control and data flow analysis on abstract syntax trees that considers statement reordering, inversion of decision control predicates and insertion of irrelevant statements. We have detected refactored program fragments after elimination of dead code fragments that are syntactically different but semantically equivalent. We have analyzed our approach on large scale Java benchmark systems having 2,16,579 lines of code and found that dead code statements stay in the source code. We found 5,831 code clone pairs from Java benchmark subject systems using proposed approach.

In the second part of thesis, we have proposed a novel Git code clone group extraction model based on transitive closure computation using Hadoop ecosystem. We have conducted an in-depth empirical study on change evolution history of code clone pairs in e-health system (OpenMRS) on Git. We have extracted code clone groups through transitive closure computation on DAG in Hadoop Distributed File System (HDFS) that provides the capability to do analysis on the code clone group dataset.

Transitive closure computation on DAG computes all the available transitive clone pairs at each depth in Git commit history between different versions of the software program located on thousands of commits. Further, to map clone pair parameters in graph data structure and to provide the capability of user friendly extraction, transformation and loading (ETL) operation on code clone genealogy, we used graph database. The computed transitive pairs are evaluated and investigated structurally and semantically

using Neo4j graph database for code clone evolution.

In our experimental study, we have identified the relationships of transitive depth between different versions of a program file for extraction of code clone evolution patterns on OpenMRS. These code clone genealogy relationships were calculated on the basis of code clone parameters as transitive depth, ratio of similarity and clone coverage ratio. We found the result dataset of commits on OpenMRS through transitive closure computation on DAG. These commits in result dataset are potential clone pairs whose clone parameters are mapped in genealogy to extract code clone evolution patterns. The result data set shows that as transitive depth increases in genealogy, the similarity between different versions of program file in genealogy decreases.

6.2 Future Research Scope

6.2.1 Semantic Code Clones

In future, we intend to extend our approach on Big Data Curated benchmark to detect semantically equivalent code clones at interprocedural level. Moreover, it would be valuable to apply formal methods of program analysis to reduce the amount of semantic differencing between different versions of program file distributed over several commit hashes in distributed version control system.

6.2.2 Clone Group Extraction

In future, we plan to extend code clone group extraction model to more distributed repositories and to include new parameters to study evolution of near miss code clones on distributed version control system. The future work will also focus on prediction

of survival time of clones in genealogy using machine learning models on distributed version control system.

List of Publications

SCI/SCIE Journals:

1. Rajkumar Tekchandani, Rajesh Bhatia, Maninder Singh, "Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis," *The Journal of Supercomputing,* pp. 1-28, 2016. *(Springer, IF 1.326) (SCI/SCIE).*
2. Rajkumar Tekchandani, Rajesh Bhatia, Maninder Singh, "Code clone genealogy detection on e-health system using Hadoop", *Computers and Electrical Engineering*," vol. 61, pp. 15-30, 2017. *(Elsevier, IF 1.570) (SCI/SCIE)*

References

[1] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[2] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Maintenance support environment based on code clone analysis," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on.* IEEE, 2002, pp. 67–76.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[4] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[5] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[6] J. R. Cordy, "Comprehending reality-practical barriers to industrial adoption of software maintenance automation," in *Program Comprehension, 2003. 11th IEEE International Workshop on.* IEEE, 2003, pp. 196–205.

[7] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on software engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[8] J. H. Johnson, "Substring matching for clone detection and change tracking." in *ICSM*, vol. 94, 1994, pp. 120–126.

[9] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[10] J. H. Johnson, "Navigating the textual redundancy web in legacy source," in *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1996, p. 16.

[11] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 368–377.

[12] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*. IEEE, 2002, pp. 87–94.

[13] A. Walenstein and A. Lakhotia, "The software similarity problem in malware analysis," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[14] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley, "The development of a software clone detector," *International Journal of Applied Software Technology*, 1995.

[15] E. Burd and M. Munro, "Investigating the maintenance implications of the replication of code," in *Software Maintenance, 1997. Proceedings., International Conference on*. IEEE, 1997, pp. 322–329.

[16] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on.* IEEE, 1995, pp. 86–95.

[17] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, "Analyzing cloning evolution in the linux kernel," *Information and Software Technology*, vol. 44, no. 13, pp. 755–765, 2002.

[18] G. Antoniol, M. D. Penta, G. Casazza, and E. Merlo, "Modeling clones evolution through time series," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01).* IEEE Computer Society, 2001, p. 273.

[19] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Transactions on Programming languages and Systems (TOPLAS)*, vol. 22, no. 2, pp. 378–415, 2000.

[20] H. A. Basit and S. Jarzabek, "Efficient token based clone detection with flexible tokenization," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* ACM, 2007, pp. 513–516.

[21] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on.* IEEE, 2008, pp. 172–181.

[22] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society, 2007, pp. 96–105.

[23] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International Static Analysis Symposium*. Springer, 2001, pp. 40–56.

[24] H. A. Basit, U. Ali, and S. Jarzabek, "Viewing simple clones from structural clones' perspective," in *Proceedings of the 5th International Workshop on Software Clones*. ACM, 2011, pp. 1–6.

[25] M. Kim and D. Notkin, "Using a clone genealogy extractor for understanding and supporting evolution of code clones," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[26] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 187–196.

[27] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*. IEEE, 2006, pp. 253–262.

[28] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 2000, pp. 155–169.

[29] Y. Higo and S. Kusumoto, "Code clone detection on specialized pdgs with heuristics," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 75–84.

[30] Y. Higo, S. Kusumoto, and 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*, vol. 20, no. 6, pp. 435–461, 2008.

[31] S. K. Abd-El-Hafiz, "A metrics-based data mining approach for software clone detection," in *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*. IEEE, 2012, pp. 35–41.

[32] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," in *Reverse engineering*. Springer, 1996, pp. 77–108.

[33] H. A. Basit and S. Jarzabek, "Detecting higher-level similarity patterns in programs," in *ACM Sigsoft Software engineering notes*, vol. 30, no. 5. ACM, 2005, pp. 156–165.

[34] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics." in *icsm*, vol. 96, 1996, p. 244.

[35] W. S. Evans, C. W. Fraser, and F. Ma, "Clone detection via structural abstraction," *Software Quality Journal*, vol. 17, no. 4, pp. 309–330, 2009.

[36] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*. IEEE, 2004, pp. 128–135.

[37] F. Lanubile and T. Mallardo, "Finding function clones in web applications," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. IEEE, 2003, pp. 379–386.

[38] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Software Metrics Symposium, 1999. Proceedings. Sixth International*. IEEE, 1999, pp. 292–303.

[39] J. Krinke, "Identifying similar code with program dependence graphs," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on.* IEEE, 2001, pp. 301–309.

[40] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2006, pp. 872–881.

[41] K. Maeda, "An extended line-based approach to detect code clones using syntactic and lexical information," in *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on.* IEEE, 2010, pp. 1237–1240.

[42] L. Zhang, D. Liu, Y. Li, and M. Zhong, "Ast-based plagiarism detection method," in *Internet of Things.* Springer, 2012, pp. 611–618.

[43] J. H. Johnson, "Visualizing textual redundancy in legacy source," in *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research.* IBM Press, 1994, p. 32.

[44] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the eighteenth international symposium on Software testing and analysis.* ACM, 2009, pp. 81–92.

[45] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating code clone genealogies at release level: An empirical study," in *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on.* IEEE, 2010, pp. 87–96.

[46] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry, "Understanding the evolution of type-3 clones: an exploratory study," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on.* IEEE, 2013, pp. 139–148.

[47] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010.

[48] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone evolution: a systematic review," *Journal of software: Evolution and Process*, vol. 25, no. 3, pp. 261–283, 2013.

[49] N. Göde and R. Koschke, "Studying clone evolution using incremental clone detection," *Journal of Software: Evolution and Process*, vol. 25, no. 2, pp. 165–192, 2013.

[50] P. Parashar, A. Kalia, and R. Bhatia, "How time-fault ratio helps in test case prioritization for regression testing," *International Journal Of software Engg.(IJSE)*, vol. 5, no. 2, pp. 25–35, 2012.

[51] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1.* IBM Press, 1993, pp. 171–183.

[52] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[53] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on.* IEEE, 1999, pp. 109–118.

[54] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, pp. 49–49, 1993.

[55] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *J. UCS*, vol. 8, no. 11, p. 1016, 2002.

[56] H. T. Jankowitz, "Detecting plagiarism in student pascale programs," *The Computer Journal*, vol. 31, no. 1, pp. 1–8, 1988.

[57] X. Yan, J. Han, and R. Afshar, "Clospan: Mining: Closed sequential patterns in large datasets," in *Proceedings of the 2003 SIAM International Conference on Data Mining*. SIAM, 2003, pp. 166–177.

[58] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[59] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective-a workbench for clone detection research," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 603–606.

[60] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "Shinobi: A tool for automatic code clone detection in the ide," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 313–314.

[61] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach."

[62] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

[63] D. Gitchell and N. Tran, "Sim: a utility for detecting similarity in computer programs," in *ACM SIGCSE Bulletin*, vol. 31, no. 1. ACM, 1999, pp. 266–270.

[64] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 2004, pp. 253–262.

[65] P. Bulychev and M. Minea, "Duplicate code detection using anti-unification," in *Spring Young Researchers Colloquium on Software Engineering*, 2008, pp. 51–54.

[66] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Clemanx: Incremental clone detection tool for evolving software," in *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, 2009, pp. 437–438.

[67] T. Kamiya, "An execution-semantic and content-and-context-based code-clone detection and analysis," in *Software Clones (IWSC), 2015 IEEE 9th International Workshop on*. IEEE, 2015, pp. 1–7.

[68] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[69] R. Garg, K. Sharma, C. Nagpal, R. Garg, R. Garg, and R. Kumar, "Ranking of software engineering metrics by fuzzy-based matrix methodology," *Software Testing, Verification and Reliability*, vol. 23, no. 2, pp. 149–168, 2013.

[70] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Laguë, "Extending software quality assessment techniques to java systems," in *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*. IEEE, 1999, pp. 49–56.

[71] A. Perumal, S. Kanmani, and E. Kodhai, "Extracting the similarity in detected software clones using metrics," in *Computer and Communication Technology (ICCCT), 2010 International Conference on*. IEEE, 2010, pp. 575–579.

[72] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, and B. V. Saranya, "Detection of type-1 and type-2 code clones using textual analysis and metrics," in *Recent Trends in Information, Telecommunication and Computing (ITC), 2010 International Conference on.* IEEE, 2010, pp. 241–243.

[73] Z. O. Li and J. Sun, "A metric space based software clone detection approach," in *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on.* IEEE, 2010, pp. 393–397.

[74] T. Lavoie, M. Eilers-Smith, and E. Merlo, "Challenging cloning related problems with gpu-based algorithms," in *Proceedings of the 4th International Workshop on Software Clones.* ACM, 2010, pp. 25–32.

[75] G. Bansal and R. Tekchandani, "Selecting a set of appropriate metrics for detecting code clones," in *Contemporary Computing (IC3), 2014 Seventh International Conference on.* IEEE, 2014, pp. 484–488.

[76] B. Joshi, P. Budhathoki, W. L. Woon, and D. Svetinovic, "Software clone detection using clustering approach," in *Proceeings, Part II, of the 22nd International Conference on Neural Information Processing-Volume 9490.* Springer-Verlag New York, Inc., 2015, pp. 520–527.

[77] S. Horwitz, *Identifying the semantic and textual differences between two versions of a program.* ACM, 1990, vol. 25, no. 6.

[78] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering.* IEEE Press, 1981, pp. 439–449.

[79] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on.* IEEE, 2008, pp. 321–330.

[80] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on.* IEEE, 2001, pp. 107–114.

[81] S. Choi, H. Park, H.-i. Lim, and T. Han, "A static api birthmark for windows binary executables," *Journal of Systems and Software*, vol. 82, no. 5, pp. 862–873, 2009.

[82] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: memory comparison-based clone detector," in *Software Engineering (ICSE), 2011 33rd International Conference on.* IEEE, 2011, pp. 301–310.

[83] P. Schugerl, "Scalable clone detection using description logic," in *Proceedings of the 5th International Workshop on Software Clones.* ACM, 2011, pp. 47–53.

[84] R. Elva and G. T. Leavens, "Semantic clone detection using method ioe-behavior," in *Proceedings of the 6th International Workshop on Software Clones.* IEEE Press, 2012, pp. 80–81.

[85] T. Kamiya, "Agec: An execution-semantic clone detection tool," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on.* IEEE, 2013, pp. 227–229.

[86] R. Tekchandani, R. K. Bhatia, and M. Singh, "Semantic code clone detection using parse trees and grammar recovery," 2013.

[87] T. Wang, K. Wang, X. Su, and P. Ma, "Detection of semantically similar code," *Frontiers of Computer Science*, vol. 8, no. 6, pp. 996–1011, 2014.

[88] L. A. Neubauer, "Kamino: Dynamic approach to semantic code clone detection," *Department of Computer Science, Columiba University, Tech. Rep. CUCS-022-14*, 2015.

[89] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *Machine Learning and Applications (ICMLA), 2016 15th IEEE International Conference on.* IEEE, 2016, pp. 1024–1028.

[90] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining.* ACM, 2016, pp. 785–794.

[91] K. Maeda, "Code clone detection using parsing actions," in *Communications and Information Technology, 2009. ISCIT 2009. 9th International Symposium on.* IEEE, 2009, pp. 762–763.

[92] N. Göde and R. Koschke, "Incremental clone detection," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on.* IEEE, 2009, pp. 219–228.

[93] C. K. Roy, "Detection and analysis of near-miss software clones," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on.* IEEE, 2009, pp. 447–450.

[94] H. A. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Transactions on Software engineering*, vol. 35, no. 4, pp. 497–514, 2009.

[95] M. Chilowicz, É. Duris, and G. Roussel, "Finding similarities in source code through factorization," *Electronic Notes in Theoretical Computer Science*, vol. 238, no. 5, pp. 47–62, 2009.

[96] H. Li and S. Thompson, "Clone detection and removal for erlang/otp within a refactoring environment," in *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation.* ACM, 2009, pp. 169–178.

[97] G. M. Selim, K. C. Foo, and Y. Zou, "Enhancing source-based clone detection using intermediate representation," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 227–236.

[98] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–9.

[99] H. Liu, Z. Ma, L. Zhang, and W. Shao, "Detecting duplications in sequence diagrams based on suffix trees," in *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*. IEEE, 2006, pp. 269–276.

[100] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 603–612.

[101] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Complete and accurate clone detection in graph-based models," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 276–286.

[102] B. Hummel, E. Juergens, and D. Steidl, "Index-based model clone detection," in *Proceedings of the 5th International Workshop on Software Clones*. ACM, 2011, pp. 21–27.

[103] H. Störrle, "Towards clone detection in uml domain models," *Software & Systems Modeling*, vol. 12, no. 2, pp. 307–329, 2013.

[104] K. Johari and A. Kaur, "Effect of software evolution on software metrics: an open source case study," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp. 1–8, 2011.

[105] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the benefits of incorporating function clone detection in a development process," in *Software Maintenance, 1997. Proceedings., International Conference on.* IEEE, 1997, pp. 314–321.

[106] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on.* IEEE, 2004, pp. 83–92.

[107] M. Kim, "Understanding and aiding code evolution by inferring change patterns," in *Companion to the proceedings of the 29th International Conference on Software Engineering.* IEEE Computer Society, 2007, pp. 101–102.

[108] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo, "A language-independent software renovation framework," *Journal of Systems and Software*, vol. 77, no. 3, pp. 225–240, 2005.

[109] M. Di Penta, "Evolution doctor: a framework to control software system evolution," in *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on.* IEEE, 2005, pp. 280–283.

[110] M. Balint, R. Marinescu, and T. Girba, "How developers copy," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on.* IEEE, 2006, pp. 56–68.

[111] M. Rieger, S. Ducasse, and M. Lanza, "Insights into system-wide code duplication," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on.* IEEE, 2004, pp. 100–109.

[112] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on.* IEEE, 2007, pp. 81–90.

[113] N. Göde, "Evolution of type-1 clones," in *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on.* IEEE, 2009, pp. 77–86.

[114] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "Analysis of the linux kernel evolution using code clone coverage," in *Proceedings of the Fourth International Workshop on Mining Software Repositories.* IEEE Computer Society, 2007, p. 22.

[115] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society, 2007, pp. 106–115.

[116] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone smells in software evolution," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on.* IEEE, 2007, pp. 24–33.

[117] D. M. Shawky and A. F. Ali, "Modeling clones evolution in open source systems through chaos theory," in *Software Technology And Engineering (Icste), 2010 2nd International Conference On*, vol. 1. IEEE, 2010, pp. V1–159.

[118] D. R. Mandel, "Chaos theory, sensitive dependence, and the logistic equation." 1995.

[119] R. K. Saha, C. K. Roy, and K. A. Schneider, "An automatic framework for extracting and classifying near-miss clone genealogies," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on.* IEEE, 2011, pp. 293–302.

[120] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on.* IEEE, 2011, pp. 273–282.

[121] S. Xie, F. Khomh, Y. Zou, and I. Keivanloo, "An empirical study on the fault-proneness of clone migration in clone genealogies," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on.* IEEE, 2014, pp. 94–103.

[122] C. K. Roy and J. R. Cordy, "Near-miss function clones in open source software: an empirical study," *Journal of Software: Evolution and Process*, vol. 22, no. 3, pp. 165–189, 2010.

[123] K. H. Rosen and K. Krithivasan, *Discrete mathematics and its applications.* McGraw-Hill New York, 1995, vol. 6.

[124] R. Tairas, J. Gray, and I. Baxter, "Visualization of clone detection results," in *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange.* ACM, 2006, pp. 50–54.

[125] E. Adar and M. Kim, "Softguess: Visualization and exploration of code clones in context," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on.* IEEE, 2007, pp. 762–766.

[126] E. Adar, "Guess: a language and interface for graph exploration," in *Proceedings of the SIGCHI conference on Human Factors in computing systems.* ACM, 2006, pp. 791–800.

[127] Z. M. Jiang and A. E. Hassan, "A framework for studying clones in large software systems," in *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on.* IEEE, 2007, pp. 203–212.

[128] Y. Zhang, H. A. Basit, S. Jarzabek, D. Anh, and M. Low, "Query-based filtering and graphical view generation for clone analysis," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on.* IEEE, 2008, pp. 376–385.

[129] Y. Fukushima, R. Kula, S. Kawaguchi, K. Fushida, M. Nagura, and H. Iida, "Code clone graph metrics for detecting diffused code clones," in *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific.* IEEE, 2009, pp. 373–380.

[130] H. Murakami, Y. Higo, and S. Kusumoto, "Clonepacker: A tool for clone set visualization," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on.* IEEE, 2015, pp. 474–478.

[131] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue, "Simultaneous modification support based on code clone analysis," in *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific.* IEEE, 2007, pp. 262–269.

[132] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," in *Proceedings of the Fourth International Workshop on Mining Software Repositories.* IEEE Computer Society, 2007, p. 18.

[133] S. Meldal, G. L. Fisher, D. J. Stearns, and P. C. Ölveczky, "Software prototyping," *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1999.

[134] J. Krinke, "Is cloned code more stable than non-cloned code?" in *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on.* IEEE, 2008, pp. 57–66.

[135] N. Gode and J. Harder, "Clone stability," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 65–74.

[136] J. Krinke, "Is cloned code older than non-cloned code?" in *Proceedings of the 5th International Workshop on Software Clones*. ACM, 2011, pp. 28–33.

[137] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 485–495.

[138] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the impact of clones on software defects," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 13–21.

[139] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 503–530, 2012.

[140] P. Kumar, V. Sehgal, Nitin, K. Shah, S. S. P. Shukla, and D. S. Chauhan, "A novel approach for security in cloud computing using hidden markov model and clustering," in *Information and Communication Technologies (WICT), 2011 World Congress on*. IEEE, 2011, pp. 810–815.

[141] L. Aversano, G. Canfora, A. De Lucia, and P. Gallucci, "Web site reuse: cloning and adapting," in *Web Site Evolution, 2001. Proceedings. 3rd International Workshop on*. IEEE, 2001, pp. 107–111.

[142] G. A. Di Lucca, M. Di Penta, and A. R. Fasolino, "An approach to identify duplicated web pages," in *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*. IEEE, 2002, pp. 481–486.

[143] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora, "Understanding cloned patterns in web applications," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on.* IEEE, 2005, pp. 333–336.

[144] D. C. Rajapakse and S. Jarzabek, "An investigation of cloning in web applications," in *International Conference on Web Engineering.* Springer, 2005, pp. 252–262.

[145] A. De Lucia, M. Risi, G. Tortora, and G. Scanniello, "Clustering algorithms and latent semantic indexing to identify similar pages in web applications," in *Web Site Evolution, 2007. WSE 2007. 9th IEEE International Workshop on.* IEEE, 2007, pp. 65–72.

[146] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, p. 391, 1990.

[147] M. Bruntink, "Aspect mining using clone class metrics," in *1st Workshop on Aspect Reverse Engineering*, 2004.

[148] M. Bruntink, A. Van Deursen, R. Van Engelen, and T. Tourwe, "On the use of clone detection for identifying crosscutting concern code," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, 2005.

[149] S. Schulze, M. Kuhlemann, and M. Rosenmüller, "Towards a refactoring guideline using code clone classification," in *Proceedings of the 2nd Workshop on Refactoring Tools.* ACM, 2008, p. 6.

[150] R. Yokomori, H. Siy, N. Yoshida, M. Noro, and K. Inoue, "Measuring the effects of aspect-oriented refactoring on component relationships: two case studies," in *Proceedings of the tenth international conference on Aspect-oriented software development.* ACM, 2011, pp. 215–226.

[151] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Aries: refactoring support tool for code clone," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–4.

[152] H. A. Basit, D. C. Rajapakse, and S. Jarzabek, "Beyond templates: a study of clones in the stl and some general implications," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on.* IEEE, 2005, pp. 451–459.

[153] S. Jarzabek and S. Li, "Unifying clones with a generative programming technique: a case study," *Journal of Software: Evolution and Process*, vol. 18, no. 4, pp. 267–292, 2006.

[154] R. Tairas, "Clone detection and refactoring," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.* ACM, 2006, pp. 780–781.

[155] R. Tairas, "Centralizing clone group representation and maintenance," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications.* ACM, 2009, pp. 781–782.

[156] R. Tairas and J. Gray, "Get to know your clones with cedar," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications.* ACM, 2009, pp. 817–818.

[157] R. Tairas and J. Gray, "An information retrieval process to aid in the analysis of code clones," *Empirical Software Engineering*, vol. 14, no. 1, pp. 33–56, 2009.

[158] R. Tiarks, R. Koschke, and R. Falke, "An assessment of type-3 clones as detected by state-of-the-art tools," in *Source Code Analysis and Manipulation, 2009.*

SCAM'09. Ninth IEEE International Working Conference on. IEEE, 2009, pp. 67–76.

[159] C. Brown and S. Thompson, "Clone detection and elimination for haskell," in *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation.* ACM, 2010, pp. 111–120.

[160] D. M. Shawky and A. F. Ali, "An approach for assessing similarity metrics used in metric-based clone detection techniques," in *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, vol. 1. IEEE, 2010, pp. 580–584.

[161] J. Krinke, N. Gold, Y. Jia, and D. Binkley, "Distinguishing copies from originals in software clones," in *Proceedings of the 4th International Workshop on Software Clones.* ACM, 2010, pp. 41–48.

[162] R. Tairas, F. Jacob, and J. Gray, "Representing clones in a localized manner," in *Proceedings of the 5th International Workshop on Software Clones.* ACM, 2011, pp. 54–60.

[163] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics," in *Proceedings of the 5th International Workshop on Software Clones.* ACM, 2011, pp. 7–13.

[164] J. Kanwal, K. Inoue, and O. Maqbool, "Refactoring patterns study in code clones during software evolution," in *Software Clones (IWSC), 2017 IEEE 11th International Workshop on.* IEEE, 2017, pp. 1–2.

[165] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth*

ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 2010, pp. 371–372.

[166] P. Sarwesh, N. S. V. Shet, and K. Chandrasekaran, "Energy-efficient network architecture for iot applications," in *Beyond the Internet of Things*. Springer, 2017, pp. 119–144.

[167] H. Chaouchi, *The internet of things: connecting objects*. John Wiley & Sons, 2013.

[168] T. Ekman and G. Hedin, "The jastadd extensible java compiler," *ACM Sigplan Notices*, vol. 42, no. 10, pp. 1–18, 2007.

[169] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. ACM, 1970, pp. 1–19.

[170] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.

[171] "Git basics." [Online]. Available: https://git-scm.com/book/en/v2/

[172] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: New Opportunities for Connected Data*. " O'Reilly Media, Inc.", 2015.

[173] "Graph database." [Online]. Available: http://neo4j.com/developer/graph-database/

[174] "Cql." [Online]. Available: http://neo4j.com/docs/stable/cypher-query-lang.html

[175] M. Acharya, R. Jain, and S. Kansal, "Vertex equitable labeling of signed graphs," *Electronic Notes in Discrete Mathematics*, vol. 63, pp. 461–468, 2017.

[176] "The perl programming language." [Online]. Available: https://www.perl.org/

[177] "Apache hive." [Online]. Available: https://hive.apache.org/

[178] "Hadoop." [Online]. Available: http://hadoop.apache.org/

[179] N. Khan, I. Yaqoob, I. A. T. Hashem, Z. Inayat, W. K. Mahmoud Ali, M. Alam, M. Shiraz, and A. Gani, "Big data: survey, technologies, opportunities, and challenges," *The Scientific World Journal*, vol. 2014, 2014.

[180] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer *et al.*, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM Sigplan Notices*, vol. 41, no. 10. ACM, 2006, pp. 169–190.

[181] "Cloudera." [Online]. Available: https://cloudera.com/.

[182] "Openmrs-core." [Online]. Available: https://bitbucket.org/Ch3ck_/openmrs-core

[183] "Openmrs." [Online]. Available: http://openmrs.org/about/