DETECTION AND ANALYSIS OF NEAR-MISS SOFTWARE CLONES

by

CHANCHAL K. ROY

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
August 2009

Copyright © Chanchal K. Roy, 2009
Abstract

Software clones are considered harmful in software maintenance and evolution. However, despite a decade of active research, there is a marked lack of work in the detection and analysis of near-miss software clones, those where minor to extensive modifications have been made to the copied fragments. In this thesis, we advance the state-of-the-art in clone detection and analysis in several ways. First, we develop a hybrid clone detection method, called NICAD, that can detect both exact and near-miss clones with high precision and recall and with reasonable performance. Second, in order to address the decade of vagueness in clone definition, we propose an editing taxonomy for clone creation that models developers’ editing activities in the copy/pasted code in a top-down fashion. NICAD is designed to address the different types of clones in the editing taxonomy. Third, we have conducted a scenario-based qualitative comparison and evaluation of all of the currently available clone detection techniques and tools in the context of a unified conceptual framework. Using the results of this study one can more easily choose the right tools to meet the requirements and constraints of any particular application, and can identify opportunities for hybridizing different techniques. The hybrid architecture of NICAD was derived from this study. Fourth, in order to evaluate and compare the available tools in a realistic setting and to avoid the challenges and huge manual effort in validating candidate clones, we have developed a mutation-based framework that automatically and efficiently measures (and
compares) the recall and precision of clone detection tools for different fine-grained clone
types of the proposed editing taxonomy. We have evaluated NICAD using this framework
and found that it is capable of detecting different types of clones with high precision and
recall. Finally, we have conducted a large scale empirical study of cloning in open source
systems, both to evaluate NICAD and to study the cloning characteristics of these sys-
tems in several different dimensions. The study has demonstrated that NICAD is capable
of accurately finding both exact and near-miss function clones even in large systems and
different languages, and that there seem to be a large number of clones in those systems.
Co-Authorship

Part of Chapter 2 has been published in a paper co-authored with James R. Cordy and Rainer Koschke in the journal Science of Computer Programming [184]. Other parts of this chapter have been prepared with James R. Cordy and Rainer Koschke for submitting to ACM Computing Surveys. Chapter 3 has been published in the proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC’08) co-authored with James R. Cordy [185]. Chapter 4 is a part of two other papers that have been co-authored by James R. Cordy and have been published in the proceedings of the Canadian Conference on Computer Science and Software Engineering (C3S2E’08) [189] and Mutation 2009 [191]. Chapter 5 is a product of two papers: the conference version has been published in the proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC’08) co-authored with James R. Cordy [188] and the extended journal version been published in the journal Science of Computer Programming co-authored by James R. Cordy and Rainer Koschke [184]. Chapter 6 has been published at Mutation 2009 co-authored with James R. Cordy [191]. Chapter 7 is a product of two papers: the conference version has been published in the proceedings of the 15th Working Conference on Reverse Engineering (WCRE’08) co-authored with James R. Cordy [186] and the extended journal version has been accepted for publication in the Journal of Software Maintenance and
Evolution: Research and Practice, again co-authored with James R. Cordy [187]. This thesis will appear in the proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM’09) as a doctoral symposium paper [190]. For all the papers I was the primary author and conducted the research under the supervision of James R. Cordy.
Statement of Originality

I hereby certify that the research presented in this dissertation is my own, conducted under the supervision of James R. Cordy. Ideas and techniques that are not a product of my own work are cited, or, in cases where citations are not available, are presented using language that indicates they existed prior to this work.
Dedication

To my wonderful daughter Prisha, whose arrival on this Earth has made our lives happier than ever before and worked as a source of inspiration for my study.
Acknowledgements

First of all, I would like to express my heart-felt and most sincere gratitude to my respected supervisor James R. Cordy for his constant guidance, advice, encouragement and extraordinary patience during this thesis work. Without him, this work would have been impossible, and he has in fact further unlocked the research potential within me. I also thank Rainer Koschke for his help and suggestions during the thesis work. We have been co-authoring a couple of papers produced from the thesis.

I would like to thank Thomas R. Dean and Ying (Jenny) Zou, the other two members of my Ph.D. supervisory committee for their suggestions and encouragement during the thesis work. Thanks are also due to Michael W. Godfrey, Patrick Martin and Selim Akl, the other members of my thesis examination committee for their helpful comments, insights and suggestions.

I am grateful to Natural Sciences and Engineering Research Council of Canada (NSERC), the Ontario Graduate Scholarship (OGS) program and Queen’s University for their generous financial support through scholarships, awards and bursaries that helped me to concentrate more deeply on my thesis work.

I thank the anonymous reviewers for their valuable comments and suggestions in improving the papers produced from this thesis. I also thank the tool authors who provided useful answers to our queries.
I would like to thank all of my friends and other staff members of the School of Computing who have helped me in one way or another along the way. In particular I would like to thank Ahmad Saifan, Scott Grant, Adrian Thurston, Rezoanoor (Ruben) Rahman, Asil Almonaies, M. Atif Farid, Karolina Zurowska, Gias Uddin, Hung Tam, Debby Robertson, Lynda Moulton, Richard Linley, Ben Hall and Tom Bradshaw.

I thank to the members of Kingston Bangladeshi community for their social and moral support during our stay at Kingston. In particular, I would like to thank Hafizur Rahman, Quazi Maidul Islam, Shamsul Karim, Shahed Kalam, Kalam Mir, Shakawat H. Chowdhury, Nasrin Jahan, Md. Zulkernine, Emran Al-Karim, Kamrul Islam, Salimur Choudhury and Siraj Rahman.

I express my gratefulness to my family members and relatives especially, my mother-in-law Bela Rani Karmaker, youngest brother Ujjal Roy, elder brothers Ratan K. Roy and Hriday R. Roy, sister-in-laws Kalyani Roy and Laboni Roy Karmokar (and her husband Sonjoy Karmokar) for their support and inspiration.

The most wonderful thing of my life is my daughter Prisha, who came as an angel in our lives and drove me in the right direction in successfully finishing my thesis and getting a job. I thank T.C. Nicholas Graham, the Ph.D. supervisor of my wife for his overwhelming support both during her pregnancy and after Prisha was born.

Finally, I am deeply indebted to my incomparable wife Banani Roy for more than I could ever express, due to her love, patience and support even through a tough situation like giving birth and raising a wonderful daughter during the thesis work. They (were and) are the light of my life ahead. Thanks to God for His love for us.
# Table of Contents

Abstract .......................................................... i

Co-Authorship ....................................................... iii

Statement of Originality ......................................... v

Dedication ............................................................ vi

Acknowledgements .................................................. vii

Table of Contents ................................................... ix

List of Tables ......................................................... xiii

List of Figures ......................................................... xv

## Chapter 1:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Contributions of the Thesis</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Problem Statement</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Outline of the Thesis</td>
<td>11</td>
</tr>
</tbody>
</table>
Chapter 2:

**Background and Related Work** ........................................ 13

2.1 Clone Terminology ....................................................... 14
2.2 Evidence of Clones in Software Systems ............................. 17
2.3 Why are there Clones in Software Systems? ........................ 17
2.4 Do Clones Really Matter? ............................................... 24
2.5 Other Applications of Code Similarity Detection ................. 30
2.6 Clone Detection Process ............................................... 35
2.7 Overview of Clone Detection Techniques and Tools ............. 42
2.8 Clone Presentation ...................................................... 50
2.9 Clone Evolution ........................................................ 56
2.10 Clone Management ..................................................... 58
2.11 Summary ........................................................................ 63

Chapter 3:

**NICAD: A Next Generation Code Clone Detection Tool** ........... 64

3.1 Motivation ................................................................. 65
3.2 Background ............................................................... 66
3.3 Proposed Approach ...................................................... 68
3.4 Extraction of Potential Clones ........................................ 69
3.5 Flexible Pretty-Printing ................................................ 73
3.6 Flexible Code Normalization ......................................... 76
3.7 Flexible Code Filtering ................................................ 80
3.8 Comparing the Potential Clones ...................................... 81
3.9 Reporting of Found Clones ............................................. 86
3.10 Evaluation of NICAD ................................................. 88
3.11 Time and Space Complexities ........................................ 94
3.12 Related Work .............................................................. 96
3.13 Summary ................................................................. 97

Chapter 4:
Towards an Editing Taxonomy for Clone Creation ....................... 98
  4.1 Definitions of Code Clone .............................................. 99
  4.2 Towards an Editing Taxonomy for Clone Creation .................... 102
  4.3 Summary ............................................................... 106

Chapter 5:
A Scenario-based Qualitative Comparison and Evaluation of Clone
Detection Techniques and Tools ............................................. 108
  5.1 Motivation ............................................................... 109
  5.2 Comparison of Tools ................................................... 112
  5.3 Scenario-Based Evaluation of the Techniques and Tools .............. 127
  5.4 An Example Use of the Study ......................................... 144
  5.5 Related Work .......................................................... 150
  5.6 Summary .............................................................. 151

Chapter 6:
A Mutation / Injection-based Framework for Automatically Evalu-
ating Code Clone Detection Tools ......................................... 153
  6.1 Motivation .............................................................. 154
  6.2 Mutation Operators for Cloning ..................................... 156
6.3 The Evaluation Framework ........................................ 159
6.4 An Example Use of the Framework .............................. 170
6.5 Related Work .......................................................... 172
6.6 Summary ................................................................. 175

Chapter 7:
An Empirical Study of Near-miss Function Clones in Open Source Software ........................................ 176
7.1 Motivation ................................................................. 177
7.2 Experimental Setup .................................................... 179
7.3 Experimental Results .................................................. 188
7.4 Related Work ............................................................ 211
7.5 Summary ................................................................. 213

Chapter 8:
Conclusion ................................................................. 214
8.1 Summary ................................................................. 214
8.2 Future Research Directions ......................................... 218

Bibliography .............................................................. 221
List of Tables

2.1 Taxonomy of clone detection techniques and tools . . . . . . . . . . . . . . 44

3.1 Pretty-printing and comparing . . . . . . . . . . . . . . . . . . . . . . . 74
3.2 Typical normalization of an if-then-else . . . . . . . . . . . . . . . . . . . 78
3.3 Two function clones . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
3.4 Two function non-clones . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
3.5 Cloned fragments and classes from Weltab . . . . . . . . . . . . . . . . . . 92

5.1 Usage facets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
5.2 Interaction facets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
5.3 Language facets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
5.4 Clone facets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
5.5 Technical facets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
5.6 Adjustment facets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
5.7 Basic transformation/normalization facet . . . . . . . . . . . . . . . . . . . 122
5.8 Code representation facet . . . . . . . . . . . . . . . . . . . . . . . . . . 123
5.9 Program analysis facet . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
5.10 Evaluation facet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
5.11 Tools attributes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
5.12 Meanings of the rating symbols . . . . . . . . . . . . . . . . . . . . . . . . 132
5.13 Scenario-based evaluation of the surveyed clone detection techniques and tools

6.1 Mutation operators for cloning

6.2 Overview of the subject code bases

6.3 Recall and precision of the Tools

7.1 Overview of the subject systems

7.2 Comparison and timing statistics in the systems

7.3 Percentage cloned LOC, clone pairs and clone classes

7.4 Percentage of files that have clones over a certain percentage for C systems

7.5 Percentage of files that have clones over a certain percentage for Java systems

7.6 Percentage of files that have clones over a certain percentage for C# systems

7.7 Percentage localization of clone pairs for C Systems

7.8 Percentage localization of clone pairs for Java Systems

7.9 Percentage localization of clone pairs for C# Systems

7.10 Percentage localization of remote clone pairs for C systems

7.11 Percentage localization of remote clone pairs for Java systems

7.12 Percentage localization of remote clone pairs for C# systems
# List of Figures

2.1 Clone pair and clone class ............................................. 15
2.2 Tree-diagram for the reasons for cloning ............................. 19
2.3 A generic clone detection process .................................... 36
2.4 Scatter plots ................................................................. 51
2.5 Hasse diagram ............................................................... 52
2.6 Polymetric view .............................................................. 53
2.7 Tree maps ................................................................. 54
2.8 Clones visualiser view in Eclipse .................................... 55
3.1 NICAD clone detection process ....................................... 69
3.2 Example result of #if-def resolution ................................ 71
3.3 A raw original code fragment ......................................... 71
3.4 Standard pretty-printed code fragment .............................. 71
3.5 Example original code fragment ...................................... 83
3.6 Sample XML output from NICAD ..................................... 87
3.7 HTML output of NICAD: exemplar clones ......................... 89
3.8 HTML output of NICAD: a clone pair ............................... 90
4.1 Example application of the editing taxonomy for cloning ...... 103
5.1 Taxonomy of editing scenarios for different clone types . . . . . . . . . . . 129

6.1 The proposed evaluation framework . . . . . . . . . . . . . . . . . . . . 160

7.1 Example of possible false positive clones . . . . . . . . . . . . . . . . . . 182
7.2 Proportion of cloned functions/methods in the systems . . . . . . . . . . 190
7.3 Example of small functions/methods in C# and Java . . . . . . . . . . . . 191
7.4 Percentage of exact clones by clone size . . . . . . . . . . . . . . . . . . . 192
7.5 Percentage of files that are associated with cloned code . . . . . . . . . 195
7.6 Some copy/paste change examples . . . . . . . . . . . . . . . . . . . . . . 201
7.7 Two exact clones in Java system in Spule . . . . . . . . . . . . . . . . . . . 204
7.8 Two exact clones in C# system Linq . . . . . . . . . . . . . . . . . . . . . 204
7.9 Percentage localization of clone pairs at UPIT = 0.3 . . . . . . . . . . . . 211
Chapter 1

Introduction

This chapter provides a short introduction to this thesis. After providing a general motivation to this thesis in Section 1.1, we list the contributions of this thesis in Section 1.2. We then state the problems and outline our contributions to those problems in Section 1.3. Finally, in Section 1.4 we provide an outline of the remaining chapters.

1.1 Motivation

Reusing code fragments by copying and pasting with or without minor adaptation is a common activity in software development. As a result, software systems often contain sections of code that are very similar, called software clones or code clones. Previous research shows that a significant fraction (between 7% and 23%) of the code in a typical software system has been cloned [15], and in one extreme case it was even 50% [68]. While such cloning is often intentional [125] and can be useful in many ways [121], it can also be harmful in software maintenance and evolution [110]. For example, if a bug is detected in a code fragment, all fragments similar to it should be checked for the same bug [156].
Duplicated fragments can also significantly increase the work to be done when enhancing or adapting code [162]. A recent study that worked in the context of industrial systems shows that inconsistent changes/updates to cloned code are frequent and lead to severe unexpected behavior [110]. Several other studies also show that software systems with code clones can be more difficult to maintain [107, 15] and can introduce subtle errors [47, 156]. Thus code clones are considered one of the bad “smells” of a software system [74] and it is widely believed that cloned code can make software maintenance and evolution significantly more difficult. Thus the detection, monitoring and removal of code clones is an important topic in software maintenance and evolution [145].

Many other software engineering tasks, such as program understanding (clones may carry domain knowledge), code quality analysis (fewer clones may mean better quality code), aspect mining (clones may indicate the presence of an aspect), plagiarism detection, copyright infringement investigation, software evolution analysis, code compaction (for example, in mobile devices), virus detection, and bug detection may require the extraction of syntactically or semantically similar code fragments, making clone detection (code similarity detection) an important and valuable part of software analysis in particular and software engineering in general [182]. However, despite a decade of active research, there has been a marked lack of progress in the detection and analysis of software clones, especially with respect to near-miss software clones (minor to significant editing differences between code fragments). In this thesis, we have advanced the state-of-the-art in clone detection research in the context of both exact and near-miss software clones.

1.2 Contributions of the Thesis

The contributions of this thesis are five-fold, as follows:
• First, a language-specific but lightweight hybrid clone detection method called NICAD has been proposed that can detect both exact and near-miss clones with high precision and recall and with reasonable performance.

• Second, to overcome the vagueness in clone definition, we have proposed an editing taxonomy for clone creation. NICAD was enhanced accordingly to detect different fine-grained types of clones of the editing taxonomy.

• Third, a scenario-based qualitative comparison and evaluation of all the existing clone detection techniques and tools has been provided with respect to a set of distinguishable features and a set of editing scenarios created from the editing taxonomy in the context of a coherent conceptual framework. This framework is helpful in choosing the right tool(s) to meet a given set of requirements and constraints and in designing new hybrid clone detection tools such as NICAD, which was derived from this study.

• Fourth, in order to evaluate and compare the available tools in a realistic setting, we have also developed a mutation / injection-based framework that automatically and efficiently measures (and compares) the recall and precision of clone detection tools for different fine-grained clone types created from the proposed editing taxonomy. NICAD was evaluated using this framework and found to give high precision and recall for the different fine-grained types of clones.

• Fifth, a large scale empirical study has been conducted both to evaluate NICAD and to study the cloning status of several large systems in several different languages.
CHAPTER 1. INTRODUCTION

1.3 Problem Statement

In this section, we first provide a summary of the existing clone detection techniques and tools, and then discuss their strengths and limitations, from which we then derive the motivation of our proposed hybrid clone detection method. We also point out the other key objectives of this thesis and the motivation behind them.

Considering the importance, over the decades several clone detection techniques have been proposed [184]. Some techniques for example, find clones by comparing program text (e.g., [68, 212, 149, 48]) or an alternative representation of program text (e.g., fingerprint [108, 160] or Latent Semantic Indexing [161]) with little or no code normalization (called textual/text-based approaches). Some techniques, on the other hand, use a lexer to make a token sequence for the whole program (e.g., [113, 203, 22, 155]) or for each basic block (e.g., [156]) or for each line of the program (e.g., [15]), apply token normalization and/or transformation on the sequence(s) and find clones by finding common subsequences on the token sequence(s) (called lexical/token-based approaches).

Some techniques make use of parsers to build a parse tree or an Abstract Syntax Tree (AST) and then find clones by comparing trees/sub-trees directly (e.g., [26, 25, 214]) or an alternative representative representation of trees/sub-trees (e.g., characteristic vector [105], XML [70, 208], sequence of AST nodes [137, 199]) (called syntactic/tree-based approaches). Some other techniques (e.g., [162, 173, 133, 20, 56, 63, 42]) also use ASTs, control flow graphs and/or program text (e.g., no. of lines, volume of comments etc.) to calculate some metrics for each potential clones (e.g., begin-end blocks) and find clones by comparing the metric values (called syntactic/metric-based approaches) of the potential clones. Some techniques make use of semantical knowledge by first building Program Dependency Graphs (PDGs) and find clones by comparing the PDGs (e.g., [143, 129]) (called
Semantical/PDG-based approaches), while some others are hybrid, use both syntactic (e.g., metrics) and semantic (e.g., call graph) knowledge to find clones (e.g., [150]).

One of the strengths of the textual/text-based approaches (e.g., [68]) is that they can detect clones with high user confidence [32, 207]. While performance can be improved, most often they cannot find meaningful clones, i.e., find spurious clones [137] and cannot find near-miss clones. Lexical/token-based approaches are, on the other hand, very fast w.r.t. the number of tokens, have high recall but also give many false positives. Like textual/text-based approaches they also return spurious clones unless special treatment is made. Syntactic/tree-based approaches are good at finding syntactic clones, however, tree-comparison is costly. Though computational complexity can be improved by using a suffix tree-based comparison algorithm by serializing the AST nodes [137], generating ASTs is a heavy-weight language-dependent approach and most existing syntactic/tree-based approaches are dependent on an integrated environment or on a fully-fledged standard parser for getting the ASTs. Moreover, such approaches have not been found effective in finding near-miss clones [32]. Although fast, metrics-based approaches are as heavy-weight as syntactic/tree-based approaches as they also need to generate metrics from ASTs/Flow Graphs. These techniques also return a lots of false positives as dissimilar code segments may return similar metrics values. Semantical approaches may find some certain types of near-miss clones (e.g., clones in which some statements are reordered in the copied fragment) but they are the most expensive ones and extremely heavy-weight which need to generate PDG-maker. Moreover, graph comparison is costly and have not been scaled so far. For a further detailed study on the techniques the reader is referred to [182, 138, 184].

**NICAD: A Hybrid Clone Detection Method:** From the above discussion we see that lightweight text-based techniques can find clones with high accuracy and confidence, but
detected clones often do not correspond to appropriate syntactic units [32]. Parser-based syntactic (AST-based) techniques, on the other hand, find syntactically meaningful clones but tend to be more heavyweight, requiring a full parser and subtree comparison method. Experimental results also show that parser-based techniques give low recall [32]. Moreover, neither text-based nor parser-based techniques have been found to be effective in detecting near-miss clones [32].

In this thesis, we propose a multi-pass hybrid approach, called NICAD (Chapter 3), which is language-specific but lightweight and detects both exact and near-miss clones with high precision and recall, and with reasonable performance.

NICAD works in three phases. First, all potential clones (code fragments of the target kind) are identified, pretty-printed and extracted from the subject code base. During the extraction, they are also automatically stripped of formatting and comments and pretty-printed by TXL [53] according to the language grammar’s formatting cues. Standard pretty-printing ensures consistent layout and spacing of code for later comparisons. Extracted potential clones can be further processed with three more flexible options, (1) Flexible pretty-printing—in which language and (possibly) project specific special pretty-printing is applied on the potential clones. This helps us to break different parts of a statement into several lines so that local changes to the parts of a statement can be isolated using a simple line-comparison, (2) Flexible code normalization—in which we can easily normalize parts of a statement (or whole statements of a given type) to ignore editing differences using TXL’s transformation rules, and (3) Flexible code filtering—in which we can efficiently filter out unimportant code statements from potential clones according to user preferences. Second, the preprocessed potential clones are compared line-by-line textwise using a longest common subsequence (LCS) algorithm similar to the Unix diff utility. To determine whether
two potential clones really are clones of each other, we compare their pretty-printed (and optionally normalized/filtered) text lines and use the number of unique lines in each as a measure of similarity/dissimilarity. Finally, the results from the comparison phase are reported in both XML database form with detailed information about each clone class and as an interactive HTML website where original code fragments are shown in HTML pages to assist hand validation.

This approach is lightweight in the sense that like other pure text-based approaches [188], we also find clones by comparing program text and that we use a robust island grammar instead of a fully-fledged parser. Although pretty-printing, code normalization and filtering all use TXL agile parsing and context-sensitive transformation rules, they can be done on a source file-by-file basis and are scalably independent of the program’s overall structure. The method can be applied to any language for which we have an approximate (pretty-printing) TXL grammar and (optionally) examples of the desired variances and code normalizations for the language. It is language-specific in that sense. The method is validated with both large scale empirical studies (Chapter 7) and a mutation-based framework (Chapter 6) and found effective in detecting both exact and near-miss clones with high precision and recall.

**An Editing Taxonomy for Clone Creation:** The definitions of code clone in the literature [191, 182] are inherently vague and in most cases are detection-dependent and/or task-specific. To overcome this limitation, we have proposed an editing taxonomy for clone creation (Chapter 4) that models developers’ editing activities. In particular, we are interested in those clones that are created as a result of copy/paste/modify actions by programmers. Based on this assumption, we have proposed a top-down theory of clone creation and formalized into a taxonomy of the editing actions that a programmer may undertake in the
intentional creation of clones. The taxonomy is derived from the large body of published work on existing clone definitions, clone types, clone taxonomies and studies of developer copy/paste activities [182]. We have validated the taxonomy by studying the copy/paste patterns of function clones [183] found in an empirical study (Chapter 7). NICAD is then designed to address the different types of clones in the taxonomy.

A Conceptual Framework for Qualitative Comparison of Existing Techniques and Tools: Clone detection techniques are often inadequately evaluated. The general lack of evaluation is exacerbated by the fact that there are no agreed-upon evaluation criteria or representative benchmarks [11]. In particular, accuracy measures such as precision and recall have only been roughly estimated, due both to problems in creating a validated clone benchmark against which tools can be compared, and to the enormous manual effort required to hand check large numbers of candidate clones. In an attempt to compare all clone detection techniques more uniformly, independent of tool availability, implementation limitations or language, we have conducted a scenario-based qualitative approach and organize the large number of techniques and tools in a coherent conceptual framework (Chapter 5).

First, we perform a classification and overall comparison with respect to a number of facets, each of which has a set of (possibly overlapping) attributes. Second, based on the editing taxonomy introduced in Chapter 4, we design editing scenarios to create Type 1, Type 2, Type 3 and Type 4 clones (Chapter 2). Using these scenarios we qualitatively evaluate the techniques and tools we have previously classified in Chapter 2. In particular, we estimate how well the various clone detection techniques may perform based on their published properties (either in the corresponding published papers or online documentation). Using this framework, one can efficiently and instantly choose the right set of tools for her purpose or can develop a hybrid clone detection method. The hybrid architecture of
NICAD is in fact derived from this framework. Consequently, NICAD was placed one of the best tools available in the framework.

**A Mutation-/Injection-based Framework for Evaluating Clone Detection Tools:**
Again, the conceptual framework proposed above can only provide qualitative comparison and evaluation of the clone detection techniques and tools rather than quantitative measures. In order to evaluate and compare the available tools in a realistic setting, in this thesis, we develop a mutation/injection-based framework [191] that automatically and efficiently measures (and compares) the recall and precision of clone detection tools for different fine-grained clone types of the proposed editing taxonomy (Chapter 4).

For each of the fine-grained clone types of the proposed editing taxonomy (Chapter 4), we design code mutation operators to model each type by mimicking developers’ typical editing activities in clone creation using TXL. By using these operators, randomly mutated clone fragments are generated from the original subject code base. These mutated fragments are then injected to the original code based to get thousands of mutated code bases. By tracking the large number of injected artificial clones in these mutated code bases, we then automatically measure how well (i.e., values precision and recall) these known clones are detected by a particular tool (for individual tool evaluation) or group of tools (for comparing different tools).

First, we have evaluated NICAD using this framework (where open source Weltab has been used as code base) and found that it gives high precision and recall for different fine-grained types of clones. We have then compared NICAD with two of its variants, the standard and flexible pretty-printing (PP) options with dissimilarity thresholds. We experienced that full NICAD gives both high precision and recall for Type 1, Type 2 and Type 3 clones and better than its variants.
Although the framework is initially targeted at evaluating NICAD and its variants, the objective was to design a generic framework for evaluating and comparing clone detection tools by overcoming the known challenges to objective tool comparison experiments. The resulting framework is flexible and adaptable enough to evaluate third party clone detection tools provided that the tools can be run from the command line and that they provide textual report of detected clones with full file name, and begin and end line numbers of the code fragments of found clone pairs.

An Empirical Study of Near-Miss Function Clones in Open Source Software: Again, despite a decade of active research there has been a marked lack of in-depth comparative studies of cloning, particularly in a variety of systems. There have been many empirical studies on cloning, for example every new technique comes with some sort of empirical validation [184], and empirical studies are used when comparing tools [32].

However, in both cases the focus is on validating or comparing the techniques rather than the clone properties of the subject systems themselves. Particular subject systems have also been analyzed with respect to aspects such as harmfulness/usefulness and maintenance issues of clones, taxonomies of clones or evolution studies of clones [182].

In this thesis, we examine more than twenty open source C, Java and C# systems, including the entire Linux Kernel, j2sdk-Swing and db4o, and compare their use of cloned code in several different dimensions, including language, clone size, clone similarity (i.e., with different UPI thresholds), clone location and clone density both by proportion of cloned functions and lines of cloned code (Chapter 7). In particular, we focus on the following four research questions in this study:

(1) Is NICAD capable of detecting clones from large systems? Can it detect clones from systems of different languages? (2) What is the cloning status of open source systems? Are
there many clones? Are there more near-miss clones than exact clones? (3) Are there significant differences in cloning between the different language paradigms used in open source systems? (4) Are there significant differences in cloning in large systems compared to medium and small-sized open source systems?

As regards the research questions, for the first question, we showed that NICAD is capable of accurately finding both exact and near-miss function clones even in large systems and different languages. For the second question, we can say that our results indicate a large number of exact function clones in these open source systems. We also see much higher percentages of near-miss clones, indicating significantly higher numbers of near-miss clones than exact clones in these systems. For the third question, we can say that we observed many more exact function clones in object-oriented Java and C# systems than in C systems. However, the effect of increasing the UPI threshold for near-miss clones was almost identical regardless of language paradigm. For the fourth question, we observed no significant differences in cloning related to the size of the systems. Even though Linux is huge, its cloning characteristics seem to be typical of other C systems. Similarly, the largest Java system Swing and the largest C# system db4o seem to be representative of the cloning characteristics of other systems written in their corresponding languages. The detailed results are available in an online repository [183] in a variety of formats and can be used as a benchmark.

1.4 Outline of the Thesis

In this Chapter 1, we have motivated the research problem in software clone identification and analysis along with the contributions of this thesis. The remaining chapters of the thesis are organized as follows:
**Chapter 2:** This chapter is a blend of a further motivation to this thesis, some background studies and related work.

**Chapter 3:** This chapter presents the new hybrid clone detection method NICAD that we have developed. We provide the details of the method along with a rst small empirical study that demonstrates the effectiveness of NICAD.

**Chapter 4:** In this chapter, first we show the definitional vagueness of clones in the literature and then we propose an editing taxonomy for clone creation in the form of an example. This taxonomy is used as the foundation of the other parts of this thesis.

**Chapter 5:** This chapter provides a conceptual framework for comparing and evaluating all the existing clone detection techniques and tools (including NICAD) in a scenario-based fashion. We also show how the findings of this study can be used by a potential user to select a set of tools for her purpose given a set of constraints or by a tool builder to make a hybrid clone detection method.

**Chapter 6:** In order to automatically measure the precision and recall of NICAD for different fine-grained clone types in a realistic setting, we provide a mutation-/injection-based framework in this chapter. The framework is flexible enough to evaluate and compare other third party clone detection tools.

**Chapter 7:** In this chapter we provide a large scale empirical study of open source software, both to evaluate NICAD and to explore the cloning status of open source software in several different dimensions.

**Chapter 8:** Finally, this chapter concludes the thesis along with some directions for future research.
Chapter 2

Background and Related Work

This chapter continues our motivation, provides background and surveys related work. We begin with Section 2.1, which provides an abstract definition of clones and the clone relation. We provide some third party quantitative data on the percentage of cloned code in software systems in Section 2.2. In Section 2.3 we consider why there are so many clones in systems and the root causes behind cloning. Section 2.4 discusses why clones matter in software maintenance, and Section 2.5 outlines other important software engineering issues affected by clones. We provide a generic clone detection process in Section 2.6 and present an overview of the available clone detection methods in Section 2.7 in the form of a taxonomy. We place our new hybrid method NICAD (Chapter 3) in this taxonomy. Presentation of clones is an important issue, and clone visualization approaches are presented in Section 2.8. Section 2.9 reviews studies of the evolution of clones, and Section 2.10 reviews methods for managing clones in software development, maintenance and evolution. Finally, Section 2.11 summarizes the chapter.
2.1 Clone Terminology

Clone detection tools normally report clones in the form of Clone Pairs (CP) or Clone Classes (CC) or both. Both these terms are based on a similarity relation between two or more cloned fragments. The similarity relation between the cloned fragments is an equivalence relation (i.e., a reflexive, transitive, and symmetric relation) [113]. A clone relation holds between two code fragments if (and only if) they are the same sequences, where sequences may refer to the original character text strings, strings without whitespace, sequences of tokens, transformed or normalized sequences of tokens, and so on. In the following we define clone pair and clone class in terms of the clone relation.

**Definition 1: Code Fragment.** A code fragment (CF) is any sequence of code lines (with or without comments). It can be of any granularity, e.g., function definition, begin-end block, or sequence of statements. A CF is identified by its file name and begin-end line numbers in the original code base and is denoted as a triple (CF.FileName, CF.BeginLine, CF.EndLine).

**Definition 2: Code Clone.** A code fragment CF2 is a clone of another code fragment CF1 if they are similar by some given definition of similarity, that is, \( f(CF1) = f(CF2) \) where \( f \) is the similarity function (see clone types below). Two fragments that are similar to each other form a clone pair (CF1, CF2), and when many fragments are similar, they form a clone class or clone group.

**Definition 3: Clone Types.** There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (independent of their text). The first kind of clone is often the result of copying a code fragment and pasting into another location. In the following we provide the types of clones based on both the textual (Types 1 to 3) [32]
CHAPTER 2. BACKGROUND AND RELATED WORK

and functional (Type 4)[76, 129] similarities:

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

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

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

**Type 4:** Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

![Figure 2.1: Clone pair and clone class](image)

**Definition 4: Clone Pair:** A pair of code fragments is called a clone pair if there exists a clone relation between them, i.e., a clone pair is a pair of code fragments which
are identical or similar to each other. For instance, the first code segment (a) in fragment 1 of Figure 2.1 together with segment (a) in fragment 2 form a clone pair. Clones may subsume each other and we are typically interested only in the maximally long clones. For instance, segment (a) could be joined with the subsequent segment (b) in fragment 1 of Figure 2.1 to form a larger clone together with the joined segments (a) and (b) in fragment 2. A maximally long clone pair is one whose two fragments can be extended neither to the left nor to the right to form a larger clone. For the three code fragments, Fragment 1 (F1), Fragment 2 (F2) and Fragment 3 (F3) of Figure 2.1, we have the following maximally long clone pairs: \(<F1(a + b), F2(a + b)>, <F2(b + c), F3(a + b)>, <F1(b), F3(a)>\).

We note that the clone relation defined by maximally long clone pairs is not transitive. For instance, if we have the following string modelling code fragments \(adbdadb\), we observe that \(d\) occurs three times in this string. The second and third \(d\) form a maximally long clone pair, but the first and last \(d\) do not form a maximally long clone pair because they can be extended neither to the left nor to the right to form the subsuming clone pair of the two \(abc\). Only if we do not insist on maximal length, clone pairs may be summarized to equivalence classes, also known as clone classes.

**Definition 5: Clone Class:** A clone class [162] is the equivalence class formed by the (non-maximal) clone relation, that is, it is the maximal set of code fragments in which any two of the code fragments is a clone pair. For the three code fragments of Figure 2.1, we get the clone class \(<F1(b), F2(b), F3(a)>, <F1(b), F2(b)>, <F2(b), F3(a)>, <F1(b), F3(a)>, <F1(b), F3(a)>\). A clone class is therefore simply the union of all clone pairs that have code fragments in common [180]. Clone classes may be further summarized into clone class families if they occur in the same context.
**Definition 6: Clone Class Family:** The group of all clone classes that have the same domain is called a clone class family \([180]\) or super clone \([103]\). The domain of a clone class is the set of source entities from which its source fragments stem \([180]\). The particular source entities to be considered as domains depend on the particular programming language or scope of interest, but common examples are files, functions, classes, or packages.

### 2.2 Evidence of Clones in Software Systems

Previous research has shown that a significant fraction of the code in software systems is cloned, depending on the domain and origin of the software system \([125, 156, 105]\). Baker \([15]\) found that in large systems between 13% - 20% of source code can be cloned code, Lague\(\) et al. \([145]\) have reported that between 6.4% and 7.5% of functions were cloned in the systems they studied, and Baxter et al. \([26]\) found that 12.7% of code in a large software system was cloned. Mayrand et al. \([162]\) estimated that industrial source code contains 5% – 20% duplicated code, and Kapser and Godfrey \([115]\) have reported that as much as 10% – 15% of source code of a large system was cloned. In one object-oriented COBOL system, the rate of duplicated code was found to be even higher, at 50% \([68]\).

### 2.3 Why are there Clones in Software Systems?

Given that a significant amount of code in software systems is cloned code (Section 2.2), the question arises why there are such huge percentages of clones in the systems. In fact, code clones do not occur in software systems by themselves. There are several factors that might force or influence the developers and/or maintenance engineers in making cloned code in the system. Clones can also be introduced by accident. In Figure 2.2 (the leaf
nodes) we provide the various factors for which clones can be introduced in the source
code [15, 26, 109, 113, 133, 162, 181, 124] where a short description for some of the
factors are discussed below:

2.3.1 Development Strategy

Clones can be introduced in software systems due to many different reuse and programming
approaches. Examples are:

a. Reuse Approach: Reusing code, logic, design and/or an entire system are the prime
reasons of code duplication.

Simple reuse by Copy/Paste: Reusing existing code by copying and pasting (with or
without minor modifications) is the simplest form of reuse mechanism in the development
process which results in code duplication. It is a fast way of reusing reliable semantic and
syntactic constructs. It is also one of the ways of implementing cross-cutting concerns
[124].

Forking: The term Forking is used by Kapser and Godfrey [120] to mean the reuse of
similar solutions with the hope that they will be diverged significantly with the evolution
of the system. For example, when creating a driver for a hardware family, a similar hard-
ware family may already have a driver, and thus can be reused with slight modifications.
Similarly, clones can be introduced when porting software to new platforms and so on.

Design, functionality and logic reuse: Functionality and logic can be reused if there is
already a similar solution available. For example, often there is a high similarity between
the ports of a subsystem to different architectures. The different ports of a subsystem
(especially, for operating system subsystems) are likely similar in their structure and func-
tionality. For example, Linux kernel device drivers contain large rates of duplication [85]
CHAPTER 2. BACKGROUND AND RELATED WORK

Figure 2.2: Tree-diagram for the reasons for cloning
because all the drivers have the same interface and most of them implement similar simple logic. Moreover, the design of such systems does not allow for more sophisticated forms of reuse.

b. Programming Approach Clones can be introduced by the way a system is developed. Examples are:

Merging of two similar systems: Sometimes two software systems of similar functionality are merged to produce a new one. Although these systems may have been developed by different teams, clones may occur in the merged system because of the implementations of similar functionality in both systems.

System development using a generative programming approach: Generating code with a tool using generative programming can produce many large code clones because these tools often use the same template to generate the same or similar logic.

Delay in restructuring: It is a common developer practice to delay restructuring developed code, which may ultimately introduce clones.

2.3.2 Maintenance Benefits

Clones are sometimes purposely introduced to obtain maintenance benefits. Examples are:

Risk in developing new code: Cordy [54] reports that clones frequently occur in financial software due to frequent updates and enhancements of an existing system to support similar kinds of new functionality. New financial applications do not change that much from existing ones, especially within the same financial institution. The developer is often asked to reuse the existing code by copying and adapting to the new product requirements because of the high risk of software errors when creating new code and to leverage the testing investment in existing code (70% of the software effort in the financial domain is spent on testing).
Clean and understandable software architecture: To keep the software architecture clean and understandable, sometimes software clones are intentionally introduced to the system [120].

Speed up maintenance: As two cloned code fragments are independent of each other both syntactically and semantically, they can evolve at different paces without affecting each other and testing is only required for a modified fragment. Keeping independent cloned fragments in the system may thus speed up maintenance, especially when automated regression tests are absent [181].

Ensuring robustness in life-critical systems: Redundancy is incorporated intentionally when designing life-critical systems. Often the same functionality is developed by different teams in order to reduce the probability that the implementations fail under the same circumstances.

High cost of function calls in real time programs: In real time programs, function calls may be deemed too costly. If the compiler does not offer to inline the code automatically, this will have to be done by hand, and consequently there will be clones.

2.3.3 Overcoming Underlying Limitations

Clones can be introduced due to underlying limitations of the programming languages or developers of a system.

a. Language Limitations Clones can be introduced due to the limitations of the language, especially when the language in question does not have sufficient abstraction mechanisms. Examples are:

Lack of a reuse mechanism in programming languages: Sometimes programming languages do not have sufficient abstraction mechanisms, such as inheritance, generic types...
(called templates in C++) or parameter passing (missing from, for example, assembly lan-
guage and COBOL) and consequently, developers must repeatedly implement these as
idioms. Such repeated activities can create many small and potentially frequent clones
[173, 23].

**Significant effort in writing reusable code:** It can be difficult and time consuming to
write reusable code. Perhaps it is easier to maintain two cloned fragments than to put in the
effort to produce a general and possibly more complicated solution.

**Writing Reusable code is error-prone:** Writing reusable code might be error-prone,
especially for a critical piece of code. It is therefore preferred to copy and reuse existing
code rather than make new reusable code. Introduction of new bugs can be avoided in
critical system functionality by keeping the critical piece of code untouched [87].

b. **Programmer Limitations** There are several limitations associated with programmers
that can introduce clones to the system. Examples are:

**Difficulty in understanding a large system:** It is generally difficult to understand a
large software system. This forces the developers to use the example-oriented programming
by adapting previously developed existing code.

**Time limitations on developers:** One of the major causes of code cloning in the system
is the time frame allowed to its developers. In many cases, the developers are assigned a
specific time limit to finish a certain project or part of it. Due to this time limit, developers
look for an easy way of solving the problems at hand and consequently look for similar
existing solutions. They just copy and paste the existing one and adapt it to their current
needs.

**Wrong method of measuring the developer’s productivity:** Sometimes the produc-
tivity of a developer is measured by the number of lines he/she produces per hour. In such
circumstances, the developer’s focus is to increase the number of lines of the system and hence tries to reuse the same code again and again by copying and pasting with adaptations instead of following a proper development strategy.

**Developer’s lack of knowledge in a problem domain:** Sometimes the developer is not familiar to the problem domain at hand and hence looks for existing solutions of similar problems. Once such a solution is found, the developer just adapts the existing solution to his/her needs. Because of the lack of knowledge, it is also difficult for the developer to make a new solution even after finding a similar existing solution and thus, reusing the existing one gets higher priority than making a new one.

**Lack of ownership of the code to be reused:** Code may be borrowed or shared from another subsystem which may not be modified because it may belong to a different department or even may not be modifiable (access not granted and/or stored in nonvolatile memory) [55]. In such situations, the only way of reusing the existing code is to copy and paste with required adaptations.

### 2.3.4 Cloning By Accident

Clones may be introduced by accident. Examples are:

**Protocols to interact with APIs and libraries:** The use of a particular API normally requires a series of function calls or other ordered sequences of commands. For example, when creating a button using the Java SWING API, a fixed series of commands is used to create the button, add it to a container, and assign the action listeners. Similar orderings are common with libraries as well [120]. Thus, the use of similar APIs or libraries may introduce clones.
Coincidentally implementing the same logic by different developers: It may happen that two developers were involved in implementing the same kind of logic and eventually, come up with similar procedures independently, thus leading to look-alikes more than clones.

Side effect of developers’ memories: Programmers may unintentionally repeat a common solution for similar problems using a solution pattern from his/her memory of similar problems. Clones may then unintentionally be created.

2.4 Do Clones Really Matter?

Given that there are huge percentages of clones in software systems (Section 2.2) and that the factors behind cloning (Section 2.3) are reasonable [121, 125], the question arises whether clones do really matter. In fact, while it may be beneficial to practise cloning, code clones can also have severe impacts on the quality, reusability and maintainability of a software system [110]. In the following we list some of the consequences of having cloned code in a system often stated in the literature as a reason to find and remove them. We will also cite empirical studies exploring the conjectural relations between clones and these effects. Finally, we will describe measures to remove existing clones and the associated benefits and risks.

Effect on system modification: Because of duplicated code in the system, one needs additional time and attention to understand the existing cloned implementation and concerns to be adapted, and therefore, it may become difficult to add new functionality in the system, or even to change existing ones [109, 162]. If a cloned code segment is found to contain a bug, all of its similar counterparts should be investigated for correcting the bug in question as there is no guarantee that this bug has been already eliminated from other
similar parts at the time of reusing or during maintenance. Moreover, when maintaining or
enhancing a piece of code, duplication multiplies the amount of work to be done [162, 169].

Geiger et al. [81, 80] have attempted to find a relation between code clones and change
couplings at the file level of granularity. Their study assumes that if a fragment is changed,
all of its cloned fragments (if any) should also be changed. Accordingly, if the cloned
fragments are in different files, all the associated files should be changed simultaneously
(change couplings). Change couplings have a bad impact on software evolution [81, 80]
as all associated files need a consistent change. In order to evaluate whether changed
couplings are the results of duplicated code, they used the release history analysis of the
Mozilla open source project and tried to correlate the number of clone instances to the
number of co-changes between the files containing the clone instances. There is a co-
change for two units \( a \) and \( b \) if changes to \( a \) and \( b \) are committed together. Geiger et al.
found many cases in which clones had to be updated frequently, although the study did not
reveal a statistically significant correlation between clones and change frequency [81].

Another preliminary study by Lozano et al. suggests that functions with cloned code
change more frequently than functions with non-cloned code [157]. For evaluating this re-
lation, a change-based experiment has been conducted by Lozano et al. [157]. Rather than
working on file level granularity as Geiger et al. [81] or disregarding parts of the system’s
history (considering only co-changes) as Kim et al. [125], they have focused on the method
level. They focus on those methods that had clones in the past for some time and then deter-
mine the frequencies of both changes and co-changes of such methods both when they have
clones and when they do not. Their study shows that although the frequency of co-changes
of methods with clones is lower than for methods without clones, the frequency of arbitrary
changes of methods with clones is higher than for methods without clones. Their results
seem to support the argument that although consistent changes between the cloned fragments are not carried out (developers may be unaware of other similar fragments), clones do have a negative impact on maintenance in terms of increased change frequency. However, their study was based on a small system, namely, DNSJava, developed by only two programmers, which limits the generality of their findings.

Another similar study was conducted by Aversano et al. [10]. They considered co-change analysis (Modification Transactions) extracted from source code repositories to verify how detected clones in a given release of a software system are affected by maintenance interventions, especially during software evolution activity or bug fixing. Based on the analysis from two Java software systems, ArgoUML and DNSJava, their study shows that most of the cloned code is consistently maintained, particularly while fixing bugs in cloned fragments. However, for divergent clones where clones evolve independently, consistent update was out of the question. Moreover, for maintenance activities (except bug fixing) developers tend to delay the propagation of maintenance over cloned fragments. A similar study was conducted by Krinke [142] with five open source Java, C/C++ systems. He also used the version histories of a target application and measured the percentage of consistent and inconsistent changes of the code clones over the different releases of a system. His study showed that roughly half of the changes to the clone classes are inconsistent changes. His study also showed that the occurrences of transforming inconsistently changed clone classes to consistently changed clone classes are very few. Inconsistent changes to clone classes are directly related to the maintenance problems (e.g., bug-fixing or update). A similar finding was noted by Kim et al. [125] that the number of consistent changes are fewer than anticipated in the evolution versions of a software system.

**Effect on Faults:** If a code segment contains a bug and that segment is reused by
coping and pasting without or with minor adaptations, the bug of the original segment may remain in all the pasted segments in the system and therefore, the probability of bug propagation may increase significantly in the system [107, 156]. Moreover, in many cases, only the structure of the duplicated fragment is reused with the developer’s responsibility of adapting the code to the current need. This process can be error prone and may introduce new bugs in the system.

Chou et al. [47] investigated the hypothesis that if a function, file, or directory has one error, it is more likely that is has others. They found in their analysis of the Linux and OpenBSD kernels that this phenomenon can be observed most often where programmer ignorance of interface or system rules combines with copy-and-paste. They explain the correlation of bugs and copy-and-paste primarily by programmer ignorance, but they also note that – in addition to ignorance – the prevalence of copy-and-paste error clustering among different device drivers and versions suggests that programmers believe that “working” code is correct code. They note that if the copied code is incorrect, or it is placed into a context it was not intended for, the assumption of goodness is violated.

Li et al. [156] use clone detection to find bugs when programmers copy code but rename identifiers in the pasted code inconsistently. On average, 13% of the clones flagged as copy-and-paste bugs by their technique turned out to be real errors for the systems Linux kernel, FreeBSD, Apache, and PostgreSQL. The false positive rate of their technique is 73% on average, where on average 14% of the potential problems are still under analysis by the developers of the analyzed systems.

**Effect on cognitive effort:** Duplication also increases the cognitive effort required by the maintenance engineers to understand a large software system. There are multiple occurrences of a cloned fragment in different places of the system and the maintenance
engineers are required to examine all the different instances in order to understand the difference between them [107].

**Effect on design:** Cloning may also introduce bad design, lack of good inheritance structure or abstraction. Consequently, it becomes difficult to reuse part of the implementation in future projects. It also badly impacts on the maintainability of the software [169].

**Effect on resource requirements:** Code duplication introduces higher growth rate of the system size. While system size may not be a big problem for some domains, others (e.g., telecommunication switch or compact devices) may require costly hardware upgrade with a software upgrade. Compilation times will increase if more code has to be translated which has a detrimental effect on the edit-compile-test cycle. The overall effect of cloning has been described by Johnson [109] as a form of software aging or “hardening of the arteries” where even small changes on the architectural level become very difficult to achieve in the actual code.

**Effect on refactoring:** In order to overcome such difficulties with clones, automatic refactoring [19, 26] or aiding developers with manual refactorings [94] of duplicated code has been proposed. However, there are some recent studies [126, 125, 121] that show that refactoring of duplicated code may not always be desirable for software maintenance. The disadvantages of cloning are sometimes less than the costs of abstraction, which include difficulty of creation and use [201], decreased program comprehensibility [198] and increased system size [19].

In an ethnographic study with researchers at IBM T. J. Watson Research Center, Kim et al. have observed that “skilled programmers often created and managed code clones with clear intent” and “refactorings may not always improve the software with respect to clones” [124]. Yet, whether these findings extend to industrial programmers is an open question.
In a large scale case study, Kapser and Godfrey have noticed that code cloning is often used consciously [120, 121]. In several case studies [116, 118, 115], Kapser and Godfrey observed eight patterns of cloning [120]. For each of the patterns, they studied both the advantages and disadvantages of these patterns of cloning to software development and maintenance. They conclude that not all the cloning patterns are harmful to software maintenance and some of the patterns are even beneficial to software development and maintenance. Therefore, before attempting any refactoring, concerns such as stability, code ownership, and design clarity need to be considered.

Cordy also reports that cloning is frequently used in large financial software for reusing existing designs or for separating dependencies in custom views of data that modules or applications may have [54]. Existing code is tested extensively and therefore, reusing such code prevents the introduction of new bugs to the system, and at the same time limits testing to a subset of the copied code that is modified. He also observes that fixing bugs in all of the similar segments may not be expected by the developers as this may break unknowingly dependent code. Therefore, changing or refactoring clones may not be advisable from a risk management point of view.

From the above discourses and studies, it seems that the argument “Cloning is harmful” is still an open issue and more studies are required to come to any final conclusion. However, from the discussion, it is obvious that while cloning is often intentional and can be useful in many ways, clones need to be detected to overcome the maintenance difficulties posed by such clones. Furthermore, a recent study [110] that worked with both large industrial and open source software concluded that inconsistent changes to code clones are frequent and can lead to severe unexpected behavior. On the other hand, research has mostly focused on the relation of cloning with faults and change coupling/frequency. Other
maintenance aspects such as the effects on program understanding, design, and resources have received too little attention. Experiments need to be conducted on the costs, benefits, and associated risks of refactoring clones. These experiments must take into account the different types of clones and their frequency.

2.5 Other Applications of Code Similarity Detection

In addition to the immediate applications of clone detection techniques to clone refactoring, there are several other domains in which clone detection techniques can be helpful. There are also other areas related to clone detection from which clone detection techniques themselves can potentially benefit. In this Section, we provide a list of applications and related areas of clone detection research.

**Detects library candidates:** Davey et al. [56] and Burd & Munro [40] have noticed that a code fragment that has been copied and reused multiple times in the system apparently proves its usability. As a result, this fragment can be incorporated in a library, to announce its reuse potential officially.

**Plagiarism Detection:** One of the closely related areas of clone detection is plagiarism detection [195, 176, 73, 160, 100, 159, 28]. In clone detection existing code is copied and then reused with or without modifications or adaptations for various reasons (c.f., Section 2.3). On the other hand, for plagiarism detection, copied code is disguised intentionally and therefore, it is more difficult to detect. Clone detection techniques can be used in the domain of plagiarism detection if extensive normalization is applied to the source code for comparison. Clone detection tools such as token-based *CCFinder* [113] and metrics-based *CLAN* [164] have been applied in detecting plagiarism. Unfortunately, to date it is not clear how good they are in doing so.
Clone detection techniques, on the other hand may benefit from plagiarism detection tools. In a tool comparison study, Burd and Bailey [41] evaluate three clone detection tools and two plagiarism detection tools, *JPlag* [176] and Moss [4]. From their study, it is found that plagiarism detection tools show more or less similar precision and recall compared to the clone detection tools even though these tools detect clones across files only. However, plagiarism detection tools are designed to measure the degree of similarity between a pair of programs in order to detect the degree of plagiarism between them and therefore, are not well suited to use directly for clone detection from a performance point of view. Clone detection tools work within the scope of intra and inter-file levels with various clone granularities. If the plagiarism detection tools are directly used to find code clones within a single program, they need to compare all possible pairs of code fragments. A system with *n* statements requires a total of $O(n^3)$ pairwise comparisons. This level of computational complexity is impractical for very large systems [156].

**Origin Analysis:** A closely related area of clone detection is origin analysis [202] in which two versions of a system are analyzed to derive a model of where, how and why structural changes have occurred. Clone detection techniques may help origin analysis research [83] and at the same time origin analysis techniques may assist clone detection research. As with plagiarism detection, the critical difference between the scopes of their detection approaches makes them infeasible to assist one another directly.

**Merging:** Another related research area to clone detection is the problem of merging [98] two existing similar systems to produce a new one. Similar to origin analysis where two different versions of a software system are analyzed, merging works with two different variants of similar systems [83]. To establish the relation between the pair of systems, clones from both the systems are compared and analyzed. The comparison is only between
the systems. Unlike clone detection, clone analysis within a system is irrelevant for merging. However, both merging and clone detection require robust comparison techniques and each of them can benefit from the other by sharing their comparison algorithms and analysis approaches.

**Software Evolution:** A very similar problem to origin analysis and merging is software evolution analysis. As with origin analysis, two or more different versions of a software systems are mapped to find a relation between them, with a view to observe their evolution behavior. By detecting clones in each of the versions and then mapping to similar clone classes such a relation can be established [202, 84, 205].

**Multi-version Program Analysis:** Clone detection techniques can be used for multi-version program analysis and vice versa. The fundamental requirement of multi-version program analysis is that the elements of one version of a program be mapped to the elements of the other version of that program. Clone detection techniques can be used for establishing such a matching relation, and at the same time, other multi-version matching techniques [127] could be adapted to clone detection.

**Bug Detection:** There is also a close relation between clone detection and software bug detection. Copy-pasted software bugs, especially, can be successfully detected by clone detection tools [156, 104, 99]. However, it is not yet clear how bug detection techniques can help clone detection research.

**Aspect Mining:** Clone detection can also be applied to aspect mining research [122] and vice versa. On a study by Bruntink et al. [36] clone detection techniques are evaluated for finding cross-cutting concerns. Their analysis shows that even though there is a fundamental difference between clone detection and aspect mining, one can get benefit from the other. Classical clone detection techniques are designed for finding similar code fragments
based on the program texts. On the other hand, a cross-cutting concern is scattered (or tangle
d) in different places and their implementation might not be textually similar. Rather,
cross-cutting concerns tend to preserve semantic similarity between the scattered or tangle
code fragments and therefore, a semantic clone detection technique might be more
effective in finding aspects than the classic clone detectors. However, it is not yet clear how
aspect mining techniques can assist clone detection research.

**Program Understanding:** Clone detection techniques may assist in understanding a
software system. As clones hold important domain knowledge, one may achieve an overall
understanding of the entire system by understanding the clones of a system. For example,
Johnson [107] visualizes the redundant substrings to ease the task of comprehending large
legacy systems. Program comprehension techniques, such as search-based techniques [91]
or concept analysis [175] may greatly help clone detection research. However, empirical
evidence on the effectiveness of such approaches to clone detection is required.

**Code compaction:** Clone detection techniques can assist with fitting code into compact
devices (e.g., mobile devices) by reducing source code size [45, 58].

**Malicious software detection:** The possibility of detecting malicious software with
code detection techniques has also been investigated. By comparing one malicious soft-
ware family to another, it is possible to find the evidence where parts of one software system
match parts of another [211]. Detecting self-mutating malware (a particular form of code
obfuscation) with clone detection techniques has also been attempted by Bruschi et al. [37].

**Copyright infringement:** The problem of detecting source code copyright infringemen
t is viewed as a code similarity measuring problem between software systems. Clone
detection tools can, therefore, be applied or can easily be adapted to detecting copyright
infringement [15, 113].
Product Line Analysis: Clone detection techniques may be used in the area of software product line analysis (PLA) [44] by reengineering existing systems. Software product lines (SPL) is a well known engineering technique with which product families are developed easily from existing reusable software assets using a common means of production. Clone detection techniques can be used in crafting an SPL from existing systems [128] or managing a product line [27]. Koleilat and Shaft [128] use clone detection tools for product line analysis. After identifying a family of applications (for their case, a device driver family from the Linux Kernel was used), they determine whether the applications exhibit reusable assets. They use CCFinder to visualize the whole Linux driver subsystem and identify the families with large amount of cloning assuming that higher the cloning rate the more chances of the existence of reusable assets. They find that clones hold important domain knowledge and when a number of clones appear to be within a defined set of applications, it could mean that these applications belong to a certain family. After a further manual analysis with the aid of CLICS [119], they find several reusable objects and then use them for product line design of device drivers. This will save the developer a lot of time understanding the domain and creating a new driver. Baxter and Churchett [27] use clone detection techniques for enhancing product line development. Clones indicate the presence of a useful problem domain concept, and at the same time provide an example implementation. Differences between the cloned copies identify parameters or points of variation. Product line development is thus benefited with clone detection in removing redundant code, improving maintainability, identifying domain concept for use in the present or the next, and identifying parameterized reusable implementations. However, both these studies are in the primary stage and unpublished. Further empirical studies are required to validate the above statements.
From the above discussion it is clear that code similarity analysis is an important research topic in software engineering in general, and clone detection techniques can help, either directly or by adapting the underlying techniques, to many of these related problems.

## 2.6 Clone Detection Process

Clone detection is a non-trivial problem. A clone detector must try to find pieces of code of high similarity in a system’s source text. The main problem is that it is not known beforehand which code fragments may be repeated. Thus the detector really should compare every possible fragment with every other possible fragment. Such a comparison is prohibitively expensive from a computational point of view and thus, several measures are used to reduce the domain of comparison before performing the actual comparisons. Even after identifying potentially cloned fragments, further analysis and tool support may be required to identify the actual clones. Nevertheless, over the decade many different clone detection approaches have been proposed because of the importance of clone detection. In this section, we provide an overall summary of the basic steps in a clone detection process. This generic overall picture allows us to compare and evaluate clone detection tools with respect to their underlying mechanisms for the individual steps and their level of support for these steps.

Figure 2.3 shows the set of steps that a typical clone detector may follow in general (although not necessarily). The generic process shown is a generalization unifying the steps of existing techniques, and thus not all techniques include all the steps. In the following subsections, we provide a short description of each of the phases.
CHAPTER 2. BACKGROUND AND RELATED WORK

Preprocessing

Remover uninteresting code, determine source and comparison units/granularities.

Match Detection

Transformed comparison units (and/or metrics calculated for those units) are compared to find similar source units in the transformed code.

Transformation

One or more extraction and/or transformation techniques are applied to the preprocessed code to obtain an intermediate representation of the code.

Formatting

Clone pair/class locations of the transformed code are mapped to the original code base by line numbers and file location.

Post-processing: Filtering

In this post-processing phase, clones are extracted from the source, visualized with tools and manually analyzed to filter out false positives.

Aggregation

In order to reduce the amount of data or for ease of analysis, clone pairs (if not already clone classes) are aggregated to form clone classes or families.

Filtered Clone Classes

Clone Pairs/Classes

Clones on Transformed Code

Match Detection

Transformed Code

Preprocessed Code

Preprocessing

Get the Original Code.

Code Base

Figure 2.3: A generic clone detection process
2.6.1 Preprocessing

At the beginning of any clone detection approach, the source code is partitioned and the domain of the comparison is determined. There are three main objectives in this phase:

**Remove uninteresting parts:** All the source code uninteresting to the comparison phase is filtered out in this phase. For example, partitioning is applied to embedded code to separate different languages (e.g., SQL embedded in Java code, or Assembler in C code). This is especially important if the tool is not language independent. Similarly, generated code (e.g., LEX- and YACC-generated code) and sections of source code that are likely to produce many false positives (such as table initialization) can be removed from the source code before proceeding to the next phase [181].

**Determine source units:** After removing the uninteresting code, the remaining source code is partitioned into a set of disjoint fragments called source units. These units are the largest source fragments that may be involved in direct clone relations with each other. Source units can be at any level of granularity, for example, files, classes, functions/methods, begin-end blocks, statements, or sequences of source lines.

**Determine comparison units / granularity:** Source units may need to be further partitioned into smaller units depending on the comparison technique used by the tool. For example, source units may be subdivided into lines or even tokens for comparison. Comparison units can also be derived from the syntactic structure of the source unit. For example, an *if-statement* can be further partitioned into *conditional expression*, *then* and *else* blocks. The order of comparison units within their corresponding source unit may or may not be important, depending on the comparison technique. Source units may themselves be used as comparison units. For example, in a metrics-based tool, metrics values can be computed from source units of any granularity and therefore, subdivision of source units is
not required in such approaches.

2.6.2 Transformation

Once the units of comparison are determined, if the comparison technique is other than textual, the source code of the comparison units is transformed to an appropriate intermediate representation for comparison. This transformation of the source code into an intermediate representation is often called extraction in the reverse engineering community.

Some tools support additional normalizing transformations following extraction in order to detect superficially different clones. These normalizations can vary from very simple normalizations, such as removal of whitespace and comments [14], to complex normalizations, involving source code transformations [185]. Such normalizations may be done either before or after extraction of the intermediate representation.

a. Extraction

Extraction transforms source code to the form suitable as input to the actual comparison algorithm. Depending on the tool, it typically involves one or more of the following steps.

Tokenization: In case of token-based approaches, each line of the source is divided into tokens according to the lexical rules of the programming language of interest. The tokens of lines or files then form the token sequences to be compared. All whitespace (including line breaks and tabs) and comments between tokens are removed from the token sequences. *CCFinder* [113] and *Dup* [14] are the leading tools that use this kind of tokenization.

Parsing: In case of syntactic approaches, the entire source code base is parsed to build a parse tree or (possibly annotated) abstract syntax tree (AST). The source units to be compared are then represented as subtrees of the parse tree or the AST, and comparison
algorithms look for similar subtrees to mark as clones [26, 208, 214]. Metrics-based approaches may also use a parse tree representation to find clones based on metrics for subtrees [133, 162].

**Control and Data Flow Analysis:** Semantics-aware approaches generate program dependence graphs (PDGs) from the source code. The nodes of a PDG represent the statements and conditions of a program, while edges represent control and data dependencies. Source units to be compared are represented as subgraphs of these PDGs. The techniques then look for isomorphic subgraphs to find clones [129, 143]. Some metrics-based approaches use PDG subgraphs to calculate data and control flow metrics [133, 162].

**b. Normalization**

Normalization is an optional step intended to eliminate superficial differences such as differences in whitespace, commenting, formatting or identifier names.

**Removal of whitespace:** Almost all approaches disregard whitespace, although line-based approaches retain line breaks. Some metrics-based approaches however use formatting and layout as part of their comparison. Davey et al. [56] use the indentation pattern of pretty printed source text as one of the features of their attribute vectors, and Mayrand et al. [162] use layout metrics such as the number of non-blank lines.

**Removal of comments:** Most approaches remove and ignore comments in the actual comparison. However, Marcus and Maletic [161] explicitly use comments as part of their concept similarity method, and Mayrand et al. [162] use the number of comments as one of their metrics.

**Normalizing identifiers:** Most approaches apply an identifier normalization before comparison in order to identify parametric Type 2 clones. In general, all identifiers in the source code are replaced by the same single identifier in such normalizations. However,
Baker [14] uses an order-sensitive indexing scheme to normalize for detection of consistently renamed Type 2 clones.

Pretty-printing of source code: Pretty printing is a simple way of reorganizing the source code to a standard form that removes differences in layout and spacing. Pretty printing is normally used in text-based clone detection approaches to find clones that differ only in spacing and layout. Cordy et al. [52] use an island grammar [170] to generate a separate pretty-printed text file for each potentially cloned source unit.

Structural transformations: Other transformations may be applied that actually change the structure of the code, so that minor variations of the same syntactic form may be treated as similar [113, 171, 185]. For instance, Kamiya et al. [113] remove keywords such as static from C declarations.

2.6.3 Match Detection

The transformed code is then fed into a comparison algorithm where transformed comparison units are compared to each other to find matches. Often adjacent similar comparison units are joined to form larger units. For techniques/tools of fixed granularity (those with a predetermined clone unit, such as a function or block), all the comparison units that belong to the target granularity clone unit are aggregated. For free granularity techniques/tools (those with no predetermined target clone unit) aggregation is continued as long as the similarity of the aggregated sequence of comparison units is above a given threshold, yielding the longest possible similar sequences.

The output of match detection is a list of matches in the transformed code which is represented or aggregated to form a set of candidate clone pairs. Each clone pair is represented as the source coordinates of each of the matched fragments in the transformed code.
In addition to simple normalized text comparison, popular matching algorithms used in clone detection include suffix-trees [135, 163, 14, 113], dynamic pattern matching (DPM) [68, 133] and hash-value comparison [26, 162].

### 2.6.4 Formatting

In this phase, the clone pair list for the transformed code obtained by the comparison algorithm is converted to a corresponding clone pair list for the original code base. Source coordinates of each clone pair obtained in the comparison phase are mapped to their positions in the original source files.

### 2.6.5 Post-processing / Filtering

In this phase, clones are ranked or filtered using manual analysis or automated heuristics.

**Manual Analysis:** After extracting the original source code, clones are subjected to a manual analysis where false positive clones or spurious clones [137] are filtered out by a human expert. Visualization of the cloned source code in a suitable format (e.g., as an HTML web page [185]) can help speed up this manual filtering step.

**Automated Heuristics:** Often heuristics can be defined based on length, diversity, frequency, or other characteristics of clones in order to rank or filter out clone candidates automatically [113, 112].

### 2.6.6 Aggregation

While some tools directly identify clone classes, most return only clone pairs as the result. In order to reduce the amount of data, perform subsequent analyses or gather overview statistics, clones may be aggregated into clone classes.
2.7 Overview of Clone Detection Techniques and Tools

Considering the importance of detecting clones, many clone detection approaches have been proposed in the literature. Based on the level of analysis applied to the source code, the techniques can roughly be classified into four main categories: textual, lexical, syntactic, and semantic. In this section we summarize the state of the art in automated clone detection by introducing and clustering available clone-detection tools and techniques by category.

The techniques can be distinguished primarily by the type of information their analysis is based on and the kinds of analysis techniques that they use. Table 2.1 provides a high-level overview of the techniques and tools in the form of a taxonomy where the first column shows the underlying approach of the tools/techniques, the second column shows their one sentence description, the third column either shows the name of the corresponding tool or (if no tool name is found) the last name of the first author has been used as the tool name, and the fourth column shows the corresponding citation(s).

2.7.1 Textual Approaches

Textual approaches (or text-based techniques) use little or no transformation / normalization on the source code before the actual comparison, and in most cases raw source code is used directly in the clone detection process. Johnson pioneered text-based clone detection. His approach [107, 108] uses “fingerprints” on substrings of the source code. First, code fragments of a fixed number of lines (the window) are hashed. A sliding window technique in combination with an incremental hash function is used to identify sequences of lines having the same hash value as clones. To find clones of different lengths, the sliding window technique is applied repeatedly with various lengths. Manber [160] also uses fingerprints, based on subsequences marked by leading keywords, to identify similar files.
One of the newer text-based clone detection approaches is that of Ducasse et al. [68, 181]. The technique is based on dot plots. A dot plot – also known as a scatter plot – is a two-dimensional chart where both axes list source entities. In the case of the approach by Ducasse et al., comparison entities are the lines of a program. There is a dot at coordinate \((x, y)\) if \(x\) and \(y\) are equal. Two lines must have the same hash value to be considered equal. Dot plots can be used to visualize clone information, where clones can be identified as diagonals in dot plots. The detection of clones in dot plots can be automated, and Ducasse et al. use string-based dynamic pattern matching on dot plots to compare whole lines that have been normalized to ignore whitespace and comments. Diagonals with gaps indicate possible Type 3 clones, and a pattern matcher is run over the matrix to find diagonals with holes up to a certain size.

An extension of the Ducasse et al. approach is used by Wettel & Marinescu [212] to find near-miss clones using dot plots. Starting with lines having the same hash value, the algorithm chains together neighboring lines to identify certain kinds of Type 3 clones. SDD [149] is another similar approach that applies an n-neighbor approach in finding near-miss clones.

NICAD [185, 186] is also text-based, but exploits the benefits of tree-based structural analysis based on lightweight parsing to implement flexible pretty-printing, code normalization, source transformation and code filtering.

Marcus and Maletic [161] apply latent semantic indexing (LSI) to source text in order to find high level concept clones, such as abstract data types (ADTs), in the source code. This information retrieval approach limits its comparison to comments and identifiers, returning two code fragments as potential clones or a cluster of potential clones when there is a high level of similarity between their sets of identifiers and comments.
<table>
<thead>
<tr>
<th>One Sentence Description</th>
<th>Tool/1st Author</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Text-Based</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hashing of strings per line, then textual comparison</td>
<td>Johnson</td>
<td>[108, 107, 109]</td>
</tr>
<tr>
<td>Hashing of strings per line, then visual comparison using dotplots</td>
<td>Duploc</td>
<td>[68]</td>
</tr>
<tr>
<td>Finds similar files with approximate fingerprints</td>
<td>sif</td>
<td>[160]</td>
</tr>
<tr>
<td>Composes smaller isolated fragments of duplication with scatter-plot</td>
<td>DuDe</td>
<td>[212]</td>
</tr>
<tr>
<td>Data structure of an inverted index and an index with n-neighbor distance concept</td>
<td>SDD</td>
<td>[149]</td>
</tr>
<tr>
<td>Latent semantic indexing for identifiers and comments</td>
<td>Marcus</td>
<td>[161]</td>
</tr>
<tr>
<td>Syntactic pretty-printing, then textual comparison with thresholds</td>
<td>Basic NICAD</td>
<td>[186]</td>
</tr>
<tr>
<td>Syntactic pretty-printing with flexible code normalization and filtering, then textual comparison with thresholds</td>
<td>Full NICAD</td>
<td>[185]</td>
</tr>
<tr>
<td>Transformation to a middle format of atomic instructions and edit distance algorithm</td>
<td>Nasehi</td>
<td>[171]</td>
</tr>
<tr>
<td>Textual comparison with flexible options (e.g., ignore all identifiers)</td>
<td>Simian</td>
<td>[192]</td>
</tr>
<tr>
<td><strong>Token-Based</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suffix trees for tokens per line</td>
<td>Dup</td>
<td>[15, 13, 14]</td>
</tr>
<tr>
<td>Token normalizations, then suffix-tree based search</td>
<td>CCFinder(X)</td>
<td>[113, 112]</td>
</tr>
<tr>
<td>Distributed implementation of CCFinder for very large systems</td>
<td>D-CCFinder</td>
<td>[155]</td>
</tr>
<tr>
<td>Uses CCFinder’s non-gapped clones to find gapped clones in interactive and visual way using a gap-and-clone scatter plot</td>
<td>GeX/Gemini</td>
<td>[203, 112]</td>
</tr>
<tr>
<td>Flexible tokenization and suffix-array comparison</td>
<td>RTF</td>
<td>[22]</td>
</tr>
<tr>
<td>Data mining for frequent token sequences</td>
<td>CP-Miner</td>
<td>[156]</td>
</tr>
<tr>
<td>Real-time token comparison in IDEs with suffix-array</td>
<td>SHINOBI</td>
<td>[213]</td>
</tr>
<tr>
<td>Karp-Rabin string matching algorithm with frequency table of tokens</td>
<td>CPD</td>
<td>[174]</td>
</tr>
<tr>
<td>Normalized token comparison integrated with Visual Studio</td>
<td>Clone Detection</td>
<td>[50]</td>
</tr>
<tr>
<td>Normalized token comparison with suffix-tree</td>
<td>clones</td>
<td>[25, 137]</td>
</tr>
<tr>
<td><em>clones</em> is adapted to detect clones over multiple versions at a time</td>
<td>iClones</td>
<td>[82]</td>
</tr>
<tr>
<td><strong>Tree-Based</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hashing of syntax trees and tree comparison</td>
<td>CloneDr</td>
<td>[26]</td>
</tr>
<tr>
<td>Derivation of syntax patterns and pattern matching</td>
<td>Asta</td>
<td>[70]</td>
</tr>
<tr>
<td>Hashing of syntax trees and tree comparison</td>
<td>cdiff</td>
<td>[214]</td>
</tr>
<tr>
<td>Serialization of syntax trees and suffix-tree detection</td>
<td>cpdetector</td>
<td>[137, 71]</td>
</tr>
<tr>
<td>Metrics for syntax trees and metric vector comparison with hashing</td>
<td>Deckard</td>
<td>[105]</td>
</tr>
<tr>
<td>Suffix-tree comparison of AST-nodes</td>
<td>Tairas</td>
<td>[199]</td>
</tr>
<tr>
<td>XML representation of ASTs with frequent itemsets techniques of data mining</td>
<td>CloneDetection</td>
<td>[208]</td>
</tr>
<tr>
<td>XML representation of ASTs and anti-unification/code abstraction</td>
<td>CloneDigger</td>
<td>[38]</td>
</tr>
<tr>
<td>Token sequence of CodeDOM graphs with levenshtein distance</td>
<td>C2D2</td>
<td>[141]</td>
</tr>
<tr>
<td>Token-sequence of AST-nodes and lossless data compression algorithm</td>
<td>Juillerat</td>
<td>[111]</td>
</tr>
<tr>
<td>Subtree comparison obtained from ANTLR</td>
<td>SimScan</td>
<td>[193]</td>
</tr>
<tr>
<td>Like cpdetector but works on the nodes of parse-trees</td>
<td>clast</td>
<td>[25]</td>
</tr>
<tr>
<td>Like CloneDr with a different intermediate representation [136] of ASTs</td>
<td>ccdmil</td>
<td>[29, 25]</td>
</tr>
<tr>
<td>AST to FAMIX and then tree matching</td>
<td>Google</td>
<td>[194]</td>
</tr>
<tr>
<td><strong>Metrics-Based</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clustering feature vector of procedures with neural net</td>
<td>Davey</td>
<td>[56]</td>
</tr>
<tr>
<td>Comparing metrics for functions/begin-end blocks</td>
<td>[133, 162, 173, 134, 55, 165, 166, 7, 8]</td>
<td></td>
</tr>
<tr>
<td>Comparing metrics for web sites</td>
<td>[42, 63]</td>
<td></td>
</tr>
<tr>
<td><strong>Graph-Based</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approximative search for similar subgraphs in PDGs</td>
<td>Duplix [143], GPLAG [153]</td>
<td></td>
</tr>
<tr>
<td>Searching similar subgraphs in PDGs with slicing</td>
<td>Komondoor</td>
<td>[129]</td>
</tr>
<tr>
<td>Mapping PDG subgraphs to structured syntax and reuse Deckard</td>
<td>Gabel</td>
<td>[76]</td>
</tr>
</tbody>
</table>
2.7.2 Lexical Approaches

Lexical approaches (or token-based techniques) begin by transforming the source code into a sequence of lexical “tokens” using compiler-style lexical analysis. The sequence is then scanned for duplicated subsequences of tokens and the corresponding original code is returned as clones. Lexical approaches are generally more robust over minor code changes such as formatting, spacing, and renaming than textual techniques.

Efficient token-based clone detection was pioneered by Baker. In Baker’s tool Dup[15, 14], lines of source files are first divided into tokens by a lexical analyzer. Tokens are split into parameter tokens (identifiers and literals) and non-parameter tokens, with the non-parameter tokens of a line summarized using a hashing functor, and the parameter tokens are encoded using a position index for their occurrence in the line. This encoding abstracts away from concrete names and values of parameters, but not from their order, allowing for consistently parameter-substituted Type 2 clones to be found. All prefixes of the resulting sequence of symbols are then represented by a suffix tree, a tree where suffixes share the same set of edges if they have a common prefix. If two suffixes have a common prefix, obviously the prefix occurs more than once and can be considered a clone.

The technique allows one to detect Type 1 and Type 2 clones, and Type 3 clones can be found by concatenating Type 1 or Type 2 clones if they are lexically not farther than a user-defined threshold away from each other. These can be summarized using a dynamic-programming technique [16]. Kamiya et al. later extended this technique in CCFinder [113], using additional source normalizations to remove superficial differences such as changes in statement bracketing (e.g., \( \text{if}(a) \ b=2; \) vs. \( \text{if}(a) \ \{b=2;\} \)). CCFinder is itself used as the basis of other techniques, such as Gemini [203], which visualizes near-miss clones using scatter plots, and RTF [22], which uses a more memory-efficient suffix-array in place
of suffix trees and allows the user to tailor tokenization for better clone detection.

CP-Miner [156] is another state-of-the-art token-based technique, which uses frequent subsequence data mining to find similar sequences of tokenized statements. A token- and line-based technique has also been used by Cordy et al. [52, 198] to detect near-miss clones in HTML web pages. An island grammar is used to identify and extract all structural fragments of cloning interest, using pretty-printing to eliminate formatting and isolate differences between clones to as few lines as possible. Extracted fragments are then compared to each other line-by-line using the Unix \texttt{diff} algorithm to assess similarity.

Because syntax is not taken into account, clones found by token-based techniques may overlap different syntactic units. However, using either preprocessing [52, 79, 185] or post-processing [93], clones corresponding to syntactic blocks can be found if block delimiters are known or lightweight syntactic analysis such as island parsing [170] is added.

### 2.7.3 Syntactic Approaches

Syntactic approaches use a parser to convert source programs into parse trees or abstract syntax trees (ASTs) which can then be processed using either tree-matching or structural metrics to find clones.

**Tree-matching Approaches:** Tree-matching approaches (or \textit{tree-based techniques}) find clones by finding similar subtrees. Variable names, literal values and other leaves (tokens) in the source may be abstracted in the tree representation, allowing for more sophisticated detection of clones. One of the pioneering tree-matching clone detection techniques is Baxter et al.’s \textit{CloneDr} [26]. A compiler generator is used to generate a constructor for annotated parse trees. Subtrees are then hashed into buckets. Only within the same bucket, subtrees are compared to each other by a tolerant tree matching. The hashing is optional.
but reduces the number of necessary tree comparisons drastically.

This approach has been adapted by the AST-based clone detectors of Bauhaus [25] as ccdiml. The main differences from CloneDr are ccdiml’s explicit modeling of sequences, which eases the search for groups of subtrees that together form clones, and its exact matching of trees. Yang [214] has proposed a dynamic programming approach for handling syntactic differences in comparing similar subtrees. (cdiff is not a clone detection tool in itself but the underlying technique could be used in clone detection). Wahler et al. [208] find exact and parameterized clones at a more abstract level by converting the AST to XML and using a data mining technique to find clones. Structural abstraction, which allows for variation in arbitrary subtrees rather than just leaves (tokens), has been proposed by Evans et al. [70] for handling exact and near-miss clones with gaps.

To avoid the complexity of full subtree comparison, recent approaches use alternative tree representations. In the approach of Koschke et al. [137, 71], AST subtrees are serialized as AST node sequences for which a suffix tree is then constructed. This idea allows to find syntactic clones at the speed of token-based techniques. A function-level clone detection approach based on suffix trees has been proposed by Tairas and Gray based on Microsoft’s new Phoenix framework [199].

An approach for detecting similar trees has been presented by Jiang et al. [105] in their tool Deckard. In their approach, certain characteristic vectors are computed to approximate the structure of ASTs in a Euclidean space. Locality sensitive hashing (LSH) is then used to cluster similar vectors using the Euclidean distance metric (and thus can also be classified as a metrics-based techniques) and thus finds corresponding clones.

**Metrics-based Approaches:** Metrics-based techniques gather a number of metrics for code fragments and then compare metrics vectors rather than code or ASTs directly. One
popular technique involves *fingerprinting functions*, metrics calculated for syntactic units such as a class, function, method and statement that yield values that can be compared to find clones of these units. In most cases, the source code is first parsed to an AST or control flow graph (CFG) on which the metrics are then calculated. Mayrand et al. [162] use several metrics to identify functions with similar metrics values as code clones. Metrics are calculated from names, layout, expressions, and (simple) control flow of functions. A function clone is identified as a pair of whole function bodies with similar metrics values. Patenaude et al. use very similar method-level metrics to extend the Bell Canada Datrix tool to find Java clones [173].

Kontogiannis et al. [133] have proposed two different ways of detecting clones. One approach uses direct comparison of metrics values as a surrogate for similarity at the granularity of `begin − end` blocks. Five well known metrics that capture data and control flow properties are used. The second approach uses a dynamic programming (DP) technique to compare `begin − end` blocks on a statement-by-statement basis using minimum edit distance. The hypothesis is that pairs with a small edit distance are likely to be clones caused by cut-and-paste activities. A similar approach is applied by Balazinska et al. [20] in their tool SMC (similar methods classifier), using a hybrid approach that combines characterization metrics with dynamic matching.

Davey et al. [56] detect exact, parameterized, and near-miss clones by first computing certain features of code blocks and then training neural networks to find similar blocks based on the features. Metrics-based approaches have also been applied to finding duplicate web pages and clones in web documents [42, 63].
2.7.4 Semantic Approaches

Semantics-aware approaches have also been proposed, using static program analysis to provide more precise information than simply syntactic similarity.

In some approaches, the program is represented as a program dependency graph (PDG). The nodes of this graph represent expressions and statements, while the edges represent control and data dependencies. This representation abstracts from the lexical order in which expressions and statements occur to the extent that they are semantically independent. The search for clones is then turned into the problem of finding isomorphic subgraphs (for which only approximate efficient algorithms exist) [129, 143, 153]. One of the leading PDG-based clone detection tools is proposed by Komondoor and Horwitz [129], which finds isomorphic PDG subgraphs using (backward) program slicing. Krinke [143] uses an iterative approach (k-length patch matching) for detecting maximally similar subgraphs in the PDG. Liu et al. [153] have developed a plagiarism detector based on PDGs. Another recent study by Gabel et al. [76] maps PDG subgraphs to related structured syntax and then finds clones using Deckard.

2.7.5 Hybrids

In addition to the above, there are also clone detection techniques that use a combination of syntactic and semantic characteristics. Leitao [150] provides a hybrid approach that combines syntactic techniques based on AST metrics and semantic techniques (using call graphs) in combination with specialized comparison functions.
2.8 Clone Presentation

Almost all the clone detection tools report clone information in the form of clone pairs and/or clone classes in a textual format where only the basic information about the clones such as the file name, begin and end line numbers of clones are provided. The returned clones also differ in several contexts such as types of clones, degree of similarity, granularity and size. Moreover, there are huge numbers of clones in large systems. For example, CCFinder found 13,062 clone pairs in Apache httpd [119]. Because of insufficient information on the returned clones, their various contexts and huge numbers, the visualization of clones becomes difficult. For the proper use of the detected clones, especially for clone management, the aid of a visualization tool is crucial. In the following we list some of the visualization approaches that have been proposed in the literature.

One of the popular approaches is the scatter plot (also known as dot plots) [48], which involves visualization of clones in the form of two-dimensional charts where all software units are listed on both axes [15, 68, 180, 204] (Figure 2.4). A dot is used if two software units are similar for providing the clone pair information as a diagonal line segment with different granularities of software units. Scatter-plots are useful to select and view clones, as well as zoom in on regions of the plot. However, the scalability issue limits its applicability to visualize clones of many software units. This limitation is overcome by providing an enhanced scatter plot by Higo et al. [92]. They show that enhanced scatter plot is also good in understanding the state of the clones over different versions of a software. Another significant benefit of enhanced scatter-plot over the classical scatter plots [15, 68, 180] is that uninteresting code clones are automatically filtered out before displaying the results. Moreover, the directory (package) separators are differently shown from the file ones. This variation in separators enables users to know the boundaries of directories,
to find out directories that contain many clones and directories that share many clones with other directories. However, the order of the listed software units is still a problem with this approach as this has an impact on the visual patterns. While there is a “natural” order for lines (i.e., lexical order) within a file, it is not clear how to order more coarse-grained units such as functions, files, and packages. Lexical order of their names is in most cases as arbitrary as random order.

Johnson [108] has applied Hasse diagrams (Figure 2.5) for visualizing the cloning relationships (textual similarity) between files. A Hasse diagram (named after a German mathematician) consists of nodes and edges, and is used to draw a partial order among sets as an acyclic graph. Directed arcs connect nodes that are related by the order relation and for which no other directed path exists. For each of the clones and its related cluster of files, the copied source text and the source files are shown as nodes and the relations between

![Figure 2.4: Scatter plots for three clones (from [139])](image-url)
clones are shown as edges. The height of a node in the graph is determined by its size, the large files or code segments are towards the bottom, similar segments of code towards the top.

Later on Johnson proposed to navigate the web of files and clone classes via hyperlinked web pages [106]. The hyperlink functionality of HTML enables users to jump freely between source files having clone relations with each other or fragments included in the same clone set. Although hyperlinks are very nice to navigate, there is no functionality to see the state of code clones over the system. This approach also lacks the overview and selection features to find one’s way in the mass of duplication data.

Lanza and Ducasse’s polymetric views [148] (Figure 2.6) have been successfully used in visualizing various aspects of clones in one view by Rieger et al. [180]. Polymetric views allow one to investigate the clones of a system at different levels of abstraction, thus, providing progressively more information about the cloning in the software. A polymetric view is based on the graph metaphor and representation where a node represents a software unit and an edge a cloning relation. Visually, additional information can be attached to the graph by the degrees of freedom for the position (X/Y in the two-dimensional space), color of nodes and edges, thickness of edges, breadth and width of nodes. Rieger et al. propose
a fixed set of metric combinations to be mapped onto graphical aspects to present the clone information from different perspective for different tasks.

Beyond polymetric views, Rieger et al. [180] propose a variation of tree maps to show the degree of cloning along with the system decomposition (Figure 2.7). Tree maps display information about entities with a hierarchical relationship in a fixed space (for instance, the whole system on one screen) where the leaves of the hierarchy contain a metric to be visualized. Each inner node aggregates the metric values of its descendants. Each node is represented through a piece of the available space. The space of a descendent node is completely contained in the space of its ancestor. There is no overlap in space for nodes that are not in an ancestor/descendant relation. This is how the hierarchy is presented. Essentially the hierarchy is projected onto the two dimensional space seen from the root of the hierarchy. In order to show the hierarchy clearly, the space of each node appears as rectangle where the direction of subdivision of nested nodes is alternated horizontally and
CHAPTER 2. BACKGROUND AND RELATED WORK

Figure 2.7: A system decomposition whose leaves are annotated with the number of cloned lines of code and its corresponding tree map (from [139])

vertically at each level. The space of each rectangle is proportional to the metric.

The tree map visualization was originally proposed by Shneiderman in the early 1990s to show space consumption of a hard disk with a hierarchical file system [196]. While space is used very efficiently, problems arise when the hierarchy is deeply nested.

Tairas et al. [200] have created an Eclipse plugin to present clone information. One of their visualizations is the clone visualiser view, a window showing the distribution of clone classes among files (Figure 2.8). A bar in this view represents a source file, a stripe within a bar a cloned code segment, and its colors the set of clone classes in which the segment is a member.

A clone comprehension tool, CLICS is developed by Kapser and Godfrey [119]. CLICS uses a taxonomy of clone types [118] to categorize clones and generate statistics about different types of clones in the system. CLICS is designed to display the structures in the source files and the system architecture with cloning information. Such visualization enables users to obtain cloning information that they are interested in. Moreover, CLICS provides query-based visualization of clones. Scatter plot visualization was not implemented
CHAPTER 2. BACKGROUND AND RELATED WORK

Figure 2.8: Clones visualiser view in Eclipse (from [200])

in CLICS because of its limited scalability.

Extending GUESS [2] Adar and Kim [3] provide a code clone exploration tool, Soft-GUESS. It consists of a code library and a number of mini-applications that supports the analysis of code-clones in the context of system dependencies, authorship information, package structures and other system features. SoftGUESS supports the visualization of code clones in a single version of a program as well as views of changing clone over multiple versions of the program.

Jiang et al. [103] extended the concept of coupling and cohesion to code cloning by visualizing the clone relations at the architecture level. Their framework is useful in generating data to investigate and manage cloning activities within and across subsystems. Jiang and Hassan [102] have also proposed a framework for understanding clone information in large software systems. They use a data mining technique framework that mines clone information from the clone candidates produced by CCFinder. First, a lightweight textual similarity is applied to filter out false positive clones. Second, various levels of system abstraction are used for further scaling down the filtered clone candidates. Finally, an interactive visualization is provided to present, explore and query the clone candidates as with the directory structure of a software system.
In addition to scatter-plots, Gemini [93], which uses the output of \textit{CCFinder}, also provides visualization through metrics graphs and file similarity tables. This allows one to browse the code clones either pair by pair or using clone classes. Aries [94] (which is also based on the output of \textit{CCFinder}), is a refactoring support environment for clones using metrics-based querying. Users can query for clones matching a variety of metrics and thresholds. While Aries provides the capability to refine the displayed clones using queries, it does not support data set refinement or views mapping clones to system architecture.

2.9 Clone Evolution

Whether clones are hindrances to software development and its maintenance may be also be investigated by looking of the evolution of clones in time. There are several studies that look at how clones evolve in different versions of a software system, that is, when they come into existence, what changes happen to them in their life-time, and whether they disappear eventually.

Analysis of clone evolution was first performed by Laguë et al. [145] for evaluating whether a clone detection tool could help when integrated in the development process. The clones of six versions of a large telecommunication system were analyzed to investigate how function clones evolve with the evolution of the system. Specially, they checked how many clones were added, modified and deleted in the next version compared to the previous one. They also checked how many clones were never modified during the entire evolution process. From their study, it is observed that with the evolution of the software, a significant number of clones are removed but the overall number of clones increases over time in the evolved system. However, they did not address how elements in a group of code clones change with respect to other elements in the group.
For monitoring and predicting the evolution of clones, Antoniol et al. [7] propose to model cloning using time series. By analyzing the clones of several versions of a medium scale software system, they derive a prediction model for cloning. The proposed model is validated with 27 subsequent versions of mSQL. They found that time series can predict the clone percentage of subsequent releases with an average error rate below 4%.

In a follow-up study, Antoniol et al. [8] analyzed the clones of different releases of the Linux Kernel and observed that cloning is largely limited to subsystems. While most of the clone classes are scattered within a subsystem, very few of them are scattered across subsystems. In the Linux kernel, many clones can be found in subsystems of device drivers and hardware architectures. A newly introduced hardware architecture or device driver is often derived from existing similar ones and therefore, shows a higher cloning rate. However, the overall number of clones over versions seems stable as some clones are also removed during the evolution process. Godfrey and Tu [86] show similar results and conclude that cloning is a common and steady practice in the Linux kernel.

Li et al. [156] studied clone evolution in Linux and FreeBSD while evaluating their tool, CP-Miner [156]. Their study shows that the cloning rate does increase steadily over time. For a period of 10 years, the cloning rate has increased from about 17% to about 22% for the Linux kernel (similar observations for FreeBSD). However, the increasing rate was limited to a few modules, *drivers & arch* in Linux and *sys* in FreeBSD, instead of the entire system. As a supporting argument to this phenomenon, they mentioned that Linux supports more and more similar device drivers during that period.

Kim et al. [125] apply a different strategy in studying clone evolution. Rather than the usual approaches of using the number of added, removed or modified clones in each version, they studied clone genealogies, the evolution of clone classes over different versions
of a software system. They studied for example, how each element of a group of clones is changed with respect to the other elements in the same group over the different releases of a system. Based on the findings from the different releases of two open source Java systems, they conclude that there are many volatile clones (cf. Section 2.1) in the system and an immediate refactoring of such short-lived clones is not required as they may diverge from each other with the evolution of the system. Moreover, there exist several long-lived clones (cf. Section 2.1) in the system which cannot be refactored because of the limitations of the underlying programming languages.

Concerning the consistency of changes in cloned code, there are currently two dissenting studies. A study by Aversano et al. [10] suggests for both bug fixing and for evolution purposes, most of the cloned code is consistently maintained during the same cochange or during cochanges close in time. A later study by Krinke [142] showed that clone groups are consistently changed in roughly only half of the time. The study also showed that later changes do not make up for missed changes in general, because inconsistently changed clone groups that become consistently changed clone groups later can be found rarely. Further studies are required to investigate these conflicting results.

2.10 Clone Management

One purpose of clone detection is to remove clones from the system by refactoring to improve the system’s overall quality. However, it may also be possible to avoid clones right from the beginning in the development process. Rather than removing them later, existing clones can be managed during the evolution of a system. In the following subsections we briefly discuss these issues.
2.10.1 Corrective Clone Management: Removal of Code Clones

With clone refactoring, we can decrease the complexity, reduce potential sources of errors emerging from duplicated code, and increase understandability of software systems.

The simplest way of refactoring clones is the refactoring Extract Method [72, 95, 204, 111, 130, 26, 132] that replaces the cloned code by a function call to a newly created function created from the shared code of the clone fragments. Type 1 and Type 2 clones, where clones are exact copies or differ only in identifiers are suitable for such simple functional abstraction. Fanta and Rajlich [72] remove function and class clones from industrial object-oriented by aid of automated restructuring tools. Higo et al.’s CCShaper [95] filters the output of CCFinder to find good candidates for the Extract Method and Pull Up Method refactoring patterns. Higo et al. [94] also use metrics calculated from clone information and architectural data, and remove clones with Extract Method and Pull Up Method refactorings. Komondoor and Horwitz developed a semantics preserving procedure extraction algorithm that works on PDG-based clones [131, 130]. Because of the semantic-based approach, clones with statement reordering can also be refactored with their tool. After detecting clones with an AST-based approach, Juillerat & Hirsbrunner [111] also use the Extract Method refactoring for Java language. In addition to consecutive statements, their method removes clones hidden in sub-expressions. Moreover, for object-oriented systems, Replace Conditional with Polymorphism refactoring may also be applied [207].

However, all the above refactorings impose some kind of restricted precondition. For example, the Extract Method refactoring is applicable only to clones consisting of consecutive statements. Although, in some cases, it is possible to reorder the statements [130], limitations of the programming language, such as Java’s inability to pass back multiple values, can make such a refactoring impossible. Similarly, for the refactoring Pull Up Method,
the shared code must be found as similar methods that have a common superclass.

For languages with preprocessors, it is possible to use macros to replace cloned code [26]. It is also possible to use conditional compilation if a preprocessor is available. However, new problems may be introduced if there is excessive use of macros and conditional compilation. Because the compiler is not able to check for certain common mistakes in presence of macros and because macros may obfuscate the source code, developers normally avoid introducing macros [69]. Alternative approaches of removing clones are therefore proposed at the design level. One may use design patterns to avoid clones by better design [19, 18]. CloRT [20, 19] finds clones using software metrics, and a dynamic pattern matching algorithm determines whether the Strategy or Template design pattern applies to these clones, factors out the common parts of the methods, and parameterizes the differences with respect to the design patterns. However, this approach is largely manual and requires human expertise.

Finding clones in web documents and resolving them using the traditional reuse techniques of dynamic web sites has been proposed by Synytskyy et al. [198]. They use a multi-pass approach to resolve clones incrementally, using several different resolution methods, resolving each clone encountered with the most appropriate resolution method available. However, their method works only with identical clones.

Although automatic support for clone refactoring has been proposed (e.g., [19]) and, sometimes, clones tend to be refactored during software evolution [8], refactoring of clones is a risky activity and potential source of faults. For this reason, developers are almost always reluctant in performing it [54]. As an alternative approach optimization of clone refactoring with constraints and conditions can applied [34].
2.10.2 Preventive Clone Management: Avoidance of Code Clones

Rather than detecting and removing clones after the fact, sometimes it would have been much better if they were never created in the first place. In order to avoid clones, we must understand the reasons for cloning. Until we sufficiently know and master the reasons, we should at least check for clones on a regular basis, possibly even on every version control system check-in or even whenever a programmer save an edited file in an integrated development environment. They may be rejected or at least they should be documented if they cannot be avoided.

Laguë et al. have explored two ways of how to use a clone detection tool in the development process [145] for avoiding clones. One way is preventive control where a new function is added to the system only after being confirmed that this new function is not a clone to any existing one or there are specific reasons of adding that function as a clone to the system. The other way is problem mining where any modification to a function must be consistently propagated to all of its similar functions in the system. Therefore, no clones are created unnecessarily, and the probability of update anomalies is reduced significantly. A very similar idea of “Code Clone Change Conflict Detection (C4D)” is described by Borkowski [33].

2.10.3 Compensative Clone Management

An alternative way of clone removal or avoidance is the management of clones in the software systems. Refactoring of some existing clones may not always be practical, feasible or cost-effective. However, it is possible to manage or track the existing clones either in individual version or evolving versions of a system. One of the first attempts towards this approach is simultaneous editing that simplifies the repetitive text editing tasks of a
CHAPTER 2. BACKGROUND AND RELATED WORK

System [167]. Regions to be linked (repetitive text records) are provided by the user through selection or by specifying a text pattern. If any edit is made to any of the linked records, the user can see equivalent edits applied simultaneously to all other records. A very similar approach is proposed by Toomin et al. [201] for managing duplicated code with an editor-based interaction technique. With their tool Codelink, clone regions are manually selected and linked together by a user. The tool can then provide enhanced visualization and editing facilities to the programmer by allowing him or her to understand or modify many clones as one.

Attempts have also been made to track the clones of the different versions of a software system. Duala-Ekoko and Robillard [66] have proposed a tool called CloneTracker for tracking clones in evolving software. For a particular software system, the output of a clone detection tool is used to produce an abstract model of the clone regions for different clone classes. This abstraction of clone regions is called, clone region descriptor (CRDs). CRDs describe clone regions within methods in a robust way that is independent from the exact text of the clone regions or its location in a file. Having the CRDs, their tool can automatically track clones as code evolves, notify developers of modifications to clone regions, and support simultaneous editing of clone regions. A similar attempt has been undertaken by Chiu et al. [46]. However, there are several fundamental differences between the two approaches. For example, instead of creating such CRDs they use the built-in support for accommodating line changes, file renaming etc.

The shape of the code structure may vary depending on the shape of an organization [49]. Thus, it may be beneficial to identify and understand the patterns of how a developing team deals with duplicated code. These patterns can help in better understanding both the project’s structure and its developing team, and thus, the cloning phenomenon [78]. Based
on this motivation, Balint et al. [21] correlate code clones with time of modification and with the developer that modified it for detecting the patterns of how developers copy. Based on these patterns they develop a visualization tool called *Clone Evolution View* to represent the evolution of code clones.

There is a recent work by Bakota et al. [17] that introduces metrics for defining the similarity mapping between the clones of the different versions of a system and based on the mapping defines the notion of dynamic clone smells. The approach is validated with 12 versions of the Mozilla Firefox web browser.

### 2.11 Summary

In this chapter, we have provided further motivation for this thesis, along with background material and related work. After defining the terminology of clones, we observed that significant amounts of code in software systems is cloned code. We then showed that there are actually many valid reasons for such a large percentage of clones in the software systems. We attempted to answer the question of whether such clones actually cause any problems in software development, maintenance and evolution. We concluded that while clones can be useful in the short term, they might be harmful and can lead to severe unexpected behaviour during software maintenance and evolution. Thus one needs to be aware of the clones in software systems. We reviewed some other applications of code similarity detection which also underline the importance of clone detection research. We then presented a generic clone detection process and an overview of available clone detection techniques in the form of a taxonomy. We reviewed the different ways of presenting clones to the users of clone detection tools. Finally, we reviewed studies of clone evolution and ways of managing clones.
Chapter 3

NICAD: A Next Generation Code Clone Detection Tool

This chapter introduces a new language-specific but lightweight clone detection approach, called NICAD. We introduce the theory and practice of the method and demonstrate its use in finding function clones in C, Java and C# code. Early evaluations indicate that the method is capable of finding exact and near-miss clones with high precision and recall.

First, we provide the motivation behind the work in Section 3.1, and then following a short introduction to TXL, agile parsing and island grammars in Section 3.2, we provide an overall summary of our approach in Section 3.3. In Section 3.4 we describe our flexible multi-granular method for extraction of potential clones using TXL. In Section 3.5 we discuss adapting pretty-printing to eliminate noise, standardize formatting and isolate variance to lines using agile parsing, and in Sections 3.6 and 3.7 we add TXL rules to allow for flexible code abstraction and filtering of potential clones respectively. In Section 3.8 we discuss optimizing the finding of near-miss clones using simple threshold-based text-line comparison of potential clones with dynamic clustering, and generating output in
Section 3.9. Section 3.10 reports our first empirical results in using the framework to detect known function clones in two C programs along with an introduction to other evaluations we have done so far. We provide a brief description of the time and space complexity issues of NICAD in Section 3.11. Finally, Section 3.12 discusses the relation of our work with previous techniques, and Section 3.13 concludes the chapter with our next steps.

3.1 Motivation

Considering that sometimes clones might be harmful [110] and that programmers intentionally do cloning [125], over the past decade several techniques and tools for detecting code clones have been proposed (Section 3.12). Recent comparative and evaluation studies of some of the state-of-the-art tools [32, 206, 188] show that lightweight text-based techniques can find clones with high accuracy and confidence, but the detected clones often do not correspond to appropriate syntactic units. Moreover, formatting differences between code fragments is a challenge to text-based approaches, and many exact and near-miss clones may not be detected with such tools. Parser-based syntactic (AST-based) techniques, on the other hand, find syntactically meaningful clones but tend to be more heavyweight, requiring a full parser and subtree comparison algorithm. Moreover, many of the existing tools depend on their associated IDEs and thus it is infeasible to use them as stand-alone tools. A recent study [185] also shows that an AST-based tool can sometimes not detect even exact function clones in a small-sized program. While other tools might detect these clones well, that particular instance at least hints at the implementation difficulty of AST-based tools. Finally, neither text-based nor parser-based techniques have been found to be effective in detecting near-miss clones (those with a few minor editing changes to the program) [32].
In this chapter, we propose a multi-pass approach which is parser-based and language-specific but lightweight, using simple text line rather than subtree comparison to achieve good time and space complexity. We exploit the benefits of TXL [53] to efficiently identify and extract potential syntactic clones with pretty-printing to eliminate formatting differences and noise. TXL’s agile parsing [57] allows us to flexibly select granularity, and to tune the pretty-printing of potential clones to introduce additional line breaks such that potential variances within statements and other structures can be accurately reflected using a simple text line comparison. TXL’s transformation rules allow us to add flexible code normalization and filtering of uninteresting or irrelevant sections in the potential clones, yielding accurate minimal differences that are easily traced back to original source using source coordinates.

Our approach is lightweight in the sense that, like other text-based techniques (e.g., Duploc [68]), we work directly on program source text. Although pretty-printing, code normalization and filtering all use TXL’s agile parsing and transformation rules, they can be done on a source file-by-file basis and are scalably independent of the program’s overall structure. The method can be applied to any language for which we have an approximate (pretty-printing) TXL grammar and (optionally) examples of the desired variances and code normalizations for the language. It is language-specific in that sense.

3.2 Background

Our approach is based on lightweight agile parsing techniques supported by the TXL source transformation system. TXL [53] is a special-purpose programming language designed to provide rule-based source transformation using functional specification and interpretation. TXL programs have two main parts: a context-free grammar that describes the syntactic
structure of inputs to be transformed, and a set of context-sensitive, example-like transformation rules organized in functional programming style.

TXL operates in three phases: parse, transform, and unpars. The parsing phase creates an internal representation of the input as a parse tree under control of a context-free grammar like the one for C if-then-else statement below. TXL grammars specify not only input forms for parsing, but also output pretty-printing for unparsing using the special markers [IN] (indent), [EX] (exdent) and [NL] (newline).

```plaintext
define if_statement
  'if { [expr] } [IN][NL]
  [statement] [EX]
  [opt else_statement]
end define

define else_statement
  'else [IN][NL]
  [statement] [EX]
end define
```

The transformation phase transforms the parse trees created by the parser under control of a set of example-like transformation rules that easily express normalization and abstraction for clone detection, for example to anonymize if conditions or normalize identifiers. Finally, the TXL unparsing phase unparses the transformed parse tree to text output with standard spacing and pretty-printing under control of the grammar, and ignoring all input formatting. Commenting and spacing in the input are ignored by default (although it can be preserved if desired).

TXL supports agile parsing [57], which allows nonterminal definitions to be modified by grammar overrides, allowing programs to easily specify different interpretations of syntax and different pretty-printing in different programs based on the same grammar. For
example, we override a program using the C grammar with the if-then-else definition above to modify the pretty-printing to eliminate indenting and newlines in the output:

```plaintext
redefine if_statement
  'if ( [expr] ) [statement] [opt else_statement]
end redefine

redefine else_statement
  'else [statement]
end redefine
```

Island grammars [61] are a grammar-based method for separating interesting parts of a program (features we are interested in) from uninteresting parts (other features, which need not be precisely parsed). In our context, island grammars provide a simple mechanism to identify the interesting elements to be compared as potential clones. Island grammars also provide robustness [170] by allowing us to use semi-parsing for the language (or dialect) of interest. Island grammars can be coded in TXL either directly in the language grammar or using grammar overrides to specify a dialect of the language in which the islands are embedded.

### 3.3 Proposed Approach

Figure 3.1 represents a conceptual diagram of our new clone detection process. We call our prototype NICAD, a loose acronym for Accurate Detection of Near-miss Intentional Clones. The major distinguishing characteristics of our method are the identification and extraction of the set of potential clones, the standard and flexible pretty-printing, normalization and filtering of the potential clone set, the clustering of potential clones to minimize comparison cost, and the reporting of results in terms of original source. The current version of NICAD is a significant improvement and adaptation of the original [185]. NICAD
works in three phases: Extraction, in which all potential clones (code fragments of the target kind) are identified, pretty-printed and extracted, Comparison, in which the potential clones are clustered and compared, and Reporting, in which the clone pairs and classes found in the comparison phase are related to original source and presented for human inspection. After the Extraction phase, the user can optionally apply flexible pretty-printing, code normalization/abstraction and filtering for further removal of the potential editing differences between the code fragments. In the following sections we provide a detailed description of each component of this new process.

3.4 Extraction of Potential Clones

Every clone extraction tool designates — sometimes implicitly — the notion of a “minimal clone”, the smallest piece of code that the tool considers to be worthwhile to examine on its own. This step is important for two reasons: it reduces the amount of work the clone detector has to do, and makes the results more usable and relevant. The amount of work is cut because the tool does not spend time looking for clones of program entities that are too small, and the results are improved because they are not polluted with information about the “cloning” of single tokens or (small groups of) statements.

The clone extractor is responsible for enforcing these minimal clone restrictions. Its
task is to extract potential structural clones from the source code for further study, and it is responsible for extracting features no smaller than the designated minimal clone.

The definition of a minimal clone can vary from language to language and application to application. In the case of C systems, researchers often choose individual structured blocks that are at least 6 lines of source code as minimal clones. However, one could choose any level of structural granularity. For example, we could choose only the whole functions of a system, or the structured statements, or begin-end blocks of a certain minimum size.

In our system, we exploit agile parsing and the TXL extract function to enumerate our potential clones. The extract function, denoted \([ \hat{} ]\) in TXL, automatically extracts a set of all embedded instances of one grammatical type (e.g., statement), given an instance of another (e.g., function definition). Using grammar overrides, we modify the grammar to capture our minimal clones in special nonterminals which can be extracted using a single invocation of the extract function. Each potential clone is extracted only once, but if the potential clone we are interested in is nested, the inner candidate is listed twice: once inside its parent and once on its own. All extracted potential clones are stored as text files, annotated with the original source file names and beginning and ending source line numbers of their origin.

### 3.4.1 If-def Resolution

Currently NICAD can efficiently handle C, Java, and C# systems. In our previous work we preserved C #ifdef preprocessor directives in the comparison, but because we are using a parser, this limits us to "well-behaved" preprocessor directives. In the current version, we generalize to all C programs by adapting the method of Antoniol et al. [8] to resolve #ifdef directives without losing the original source. The strategy is simple. For each #ifdef
static void wake_up_func(unsigned long data) {
    // ifdef DECLARE_WAITQUEUE
    wait_queue_head_t *sleep_wait_p =
        (wait_queue_head_t*)data;
    // else
    // struct wait_queue **sleep_wait_p =
    //     (struct wait_queue **)data;
    // endif
    wake_up(sleep_wait_p);
}

Figure 3.2: Example result of #if-def resolution

static int seq_pr(struct seq_file *s, const struct ip_conn *ctr) {
    return sprintf(s, "pkts=%d bytes=%d ",
                   ctr->pkts, ctr->bytes);
}

Figure 3.3: A raw original code fragment

directive, we keep the then part and comment out the else part and associated preprocessor lines. Figure 3.2 shows an example of the result of #if-def resolution. While resolving #if-def in this way is not an ideal solution, we can at least maintain uniformity in the copy/paste code fragments (see [8] for details), while making it possible to parse any C program.

static int seq_pr(struct seq_file *s, const struct ip_conn *ctr) {
    return sprintf(s, "pkts=%d bytes=%d ",
                   ctr->pkts, ctr->bytes);
}

Figure 3.4: Standard pretty-printed version of the code fragment of Figure 3.3
3.4.2 Standard Pretty-Printing

When potential clones are extracted to the XML file, they are also automatically stripped of formatting and comments and pretty-printed by TXL according to the language grammar’s formatting cues. Standard pretty-printing ensures consistent layout and spacing of code for later comparisons. When code is cloned, it is often changed — whitespace and comments are inserted or removed, block markers are moved around to suit the developer’s taste, and so on. Whitespace and comment removal can address these changes to some degree, but does not necessarily eliminate them. Standard pretty-printing guarantees that all code has uniform layout and line breaks, yielding an improvement in comparison accuracy. For example, the raw code fragment shown in Figure 3.3 (which is actually a code clone from the Linux kernel), we get the pretty-printed code fragment shown in Figure 3.4. Whatever formatting changes are made in the code block of Figure 3.3, we will get the same pretty-printed output as Figure 3.4, and thus NICAD can detect code clones that differ in formatting, layout and comments.

3.4.3 Splitting and Sorting

The XML file containing all the annotated potential clones is then split to separate files by assigning a unique ID (1, 2, 3, ...) to each potential clone. For each potential clone $N$, two files are generated, one (named $N.pc$) for the pretty-printed code and one (named $N.info$) for its corresponding original source coordinates (file name and line numbers). At this point, once all the potential clones have been extracted, flexible pretty-printing, code normalization and filtering can be applied to the potential clones if desired (see following sections). The potential clones are then sorted into descending order by size (number of pretty-printed lines) to assist in clustering and comparison.
CHAPTER 3. NICAD: CLONE DETECTION TOOL

3.5 Flexible Pretty-Printing

In addition to removing the formatting and layout differences between code segments using standard pretty-printing, we can also exploit TXL’s agile parsing to introduce flexible pretty-printing specific to clone detection. This special pretty-printing helps us to break different parts of a statement into several lines so that local changes to the parts of a statement can be isolated using a simple line-comparison. Unlike token-based techniques (e.g., CCFinder) where each token is an item of the token-sequence, we allow different parts of a statement to be compared at different granularities, appropriate to the particular language structure. An item to be compared in our method may contain one token, or several tokens, according to the particular pretty-printing rules we choose. Because we use a fast text-line-based technique for comparing potential clones to each other, pretty-printing to spread the code over more lines increases the granularity and allows us to choose the appropriate granularity for each particular language and context.

To see the effectiveness of such feature-specific pretty-printing, let us consider three code fragments, each only with only a single fragment (a for loop header).

Segment 1: for (i=0; i<10; i++)
Segment 2: for (i=1; i<10; i++)
Segment 3: for (j=2; j<100; j++)

With a typical line-based technique (e.g., Duploc [68]) with no normalization/transformation option selected, all the three segments are different. With a classical token-based technique (e.g., CCFinder [113]) with identifier normalizations, all the segments will be similar and returned as clones. With a classical AST-based technique (e.g., CloneDR [26]) where a hashing is used to ignore the leaves of the trees, these three segments will be still similar and will be returned as clones. In fact, using the general definition of clones where
only syntactic similarity is considered (e.g., the Type 2 clones of Bellon’s taxonomy [32]), these are indeed clones and form a clone class.

In this study, we are primarily interested in accurately finding copied fragments that have been edited after pasting but neither consistent renaming nor renaming of identifiers and literals are done (It is of course important and interesting to find such consistently renamed or renamed clones [156] and we have adapted NICAD to detect them as well). Thus, in our study, while segment 1 and segment 2 form a clone pair (only the initialization value of \( i \) changes), segment 3 probably does not form any clone pairs with segment 1 or segment 2, because there are several changes, including the variable name.

With a careful pretty-printing (and an appropriate similarity threshold) for the code segments, we can more clearly see that while segment 1 and segment 2 form a clone pair, segment 3 neither forms clone pair with segment 1 nor segment 2. Table 3.1 shows a comparison where our pretty-printing has been used to break all for headers into into four separate lines. Using a naive line-by-line text comparison of the segments and a reasonable similarity threshold, we can accurately determine the similarity between the segments. If we now consider a size-sensitive similarity threshold (see Section 3.8) of 70%, we can see that only segment 1 and segment 2 (with similarity 75%) form a clone pair. Both segment 1 and segment 3, and segment 2 and segment 3 are only 25% similar, and thus cannot form clone pairs. We could further break down the code segments such that each token is

Table 3.1: Pretty-printing and comparing

<table>
<thead>
<tr>
<th>Line No.</th>
<th>Segment1</th>
<th>Segment2</th>
<th>Segment3</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S1&amp;S2</td>
</tr>
<tr>
<td>1</td>
<td>for (</td>
<td>for (</td>
<td>for (</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>i = 0;</td>
<td>i = 1;</td>
<td>j = 2;</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>i &lt; 10;</td>
<td>i &lt; 10;</td>
<td>j &lt; 100;</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>i++;</td>
<td>i++;</td>
<td>j++;</td>
<td>1</td>
</tr>
<tr>
<td>Total Matches</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Total Mismatches</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
formatted to a separate line and then catch more changes in the code segments. However, in that case, we would have to reduce the similarity threshold, and at the same time, time complexity would increase.

In TXL specifying such precise flexible pretty-printing is simple. One needs only to override the grammar definition of the intended statement. For example, consider the following TXL grammar definition for for headers in C.

```txl
define for_head
  'for ([opt expr]; [opt expr]; [opt expr])
end define
```

Using grammar overrides we can redefine the grammar to break for headers into four parts, each on a different line (as in Table 3.1), simply by adding “[NL]” (new line) formatting cues to the grammar for each part as follows.

```txl
redefine for_head
  'for( [NL]
  [opt expr] '; [NL]
  [opt expr] '; [NL]
  [opt expr] ) [NL]
  [statement]
end redefine
```

Let us consider another example, a function call, say `foo(len, sum)`. A typical approach would either normalize this call as `id()`, or as `id(id, id)` and then apply an exact matching algorithm. In the first case, all function calls in the code are treated as the same and thus clone detection may generate many false positives. In the second case, all function calls that have two parameters will match, again generating many false matches and in the same time may miss some potential matches (e.g., overloaded function calls in C++ where only number of parameters is changed).

Using flexible pretty-printing, we can stay in the middle. For example, we may consider that for a function call, at least the function name or the number of parameters (and possibly
names of the parameters) should be same to have a match with other function calls. We can do this using the same pretty-printing technique, using TXL formatting to break function calls into two lines, the function name, \texttt{foo} in one line and its parameters \((\texttt{len}, \texttt{sum})\) on another. If there is another function call in another segment, say \texttt{foo}(\texttt{len}, \texttt{sum}, \texttt{product}) (function overloading), it will similarly be broken into two lines by the pretty-printer. Using a naive text-line-based comparison of the two pretty-printed segments, we can see that these two function calls are neither exactly similar nor completely dissimilar. They are in fact in the middle, neither similar (as in normalizing both to \texttt{id()}) nor dissimilar (as in normalizing the first call to \texttt{id(id, id)} and the second to \texttt{id(id, id, id)}). Using size-sensitive similarity, they are 50\% similar. In the same way, we can assign similarity values to each of the statements in the two code segments and more accurately assess the possibility that they are clones.

In the prototype implementation of our method, the user can either select from a set of flags controlling the breaking level for each of the statements on how the pretty-printing should be performed, or can change the TXL grammar overrides by hand to achieve other custom granularities. Although the examples above are shown on a statement by statement basis, in practice, a sequence of such statements of a code block or a function are pretty-printed and compared for detecting clones.

### 3.6 Flexible Code Normalization

While flexible pretty-printing helps us to find near-miss clones, it may require us to adjust the similarity threshold as well (e.g., the more fine-grained the break-up of a statement, the higher the dissimilarity threshold might be). In addition, as with all other methods, it may not be possible to predict in advance where possible changes might have been made.
in a cloned fragment. Using TXL transformation rules in addition to its parsing and pretty-printing capabilities, we can easily normalize parts of a statement (or whole statements of a given type) to ignore editing differences. By adding normalization to our pretty-printing, we can detect near-miss clones even with a 100% similarity threshold.

Unlike other classical token-based methods, our flexible normalization is not simply limited to global replacement, for example of all identifiers and literals, or simple abstraction, for example of loop bodies. Using TXL rules we can choose to normalize only certain parts of a statement or only statements of a certain type, for example, only if statements. Thus we can choose to normalize only those parts that we expect to vary. Using TXL patterns, we can also provide flexibility in applying the normalization, for example choosing only to normalize within a certain type of statement or within a certain level of nesting. In this way, we can be sure of the locations of potential changes in a detected cloned fragment when 100% similarity is used in the comparison phase.

We can also apply both flexible pretty-printing and flexible code normalization together in combination with a dissimilarity threshold (e.g., 90% similarity). This allows us to find near-miss clones that may have changed not only in the normalized places but also some other arbitrary parts that we could not anticipate in code normalization. To see the effectiveness of such flexible code normalization, let us consider the case of normalizing conditional statements in C.

TXL rules allow us to easily customize and localize normalizations by language feature and context. As an example, we can choose to custom normalize conditional statements. In general there are several conditional statements in a programming language, and using TXL rules we can target only certain ones, and normalize each of them in the most appropriate way. For example, in the case of the C if-then-else statement, we can choose just to
Table 3.2: Typical normalization of an if-then-else

<table>
<thead>
<tr>
<th>Original Statement</th>
<th>Typical Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ((x &lt; (n + y))) (m = (y + c) - x;) (\text{else} \ m = y;)</td>
<td>if ((id &lt; (id + id))) (id = (id + id) - id;) (\text{else} \ id = id;)</td>
</tr>
</tbody>
</table>

normalize the control part of the statement.

Consider the C if-then-else statement of Table 3.2 (left column). A typical token-based approach will apply global lexical normalization and produce something like the code shown in the right column of Table 3.2. As general normalization is applied to the entire source, such an approach can produce many false positives [32]. In our method, we apply neither general lexical normalization nor exact string matching. As with flexible pretty-printing, we stay in the middle. For a conditional statement, we can first focus on the control part, keeping the other parts unchanged (or left to other custom normalizations).

In our example, we could simply normalize the control part, \((x < (n+y))\) to AnyControl. Clones obtained using such a normalization will have the possibility of having only the control parts edited. However, it may also generate false positives as it normalizes the entire control part. So instead, we could apply the traditional normalization, but only on the control part. For example, \((x < (n + y))\) could be normalized to \((id < (id + id))\) and keep the other parts of the if-then-else statement unchanged. (In TXL, this can be implemented using simply the scoped application of a transformation subrule instead of a direct change.) As only the control part is normalized and other parts are unchanged, there will likely be fewer false positives than the general normalization. However, since all the identifiers of the control part are normalized, this could still return false positives.

So we can consider an even more restricted normalization. We can apply normalization
only on the right hand side of the control expression, for example to normalize \( (x < (n+y)) \) to \( (x < (id + id)) \), keeping the left part unchanged. This will allow identifier changes on the major part of the control, and at the same time avoid false positives as the left-part is unchanged. However, all the above normalizations can miss clones that have been structurally modified in the control part only (e.g., \( (x < (n + y)) \) could be changed to \( (x < (n*y))) \).

In the end we might refine to a more general normalization. For example, we can normalize \( (x < (n * y)) \) to \( (x < rightControl) \). While this will allow more structural changes, it will avoid false positives in two ways. First, its left part is unchanged and second, other parts of the if-then-else statement are either unchanged or normalized by other applicable options. Moreover, clones obtained with this normalization can indicate that possible changes are made on the right-part of the control. To make it more generalized, we can normalize the control to \( (leftControl < rightControl) \) or just simply \( AnyControl \) as mentioned earlier. In a similar way, we can custom normalize the control parts of other statements appropriately to their context. For example, the control part in the for-loop, while-loop or even in the assignment statement (e.g. \( x = (x > 0) : 1 : 0 \)) can be normalized either in the same way, or in different ways using other TXL rules. One has the option of normalizing the different types of controls to different IDs, or to a single ID. For example, the following TXL rule normalizes the control part of all the if-then-else statements to the specific ID AnyIfControl.

```latex
rule ifElseNormalization
    replace $ [statement]
        'if ( Expr1 [expr])
            ThenPart [statement]
            OptElsePart [opt else_statement]
    by
        'if ('AnyIfControl)
```
With this rule, the code segment of Table 3.2 (left-column) will be transformed to:

```plaintext
if (AnyIfControl)
    m = (y + c)-x;
else
    m = y;
```

Such a normalization of the control part allows arbitrary changes in the control conditions of the copy/pasted segments and our method can efficiently detect them as clones.

As in Basit et al. [22], we also allow for flexible tokenization. We provide an option for equating different token classes, for example to assign the same ID to different but similar data types such as `int, short, long, float, double` depending on user choice.

### 3.7 Flexible Code Filtering

Using agile parsing and source transformation we can also efficiently filter out code statements from potential clones according to user preferences. We can filter at any stage, either while extracting potential clones or separately following extraction. While extracting, we can use agile parsing and island grammars to filter out uninteresting statements from potential clones.

To filter out a certain type of statement after extracting potential clones, we can use TXL rules. Filtering out statements is simple and efficient in TXL, simply replacing the uninteresting statement by an empty one. For example, declaration and initialization statements are not major factors in affecting the logic of a code segment and thus could be ignored before comparison using a TXL rule to remove them. In a similar way, when searching for
function clones, the function name and parameters can often be ignored. A sample TXL rule for filtering out all the declaration statement could be as follows:

```txl
rule declarationFiltering
    replace [repeat declaration_and_statement]
        DeclarationPart [declaration]
        Rest [repeat declaration_and_statement]
    by
        Rest
end rule
```

### 3.8 Comparing the Potential Clones

Once the potential clones are extracted with preprocessing (with or without flexible pretty-printing, code normalization and filtering), they are fed to a comparison algorithm. In this paper, we have used a Longest Common Subsequence (LCS) algorithm for comparing the text lines of potential clones. The details of the algorithm are described elsewhere [96, 97], here we only provide an example of what it does and how it helps in finding clones. The LCS algorithm takes two sequences of items (each item is considered as string) as input and produces the longest sequence of items that is present in both sequences in the same order. For example, consider the following two sequences of items where each item/letter represents a string:

```
Sequence 1: a b c d f g h j q z
Sequence 2: a b c d e f g i j k r x y z
```

The LCS algorithm will produce a new sequence \( I < S \) which can be obtained from the first sequence by deleting some items, and from the second sequence by deleting some others. The condition is that \( \text{length}(I < S) \) should be as long as possible. For the above two sequences, we find \( I < S \) as \( a b c df gjz \). Once the sequence is determined, we use
the number of unique items for both sequences as a measure of similarity. In the following, we discuss how this algorithm helps us in finding clones. The algorithm is similar to the one employed by the Unix `diff` utility that we have used in our previous work, but has been reimplemented to be more efficient in our application.

To determine whether two potential clones really are clones of each other, we compare their pretty-printed and normalized sequences of text lines as items using LCS. Once we get the longest common subsequence of the two sequences, we determine the number of unique items in each potential clone. For an item to be declared “common” between the two potential clone sequences, it is not enough for it to occur in both sequences; it has to be an item that occurs in the longest common subsequence of the two sequences. Correspondingly, the items that have to be deleted from the sequences to generate the longest common subsequence are considered to be unique to their respective sequences. In fact, an item that occurs in both sequences might be considered as unique to both of them if it is not part of the longest common subsequence. We then compute the percentage of unique items for each potential clone (i.e., item-sequence) using the following equation:

$$\text{Unique Percentage of Items (UPI)} = \frac{\text{No. of Unique Items} \times 100}{\text{Total No. of Items}}$$

If these ratios for both line sequences are either zero or below a certain threshold, the sequences are considered to be clones of each other.

For example, consider the code segment in Figure 3.5. Imagine that it is copied and edited in two places (e.g., `i` is replaced with `i * i` for the assignment statements in the copied fragment), and assume that the UPI threshold (UPIT) is 30% for considering the two segments as clones.

Following flexible pretty-printing (in this case without code normalization or filtering) of both segments, the corresponding sequences of the fragments are shown in Table 3.3.
void sumTimes (int n) {
    float sum=0.0;
    double product =1.0; // C1
    for (int i=1; i<=n; i++) {
        sum=sum + i ; //C2
        product = product * i;
        fun(sum, product); //C3
    }
}

Figure 3.5: Example original code fragment

After finding the longest common subsequence, we calculate the UPI values for both the sequences. Since both sequences have UPI values (11%) below our assumed threshold (30%), they are considered to be clones.

Now consider another copy with several unique lines added, and some lines modified, including the function name and its parameters. From Table 3.4, we see that while the UPI value for the original segment (22.22%) is below the assumed UPI threshold (30%), the UPI value for the copied segment (50%) is above the assumed threshold and thus they are not considered to be clones.

From the above examples, we see that the UPI threshold is size-sensitive w.r.t. the number of items in the sequence (i.e., number of lines in the preprocessed potential clones, although one can choose in terms of original lines of code). For example, if UPI threshold is 30%, a potential clone of 10 lines can have a maximum of \((10 \times 30/100) = 3\) three unique lines compare to its counterpart potential clone. In addition to this size-sensitive threshold, one can also use other size-sensitive thresholds such as maximum gap size and maximum number of gaps in a sequence.

**Clustering and Overall Algorithm:** Because the LCS algorithm can only compare two potential clones at a time, in principle each potential clone needs to be compared with all of
CHAPTER 3. NICAD: CLONE DETECTION TOOL

Table 3.3: Two function clones

<table>
<thead>
<tr>
<th>Item No.</th>
<th>Sequence 1 (Original Segment)</th>
<th>Sequence 2 (Copied and Edited Segment)</th>
<th>Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>void</td>
<td>void</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>sumTimes</td>
<td>sumTimes</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>(int n) {</td>
<td>(int n) {</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>float sum=</td>
<td>float sum=</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0.0;</td>
<td>0.0;</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>double product =</td>
<td>double product =</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1.0;</td>
<td>1.0;</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>for (</td>
<td>for (</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>int i=1;</td>
<td>int i=1;</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>&lt; n;</td>
<td>&lt; n;</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>i++;</td>
<td>i++;</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>sum=</td>
<td>sum=</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>sum + i;</td>
<td>sum + (i * i);</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>product=</td>
<td>product=</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>product * i;</td>
<td>product * (i * i);</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>fun</td>
<td>fun</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>(sum, product);</td>
<td>(sum, product);</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>} }</td>
<td>} }</td>
<td>1</td>
</tr>
</tbody>
</table>

Total Items = 18
Unique Items = 2
UPI = 11.11%

Table 3.4: Two function non-clones

<table>
<thead>
<tr>
<th>Item No.</th>
<th>Sequence 1 (Original Segment)</th>
<th>Sequence 2 (Copied and Edited Segment)</th>
<th>Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>void</td>
<td>void</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>sumTimes</td>
<td>sumTimesExtended (int n, int m, int x, int y)</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>(int n) {</td>
<td>{ float sum=</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>float sum=</td>
<td>float sum=</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0.0;</td>
<td>0.0;</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>double product =</td>
<td>double product =</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1.0;</td>
<td>1.0;</td>
<td>1</td>
</tr>
<tr>
<td>8/18</td>
<td>for (</td>
<td>for (</td>
<td>1</td>
</tr>
<tr>
<td>9/19</td>
<td>int i=1;</td>
<td>int i=1;</td>
<td>1</td>
</tr>
<tr>
<td>10/20</td>
<td>&lt; n;</td>
<td>&lt; n;</td>
<td>1</td>
</tr>
<tr>
<td>11/21</td>
<td>i++;</td>
<td>i++;</td>
<td>1</td>
</tr>
<tr>
<td>12/22</td>
<td>sum=</td>
<td>sum=</td>
<td>1</td>
</tr>
<tr>
<td>13/23</td>
<td>sum + i;</td>
<td>sum + (i * i);</td>
<td>0</td>
</tr>
<tr>
<td>14/24</td>
<td>product=</td>
<td>product=</td>
<td>1</td>
</tr>
<tr>
<td>15/25</td>
<td>product * i;</td>
<td>product * (i * i);</td>
<td>0</td>
</tr>
<tr>
<td>16/26</td>
<td>fun</td>
<td>fun</td>
<td>1</td>
</tr>
<tr>
<td>17/27</td>
<td>(sum, product);</td>
<td>(sum, product);</td>
<td>1</td>
</tr>
<tr>
<td>18/28</td>
<td>} }</td>
<td>} }</td>
<td>1</td>
</tr>
</tbody>
</table>

Total Items = 18
Unique Items = 4
UPI = 22.22%

Total Items = 28
Unique Items = 14
UPI = 50%
the others, making the comparisons very expensive. We have used a number of strategies to reduce the number of comparisons, based on the UPI thresholds chosen by the user. If the UPI threshold is 0% (i.e., we are looking for exact matches only), then only potential clones of the exact same size (number of pretty-printed lines) are compared to each other. If UPI threshold is greater than 0%, then a potential clone \( x \) is compared to another potential clone \( y \) if and only if \( \text{size}(y) \) in lines is in the range between \( \text{size}(x) - \text{size}(x) \times UPI/100 \) and \( \text{size}(x) + \text{size}(x) \times UPI/100 \). In essence, this implements dynamic clustering based on the size (i.e., number of lines in the corresponding sequences) of the potential clones and the UPI threshold. In addition, potential clones that are below a certain threshold size (Minimum Clone Size) are either not extracted as potential clones or not compared.

**Generating Clone Classes:** Once all potential clones are compared and a correspondingly ordered clone pair database is formed. It is a simple step to generate clone classes from the database. If a potential clone \( P_i \), forms a clone pair with another potential clone \( P_j \), then all the other potential clones that form a clone pair with \( P_j \) are also included in the clone class and this continues recursively. As we maintain a unique ID for each potential clone and as the clone pair database is ordered by size, one can directly form all clone classes from the database without any computational bottleneck or post-processing. Note that when using this method with a UPI threshold greater than zero, the relation between clones is no longer strictly an equivalence relation since transitivity does not hold, and in the theoretical worst case all instances could be subsumed by the same class. Fortunately, in practice we have not observed this to be an issue.

**Clone Classes Using Exemplars:** As in our previous work [52], we also provide the option of applying an even more efficient method to reduce the number of comparisons and find clone classes directly. Although in theory this approach might seem a little arbitrary
and clearly can miss some clones, in practice we have found it to be very useful and misses very few. The method uses the first clone of each kind as an exemplar or distinguished representative of its clone class. When two potential clones $x$ and $y$ are compared and found to be clones, $y$ is marked as being in the class $x$. That is, $x$ is considered to be the exemplar for the pair. After this finding $y$ is removed from the comparison set and never compared to anything else. Instead, $x$ is compared to all other potential clones of the same cluster and any other matching potential clones are directly added to the class of $x$. For systems rich in clones, this optimization can reduce comparisons by a large factor. However, once again this is an approximation and in the theoretical worst case it might be possible that $x$ is an exemplar of both $y$ and $z$ even though $y$ and $z$ should not form a clone pair.

**Parallelizing Comparisons using Multiple Processors:** Depending on user hardware and the size of the subject system, NICAD can distribute the comparisons to multiple processors. Once the list of potential clones for an exemplar is determined and cached (if not already), NICAD automatically splits the comparison array according to the number of processors and sends them to different processors along with the exemplar. Results from all the worker processors are then gathered to get the complete clone class (if any) for that exemplar.

### 3.9 Reporting of Found Clones

The last phase of NICAD is the Reporting phase, which takes the database of clone classes discovered in the Comparison phase and provides output in either XML document format or as a visual interactive HTML website. As part of this phase, the original raw source code of each cloned fragment is recovered using the source coordinate information file for the
<clones systemname="weltab" minclonesize="3" withupi="10%" nfragments="45" nClasses="11">
  <class id="1" nfragments="2">
    <source file="weltab/src/cnv1a.c" startline="25" endline="381" withupi="0"/>
    <source file="weltab/src/cnv1.c" startline="33" endline="383" withupi="0.05"/>
  </class>
  ...
  <class id="4" nfragments="3">
    <source file="weltab/src/lansxx.c" startline="174" endline="365" withupi="0"/>
    <source file="weltab/src/lans.c" startline="176" endline="368" withupi="0.1"/>
    <source file="weltab/src/ejcn88.c" startline="182" endline="374" withupi="0.1"/>
  </class>
  ...
</clones>

Figure 3.6: Sample XML output from NICAD

corresponding potential clone. Thus all NICAD reports show original code rather than the pretty-printed version that was actually compared. This assists in human validation of the clone classes reported.

3.9.1 XML Output

Rather than traditional unstructured textual output, as in original NICAD, output is now provided as an XML document augmented with detailed information about the system and each class. Figure 3.6 shows a partial example of the XML output for NICAD’s clone analysis of the weltab [31] open source system.

The first line of the XML output shows that the subject system was weltab where the minimum clone size was set to 3 lines of pretty-printed code and that UPI threshold was set to 0.10 (10%). It also reports that 45 fragments/functions were detected as cloned fragments in 11 classes. The second line begins the first clone class and reports that it has two cloned fragments. The file names and source coordinates of the cloned source fragments are given in the class element. For each cloned fragment, the dissimilarity factor to the exemplar (the first element of the class) is reported as attribute withupi. For example, the second element of the first clone class has a withupi attribute of 0.05, indicating that the
code fragment differs from the exemplar in 5% of its pretty-printed lines. The first element of each reported class is the exemplar itself, and thus always has a dissimilarity factor of zero.

3.9.2 HTML Output

NICAD also provides results as an interactive HTML website for exploring the clone classes. It generates an HTML page showing the the original code of the first code segment as an exemplar for each clone class, and each of the clone classes is linked to a number of secondary clone report pages that shows the original code for the other members of the same class including the exemplar. For example, Figure 3.7 shows (a part) the leading code fragments (the exemplars) of the clone classes/pairs. If one wants to see all the members of a certain clone class, say for the first clone class in the figure, she has to click on the [All Clones in Class] for Report 1. Figure 3.8 shows the members of the first clone class/pair. As in our XML output, we also show the original file name, source coordinates (begin and end line numbers) and dissimilarity factors for each cloned fragment.

3.10 Evaluation of NICAD

We have evaluated NICAD in three phases: first, with an early evaluation study with two small to medium-size systems, second, with a mutation-based automatic framework and third, with large scale open source systems of different languages. In the following we provide a short introduction of these studies.
Figure 3.7: HTML output of NICAD: exemplar clones
Figure 3.8: HTML output of NICAD: a clone pair
3.10.1 First Study: Abyss and Weltab

In our early study, we evaluated our approach on two small to medium-size C programs with promising results. The first program was Abyss [1], a small web server written in approximately 1,500 lines of C code. The second program was Weltab [31], which is an election results program of approximately 11,000 lines. We have chosen these two systems as our first testbed for two reasons. First, in this early test we wanted to manually verify the clones found, which is obviously difficult or impossible with large systems. Second, there are already existing published results for function clones available for these systems [199] that we can compare with. Moreover, Weltab has already been used in a well known tool comparison experiment [32].

Although our method can be used for finding clones of any granularity (Section 3.4), in our first evaluation study we only considered clones of function granularity since there are already detailed published results available (with file and function names of all clones found). Tairas and Gray [199] report exact function clones in these systems allowing differences in function names and data types only.

For Abyss there are two previously reported clone classes/pairs (Class 1: ConfGetToken in conf.c and GetToken in http.c, and Class 2: ThreadRun in thread.c and ThreadStop in thread.c). Using our new method we accurately found both of these two classes and one additional valid near-miss clone class (ConfNextToken in conf.c and NextToken in http.c) not previously reported.

In case of Weltab we obtained eight exact match clone classes from 27 cloned fragments using standard pretty-printing whereas Tairas and Gray [199] obtain only five exact match clone classes (which they report as four groups since they allow function name changes).
The additional clone pairs/classes we obtained are (prtpag in ejcn88.c and lans.c), (whoen-
trer in poll.c and spol.c) and (shead in samp.c and sped.c). We confirmed our additional
findings by checking the results submitted to the Bellon et al. tool comparison experiment
[31, 32] by a metrics-based tool owner. This is interesting because in this case an AST-
based technique did not find even exact clones found by our method (although of course,
other AST-based techniques might accurately detect these clones). By relaxing data types,
they were able to detect three more classes that we also detected using a UPI threshold of
5%.

With Weltab we have also experimented with varying different options to assess their
effect in finding cloned fragments and clone classes. In Table 3.5, we provide the the num-
ber of cloned fragments and clone classes found from Weltab using our method depending
on the different options we used. The first column of the table shows the UPI threshold
we used, the second column shows the number of cloned fragments (CF) and clone classes
(CC) found when only standard pretty-printing option is used (std. PP), the third column
(Flex. PP) shows CF and CC when we used flexible pretty-printing for assignment, if and
for loop statements, the fourth column (Ctl. Norm) shows CF and CC when we normal-
ize the control parts of if, for loop and while loop statements, the fifth column (FunDef.
Norm) shows the CF and CC when we normalized/filtered function definition names and
the last column (ExpR. Norm) shows CF and CC when we have normalized the right part
of assignment statements.

<table>
<thead>
<tr>
<th>UPI</th>
<th>Std. PP CF</th>
<th>CC</th>
<th>Flex. PP CF</th>
<th>CC</th>
<th>Ctl. Norm. CF</th>
<th>CC</th>
<th>FunDef. Norm CF</th>
<th>CC</th>
<th>ExpR. Norm CF</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>27</td>
<td>8</td>
<td>27</td>
<td>8</td>
<td>27</td>
<td>7</td>
<td>27</td>
<td>7</td>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td>5%</td>
<td>42</td>
<td>12</td>
<td>44</td>
<td>11</td>
<td>44</td>
<td>12</td>
<td>42</td>
<td>11</td>
<td>42</td>
<td>12</td>
</tr>
<tr>
<td>10%</td>
<td>45</td>
<td>11</td>
<td>55</td>
<td>15</td>
<td>47</td>
<td>12</td>
<td>47</td>
<td>11</td>
<td>47</td>
<td>12</td>
</tr>
<tr>
<td>20%</td>
<td>59</td>
<td>16</td>
<td>66</td>
<td>19</td>
<td>59</td>
<td>16</td>
<td>63</td>
<td>18</td>
<td>64</td>
<td>18</td>
</tr>
<tr>
<td>30%</td>
<td>68</td>
<td>20</td>
<td>79</td>
<td>25</td>
<td>70</td>
<td>21</td>
<td>72</td>
<td>22</td>
<td>70</td>
<td>21</td>
</tr>
</tbody>
</table>
From the table (first row), we see that when we use 0% UPI threshold (i.e., looking for exact match), both standard pretty-printing and flexible pretty-printing return the same number of clone pairs and classes since for exact match clones flexible pretty-printing does not have any effect. However, when we normalize the control part of some statements, we get two more clone pairs and one more clone class. When we normalize the function definition names, we get the same number of clone pairs and one less clone class. This is because, due to the normalizing of function names, two different clone classes become one clone class. On the other hand, when we normalize the right-part of assignment statements, we get five more clone pairs and two more clone classes. When we increase the UPI threshold, we get more clone pairs and classes. Given the fact that we are only looking for function clones, not all the potential changes can be captured with code normalization or filtering, so we have tried using different UPI thresholds also with good results.

In these first tests we have not found any false positives using our method except when we use a very large UPI threshold. Although we have manually examined all the functions of the both systems to assess whether our method has missed any, we of course cannot be completely sure. To further validate the method, we decided to inject a substantial number of new clones of different types using editing scenarios [188] into the systems and attempted to find them using our method. Our method was able to effectively find all injected clones of the different types.

### 3.10.2 Mutation-/ Injection-based Automatic Framework

Although NICAD was placed as one of the best tools available in our scenario-based evaluation (Chapter 5) and showed high precision both in the above evaluation study and later on a large scale empirical study (Chapter 7), it was not possible to accurately measure its recall
with respect to large systems due to the lack of a valid benchmark and practical difficulties of oracling a large system. In order to overcome these difficulties and to measure precision and recall of NICAD automatically and with respect to different fine-grained types of clones, we have built a mutation- Injection-based framework and evaluated NICAD (Chapter 6). Case study results show that NICAD gives both high precision and recall for different fine-grained types of clones.

### 3.10.3 Large Scale Empirical Evaluation

Although NICAD was found to give high precision and recall both in the first evaluation study above and in the mutation-based framework (Chapter 6), its scalability was unknown for large systems and different languages. In order to check both the scalability and portability of NICAD and to study the cloning status of large systems in different dimensions, we have conducted a large scale empirical study (Chapter 7) with NICAD. Experimental results show that NICAD is capable of detecting clones from large systems (including the Linux kernel) of different languages and that there seem to be a large amount of copy/pasted function clones in those systems.

### 3.11 Time and Space Complexities

While it is hard to estimate the exact computational and space complexities of this multi-phase detection approach, we can provide an overall estimate. Resolving \textit{if-def} (for C systems only) and extraction of potential clones with standard pretty-printing rules is clearly linear in time and space over the total size of the system, requiring two linear passes for C systems (one \textit{if-def} resolution and one for extraction) and one linear pass for Java (for
CHAPTER 3. NICAD: CLONE DETECTION TOOL

extraction of potential clones). If one wishes to apply flexible pretty-printing, code normalization and filtering, there will be another pass on the extracted potential clones.

Thus the only real performance issue is the comparison of extracted potential clones. In the worst case, one can artificially create a scenario where every potential clone must be compared with every other, requiring quadratic time and linear space in the number of potential clones. In the best case, with the optimizations above, if full code normalization allows us to use direct text comparison of only equally sized potential clones then we can do the whole set of comparisons in linear time. In practice the usual case lies somewhere between, closer to linear than quadratic since the dynamic clusters tend to be very small.

When full flexible code normalization is used, we can use exact text line comparison between potential clone pairs and the time for pair comparison is linear in the size (w.r.t. number of lines) of the potential clones.

The LCS algorithm used in our prototype for individual pair comparisons has a quadratic worst case time and space complexity. However, in our method the potential clones to be compared are orders of magnitude smaller than the entire system, and the individual comparison time and space is effectively approximated by a small constant per individual comparison.

For example, when we had 125 potential clones to compare for Weltab using flexible pretty-printing of some statements (Section 3.10), there were 15,500 potential comparisons to make. However, using our method with a UPI threshold of 0%, it required only 139 comparisons. Further, when we increased the threshold to 10%, it still required only 295 comparisons, when the threshold was 20%, 464 comparisons, and when the threshold was 30%, 581 comparisons. For the small/medium-sized systems we studied with, no entire computation took more than several seconds. In Section 7.2.4 we have provided such
statistics for large systems of different languages which show that NICAD is capable of handling large systems.

### 3.12 Related Work

Most text-based approaches [68, 212] are related to our work in the sense that like them we also find clones by comparing program text. Although many of them can find near-miss clones, these approaches do not find syntactic clones and do not provide for accurate approximation in any way similar to our flexible pretty-printing, normalization and filtering.

Most lexical approaches [113, 22, 15, 156] (also called token-based approaches) are related to our work in the sense that like them we also use a similar sequence matching algorithm, and like them, we can also apply token transformation on the input. However, we do not use a generalized tokenization of identifiers and literals (e.g., normalizing all identifiers to a unique \( id \)) as they do. Moreover, special treatment or post-processing is required to find syntactic clones or gapped clones with such tools, which calls for the use of further “helper” tools such as \( CLICS \) [115] or extended implementation in \( Gemini \) [203]. We deal with these issues easily in our pre-processing and comparison phases.

Syntactical approaches [26, 105, 137, 199, 70] (also called tree-based approaches) are also related to ours in the sense that they are also parser-based and are aimed at syntactic clones. However, these are heavily dependent on fully-fledged parsers and find clones on ASTs or sequences of AST nodes (suffix trees) [137, 199], whereas we work on (pretty-printed) program text. Our method adds more flexible and restricted code normalization and filtering, and provides for the post-normalization inexact matching necessary to find many non-structural near-misses. \( Asta \) [70] can find near-miss clones based on structural abstraction much like ours, but using arbitrary pattern matching on ASTs.
3.13 Summary

In this chapter we have presented a new clone detection method based on a two stage approach: identification and normalization of potential clones using flexible pretty-printing and code normalization, followed by simple user-guided threshold-based text-line comparison of potential clones using dynamic clustering. Early evaluations demonstrate that this new method can do at least as well as existing methods in finding and classifying function clones in C.

However, validating the method with such small systems of only one language is not enough. Furthermore, we could not accurately measure the recall of our method for large systems and for different types of clones because of the challenges involved in measuring it. In order to examine whether NICAD can work with large systems and of different languages, we have conducted large scale empirical studies (Chapter 7) and found that NICAD is capable of accurately finding both exact and near-miss function clones even in large systems of different languages and that there are huge number of clones in those systems. NICAD is also placed as one of the best state-of-the-art tools in another study (Chapter 5) that provides a scenario-based qualitative comparison and evaluation of existing techniques and tools. In order to measure recall (also precision) accurately and automatically and for different types of fine-grained clone types (created from the editing taxonomy for clone creation in Chapter 4), we have developed a mutation / injection-based framework (Chapter 6). We have evaluated NICAD with this framework and found that NICAD is capable of detecting different types of fine-grained clones with high precision and recall. While comparing three different variants of NICAD, we noticed that full NICAD gives better results than the others (Chapter 6).
Chapter 4

Towards an Editing Taxonomy for Clone Creation

Clone detection is a research problem where there is no generally agreed-upon definition of suitable output. Extensive research on clone detection is done without knowing clearly what a clone is, and with no specific and universal task context. In almost all the cases, the definitions and types of the clones depend on the underlying algorithms and thresholds used by the particular tool. It is therefore crucial to know what exactly a clone means, and for what purpose before going to the details of detecting it.

In this chapter, we first show the definitional vagueness of clones from the literature in Section 4.1 and then propose an editing taxonomy in the form of a top-down theory of clone creation in Section 4.2. Finally, Section 4.3 summarizes the chapter outlining the limitations of this proposed editing taxonomy.


4.1 Definitions of Code Clone

A code fragment that has identical or similar code fragment(s) to it in the source code, in general, is termed as code clone. A copied fragment can be used with or without minor modifications in a system by the developer. If there are no modifications or the modifications are within a certain level in the copied fragment then the original and copied fragments are called code clones and they form a clone pair. However, there is no single or generic definition for code clone and all the proposed clone detection methods use their own definition for code clone [146]. In the following we provide clone definitions from the literature along with their associated vagueness.

Baxter et al. [26] define clone clones as the segments of code that are similar according to some definition of similarity. While they provide a threshold-based definition of tree similarity for near-miss clones, there is no specific definition of detection independent clone similarity. A vaguer definition is provided by Kamiya et al. [113] who define clones as the portions of source file(s) that are “identical” or “similar” to each other. While by the term “identical” they mean “exact copy clones”, there is no formal definition of the term “similar”. A similarly vague definition is proposed and used by Burd et al. [41] in their tool evaluation experiment where a code segment is termed as a clone if there is/are second or more other occurrences of that segment in the source code with or without “minor” modifications. It is not specified what is meant by “minor”. However, as in Baxter et al. [26], detection dependent threshold-based definitions of the terms “similar” or “minor” are attempted by several authors [134, 156, 118]. Attempts of automatically combining multiple detector result sets are also considered to overcome such similarity problems [32, 29, 168, 134]. These approaches may help in evaluating the tools in question, but still leaves open the question of how well the results match what human judges would decide.
In order to avoid such ambiguities related to the terms “similar” or “minor”, a categorization to clone definitions is attempted in the form of taxonomies. For instance, Mayrand et al. [162] provide an ordinal scale of eight distinct types of clones, of which some have simple, crisp definitions. For example, the category “DistinctName” refers to the clones where only identifier names may differ between the cloned segments. However, their ordinal scale does not provide a sound definition of clone. For instance, they define a category “SimilarExpression” to identify clones with expressions that differ but yet are still “similar”. Balazinska et al. [20] provide 18 different categories of clones based on what kind of syntax elements have been changed and also how much of the methods has been duplicated. While most of their categories are specific to a single change in the code, they still have vague categories “One long difference” means one unit token-sequence difference in an expression or in a statement or in another part of the function body, “Two long differences” means changes in two units and “Several long differences” means changes in three or more.

The issue of minimum clone size is also questionable. Some studies show that for a token-based technique, e.g., CCFinder, a threshold of 30 tokens is reasonable as the minimum clone size [113, 115, 125]. Other studies argue that measuring clone size with respect to the number of lines could be a better option. However, there is also disagreement on the minimum clone size with respect to number of lines. For instance, in Bellon’s tool comparison experiment [32, 30], the minimum clone size was set to 6 unprocessed lines of code. On the other hand, Baker [15] has used a minimum threshold of 15 non-commented lines, while Johnson [109] used 50 lines. Some studies consider the number of AST/PDG nodes as the measure of clone size and provide a measure of thresholds [156, 130]. Some studies work with only function clones and limit their clone size to the function body of
any size [162, 145].

Human judgement of code clones is also a major issue and varies among experts [209]. Walenstein et al. [209] experienced that for more than 60% of automatically detected clones, three experts disagreed whether the fragments are really code clones or not. The reason for the disagreement may partly be attributed to the specific given guidelines to find clones worth to be removed. Yet, the question remains whether there is an agreement by the involved parties on a task-independent definition of clone at all. A brief probe of many leading researchers in software clone management at the Dagstuhl seminar 06301 “Duplication, Redundancy, and Similarity in Software” 2007 revealed the discrepancies about the notions of a clone once again [114]. Less than half of the clone candidates that Kapser presented to these experts had 80% agreement amongst the judges. Judges appeared to differ primarily in their criteria for judgement rather than their interpretation of the clone candidates.

As we see from the above discussion, the definition of clone and its minimum size depend on the detection approach. The first attempt to define a detection independent definition of exact and near-miss clones was considered by Giesecke [77] where he pointed out the following desirable properties of a clone definition:

**Independent of a Programming Language:** Rather than finding code clones that are based on the text, syntax, and structure of a particular programming language, we want to find logic clones i.e., we want to detect duplication of logic, the essential property of a program. If such a generic modeling of code clones can be determined, the code clone detection problem will be independent of programming languages and we can then overcome most of the limitations of language-based approaches.

**Independent of a Detection Approach:** The detection of a particular type of clone should
be independent of the detection approaches. A developer can identify whether two code fragments form a clone or not. The detection approach should perform in such a way that it can replicate the detection capability of the human arbiter in algorithmic form.

**Describe a Continuum of Clones from Exact to Non-Exact:** Once a code fragment is copied, it can be used without being changed or there might be different levels of editing in order to fit the programmer’s need. As a result of extensive editing, two fragments may evolve to be completely different. But even for fragments where the common origin is almost unrecognizable, similarity knowledge is still valuable for a range of maintenance tasks.

### 4.2 Towards an Editing Taxonomy for Clone Creation

While each of the approaches outlined in the previous section above may help in evaluating the tools in question, they still leave open the question of how well the results might match what human judges would decide, and make it difficult to compare methods on an objective basis.

What we need is an evaluation system that can be used independently of detection method, and that demonstrably matches human judgement well. Given the problems with validating existing clones, we propose instead to use a theory of clone creation to mimic the actions of a development programmer in synthesizing new clones that can be used to objectively evaluate detection methods with no need for hand validation.

Intuitively, in most cases the “clones” we are looking for are those created as a result of copy/paste/modify actions by programmers. In our work we begin with this assumption,
# CHAPTER 4. TOWARDS AN EDITING TAXONOMY

## Arbitrary Renaming

- **Delete one or more lines**
- **Systematic renaming**
- **Expressions for parameters**
- **Reordering of statements**
- **AND control replacements**

## Further editing of whole lines

- **Small deletion within a line**
- **Small insertion within a line**
- **Comments and whitespace changes**
- **Formatting change**
- **Reuse by copy & paste**

## Insert one or more lines

- **Small deletion within a line**

## Reordering of declaration statements

- **Small deletion within a line**

## Figure 4.1: Example application of the editing taxonomy for cloning
and use it as the basis of a top-down theory of clones, which we have formalized into a
taxonomy of the editing actions that a programmer may undertake in the intentional cre-
ation of a clone [188]. Our taxonomy is derived from the large body of published work on
existing clone definitions [26, 76, 113, 129, 156], clone types [32, 134], clone taxonomies
[20, 118, 162], studies of developer copy/paste activities [124] and other empirical studies
[10, 21, 120, 125]. We have validated the taxonomy by studying the copy/paste patterns
of function clones [183] in an empirical study that analyzed 17 open source C and Java
systems including the entire Linux Kernel (6265 KLOC C, 154977 functions), Apache
httpd (275 KLOC C, 4301 functions) and j2sdk-swing (204 KLOC Java, 10971 methods)
[186, 187].

Figure 4.1 demonstrates the use of our proposed editing taxonomy for code fragments
at the function-level of granularity. The taxonomy is demonstrated on a simple example
original function (a) that calculates the sum and product of a loop variable and calls another
function with these values as parameters. Although the editing steps are demonstrated at
function-level granularity, they are general enough to be applicable to any granularity of
code fragment.

As an example of the creation of clones using the taxonomy, in Figure 4.1 we can
see that code fragment (b) is obtained by copying/pasting the original (a) and modifying
whitespace and comments. Similarly, code fragment (c) is obtained from (a) by modifying
code formatting. When comments are ignored and standard pretty-printing is used, all these
code fragments are exactly the same. Thus, clone fragment pairs (a, b) and (a, c) are exact
clone pairs and together form the clone class (a, b, c). Mapping to the literature, these
editing actions create Type 1 clones.

Moving to the next layer, code fragment (d) is obtained by consistently renaming the
identifiers (function names, variables, data types and literal values) of code fragment (c).
Similarly, code fragment (e) is also obtained from (c) with renaming, but not completely
consistently (in this case reordering the arguments of function \textit{foo}). If renaming is ignored,
(c, d) and (c, e) form clone pairs and the clone class (c, d, e). Again mapping to the
literature, these are \textit{Type 2} clones. If a detection approach handles both \textit{Type 1} and \textit{Type}
2 clones, then fragments (a) and (b) also form clone pairs with each of these, forming the
clone class (a, b, c, d, e). Code fragment (f) is created from (d) by replacing an identifier
with an expression — in this case the loop variable \textit{i} is replaced with \((i \ast i)\) inside the loop.
Clone detectors such as NICAD \cite{185} and Asta \cite{70}) that can abstract expressions, will
form the clone pair (d, f) for these, and with renaming, all of (a), (b), (c), (d) and (e) form
clone pairs with (f), yielding the clone class (a, b, c, d, e, f).

At the next editing layer, code fragment (g) is obtained by inserting another argument to
the function \textit{fun} (small insertion within a line), code fragment (h) is obtained by deleting an
argument of the function \textit{fun} (small deletion within a line), code fragment (i) is obtained by
adding a new line (insert one or more lines), (j) is obtained by deleting a line (delete one or
more lines), and code fragment (k) is obtained by modifying a line (extensive modification
of one or more lines). Mapping to the literature, these are all \textit{Type 3} clones of code fragment
(f). As usual, a robust method that does not differentiate clone types should detect all of
these fragments (including previous ones) as clone pairs, forming the clone class (a, b, c,
d, e, f, g, h, i, j, k).

While clones created by the editing activities so far are somewhat textually similar,
there are some editing activities for which the copy/pasted code fragment might not be
similar to the original on a text-line basis, but rather by functional behavior. For example,
in the next layer of the taxonomy, example fragment (n) is obtained from (i) by replacing
a *for-loop* with a *while-loop*. While textually different, these fragments still do the same thing. Similarly, when statements are reordered in the copied fragment, the result might be completely different on text-line basis but may still do the same thing, especially if the reordering happens between data-independent statements. For example, code fragment (l) is obtained by reordering only the declaration statements of (i). However, if reordering happens between data-dependent statements, they may not produce the same output (and may hint at a potential bug). For example, fragment (m) is obtained by moving the function call statement before the *sum* and *product* assignments, which will definitely produce different output. Researchers call such clones *semantic clones* [76, 129] or *Type 4* clones. As usual, an ideal detector should consider these fragments to be clone pairs with all the other example fragments, forming the clone class (a, b, c, d, e, f, g, h, i, j, k, l, m, n). Another kind of semantic clone, called an *intertwined clone* [76, 129], is also a *Type 4* member of the editing taxonomy.

### 4.3 Summary

In this chapter we have shown the existing vagueness in clone definitions and propose a top-down theory of clone creation in an attempt to provide a generic taxonomy of clone types. Based on this editing taxonomy we have then enhanced NICAD so that it can detect different fine-grained clone types created from this taxonomy (see Chapter 6 for the statistical results). This editing taxonomy has also been used to conduct a scenario-based qualitative comparison and evaluation of existing clone detection techniques and tools (Chapter 5). Furthermore, this taxonomy has been used as a vehicle for building the mutation operators for cloning in the mutation / injection-based automatic framework for tool evaluation (Chapter 6). A subset of this taxonomy has also been used in conducting a large scale
empirical study with open source systems of different languages (Chapter 7).

While we claim that the proposed taxonomy is a comprehensive one, we cannot guarantee that clones created by this editing taxonomy are representative for evaluation for any particular task, such as software maintenance by refactoring. What we can guarantee with some confidence is that to our knowledge there are no clone types in the literature that cannot be created using our editing taxonomy. At the same time, we acknowledge that our taxonomy still uses some vague terms and our definition of clone is still detection dependent to some extent. For full details on different clone types and taxonomies with concrete examples, the reader is referred to our technical report [182].
Chapter 5

A Scenario-based Qualitative Comparison and Evaluation of Clone Detection Techniques and Tools

Over the last decade many techniques and tools for software clone detection have been proposed. In this chapter, we provide a qualitative comparison and evaluation of the current state-of-the-art in clone detection techniques and tools, and organize the large amount of information into a coherent conceptual framework. Tempering the findings of this study one might choose the most appropriate clone detection tool or technique in the context of a particular set of goals and constraints or can have a clear idea to make a new hybrid clone detection method. In fact, the hybrid architecture of our tool NICAD is the result of this study. The primary contributions of this chapter are:

- a schema for classifying clone detection techniques and tools and a classification of current clone detectors based on this schema, and
• a taxonomy of editing scenarios that produce different clone types and a qualitative evaluation of current clone detectors based on this taxonomy.

After motivating the study in Section 5.1, we provide an overall comparison of the techniques and tools in terms several general criteria organized into facets in Section 5.2, which is based on the taxonomy of clone detection techniques and tools presented in Section 2.7 of Chapter 2. Section 5.3 introduces our taxonomy of hypothetical editing scenarios and presents our qualitative evaluation result, an analysis of the techniques and tools in terms of their estimated ability to detect clones created by each scenario. An example discussion on how the results of this study can be useful to a potential user or tool builder is presented in Section 5.4. Section 5.5 relates our work to that of others, and finally, Section 5.6 concludes this chapter. The reader is referred to some introductory background terms (Section 2.1 of Chapter 2), a generic clone detection process (Section 2.6 of Chapter 2) and an overview of the clone detection techniques and tools (Section 2.7 of Chapter 2) as a background study to this chapter.

5.1 Motivation

Over the last decade many techniques and tools for software clone detection have been proposed. The abundance of techniques and tools calls for quantitative comparison and evaluation, and there have been a number of evaluation studies to relate them. The most recent study, by Bellon et al. [32], provides a comprehensive quantitative evaluation of six clone detectors in detecting known observed clones in a number of open source software systems written in C and Java. Other studies have evaluated clone detection tools in other contexts [137, 41, 206, 207]. These studies have not only provided significant contributions
to the clone detection research, but have also exposed how challenging it is to compare different tools, due to the diverse nature of the detection techniques, the lack of standard similarity definitions, the absence of benchmarks, the diversity of target languages, and the sensitivity to tuning parameters [11]. To date no comparative evaluation has considered all of the different techniques available. Each study has chosen a number of state-of-the-art tools and compared them using precision, recall, computational complexity and memory use. There is also as yet no third party evaluation of the most recent tools, such as CP-Miner [156], Deckard [105], cpdetector [137], RTF [22], Asta [70] and NICAD [185].

In this chapter, we provide a comprehensive qualitative comparison and evaluation of all of the currently available clone detection techniques and tools in the context of a unified conceptual framework. Beginning with a basic introduction to clone detection background and terminology (Section 2.1 of Chapter 2), we organize the current techniques and tools into a taxonomy (Section 2.7 of Chapter 2) based on a generic clone detection process model (Section 2.6 of Chapter 2). We then classify, compare and evaluate the techniques and tools in two different dimensions.

First, we perform a classification and overall comparison with respect to a number of facets, each of which has a set of (possibly overlapping) attributes. Second, we define a taxonomy of editing scenarios designed to create Type 1, Type 2, Type 3, and Type 4 clones, which we use to qualitatively evaluate the techniques and tools we have previously classified. In particular, we estimate how well the various clone detection techniques may perform based on their published properties (either in the corresponding published papers or online documentation). In order to estimate maximal potential, we have assumed the most lenient settings of any tunable parameters of the techniques and tools. Thus, this is not an actual evaluation, rather it provides an overall picture of the potential of each
technique and tool in handling clones resulting from each of the scenarios. Our compari-
son is not intended to be a concrete experiment, and could not be comprehensive or truly
predictive and qualitative if it were cast as one, bound to target languages, platforms and
implementations. Finally, we provide two examples of how one might use the results of
this study to identify one or more appropriate clone detectors given a set of constraints and
goals or can have a clear idea to make a new hybrid clone detection method. In fact, the
hybrid architecture of our tool NICAD is the result of this study.

In contrast to previous studies, which concentrate on empirically evaluating tools, we
aim to identify the essential strengths and weaknesses of both individual tools and tech-
niques and alternative approaches in general. Our goal is to provide a complete catalogue
of available technology and its potential to recognize “real” clones, that is, those that could
be created by the editing operations typical of actual intentional code reuse.

To the best of our knowledge, this paper is the first study of the area, other than
Koschke’s recent overview [139, 140, 138], that provides a complete comparison of all
available clone detection techniques. For an even more complete in-depth overview of the
area, readers are referred to our recent technical report [182].

Our work particularly differs from previous surveys in our use of editing scenarios as
a basis for estimating the ability of techniques to detect intentional rather than observed
clones, in the evaluation of techniques for which no runnable tools as yet exist, in the
inclusion of a number of new techniques and tools that have not been previously reviewed,
and in the comparison of techniques independent of environment and target language. Our
goal is not only to provide the current comparative status of the tools and techniques, but
also to make an evaluation indicative of future potential (e.g., when one aims to develop a
new hybrid technique) rather than simply present implementation.
5.2 Comparison of Tools

Clone detection tools are multivariate, and therefore their study requires a systematic scheme for describing their properties. In this comparison, we will describe the properties of clone detection tools according to such a systematic classification. Our classification scheme is outlined first, and then we classify and compare the techniques and tools using it.

The properties are organized into facets, each of which may have different, but not necessarily disjoint attribute values. Related facets are grouped into categories. We first introduce the categories, facets, and attributes and then classify the tools and techniques in this scheme.

In order to provide a comparison of both general techniques and individual tools, we gather citations of the same category together using a category annotation, $T$ for text-based, $L$ for lexical (token-based), $S$ for syntactic (tree-based), $M$ for metrics-based and $G$ for graph (PDG)-based, with combinations for hybrids. While the citations for the different facets and attributes in Tables 5.1 to 5.10 may not be complete, we provide the values for all facets and attributes of the individual tools and techniques in Table 5.11.

5.2.1 Usage Facets

The category Usage groups facets relevant to the usage of a technique or tool. Table 5.1 lists the usage facets. The second column in the table gives the full name of the facet, and the first column gives the mnemonic abbreviation we use to refer to it. Unique identifiers for the facet’s attribute values are found in the third column. The last column gives short descriptions of the attribute values along with the citations of the corresponding techniques and tools.

**Platform:** This facet describes the execution platform for which the tool is available.
Table 5.1: Usage facets

<table>
<thead>
<tr>
<th>Abb.</th>
<th>Facet</th>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Platform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pa</td>
<td>The tool is platform independent</td>
<td>T [149], L [174], S [38]</td>
<td></td>
</tr>
<tr>
<td>Pb</td>
<td>The tool has been run on Linux/Unix</td>
<td>T [185, 186], L [15], S [105], M [173]</td>
<td></td>
</tr>
<tr>
<td>Pc</td>
<td>The tool has been run on Windows</td>
<td>L [22, 144, 50, 213], LS [141], S [199, 208], G [129, 153, 76]</td>
<td></td>
</tr>
<tr>
<td>Pd</td>
<td>The tool has been run on both Windows and Linux/Unix</td>
<td>T [192], L [113, 203], S [25, 137, 26, 193]</td>
<td></td>
</tr>
<tr>
<td>Pe</td>
<td>Others / Information not available</td>
<td>T [109, 68, 161, 160, 212, 171], L [156], S [70, 214, 111], M [133, 162, 56, 134], G [143]</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>External Dependencies</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Da</td>
<td>Possibly the tool has no external dependencies</td>
<td>T [186, 68, 192, 185, 109, 160, 212, 149], L [113, 22, 203, 112, 15, 213], S [105, 214]</td>
<td></td>
</tr>
<tr>
<td>Db</td>
<td>The tool seems to have external dependencies or to be a part of a larger tool set</td>
<td>T [161] (PROCSSI), T [171] (recoder), L [174] (PMD), L [156] (CloSpan), L [50] (ConQAT), S [26] (DMS [24]), S [25] (Bauhaus), [38] (CPython, ANTLR), S [199] (Microsoft Phoenix Framework), LS [141] (CodeDOM of .Net), S [208] (JAML), LS [137] (Bauhaus), S [70] (JavaML and Iscsc), S [193] (ANTLR), M [162, 173] (Datrrix), G [143] (VALSOFT), G [129, 153, 76] (CodeSurfer), G [143] (Krinke and Snelting validation framework)</td>
<td></td>
</tr>
<tr>
<td>Dc</td>
<td>Others / Information not available</td>
<td>S [111], M [133, 56, 134]</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Availability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A.a</td>
<td>The tool is open source</td>
<td>T [149], L [174, 50], S [38]</td>
<td></td>
</tr>
<tr>
<td>A.b</td>
<td>The tool is freely available for research in binary form</td>
<td>T [68, 192], L [113, 203, 112, 50], S [193]</td>
<td></td>
</tr>
<tr>
<td>A.c</td>
<td>The tool is commercially available</td>
<td>S [26]</td>
<td></td>
</tr>
<tr>
<td>A.d</td>
<td>There is a free evaluation license</td>
<td>S [26, 25]</td>
<td></td>
</tr>
<tr>
<td>A.e</td>
<td>Probably evaluation version is available on request</td>
<td>T [185, 186, 212], L [15, 22, 156], S [137, 105], G [129, 153, 76, 143]</td>
<td></td>
</tr>
<tr>
<td>A.f</td>
<td>Others / Information not available / Possibly not available</td>
<td>T [109, 160, 161, 171], L [213], S [70, 214, 199, 208, 141, 111], M [133, 133, 56, 173, 134]</td>
<td></td>
</tr>
</tbody>
</table>

**External Dependencies:** The External Dependencies facet states whether the tool requires a special environment or additional other tools to work.

**Availability:** This facet indicates the kind of license under which the tool is available.

### 5.2.2 Interaction Facets

The interaction category deals with how a user interacts with the clone detection tool (cf. Table 5.2), an important consideration when adopting a tool.

**User Interface:** This facet describes whether the tool supports interactivity or whether
Table 5.2: Interaction facets

<table>
<thead>
<tr>
<th>Abb.</th>
<th>Facet</th>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>User Interface</td>
<td>U.a</td>
<td>May be used as command line tool T [185, 212, 192], L [113], S [38]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U.b</td>
<td>Provides a graphical user interface T [149, 171], L [213] (Clone List and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>File Info Views), L [203, 50], S [70], M [56]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U.c</td>
<td>Both command line tool and graphical user interface (U.a) &amp; (U.b) L [112,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>113, 174, 25], S [26, 137, 193]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>U.d</td>
<td>Not precisely mentioned: See Table 5.11 for the remaining list</td>
</tr>
<tr>
<td>O</td>
<td>Nature of Output</td>
<td>O.a</td>
<td>Emits results textually providing only the source coordinates of the cloned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fragments (e.g., file name and begin-end line numbers of the cloned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fragments) T [160, 161, 192], L [22], S [105, 137], M [43, 133, 134]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O.b</td>
<td>Emits results graphically providing the original source of the cloned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fragments in a suitable format (e.g., HTML) or provides overall abstracted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>visual representation (e.g., dot-plot). T [48, 149, 171], L [52, 203, 213,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>50], S [214, 70, 38, 79, 199], M [42]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O.c</td>
<td>Both textual source coordinates of the cloned fragments and original</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>source in suitable format or abstracted visual representation ( both O.a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[26, 38, 193, 25, 29]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>O.d</td>
<td>Not precisely mentioned: See Table 5.11 for the remaining list</td>
</tr>
<tr>
<td>I</td>
<td>IDE Support</td>
<td>I.a</td>
<td>Is a Plug-in for Eclipse T [149, 65, 193, 99], S [38]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I.b</td>
<td>Integrated/Dependent in other IDE S [199] (MS Phoenix framework), L [213,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>144] (Visual Studio 2005), [193] (several IDEs)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I.c</td>
<td>Others: All other tools (Table 5.11) except listed here possibly have no</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IDE support</td>
</tr>
</tbody>
</table>

it is used in batch mode.

**Output:** The Output facet indicates the kind of output supported by the particular tool. Some tools provide cloning information textually with file name and begin-end line numbers of the cloned fragments, some provide the original source of the cloned fragments in a suitable format, some show the abstracted view of the cloned code (e.g., scatter-plot view) and some provide a combination of these.

**IDE Support:** The Plug-in Support facet indicates whether the tool is part of an integrated development environment (IDE). Only a few tools provide direct IDE support.
### Table 5.3: Language facets

<table>
<thead>
<tr>
<th>Abb.</th>
<th>Facet</th>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LP.a</td>
<td>Language Paradigm</td>
<td>LP.a</td>
<td>Applied to only procedural languages [109, 108, 160, 161, 185], S [214, 79, 199], M [43, 56, 133, 134, 162], G [129, 143]</td>
</tr>
<tr>
<td>LP.b</td>
<td>Language Paradigm</td>
<td>LP.b</td>
<td>Applied to only object-oriented languages [171], L [50], S [70, 38, 208, 141, 111, 193], M [173, 20]</td>
</tr>
<tr>
<td>LP.c</td>
<td>Language Paradigm</td>
<td>LP.c</td>
<td>Applied to both procedural and object-oriented languages [68, 149, 186, 212, 192], L [15, 113, 22, 112, 203, 156, 213, 174], S [26, 105, 25, 71], G [153, 76]</td>
</tr>
<tr>
<td>LP.d</td>
<td>Language Paradigm</td>
<td>LP.d</td>
<td>Applied to web languages [192], [63, 62, 60, 179, 52, 42, 177, 147, 123, 89]</td>
</tr>
<tr>
<td>LP.e</td>
<td>Language Paradigm</td>
<td>LP.e</td>
<td>Applied to only functional languages [151]</td>
</tr>
<tr>
<td>LP.f</td>
<td>Language Paradigm</td>
<td>LP.f</td>
<td>Applied to modeling languages L [154] (Sequence Diagram), G [59] (Simulink)</td>
</tr>
<tr>
<td>LP.g</td>
<td>Language Paradigm</td>
<td>LP.g</td>
<td>Applied to Lisp-like languages SMG [150] (hybrid)</td>
</tr>
<tr>
<td>LP.h</td>
<td>Language Paradigm</td>
<td>LP.h</td>
<td>Applied to assembler code [51, 64, 58, 75]</td>
</tr>
<tr>
<td>LP.i</td>
<td>Language Paradigm</td>
<td>LP.i</td>
<td>Applied to Java Byte Code [12]</td>
</tr>
<tr>
<td>LP.j</td>
<td>Language Paradigm</td>
<td>LP.j</td>
<td>Applicable across different languages LS [141] (C# and Visual Basic.NET)</td>
</tr>
<tr>
<td>LS.a</td>
<td>Language Support</td>
<td>LS.a</td>
<td>Is language independent [149], T [192] (several other language-specific lexical options too) L [50] (several other language-specific lexical options too)</td>
</tr>
<tr>
<td>LS.c</td>
<td>Language Support</td>
<td>LS.c</td>
<td>Experimented with “C++” [68], L [113, 112, 203, 156, 213, 174], S [25, 26, 208], G [153, 76]</td>
</tr>
<tr>
<td>LS.d</td>
<td>Language Support</td>
<td>LS.d</td>
<td>Experimented with “C#” [187] L [213, 50], L [113, 112, 203], S [70], LS [141]</td>
</tr>
<tr>
<td>LS.e</td>
<td>Language Support</td>
<td>LS.e</td>
<td>Experimented with “Java” [68, 212, 149, 186, 171], L [113, 15, 22, 174], S [26, 70, 105, 25, 38, 71, 208, 111, 193], M [173, 20], G [153]</td>
</tr>
<tr>
<td>LS.f</td>
<td>Language Support</td>
<td>LS.f</td>
<td>Experimented with “COBOL” [68], L [113, 112, 203], S [25, 26]</td>
</tr>
<tr>
<td>LS.g</td>
<td>Language Support</td>
<td>LS.g</td>
<td>Experimented with “Python” S [38]</td>
</tr>
<tr>
<td>LS.h</td>
<td>Language Support</td>
<td>LS.h</td>
<td>Experimented with “HTML” L [52, 198]</td>
</tr>
<tr>
<td>LS.i</td>
<td>Language Support</td>
<td>LS.i</td>
<td>Experimented with “Visual Basic” L [113, 112, 203, 213], S [141]</td>
</tr>
</tbody>
</table>
5.2.3 Language Facets

The language category deals with the programming languages that can be analyzed using the tool. Table 5.3 summarizes these facets and their attribute values.

**Language Paradigm:** The *Language Paradigm* facet indicates the programming paradigm targeted by the tool.

**Language Support:** Facet *Language Support* refines *Language Paradigm* to the set of particular languages.

5.2.4 Clone Information Facets

The clone information category gathers facets that characterize the kinds of clone information the tool is able to emit (cf. Table 5.4). The richer this information and more refined its structure, the more useful it is for further processing.

**Clone Relation:** The *Clone Relation* facet concerns how clones are reported— as clone pairs, clone classes, or both. Clone classes can be more useful than clone pairs, for example reducing the number of cases to be investigated for refactoring. Techniques that provide clone classes directly (e.g., *RTF* [22]) may therefore be better for maintenance than those that return only clone pairs (e.g., *Dup* [15]) or require post-processing to group clones into classes (e.g., *CCFinder* [113]).

**Clone Granularity:** The facet *Clone Granularity* indicates the granularity of the returned clones – *free* (i.e., no syntactic boundaries), *fixed* (i.e., within predefined syntactic boundaries such as method or block) or both. Both granularities have advantages and disadvantages. For example, techniques that return only function clones are useful for architectural refactoring, but may miss opportunities to introduce new methods for common statement sequences. A tool that handles multiple granularities may be more useful for
### Table 5.4: Clone facets

<table>
<thead>
<tr>
<th>Abb.</th>
<th>Facet</th>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R</strong></td>
<td>Clone Relation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.a</td>
<td>Yields clone pairs</td>
<td></td>
<td>T [68, 212, 149, 171], L [15, 113, 112, 174], S [26, 137, 25, 38, 70, 199, 141, 193], M [162, 42, 43, 133, 173, 134], G [129, 143, 153]</td>
</tr>
<tr>
<td>R.b</td>
<td>Yields clone classes</td>
<td></td>
<td>T [185, 186, 109, 108, 160, 161, 192], L [22, 52, 213, 156, 50], S [105, 208], M [20, 56], G [153, 59, 76]</td>
</tr>
<tr>
<td>R.c</td>
<td>Yields both clone pairs and clone classes directly by the comparison algorithm (note: None can directly find both clone pairs and clone classes.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R.d</td>
<td>Groups clone pairs in classes in post-processing</td>
<td></td>
<td>T [68], L [113, 112], S [26, 199, 193, 25], M [20], G [129]</td>
</tr>
<tr>
<td>R.e</td>
<td>Others</td>
<td></td>
<td>S [214, 111]</td>
</tr>
<tr>
<td><strong>G</strong></td>
<td>Clone Granularity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G.b</td>
<td>Fixed, Function/Method</td>
<td></td>
<td>T [186, 185], S [199, 141], M [162, 20, 56, 42, 43, 133, 134], G [129, 153, 76]</td>
</tr>
<tr>
<td>G.c</td>
<td>Fixed, begin-end block</td>
<td></td>
<td>T [186, 185], L [52], M [133]</td>
</tr>
<tr>
<td>G.d</td>
<td>Fixed, any structured block</td>
<td></td>
<td>T [186, 185], L [52]</td>
</tr>
<tr>
<td>G.e</td>
<td>Fixed, Class</td>
<td></td>
<td>S [194]</td>
</tr>
<tr>
<td>G.f</td>
<td>Fixed, File</td>
<td></td>
<td>T [160], S [214]</td>
</tr>
<tr>
<td>G.g</td>
<td>Others</td>
<td></td>
<td>L [156] (Basic Block), S [111] (sub-statement)</td>
</tr>
<tr>
<td><strong>CT</strong></td>
<td>Clone Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CT.a</td>
<td>Type 1 or subset of Type 1: All the tools/techniques listed in Table 5.11 can detect such clones (or a subset) with some limitations.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CT.b</td>
<td>Type 2 or subset of Type 2: Except some text-based techniques/tools [109, 68, 212, 149] and one tree-based [111], all others are able to detect such clones (or a subset) with some limitations.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CT.c</td>
<td>Type 3 or subset of Type 3. Some techniques/tools might have some limitations T [68, 212, 149, 160, 185, 186, 109, 161], L [156, 203], S [26, 105, 70, 79, 25, 38, 193], M [162, 20, 133, 171, 42, 43], G [129, 143, 153, 76]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CT.d</td>
<td>Type 4 or subset of Type 4. Some techniques/tools might have some limitations T [161], G [129, 143, 153, 76]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CT.e</td>
<td>Others</td>
<td></td>
<td>T [161] (ADT), T [48] (Visualization), S [214] (Visualization)</td>
</tr>
</tbody>
</table>
CHAPTER 5. QUALITATIVE COMPARISON AND EVALUATION

general reengineering.

**Clone Type:** The *Clone Type* facet considers the types of clones that a technique can detect. While all techniques can detect exact clones, only few tools (e.g., *Dup* [15]) can find parameterized *Type 2* clones. This issue is discussed in detail in the context of edit-based scenarios later in the paper.

### 5.2.5 Technical Aspect Facets

The technical aspect facets category relates to the comparison algorithms, their complexity, and their unit of comparison (cf. Table 5.5).

**Comparison Algorithm:** The *Comparison Algorithm* facet identifies the different algorithms used in clone detection. For example, the suffix-tree algorithm finds all equal subsequences in a sequence composed of a fixed alphabet (e.g., characters, tokens, hash values of lines) in linear time and space, but can handle only exact sequences. On the other hand, data mining algorithms are well suited to handle arbitrary gaps in the subsequences.

**Comparison Granularity:** Different techniques work at different levels of comparison granularity, from single tokens and source lines to entire AST subtrees and PDG subgraphs. The facet *Comparison Granularity* refers to the granularity of the technique in the comparison phase. The choice of granularity is crucial to the complexity of the algorithm and the returned clone types and determines also the kinds of transformation and comparison required. For example, a token-based technique may be more expensive in terms of time and space complexity than a line-based one because a source line generally contains several tokens. On the other hand, a token representation is well suited to normalization and transformation, so minor differences in coding style are effectively removed, yielding more clones. Similarly, although subgraph comparison can be very costly, PDG-based
techniques are good at finding more semantics-aware clones.

**Worst Case Computational Complexity:** The overall computational complexity of a clone detection technique is a major concern, since a practical technique should scale up to detect clones in large software systems with millions of lines of code. The complexity of an approach depends on the kinds of transformations, the comparison algorithm used, and the granularity of its use. The facet *Computational Complexity* indicates the overall computational complexity of a particular technique/tool.

### 5.2.6 Adjustment Facets

The adjustments category relates to ways the tool allows a user to make adjustments to the search. Adjustments are offered by way of heuristics that may be turned on and off, thresholds that may be set, and various kinds of pre- and post-processing (cf. Table 5.6).

**Pre-/Post-Processing:** The facet *Pre-/Post-Processing* refers to any special pre- or post-processing (e.g., pretty printing) as outlined in Sections 2.6.1 and 2.6.5 that are required other than the usual filtering of whitespace and comments with light-weight parsing or regular expressions [68, 212].

**Heuristics/Thresholds:** This facet indicates whether there are any thresholds or heuristics used by a particular technique/tool that may be manipulated by a user.

### 5.2.7 Processing Facets

The processing category includes facets that characterize the ways a tool analyzes, represents, and transforms the program for the comparison.

**Basic Transformation/Normalization:** Noise (e.g., comments) filtering, normalization and transformation of program elements are important steps in clone detection tools,
### Table 5.5: Technical facets

<table>
<thead>
<tr>
<th>Abb.</th>
<th>Facet</th>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA.a</td>
<td>Suffix tree</td>
<td>L [113, 15, 50], S [38, 137, 199], G [154]</td>
<td></td>
</tr>
<tr>
<td>CA.b</td>
<td>Suffix array</td>
<td>L [213, 22]</td>
<td></td>
</tr>
<tr>
<td>CA.c</td>
<td>AST-based Suffixtree</td>
<td>S [137, 199]</td>
<td></td>
</tr>
<tr>
<td>CA.d</td>
<td>dotplot/scatter plot</td>
<td>T [212, 48], L [203]</td>
<td></td>
</tr>
<tr>
<td>CA.e</td>
<td>Dynamic pattern matching</td>
<td>T [68], M [20, 43, 133]</td>
<td></td>
</tr>
<tr>
<td>CA.f</td>
<td>Data Mining</td>
<td>L [156] (Frequent Sub Sequence), S [208] (Frequent Itemset)</td>
<td></td>
</tr>
<tr>
<td>CA.g</td>
<td>Information Retrieval</td>
<td>L [161] (Latent Semantic Indexing)</td>
<td></td>
</tr>
<tr>
<td>CA.h</td>
<td>Hash-value comparison</td>
<td>S [26, 105, 25]</td>
<td></td>
</tr>
<tr>
<td>CA.i</td>
<td>Fingerprinting</td>
<td>T [109, 160, 108]</td>
<td></td>
</tr>
<tr>
<td>CA.j</td>
<td>Neural Networks</td>
<td>M [56]</td>
<td></td>
</tr>
<tr>
<td>CA.k</td>
<td>Graph matching</td>
<td>G [143, 153], G [129] (slicing), G [59] (model)</td>
<td></td>
</tr>
<tr>
<td>CA.l</td>
<td>Sub-tree matching</td>
<td>S [26] (hashing), S [25]</td>
<td></td>
</tr>
<tr>
<td>CA.m</td>
<td>Euclidean distance</td>
<td>M [134, 63]</td>
<td></td>
</tr>
<tr>
<td>CA.n</td>
<td>Levenshtein distance</td>
<td>LS [141]</td>
<td></td>
</tr>
<tr>
<td>CA.o</td>
<td>Other sequence matching</td>
<td>T [149] (n-neighbor), T [185] (similar to diff), T [171] (Edit distance), L [52] (diff), S [214, 79] (dynamic prog.),</td>
<td></td>
</tr>
<tr>
<td>CA.p</td>
<td>Hybrid SMG</td>
<td>[150]</td>
<td></td>
</tr>
<tr>
<td>CU.a</td>
<td>Line</td>
<td>T [68, 212, 48, 185, 192], L [52], L [15] (p-tokens of line)</td>
<td></td>
</tr>
<tr>
<td>CU.b</td>
<td>Substring/fingerprint</td>
<td>T [108, 109, 160] (multi-line), T [149] (multi-word)</td>
<td></td>
</tr>
<tr>
<td>CU.c</td>
<td>Identifiers and Comments</td>
<td>T [161]</td>
<td></td>
</tr>
<tr>
<td>CU.d</td>
<td>Tokens</td>
<td>L [113, 22, 15, 213, 174, 50, 25], S [137, 199] (tokens of suffix trees), S [141] (Tokens of codeDOM graph), S [111] (Tokens of ASTs)</td>
<td></td>
</tr>
<tr>
<td>CU.e</td>
<td>Statements</td>
<td>L [156], S [208]</td>
<td></td>
</tr>
<tr>
<td>CU.f</td>
<td>Subtree</td>
<td>S [26, 25, 214, 70, 105, 25, 38, 193]</td>
<td></td>
</tr>
<tr>
<td>CU.g</td>
<td>Subgraph</td>
<td>G [129, 143, 153]</td>
<td></td>
</tr>
<tr>
<td>CU.h</td>
<td>Begin-End Blocks</td>
<td>M [133]</td>
<td></td>
</tr>
<tr>
<td>CU.i</td>
<td>Methods</td>
<td>S [199], M [162, 20, 42, 173, 43, 133, 134]</td>
<td></td>
</tr>
<tr>
<td>CU.j</td>
<td>Files</td>
<td>T [160], S [214]</td>
<td></td>
</tr>
<tr>
<td>CU.k</td>
<td>Others</td>
<td>T [171] (Atomic instructions) L [203] (uses non-gapped clones),</td>
<td></td>
</tr>
<tr>
<td>CC.a</td>
<td>Linear</td>
<td>T [149], L [113, 15, 22, 213, 50, 25], S [137], [137, 71]</td>
<td></td>
</tr>
<tr>
<td>CC.b</td>
<td>Quadratic</td>
<td>T [68, 185] (wrt. no of lines/potential clones), L [203] (wrt. no. of non-gapped clones), S [26, 25, 105, 214, 79, 199, 38, 208], M [20, 42, 43, 133, 162, 56, 173, 134],(wrt. no. of methods/begin-end blocks)</td>
<td></td>
</tr>
<tr>
<td>CC.c</td>
<td>Polynomial</td>
<td>G [143, 129, 153, 59]</td>
<td></td>
</tr>
<tr>
<td>CC.d</td>
<td>Others/Not precisely defined</td>
<td>T [109, 160, 212, 161, 171, 192], L [156, 174], S [70, 141, 111, 193]</td>
<td></td>
</tr>
</tbody>
</table>
Table 5.6: Adjustment facets

<table>
<thead>
<tr>
<th>Abb.</th>
<th>Facet</th>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP</td>
<td>Pre-/Post-Processing</td>
<td>PP.a</td>
<td>Pre-processing T [212, 149], L [52, 185, 186, 68, 171], S [79, 214]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PP.b</td>
<td>Post-processing T [160], L [15, 113, 203, 156, 213], S [137, 105], G [129, 76]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PP.c</td>
<td>Others/Possibly none T [109, 161, 192], L [22, 174, 50, 25], S [26, 70, 199, 208, 38, 141, 111, 193, 25], M [20, 42, 43, 133, 162, 56, 173, 134], G [143, 153]</td>
</tr>
<tr>
<td>H</td>
<td>Heuristics/Thresholds</td>
<td>H.a</td>
<td>On clone length T [68, 212, 25], T [149] (4 words), T [108, 109] (50 lines), L [15] (15 lines), L [113, 112, 203, 22, 213, 50] (e.g., 30 tokens), LS [137]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H.b</td>
<td>On code similarity T [186, 185, 212, 161, 171], L [15, 113, 156, 22, 52, 213], S [26, 105, 25, 70, 38, 193], LS [141], M [20, 42, 43, 133, 162, 56, 173, 134], G [153, 76]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H.c</td>
<td>On gap size T [68, 149, 212, 185, 160], L [156, 203, 174], S [105, 70, 79, 208], M [20], G [143, 76]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H.d</td>
<td>On pruning T [68, 108, 109, 212], L [113, 22]*, L [156], S [137], G [153, 76]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H.e</td>
<td>Others/Possibly none T [192], S [214, 199, 111], G [129]</td>
</tr>
</tbody>
</table>

helping both in removing uninteresting clones (filtering), and in finding near-miss clones (normalization and transformation). The Basic Transformation/Normalization facet deals with this issue (cf. Table 5.7).

**Code Representation:** The Code Representation facet refers to the internal code representation after filtering, normalization and transformation (cf. Table 5.8). The complexity of the detector implementation, the bulk of which is the normalization, transformation and comparison, depends a great deal on the code representation. One should note that we have already generally classified the techniques based on overall level of analysis in Section 2.7. Here we attempt a finer-grained classification based on the actual representation used in the comparison phase. For example, although a tree-based technique, the actual code representation of cpdetector [137] is a serialized token-sequence of AST-nodes, improving the computational and space complexities of the tool from quadratic to linear using a suffix-tree based algorithm.
Table 5.7: Basic transformation/normalization facet

<table>
<thead>
<tr>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.a</td>
<td>No normalization of source code T [149, 161, 160, 212] (whitespace and single brackets) M [42]</td>
</tr>
<tr>
<td>T.b</td>
<td>Remove comments and whitespace with regular expressions or light-weight parsing T [68, 108, 212], L [15, 22]</td>
</tr>
<tr>
<td>T.c</td>
<td>Remove comments and whitespace in parsing and apply some kind of pretty-printing/text-processing to remove formatting differences between similar fragments. T [185, 186]</td>
</tr>
<tr>
<td>T.d</td>
<td>Comments are not removed but also taken into consideration for comparison T [161, 108]*, M [162]</td>
</tr>
<tr>
<td>T.e</td>
<td>Apply normalization of identifiers, types and literal values T [192], L [113, 112, 203, 156, 213, 50]</td>
</tr>
<tr>
<td>T.f</td>
<td>Identifier names (and comments) are kept and compared for finding clones T [161]</td>
</tr>
<tr>
<td>T.g</td>
<td>There is flexible normalization of the identifiers (different options are provided to the user) T [185], L [22, 174], S [105]</td>
</tr>
<tr>
<td>T.h</td>
<td>Several language dependent transformation rules are applied T[185] (Example like TXL transformation rules), T [171] (Semantic preserving transformation rules to get sequence of atomic instructions), L [113] (Token transformation rules)</td>
</tr>
<tr>
<td>T.i</td>
<td>Comments and whitespace are ignored in parsing or while generating graphs T [185, 186], S [26, 105, 214, 70, 79, 25, 137, 199, 208], M [20, 43, 133, 134, 173], G [129, 143, 153, 76]</td>
</tr>
</tbody>
</table>

Program Analysis: The facet Program Analysis indicates the kind of program analysis required for a particular technique in order to produce the intermediate representation (cf. Table 5.9). While most text-based techniques work directly on source code and token-based techniques generally require only lexical analysis, other techniques/tools can be very language-dependent (e.g., requiring a full parser).

5.2.8 Evaluation Facets

Empirical validation of tools is important, especially in terms of precision, recall, and scalability. The evaluation category deals with evaluation aspects (cf. Table 5.10). These facets can assist in choosing a well validated tool/technique, in comparing a new tool with one that has existing empirical results, or in choosing a commonly used subject system as a benchmark. They may also encourage empirical studies on promising tools and techniques.
Table 5.8: Code representation facet

<table>
<thead>
<tr>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR.a</td>
<td>Raw source without any changes: <em>Possibly none</em></td>
</tr>
<tr>
<td>CR.b</td>
<td>Filtered Strings: Effective lines of code after removing comments and whitespace (possibly line breaks are not removed) T [68, 212, 149, 109, 108], L [52]</td>
</tr>
<tr>
<td>CR.c</td>
<td>Line breaks are also removed in filtered strings: <em>Most token-based tools do this</em></td>
</tr>
<tr>
<td>CR.d</td>
<td>Filtered subtrings with comments, whitespace and line breaks may or may not be removed T [108, 160] (fingerprint), L [213]</td>
</tr>
<tr>
<td>CR.e</td>
<td>Fingerprinting of substrings with comments, whitespace and line breaks may or may not be removed T [192]</td>
</tr>
<tr>
<td>CR.f</td>
<td>Normalized strings/Token sequence with comments, whitespace and line breaks may or may not be removed T [68, 67], L [113, 112, 203, 22, 213, 50, 25] (token sequence)</td>
</tr>
<tr>
<td>CR.g</td>
<td>Parameterized strings/Token sequence with comments, whitespace and line breaks may or may not be removed L [15, 14], (p-token sequence), L [25]</td>
</tr>
<tr>
<td>CR.h</td>
<td>Words in context T [161]</td>
</tr>
<tr>
<td>CR.i</td>
<td>Metrics/Vectors S [105] (characteristic vector), M [162] (IRL), M [133, 20, 173, 42, 43, 134, 56], G [76]</td>
</tr>
<tr>
<td>CR.j</td>
<td>Abstract Syntax Tree (AST) or Annotated AST or AST nodes are in another form S [26, 214, 193], S <a href="IML">25</a>, S <a href="XML">70</a>, S [79](string alignment), S <a href="suffix-trees">137, 199</a></td>
</tr>
<tr>
<td>CR.k</td>
<td>PDG or variants of PDG G [129, 153] (PDG), G [143] (PDG+AST)</td>
</tr>
<tr>
<td>CR.l</td>
<td>AST/Parse-tree is in another form S [38, 208] (XML), LS <a href="CodeDOM">141</a>, S [111](Tokens of AST-nodes), M [162] (IRL)</td>
</tr>
<tr>
<td>CR.m</td>
<td>Pretty-printed text without comments T [186]</td>
</tr>
<tr>
<td>CR.n</td>
<td>Normalized/transformed Text T [185] (also pretty-printed), L [156] (Mapping statements to numbers)</td>
</tr>
<tr>
<td>CR.o</td>
<td>Hybrid SMG [150] (AST+Metrics+call graph)</td>
</tr>
<tr>
<td>CR.p</td>
<td>Others T [171] (sequence of atomic instructions), L [174] (Frequency table of tokens) G [59] (normalized graph*)</td>
</tr>
</tbody>
</table>
Table 5.9: Program analysis facet

<table>
<thead>
<tr>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA.a</td>
<td>Nothing, completely language independent T [160], T [192] (has several other language-specific options too)</td>
</tr>
<tr>
<td>PA.b</td>
<td>Only needs some regular expressions for removing comments and whitespace or so T [212]</td>
</tr>
<tr>
<td>PA.c</td>
<td>Only needs lightweight parsing for removing comments, whitespace and pretty-printing (or so) of the code T [68, 108, 109, 149, 161], M [42]</td>
</tr>
<tr>
<td>PA.d</td>
<td>Needs a lexer at least for removing comments / whitespace and to tokenize the source L [113, 112, 203, 14, 15, 22, 213, 174, 50, 25],</td>
</tr>
<tr>
<td>PA.e</td>
<td>Needs a full-fledged parser or IDE to generate parse tree/AST or to find another representation of the source L [156], S [26, 214, 70, 79, 137, 25, 38, 199, 208, 141, 111, 193, 29, 25], M [162, 20, 43, 133, 134, 56, 173],</td>
</tr>
<tr>
<td>PA.f</td>
<td>Needs specialized tool to generate Call Graphs, traditional PDGs or annotated special PDGs G [129, 143, 153], SMG [150] (call graph)</td>
</tr>
<tr>
<td>PA.g</td>
<td>Needs language dependent transformation rules also T[185] (full NICAD), T [171], L [113] (lexical)</td>
</tr>
<tr>
<td>PA.h</td>
<td>Needs only a context-free grammar for the language dialect of interest T [186] (Basic NICAD) (in TXL), L [52] (in TXL), S [105]</td>
</tr>
</tbody>
</table>

that are as yet inadequately validated.

**Empirical Validation:** This facet hints at the kind of validation that has been reported for each technique.

**Availability of Empirical Results:** The facet Availability of Empirical Results notes whether the results of the validation are available. If the results are available, other researchers may be able to replicate, compare and extend them with additional studies.

**Subject Systems:** The Subject Systems facet notes which systems have been used in the validation. If researchers conduct their empirical studies on the same systems, results can be compared more meaningfully.
### Table 5.10: Evaluation facet

<table>
<thead>
<tr>
<th>Abb.</th>
<th>Facet</th>
<th>Attr.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Empirical Validation</td>
<td>E.a</td>
<td>Yes, validated empirically in terms of precision, recall, memory and time and compared with other tools S [137, 71]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E.b</td>
<td>Validated enough in support of the claim T [68, 212, 161, 171, 185], L [15, 14, 155, 156], S [70, 26, 105], M [20, 133, 162, 134], G [153, 76]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E.c</td>
<td>Validated by other means or third party comparison study T [185, 186] (with an automatic validation framework [191, 189]), T [68] (with Bellon’s experiment [32]), L [14] (with Bellon’s experiment [32]), L [113] (with Bellon’s experiment [32]), S [26] (with Bellon’s experiment [32]), M [162] (with Bellon’s experiment [32]), G [143] (with Bellon’s experiment [32]),</td>
</tr>
<tr>
<td>AR</td>
<td>Availability of Results</td>
<td>AR.a</td>
<td>Yes, complete results T [186, 185] (see at [183]), S [38] (see at [39]), Experiment [32] (see at [31])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AR.b</td>
<td>Enough/Partial results as in the published paper (or online documents) T [108, 68, 160, 212, 149, 161, 48, 171, 192], L [15, 113, 155, 203, 112, 22, 156, 213, 174, 50, 25, 137, 82], S [26, 70, 214, 137, 71, 105, 199, 208, 141, 111, 193, 25, 29, 194], M [133, 162, 56, 173, 134], G [129, 143, 76, 153]</td>
</tr>
<tr>
<td>S</td>
<td>Subject Systems</td>
<td>S.a</td>
<td>Linux Kernel/part (C, 3M LOC) T [186], L [22, 113, 156, 25, 137], S [105], M [43], G [76]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S.b</td>
<td>JDK/Part (Java, 204K LOC) T [186, 212, 149, 192], L [113, 11, 174, 25, 137], S [105, 208, 193], M [20, 173], Experiments [32]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S.c</td>
<td>SNNS (C, 115K LOC) T [186, 212], L [11, 25, 137], S [137, 11], Experiments [32]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S.d</td>
<td>postgresql(C, 235K LOC) T [186, 212], L [156, 29, 11, 25, 137], S [137], Experiments [32, 11], G [76]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S.e</td>
<td>Apache httpd or part(C, 261K LOC) T [186, 149], L [25, 137, 156]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S.f</td>
<td>FreeBSD (C, 3M LOC) L [155, 156, 113]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S.g</td>
<td>Others T [68, 160, 109, 171], L [213, 50], S [26, 70, 214, 199, 38, 141, 111, 29, 25], M [133, 134, 162, 56], G [143, 129, 153]</td>
</tr>
</tbody>
</table>
Table 5.11: Tools attributes

<table>
<thead>
<tr>
<th>Col. 1</th>
<th>Col. 2</th>
<th>Col. 3</th>
<th>Col. 4</th>
<th>Col. 5</th>
<th>Col. 6</th>
<th>Col. 7</th>
<th>Col. 8</th>
<th>Col. 9</th>
<th>Col. 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool/Author</td>
<td>Platform</td>
<td>P Extern Dependencies</td>
<td>A Availability</td>
<td>U User Interface</td>
<td>I IDE Support</td>
<td>DL Language Paradigm</td>
<td>LS Language Support</td>
<td>R Clone Relation</td>
<td>G Clone Granularity</td>
</tr>
<tr>
<td>Johnson [109, 108, 107]</td>
<td>e a l d c c b a b d c d c a d ad bd cd d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Duploc [68]</td>
<td>e a b d c c c bcfl ad a ac e a b a cd b b c ac b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dupix [180]</td>
<td>e a l d a c e b b l ac b d b c d a d a abc ab b d d b c bd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DupliCate [23]</td>
<td>e a e c e c be a a ac d a d a abc ab b d d b c bcd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SDD [149]</td>
<td>a a b d b a c abc a a ac b a ac e b c d b be</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Marcus* [161]</td>
<td>e b l d a c a b b a acde g c d c b adh c b b be</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NICAD [186, 185]</td>
<td>b a e a e c c bde b bcd abed o a b a bc e fhg n g gc c abcde</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nisehi [171]</td>
<td>c b l b h b c e a a b abed o k d a b abed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simian [192]</td>
<td>d a b p a a e kd a b a ab q d a a c e y t y l a d b b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dap [15]</td>
<td>b a e d c c c be a a ab a ad a b ab b g d bcd b bcd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCFinder(X) [113, 112]</td>
<td>d a b c c c c bcfl ad a ac a d a b ab edh c d bc adbl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTF [22]</td>
<td>c a e d a c c be b a ab b d a c abd bg l d d b d a abaf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CP-Miner [176]</td>
<td>e b e d d c c bc e g abc i edh b cd e a n a b abdadal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHINORB* [213]</td>
<td>e b d a d b b ca d c d c c c ad c ad c c c c c c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPD [174]</td>
<td>a b c c c c bce a a ab q d d d c c g p d b b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CloneDetective [50]</td>
<td>c b ab b b b b ad b a ab q d a c a e l d d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clones [25, 127]</td>
<td>d b d e e c c c bcde ad a ab ad a c a a abd lys b d abedp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CloneDr [26]</td>
<td>c b cd c c c bcfl ad a abc h l b c b i j e bc b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asta [70]</td>
<td>e b l h b c d e a a abq i d c bc h fg j e b b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tmlr [214]</td>
<td>c a l b b e b c a c e f abc o f g b e h g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpdetector [137, 71]</td>
<td>d b e e a c c bc be a a ab a d a h ad j e a b bc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deckard [105]</td>
<td>b a e d a c c be b a ab h l b c b i j e b b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tairas [199]</td>
<td>c b l d b b a b ad b a d d b c e i j e d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CloneDetection [208]</td>
<td>c b l d d c c ce e a ab f e b c e i j e d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CloneJjigger [38]</td>
<td>a b a a c a h eg a a ab abc r b c b l c b i j e d a g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2D2 [141]</td>
<td>c b l d b d c d c d c c ad n d d c ab i j e d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jitterat [111]</td>
<td>e c l d c d b e j e g a q d d c c e i j e d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SimScan [193]</td>
<td>d b b e c ab e a ad a abc q d c b i j e d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cdiff [29, 25]</td>
<td>f c l d e b ac b a b b b abed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kontogiannis [133]</td>
<td>e c l d d e a b a b abc e h l b c b i j e d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mayrand [162]</td>
<td>e b l d d d e a b a b abc q i b c b q i c bc b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Davey [50]</td>
<td>e c l d b e a b abed j i b c b d i j e d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Patenaude [183]</td>
<td>b b l d d c b e a a ab ed q i b c b i j e d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kontogiannis [134]</td>
<td>e c l d c a o a a ab abed m i b c b i j e d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Duplic [185]</td>
<td>e b e d d e a b a a abed k g c c c c i k k f d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kontogiannis [129]</td>
<td>k e d d d e a b ab ed c g c b c i k k f d b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPLAG* [155]</td>
<td>c b e d d c c bce a b abed k g c c c bd i k f b b g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gabel [76]</td>
<td>c b e d d c c bc b b abed q i c b bcd l i f b b ad</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.2.9 Tool Classification and Attributes

In this section we provide the attribute values for the facets for each of the individual tools in our study. Table 5.11 presents a detailed overview of the available tools and techniques in the form of a taxonomy where the first column (Col. 1) groups by the underlying approach, and the second column (Col. 2) lists each tool / technique by name (or first author name for those technique without a tool name) and citations. The third column (Col. 3) gives the attribute values for the Usage facets of Table 5.1 that apply to the tool / technique, and the remaining columns give the attribute values for the other facets, Interaction, Language, Clones, Technical Aspects, Adjustments, Basic Transformation/Normalization, Code Representation, Program Analysis and Evaluation, as described in Tables 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9 and 5.10 respectively.

A particular tool / technique can have multiple attribute values for a facet, represented as a sequence of attribute letters. For example, the attribute value “acg” for facet \( F \) refers to attributes \( F.a \), \( F.c \) and \( F.g \). In order to focus the comparison, we have restricted this summary comparison to methods for procedural and object-oriented languages and have not listed tools and techniques aimed at other paradigms (such as web applications) in this summary.

5.3 Scenario-Based Evaluation of the Techniques and Tools

Clone detection techniques are often inadequately evaluated, and only a few studies have looked at some of the techniques and tools [32, 206, 207, 41]. Of these, the Bellon et al. [32] study is the most extensive to date, with a quantitative comparison of six state-of-the-art techniques, essentially all of those with tools targeted at C and Java. However, even in
that careful study, only a small proportion of the clones were oracled, and a number of other factors have been identified as potentially influencing the results [11]. The general lack of evaluation is exacerbated by the fact that there are no agreed upon evaluation criteria or representative benchmarks. Finding such universal criteria is difficult, since techniques are often designed for different purposes and each has its own tunable parameters.

In an attempt to compare all clone detection techniques more uniformly, independent of tool availability, implementation limitations or language, we have taken a predictive, scenario-based approach. We have designed a small set of hypothetical program editing scenarios representative of typical changes to copy/pasted code in the form of a top-down editing taxonomy proposed in Chapter 4.

Figure 5.1 demonstrates the use of our proposed editing taxonomy for code fragments at the function level of granularity (For details see Chapter 4). The taxonomy is demonstrated on a simple example original function (in the middle, labeled “Original Copy”) that calculates the sum and product of a loop variable and calls another function with these values as parameters. Although the editing steps are demonstrated at function-level granularity, they are general enough to be applicable to any granularity of code fragment. We assume that our primary intention is to find true clones, that is, those that actually result from copy-and-edit reuse of code. Figure 5.1 shows four scenarios, Scenario 1, Scenario 2, Scenario 3 and Scenario 4, where each scenario has several sub-scenarios. Mapping to the literature (Chapter 2), we call the clones created by these scenarios Type 1, Type 2, Type 3 and Type 4 clones respectively.

From a program comprehension point of view, finding such true clones (those are created as per the scenarios) is useful since understanding a representative copy from a clone group assists in understanding all copies in that group [108]. Moreover, replacing all the
Figure 5.1: Taxonomy of editing scenarios for different clone types
detected similar copies of a clone group by a function call to the representative copy (i.e., refactoring) can potentially improve understandability, maintainability and extensibility, and reduce the complexity of the system [74]. These scenarios could also be used to guide the development of forward clone management tools (e.g., CReN [99]).

Based on these hypothetical scenarios, we have estimated how well the various clone detection techniques may perform based on their published properties (either in the published papers or online documentation). In order to estimate maximal potential, we have assumed the most lenient settings of any tunable parameters of the techniques/tools. Thus, this is not an actual evaluation, rather it provides an overall picture of the potential of each technique and tool in handling clones resulting from each of the scenarios. Our comparison is not intended to be a concrete experiment, and could not be comprehensive or truly predictive and qualitative if it were cast as one, bound to target languages, platforms and implementations.

Table 5.13 provides an overall summary of the results of our evaluations, where the symbols represent an estimate of the ability of each technique/tool to accurately detect each (sub-) scenario with both high precision and high recall. For example, a very well (denoted with •) rating for a particular sub-scenario of a particular tool means that the subject tool (or the corresponding technique used in that tool) is capable of detecting (i.e., about 100% recall) that scenario without any false positives (i.e., about 100% precision) as per our understanding. When a tool’s tunable parameters are set to detect a sub-scenario of a particular scenario, detection of the other sub-scenarios of that scenario is not counted as false positives.

However, detecting the sub-scenarios of other scenarios may be considered as false positives. Because the taxonomy is created as a top-down theory of clones from Scenario
1 to Scenario 4, when a tool is set to detect a sub-scenario of a lower numbered scenario (e.g., Scenario 1), any detection of sub-scenario(s) of a higher numbered scenario (i.e., scenarios 2, 3 or 4) is considered as a false positive. On the other hand, when the tool is set to detect a sub-scenario of a higher numbered scenario (e.g., Scenario 3), the detection of the sub-scenarios of the lower numbered scenarios (i.e., scenarios 1 and 2) is desirable (for high recall) and should not be considered as false positives. Table 5.12 summarizes the meanings of the symbols we have used in the evaluation.

For Scenario 2, we also expect that a tool may provide different tunable parameters to detect the different sub-scenarios separately. For example, there may be a separate option for detecting consistently renamed clones from renaming where consistency is not maintained. This is important because some tools use the same technique but differ with respect to the tunable parameters for different types of clones. Moreover, some tools yield syntactic clones while others do not, leading to the lower ratings for a (sub)-scenario.

Given the fact that not all tools actually produce the expected output in practice, we have also employed our experience in tool comparison and knowledge gained from other tool comparison experiments and individual tool evaluations where applicable. Thus, the ratings in Table 5.13 for each (sub)-scenario is a balance of what is expected and what is achieved (where applicable) using a particular tool, potentially hinting the overall ability of the tool with respect to the (sub)-scenarios. Although the scenarios are represented in the language C, when we evaluate a tool that supports only object-oriented languages (e.g., Java), we imagine similar scenarios on that language to evaluate the tool (the adaptability/portability is a separate issue and discussed in the previous section).

An asterisk (*) indicates a technique/tool with special limitations (or that has some other main purpose other than clone detection) such as whole file comparison, visualization only,
### Table 5.12: Meanings of the rating symbols

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>●</td>
<td>very well</td>
<td>Detects the clones with high accuracy and confidence, i.e., with high precision and recall. Has tunable parameters for different types of clones (i.e., can detect clones of different scenarios separately). In case of Scenario 2, has separate tunable parameters for detecting clones of the sub-scenarios. When detecting clones of a sub-scenario of scenario $k$ (except for Scenario 2), detection of the clones of other sub-scenarios of $k$ is desirable for high recall. The scenarios are on a top-down fashion and thus, when detecting clones of scenario $k$, detection of clones of (sub-)scenario $l$ where $k &lt; l$ is not expected (for high precision). However, detection of clones of (sub-)scenario $j$ where $j &lt; k$ is desirable. The tool either has an option for detecting different granularities (e.g., method or begin-end block) of clones or applies several pre-/post-processing activities to avoid spurious clones [137] or at least (if the tool finds clones of free granularity) subsumes the clones of the (sub-)scenario in question. The tool is capable of detecting the clones of the (sub-)scenario with reasonable time and space (not in months for example). To our knowledge there is no empirical studies that shows that the subject tool was not capable (or performed poorly) of detecting the clone type in question.</td>
</tr>
<tr>
<td>◦</td>
<td>probably can</td>
<td>Detects the clones of (sub-)scenario but may return few false positives. May also miss some of the clones. Does not meet one or more of the criteria of the first row (for very well).</td>
</tr>
<tr>
<td>⊙</td>
<td>medium</td>
<td>Detects the clones of the (sub-)scenario but may return many false positives (about 50% for example). Does not meet many of the criteria of the first row (for very well).</td>
</tr>
<tr>
<td>○</td>
<td>low</td>
<td>Detects with lots of false positives (low precision). Also may miss many of the similar clones (low recall). Does not meet many of the criteria of the first row (for very well).</td>
</tr>
<tr>
<td>◎</td>
<td>probably cannot</td>
<td>Although there is no empirical or other sort of evidence, the underlying technique of the technique/tool might be capable of detecting clones of the (sub-)scenario in question. The tool/technique might generate lots of false positives (very low precision). The tool/technique might miss some clones (very low recall).</td>
</tr>
<tr>
<td>◎</td>
<td>cannot</td>
<td>We are not sure but as per the underlying technique of the subject technique/tool, it might be impossible to detect the clones of the (sub-)scenario in question. We do not think there are empirical studies or any sort of evidence that shows that the subject tool is capable of detecting the clones of the (sub-)scenario in question.</td>
</tr>
<tr>
<td>◍</td>
<td>cannot</td>
<td>As per the underlying technique of the subject technique/tool, it is impossible to detect the clones of the (sub-)scenario in question. There is no empirical study or any sort of evidence that the subject tool was capable of detecting the clones of the (sub-)scenario in question.</td>
</tr>
</tbody>
</table>
plagiarism detection, IDE support or other special issues discussed as applicable. In the following subsections, we consider each scenario and outline our reasoning in estimating the ability of the techniques to accurately detect them using the information from Section 5.2.

5.3.1 Scenario 1

Scenario 1: A programmer copies a function that calculates the sum and product of a loop variable and calls another function, foo() with these values as parameters three times, making changes in whitespace in the first fragment (S1(a)), changes in commenting in the second (S1(b)), and changes in formatting in the third (S1(c)) (Figure 5.1).

An ideal clone detection technique should recognize all three copy/pasted/modified fragments as clone pairs with the original or form a clone class for them along with the original. The third column under the Scenario 1 heading of Table 5.13 summarizes how well each technique is likely to work in these scenarios.

Among the text-based techniques and tools, only NICAD [186, 185] is expected to do very well on all the sub-scenarios, in part because it was designed with them in mind. NICAD applies a standard pretty-printing normalization that removes comments (scenario S1(b)) and formatting differences (scenario S1(c)), and uses a whitespace insensitive (Scenario S1(a)) text line-wise comparison to find clones. Although, linear in space and scalable [186], NICAD has a quadratic time complexity with respect to the number of extracted code fragments for comparison. Moreover, NICAD is parser-based and thus language specific. While adapting to a new language, one at least needs to get a TXL [53] grammar for that language. Other text-based tools, such as Duploc [68], DuDe [212] and Simian [192] also detect scenarios S1(a) and S1(b) very well. Unlike NICAD, Duploc does not rely on
Table 5.13: Scenario-based evaluation of the surveyed clone detection techniques and tools

<table>
<thead>
<tr>
<th>Citation</th>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
<th>Scenario 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a b c</td>
<td>a b c d</td>
<td>a b c d</td>
<td>a b c d</td>
</tr>
<tr>
<td>Johnson [109, 108]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Duploc [68]</td>
<td>● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>sif [160]*</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>DuDe [212]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>SDD [149]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Marcus [161]*</td>
<td>● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Basic NICAD [186]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Full NICAD [185]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Nasehi [171]</td>
<td>● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Simian [192]</td>
<td>● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Dup [15]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>CCFinder(X) [113, 112]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Gemini [203]*</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>RTF [22]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>CP-Miner [156]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>SHINOBI [213]*</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>CPD [174]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Clone Detective [50]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>clones/iClones [25, 137]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>CloneDr [26]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Asta [70]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>cpdetector/clast [137, 25]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Deckard [105]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Taras [199]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>CloneDetection [208]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>CloneDigger [38]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>C2D2 [141]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Juillerat [111]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>SimScan [193]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>sifdiml [29, 25]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Kontogiannis [133]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Mayrand [162]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Dagenais [55]*</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Merlo [165, 166]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Davey [56]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Patenaude [173]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Kontogiannis [134]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Antoniol [7, 8]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Duplix [143]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Komondoor [129]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>GPLAG [153]*</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
<tr>
<td>Gabel [76]</td>
<td>● ● ● ●</td>
<td>o o o o</td>
<td>o o o o</td>
<td>o o o o</td>
</tr>
</tbody>
</table>

- **Scenario 1**: very well
- **Scenario 2**: well
- **Scenario 3**: medium
- **Scenario 4**: low
- **Scenario 5**: probably can
- **Scenario 6**: probably cannot
- **Scenario 7**: cannot
robust parsing – instead it uses lightweight lexical analysis to remove comments (scenario S1(b)) and whitespace (scenario S1(a)) within lines and detects clones using string-based dynamic pattern matching. DuDe and Simian do similar things by applying regular expressions (i.e., lexical analysis again). However, all of these line-based techniques / tools are sensitive to format alterations and thus may not detect scenario S1(c). Marcus’s text-based LSI approach [161] is not designed to detect scenario S1(b), since it compares comments (and identifiers) in finding clones. Among the other text-based techniques, Johnson’s approach [107, 108, 109] should detect all three of these sub-scenarios well. Johnson applies several options for keeping/removing whitespace and comments (thus, scenarios S1(a) and S1(b) might be detected well) and uses fingerprints of substrings for finding clones (thus might not be affected by formatting, leading to detect scenario S1(c)). SDD [149] applies n-neighbor approach (i.e., allows gaps in similarity) and thus might detect these sub-scenarios too. However, allowing gaps might lead to detect false positive clones even for these exact clones.

Among the token-based techniques/tools, RTF [22] and clones [137] should detect all three Scenario 1 sub-scenarios well. RTF applies flexible tokenization and clones has a post-processor that can distinguish different types of clones by comparing identifier values and can differentiate other similar scenarios (e.g., sub-scenarios of Scenario 2). However, clones has problems if superfluous brackets are added in the copied fragment as it compares only the sequence of tokens and does not remove brackets before comparison. Token-based techniques and tools (e.g., CCFinder) in general cannot differentiate between clones of Scenario 1 and Scenario 2. Moreover, these techniques often return non-syntactic and spurious clones [137]. Baker’s Dup can also detect clones of scenarios S1(a) and S1(b) very well but cannot detect clones of scenario S1(c), since Dup summarizes all tokens of a line
at a time and thus is sensitive to formatting changes. Most other token-based techniques are not sensitive to formatting changes since they compare token-by-token.

Tree-based techniques (e.g., cpdetector) ignore formatting differences and comments and should detect all Scenario 1 sub-scenarios very well if they look for exact subtrees without ignoring tree-leaves (in most cases they ignore leaves and thus a post-processing step is required to distinguish clones of Scenario 1 and Scenario 2). However, some tree-based techniques use alternative representations of the parse-tree/AST (e.g., Deckard works on characteristic vectors of the parse tree) and may not detect them accurately (a post-processing step is required to differentiate them). Moreover, a recent study [185] shows that an AST-based exact matching function clone detection technique [199] can even miss some exact function clones.

Metrics-based techniques may return the same metrics values for other scenarios of our taxonomy and for other different fragments and thus may return false positives in our sense. Among the metrics-based approaches, Mayrand et al. [162] provide a fine-grained set of metrics for detecting function clones (and possibly also clones of begin-end blocks). Others (e.g., Antoniol et al. [8, 7] and Merlo et al. [166, 165]) also provide similar metrics with some minor differences and are expected to do well on these scenarios.

In theory, graph-based techniques should be good at all Scenario 1 sub-scenarios. However, in practice they yield many variants of the actual clone pairs and that there might be similar graphs for dissimilar code blocks, reducing precision. Thus in our view they do not do well on these scenarios. However, a new variant of the Deckard tool maps PDG subgraphs to related structured syntax before comparison and thus might do well [76].
5.3.2 Scenario 2

Scenario 2: The programmer makes four more copies of the function, using a systematic renaming of identifiers and literals in the first fragment (S2(a)), renaming the identifiers (but not necessarily systematically) in the second fragment (S2(b)), renaming data types and literal values (but not necessarily consistently) in the third fragment (S2(c)), and replacing some parameters with expressions in the fourth fragment (S2(d)) (Figure 5.1).

Once again, an ideal clone detection technique should detect all four modified fragments as clone pairs with the original or should form a clone class for them along with the original. Needless to say, code fragments created from Scenario 1 might also form clone pairs or a clone class with the code fragments of this scenario. The fourth column under the Scenario 2 heading of Table 5.13 summarizes how well each technique may work on these scenarios.

Text-based techniques and tools are not good at detecting clones created by these (sub)-scenarios. For detecting such scenarios token normalization / abstraction / transformation is required to remove the differences between differing identifiers and literals. Of the text-based techniques, only NICAD [185], Nasehi’s approach [171] and Simian [192] can detect such scenarios (although Simian cannot detect scenario S2(d)). NICAD can detect consistently renamed clones (scenario S2(a)) and other renamed clones (scenarios S2(b) and S2(c)) efficiently, and using flexible code normalization thus can detect scenario S2(d) as well. An extended version of Duploc [67] can also detect scenarios S2(b) and S2(c), but not S2(a) and S2(d). However, although these tools (i.e., NICAD, Nasehi’s approach or Simian) find clones by textual comparison, they actually use source transformations (in NICAD’s case, code abstraction and in Nasehi’s approach, a transformation of program code to atomic units) and thus a syntactic / semantic analysis is required that may not be
easily adaptable to other languages. The remaining text-based techniques cannot do well
with these scenarios since they normally compare program text without normalization or
transformation and are therefore fragile to identifier renaming.

Token-based techniques/tools are well suited to detecting clones created by Scenario 2. 
Almost all token-based techniques and tools can detect scenarios S2(a), S2(b) and S2(c)
well, but are likely to also have many false positives due to their identifier and literal nor-
malizations (or abstractions) and the detection of spurious clones [137]. However, only
Dup [14] and clones/iClones [82] are rated to be robust in detecting consistently parameter-
substituted clones (scenario S2(a)) because of their use of parameterized suffix trees. Most
of the tools (except Dup, RTF [22] and clones/iClones) cannot differentiate between Type
1 (clones of Scenario 1) and Type 2 (clones of Scenario 2). RTF and clones/iClones can
also differentiate between the sub-scenarios of Scenario 2. None of the token-based tech-
niques (except possibly CP-Miner [156] that allows arbitrary gaps in comparison) can de-
tect clones of scenario S2(d) because they neither apply structural abstraction to the pro-
gram code nor allow gaps in their comparison.

With the exception of Juillerat’s approach [111], which detects only exact clones, and
Tairas’s approach [199], which detects exact clones and a small subset of Type 2 clones,
almost all tree-based techniques may also detect scenarios S2(a), S2(b) and S2(c) very
well, because these techniques normally ignore identifiers and literals when comparing.
However, like some of the token-based approaches, some syntactic tools do not differentiate
between clones of Type 1 and Type 2. The tools CloneDr [26], ccdiml [25], cpdetector [137]
and clast [25] are known to differentiate these types. For scenario S2(d), the tree-based
tools Asta [70] and CloneDigger [38] seem to be well suited, as they can apply structural
abstraction to arbitrary subtrees.
Metrics-and graph-based techniques can also detect these scenarios, but metrics-based approaches may return many false positives because our other scenarios can yield similar metrics values. Graph-based techniques are also expected to do well in these scenarios. However, they normally return many variants of the ideal clones and that dissimilar code fragments can lead to similar graphs leading to low precision.

5.3.3 Scenario 3

Scenario 3: *The programmer makes five more copies of the function and this time makes small insertions within a line in the first fragment (S3(a)), small deletions within a line in the second fragment (S3(b)), inserts some new lines in the third fragment (S3(c)), deletes some lines from the fourth fragment (S3(d)), and makes changes to some whole lines in the fifth fragment (S2(e)) (Figure 5.1).*

We hope that an ideal clone detection technique would detect all five fragments as clone pairs with the original and form a clone class for them. Again, code fragments of Scenario 1 and Scenario 2 might also form clone pairs / classes with the code fragments of this scenario. The fifth column under the Scenario 3 heading of Table 5.13 summarizes how well each technique may work on these scenarios.

In general, text-based techniques and tools are not good at detecting Type 3 near-miss clones created using Scenario 3 unless they apply threshold-based comparison or combine smaller Type 1 and Type 2 clones in a post-processing phase. *Duploc* transforms program text to a condensed form (removing whitespace and comments) then applies string-based dynamic pattern matching with gaps, and hence can detect changes within a line. Therefore, *Duploc* is expected to do well on scenarios S3(a) and S3(b) (and possibly S3(e)). Although *DuDe* [212] is text-based, it can combine small duplicated segments to form larger ones by
allowing gaps in its scatter plot visualization. Both Basic NICAD [186] and Full NICAD [185] detect these scenarios well as they allow size-sensitive threshold-based comparison of the extracted and pretty-printed potential clones. Full NICAD [185] also uses flexible code normalization and filtering that removes many of the small differences between code fragments and thus can also detect Type 3 clones. Nasehi’s approach [171] transforms code to semantically equivalent atomic units and uses an edit distance algorithms with allowable thresholds. Thus, this approach is also expected to detect scenarios S3(a) and S3(b) well, and possibly also scenarios S3(c), S3(d), and S3(e).

Among the token-based techniques, only Gemini [203] (a post-processor / visualizer for CCFinder [113]) and CP-Miner [156] are likely to work well with these scenarios. CP-Miner uses a frequent subsequence data mining algorithm which allows it to tolerate gaps in cloned segments. Gemini on the other hand, uses output (Type 1 and Type 2 clones) from CCFinder and scatter plot visualization to detect such near-miss clones, much like DuDe.

Among the tree-based techniques, only Deckard [105] and Asta [70] are likely to do well for these scenarios. Asta derives syntax-tree patterns with placeholders for complete subtrees, which supports structural abstraction. Deckard uses the novel idea of a characteristics vector (thus Deckard can also be classified as a metrics-based tool) to approximate the structural information from ASTs in the Euclidean space. However, as with metrics-based approaches, such an approximation is challenging and vector values from two quite distinct code fragments may be similar, indicating that Deckard could return many false positives in detecting such clones. Other tree-based tools, such as CloneDr [26] and ccdiml [25], may detect scenarios S3(a) and S3(b) if their underlying similarity measure for inexact tree matching is set to tolerate them. In CloneDr, a compiler generator is used to generate an annotated parse tree (AST) and compares its subtrees by characterization metrics based
on a hash function. Source code of similar subtrees is then returned as clones. The hash function enables one to do parameterized matching and to detect gapped clones, especially if the gaps are within a line. _ecdiml_ is a variant of _CloneDR_ that has a different intermediate representation with explicit modeling of sequences, which helps in finding near-miss clones created from these scenarios.

Metrics-based techniques can find clones in these scenarios, but may yield many false positives, since many other code fragments may have similar metrics values, resulting in lower overall accuracy. However, scenarios S3(a) and S3(b) can likely be accurately detected by at least some of the metrics-based approaches, notably Mayrand’s [162], Dagenais’s [55], Merlo’s [166] and Antoniol’s [8].

Graph-based approaches primarily use control and data flow information and thus are expected to detect these scenarios well. In fact, in Bellon’s experiment [32], the graph-based tool _Duplix_ [143] was found to detect a small proportion of such near-miss clones. However, in general graph-based tools may return many variants of the ideal clones, and some of these variants can be considered as false positives, yielding a lower overall accuracy. Only the recent semantics-based approach by Gabel et al. [76] has been demonstrated to scale. Instead of comparing subgraphs of the PDGs, Gabel’s approach maps subgraphs to related structured syntax and then finds clones using _Deckard_ technique.

One should also note that although Kamiya [113], Krinke [143], Mayrand/Merlo [166, 162], and Rieger [68] mention that their approaches can also find clones of _Type 3_, according to Bellon et al.’s study [32] in practice only Krinke’s _Duplix_ actually does. In _Duplix_, however, clones of other types are found with very low recall.
5.3.4 Scenario 4

Scenario 4: The programmer makes four more copies of the function and this time reorders the data independent declarations in the first fragment (S4(a)), reorders data independent statements in the second (S4(b)), reorders data dependent statements in the third (S4(c)), and replaces a control statement with a different one in the fourth (S4(d)) (Figure 5.1).

Again, we expect that an ideal clone detection technique should be robust enough to detect such modified code fragments as clone pairs with the original or form a clone class for them. Once again, code fragments of Scenario 1, Scenario 2 and Scenario 3 might form clone pairs/clone classes with the code fragments of this scenario. The sixth column under the Scenario 4 heading of Table 5.13 summarizes how well each technique is likely to work in these scenarios.

Among the text-based techniques, only Marcus’s LSI approach [161] is likely to do well with scenarios S4(a), S4(b) and S4(c). Marcus’s approach considers only comments and identifier names in the comparison. When statements of copied fragments are reordered, comments and identifiers may not be changed and thus their approach may detect these scenarios. Nasehi’s approach [171] performs a semantics-preserving transformation for different syntactic variants of a language to the same atomic units. Thus, the representation of the atomic units of the original function with the for loop might be similar to the atomic representation of the copied function with while loop of scenario S4(d). Moreover, this approach uses an edit distance based algorithm, which allows for dissimilarity thresholds in the comparison. We therefore expect that Nasehi’s approach may be able to detect clones created by scenario S4(d). NICAD probably can detect the reordering scenarios S4(a), S4(b) and S4(c) if the total gap created by the reordering of statements is within the allowable size-sensitive difference thresholds. However, increasing the threshold might lead to
false positive clones.

Unfortunately, there is no token-based technique that can detect clones created in these scenarios well. This is obvious since these techniques/tools use exact matching on normalized token sequences and do not allow for any gaps. Reordering statements (scenarios S4(a), S4(b) and S4(c)) or replacements of one control by another equivalent variant (scenario S4(d)) obviously breaks the token sequences between the original and copied code fragments. However, some token-based tools, such as Gemini and CP-Miner, might detect scenarios S4(a), S4(b) and S4(c). Gemini uses scatter plot visualization of Type 1 and Type 2 clones from CCFinder and thus might detect scenarios S4(a), S4(b) and S4(c) by allowing gaps. CP-Miner allows for arbitrary gaps in cloned segments and thus might also detect scenarios S4(a), S4(b) and S4(c). However, there is no token-based technique that can detect scenario S4(d).

The situation is worse in the case of tree-based techniques. There is no tree-based technique or tool that can be expected to detect these scenarios, with the possible exception that CloneDr may be able to detect clones of scenario S4(a) since its subtree characterization can ignore declaration statements.

Metrics-based techniques should be able to detect scenarios S4(a) and S4(b) well, since reordering of data-independent statements might not change the metrics values. However, metrics values might change when reordering happens between data-dependent statements (scenario S4(c)) due to the underlying metrics definition. When control replacement is performed on the copied fragment (scenario S4(d)) metrics values might change significantly and thus metrics-based techniques either cannot detect scenario S4(d) or will detect it with many false positives, yielding a low overall accuracy.

It appears that only PDG-based techniques are likely to work well with scenarios S4(a)
and S4(b). PDG-based techniques use data and control flow information, which remains unchanged across reordering of declarations and data independent statements. Reordering of data dependent statements may change the data and control flow graphs however, so they may not do as well with scenario S4(c). To detect scenario S4(d), exhaustive source transformation may be necessary. However, an alternative approach is proposed in the plagiarism detection tool GPLAG [153] for finding plagiarized code similar to those created by scenario S4(d).

5.4 An Example Use of the Study

Our survey and evaluations are not just intended for experts in clone detection, but also for users and builders of tools based on clone detection techniques. As a demonstration of how this study can help, we provide two example user intentions and suggest a tool or set of tools to meet their requirements. Of course, many other combinations of the tools can be derived based on user requirements, both in terms of different scenarios and the techniques used. Such a combination might help one to understand how to design a hybrid approach to be robust across all types of clones or how to employ a set of different tools to achieve a better result. Our NICAD tool [185] is an example of such a hybrid, combining tree-based structural analysis with text-based comparison.

**Intention 1:** A tool user would like to find all types of clones (as outlined in this paper) in a large C system (the Linux kernel) with reasonable performance.

Here, the primary objective is the ability to handle large C systems while doing well in finding all the kinds of clones that may be created by the various editing scenarios outlined in Section 5.3. Let us first look for individual tools that rate reasonably well for the
scenarios. From Table 5.13 we see that the obvious set is *Gabel* [76], *GPLAG* [153], *Kontogiannis* [133], *C2D2* [141], *CP-Miner* [156], *Gemini* [203], *Nasehi* [171]. Although none of these tools is able to handle all of the scenarios, they all seem to do well with most of the scenarios.

The second requirement is that the tool should handle C systems. According to our findings in Section 5.2 (column 5 of Table 5.11), only *Gabel* [76], *GPLAG* [153], *Kontogiannis* [133], *CP-Miner* [156], and *Gemini* [203] meet this requirement.

As a third requirement, the user needs a tool capable of handling large systems. From the 7th column with column heading *Technical Aspects* and from the corresponding facet table (Table 5.5), the user can get an idea of the algorithms used and their associated complexities. In particular from the sub-column *Computational Complexity* of Table 5.11 and from the last row (*CC (Worst Case Computational Complexity)*) of Table 5.5, we see that of the set we have chosen only *Gemini* seems computationally efficient. However, although *Gemini* uses *CCFinder* in the background for finding smaller *Type 1* and *Type 2* clones which is linear w.r.t. the size of the program, finding combination of *Type 1* and *Type 2* clones to form *Type 3* clones may require superlinear time; often dynamic programming is used for this combination, which is not linear. Furthermore, *Gemini* is mainly a visualization tool and thus might not fully meet our user’s requirements.

The question now remains as to whether there are other tools in our candidate set that can handle large systems despite having non-linear (worst case) computational complexities. We can find this information from the 10th column (with column heading *Evaluation*) of Table 5.11. In particular, from the sub-column with heading *Subject Systems* and the corresponding *Evaluation* facet table (Table 5.10) we see that both *CP-Miner* and *Gabel* have already been evaluated with *Linux Kernel*, one of the largest C systems. The question again
remains whether the results of their studies are available, especially for the Linux Kernel. We can see this information in the same column Evaluation with sub-column Availability of Results and the corresponding facet table (with row heading AR (Availability of Results) in Table 5.10). We see that complete results are not available for Linux Kernel.

As the results are not available, the user needs to run the tool (either CP-Miner or Gabel) to find clones in Linux. The next crucial question now is whether the tools are available for third party use. From the Availability facet of Usage category in Table 5.11 and in more detail in Table 5.1, we observe that neither of them is available online but an evaluation version may be available on request.

The remaining question is, which tool to request first? The user can ask for both tools, or can be more specific in determining who might actually be able to release their tool. In particular, we can look to see whether the tool is standalone or has any external dependencies or is a part of required larger tool set. If the tool is standalone, it is more likely that the tool will be available upon request, otherwise it is likely that the tool may not be available, or even if available may be hard to use by a third party. Unfortunately, we see (from the same tables above) that both the tools have external dependencies. With a closer look in the description of the row External Dependencies in Table 5.1 we see that CP-Miner is dependent on CloSpan and Gabel is dependent on CodeSurfer. Given that both tools are dependent on other systems, the user might contact both the tool authors or may choose to undertake further studies on the dependencies by reading the details in Section 2.7 or the corresponding papers before contacting the tool authors. The user might also reconsider other tools because neither of the chosen two can actually detect all types of clones. Using our study and evaluations in this paper, one can identify options quickly and with minimal effort.
Intention 2: A user wants to detect clones from many systems in different languages. The user does not aim to detect all types of clones but the intention is to detect as many types as possible. The user is also willing to do some adaptation work for different languages if the tool seems appropriate for the target task at hand. Computational complexity should be reasonable but need not be ideal.

Here, the primary concern of the user is that the tool should be either language independent or adaptable to other languages with a reasonable amount of effort. However, there is a trade-off between the adaptability to different languages and the quality (e.g., capability of dealing with different types of clones) of the tool. The user is also comfortable with computationally expensive tools, within reason (i.e., not taking weeks or months to process systems).

From Table 5.9 we find that text-based tools are either language-independent or easily adaptable to other languages but the computational complexity depends on the algorithm used. Token-based tools are in most cases language-dependent (needing a lexer at least) but computationally efficient. As the concern is adaptability (and not the complexity), the user looks for a text-based tool (or set of tools) in Table 5.13 that covers most of the scenarios. Such a set of tools is *Full NICAD* [185] and *Nasehi* [171]. Before taking the final decision of which tool should be chosen, the user needs to look at some details of the tools. In particular, the user needs to know whether the chosen tools have any language dependencies or not, and in case there is any language dependency, how much effort it might take to adapt to other languages.

From Section 5.2 we know that the *Program Analysis* facet under category *Processing* has information regarding language dependencies. Two other facets (*Transformation* and *Code Representation*) of category *Processing* further hint about the adaptability of a tool to
a different language. From the sub-column *Program Analysis* of the 9th column of Table 5.11 we see that both tools have attribute value *g* for the *Program Analysis* facet. Table 5.9 tells us that this means that both of the tools use language-dependent transformation rules. Thus although they are text-based techniques, they might not be easily adaptable to other languages since they apply advanced transformation rules on the program text before the comparison. These transformations obviously need syntactic (or semantic) analysis of the source code.

In order to gain further insight into the tools, we examine the attribute values of the other two facets, *Transformation* and *Code Representation*, and find that while NICAD uses example-like code normalization rules, which may easily port to other languages, *Nasehi* applies semantics-preserving transformations to yield an equivalent set of atomic instructions, which may not be as easy to adjust. Thus NICAD may require less work to adapt than *Nasehi*.

In this situation the user may choose NICAD or may compare other attributes of the two tools to come to a final decision. In particular, the user can examine the *Language* facets (*Language Paradigm* and *Language Support*) of the two tools. From the 5th column of Table 5.11 (and the associated facet Table 5.3) we see that while NICAD can handle both procedural (e.g., C) and object-oriented (e.g., Java) systems, *Nasehi* works only with object-oriented systems (Java) and thus, *Full NICAD* may be a better choice than *Nasehi* for this purpose. Of course, other facet attributes should also be examined for a final decision.

Alternatively, the user may look for token-based tools. At a first glance at Table 5.13, we see that *CP-Miner* [156] covers most of the clone types/sub-scenarios. However, after examining its attribute values from Table 5.11 (and the associated facet tables) we find that
a full-fledged parser is required when it needs to be adapted to a different language, and that it depends on an external system (CloSpan). The user thus cannot choose CP-Miner.

Instead of giving up on which tool to choose (there are about 40 tools out there), the user can examine the Program Analysis facet table (Table 5.9) first. This table shows the different attributes (with description) of language dependency and citations to the corresponding tools. Fortunately, we see that attribute PA.h: Needs only a context-free grammar for the language dialect of interest has three citations, one text-based (denoted with T) tool, Basic NICAD [186], one token-based (denoted with L), Cordy [52], and one tree-based (denoted with S) tool, Deckard [105]. Among the three tools, Cordy only works with HTML pages and thus cannot be chosen as the user wants to find clones in different procedural and object-oriented systems.

The question now remains whether to choose the text-based Basic NICAD or the tree-based Deckard. The user then has to examine which tool covers most of the clone types. From Table 5.13 we see that Deckard covers more clone types/sub-scenarios than Basic NICAD. Furthermore, even though a tree-based tool, Deckard needs only a context-free grammar to adapt to a new language. Basic NICAD also only needs a context-free grammar, but has to be written in TXL [53] format. Of course, the user has to examine the other associated attribute values of the two tools before coming to a conclusion.

These two examples demonstrate some of the ways how our study can be used to assist in understanding the alternatives when faced with a need for clone detection. Depending on the particular intentions, a range of possibilities may present themselves, but using our summary tables, alternatives can be quickly narrowed down to focus on the one or two most appropriate to the application.
5.5 Related Work

Although there is no work in the literature that provides a property-based comparison and scenario-based evaluation of the techniques and tools similar to this study, several tool comparison experiments have been conducted to estimate the abilities of the tools in terms of precision, recall, and time and space requirements.

One of the first experiments was conducted by Bailey and Burd [41], who compared three state-of-the-art clone detection and two plagiarism detection tools. They began by validating all the clone candidates of the subject application obtained with all the techniques of their experiment to form a human oracle, which was then used to compare the different techniques in terms of several metrics to measure various aspects of the reported clones.

Although they were able to verify all the clone candidates, the limitations of the case study in terms of a single subject system, modest system size and validation subjectivity may make their findings less than definitive. Moreover, the intention of their analysis was to assist in preventative maintenance tasks, which may have influenced their clone validation process.

Considering the limitations of Burd and Bailey’s study, Bellon et al. set out to conduct a larger tool comparison experiment [32] on the same three clone detection tools used in Burd and Bailey’s study and three additional clone detection tools. They also used a more diverse set of larger software systems, consisting of four Java and four C systems totaling almost 850 KLOC. As in the study of Burd and Bailey, a human oracle validated a random sample of about 2% of the candidate clones from all the tools evenly and blindly. While their study is the most extensive to date, only a small proportion of the clone candidates were oracled and several other factors may have influenced the results [11]. Bellon’s framework has been reused in experiments by Koschke et al. [137, 71] and Ducasse et al. [67] (partially),
but without any improvements to the framework.

Rysselberghe and Demeyer [207, 206] have evaluated prototypes of three representative clone detection techniques, providing comparative results in terms of portability, kinds of duplication reported, scalability, number of false matches, and number of useless matches. However, they did not make a reference set, used relatively small subject systems (under 10 KLOC) and did not provide the reliability of the judge(s) that validated the found clones. Moreover, rather than quantitative evaluation of the detection techniques, their intention was to determine the suitability of the clone detection techniques for a particular maintenance task (refactoring) which might have influenced their clone validation.

Another interesting study has been conducted by Bruntink et al. [36], in which several clone detection techniques are evaluated in terms of finding known cross-cutting concerns in C programs with homogeneous implementations.

## 5.6 Summary

In this chapter, we have focused on clone detection techniques and tools, providing a concise but comprehensive survey and a hypothetical evaluation based on editing scenarios. A more detailed review of the entire range of clone detection research can be found in our technical report [182]. Koschke’s Dagstuhl report [138] and the corresponding book chapter [139] also provide an excellent brief overview.

We hope that the results of this study may assist new potential users of clone detection techniques in understanding the range of available techniques and tools and selecting those most appropriate for their needs. We hope it may also assist in identifying remaining open research questions, avenues for future research, and interesting combinations of techniques. The evaluation results of this chapter are based on estimating the performance of techniques
using the most lenient values of all tunable parameters, and thus our findings differ from the results of empirical studies such as Bellon et al. [32].

While in this study our goal was predictive rather than empirical, we have undertaken an experiment using our editing scenarios as the basis for generating and injecting thousands of artificial mutants which can be used to empirically evaluate and compare actual tools on a similar basis (Chapter 6).
Chapter 6

A Mutation / Injection-based
Framework for Automatically
Evaluating Code Clone Detection Tools

NICAD was found to have high precision both in our first experiment with small open source systems (Chapter 3) and in our second exhaustive experiment with large open source systems in different languages (Chapter 7). However, due to the practical difficulties outlined in Section 6.1 below, we could not accurately measure recall in those experiments. In our scenario-based evaluation (Chapter 5), NICAD was placed one of the best tools available. However, that evaluation was predictive and qualitative rather than quantitative. In order to evaluate NICAD in terms of both recall and precision and with respect to different fine-grained types of clones (Chapter 4), in this chapter, we propose a mutation / injection-based framework. The framework automatically and efficiently measures (and compares) the precision and recall of NICAD for different fine-grained types of clones, overcoming most of the existing challenges of tool evaluation. Experimental results show
that NICAD gives both high precision and high recall for different fine-grained types of clones. Although the framework is initially targeted at evaluating NICAD and its variants, the objective was to design a generic framework for evaluating and comparing clone detection tools by overcoming the known challenges to objective tool comparison experiments. The resulting framework is flexible and adaptable enough to evaluate any clone detection tool.

The chapter is organized as follows. Following a general motivation in Section 6.1, we present the mutation operators for cloning in Section 6.2 which is based on the editing taxonomy for clone creation of Chapter 4 and/or the editing scenarios of Chapter 5. In Section 6.3, we provide the details of the mutation-based evaluation framework. Section 6.4 reports the results of an example use of the framework with NICAD and its variants. While in Section 6.5 we consider existing evaluation studies and their relation to ours, Section 6.6 concludes the chapter with our plans for future research. The reader is referred to Section 2.1 of Chapter 2 for a short introduction to terminology and general types of clones as a background study of this chapter.

6.1 Motivation

Given the similarity of clone detection to information retrieval (IR) [134], precision and recall are important factors in measuring and comparing the effectiveness of these tools, especially when dealing with large systems. Unfortunately, despite a decade of active research, there has been a marked lack of in-depth evaluation in terms of precision and recall. Although in some cases precision has been measured by manually validating a very small set of randomly selected clone candidates [76, 105, 129, 143, 156], recall has not been measured so far, in part due to the practical difficulties in measuring it. While tool comparison
experiments [31, 32, 41, 207] and some individual tool evaluations [71, 134, 185] have been designed to measure precision and recall (as well as time and space), these efforts have faced serious challenges with the difficulty of creating a large enough base of reliable reference data and the expense of manually validating thousands of candidate clones.

As a result, these experiments have either used no reference data [207], have simply assumed that the union of all tool results from a set of subject systems was representative (possibly hand validating a small sample) [31, 32, 41, 71] or have manually validated clones detected in a small subject system using a human oracle [36, 121, 134, 185]. While the union reference data may give good relative measurements [32], one cannot guarantee that the subject tools have indeed detected all the clones from the subject systems. Manually validating the large number of candidate clones or manually oracleing a subject system is also a real challenge. For example, in Bellon’s experiment [32], it took 77 hours for Bellon to validate only 2% of the candidate clones. While oracleing small systems is possible [41, 134, 185], even a relatively small system such as Cook, when looking only for function clones, has nearly a million function pairs to sort through [209], an impossible task to be handled error-free by a human. Moreover, none of the experiments or individual authors report on the reliability of the human judges, and even expert judges frequently disagree in creating task-relevant reference data [209]. Furthermore, with the exception of Bellon et al. [32] (for clone types 1, 2 and 3) and Falke et al. [71] (for clone types 1 and 2) there is no work that separates precision and recall values for different types of clones.

In this chapter, we propose a mutation-based [6] approach that automatically and efficiently measures (and compares) the recall and precision of clone detection tools for different types of fine-grained copy/paste/modify clones. We propose a taxonomy of clone types that reflects a developer’s typical copy/paste editing activities, and design code mutation
operators to model each type. By using these operators to generate and track a large number of artificial clones, we then automatically measure how well (i.e., precision) and how efficiently (i.e., recall) these known clones are detected by a particular tool (for individual tool evaluation) or group of tools (for comparing different tools). We have shown an example use of the framework with NICAD and its variants and found that NICAD is capable of detecting different fine-grained types of clones with high precision and recall except Type 4 clones.

6.2 Mutation Operators for Cloning

Mutation [6] has been used in the testing community for over 25 years and is traditionally used to evaluate the effectiveness of test suites. Mutation relies on the notion of mutation operators, editing operations that systematically modify certain statement or expression forms to subtly change their meaning in a way that mimics typical programming errors. Given a program under test, the mutation operators are used to generate a set of mutants, versions of the program that have been modified by one of the operators, and test suites are evaluated on the basis of their ability to kill (that is, expose the behavioral difference in) each of the mutants.

Although mutation has been used primarily in the testing community, Bradbury et al. [35] have previously applied mutation-based analysis to the comparative evaluation of fault detection techniques for concurrent software, an area where objective evaluation is notoriously difficult. In light of the similar difficulties in evaluating clone detection techniques, we use mutation as the basis of our clone detection tool evaluation framework.

While mutation operators for generating potential “bugs” in various languages have been extensively studied, mutation operators for code cloning are to our knowledge unique
to our work. In mutation testing analysis, mutation operators are targeted to change the
original code so as to introduce new potential bugs. Analogously, mutation operators for
cloning should create new clones by modeling developers’ copy/paste editing activities,
exactly what our editing taxonomy (Section 4.2) encodes.

In this section we outline the set of mutation operators that we have developed using
the TXL source transformation language [53] for generating each of the different types of
clones (and combinations thereof) given an original code fragment. In particular, we follow
the editing taxonomy for clone creation of Chapter 4 (Figure 4.1) and the editing scenarios
of Chapter 5 (Figure 5.1) in building the mutation operators.

Table 6.1 summarizes the set of mutation operators we have developed and used in
our evaluation framework (Section 6.3). The first column of the table gives a name to the
mutation operator, the second lists the associated editing activities, the third provides an
example by reference to the fragments of the editing taxonomy (Figure 4.1) whereas the
fourth to the editing scenarios (Figure 5.1), and the fifth gives the clone type of the mutated
result.

Like the mutation operators used in mutation testing, each of our cloning operators
is targeted to make just one random change. For example, the mutation operator \textit{mSIL}
randomly does small insertion within a line, producing fragment (g) as a mutant of fragment
(f) in Figure 4.1 or mimics scenario S3(a) of Figure 5.1. However, unlike mutation testing,
we are interested in clones that can be created by sequences of editing operations, and thus
sequences of our mutation operators can be applied to produce other clones, possibly of
mixed type, as for example in the sequence of edits that leads from the original fragment (a)
to the mutated fragment (n) in Figure 4.1 (solid black arrows). By applying the operators in
random sequences to a code fragment, we can create hundreds of different cloned versions
Table 6.1: Mutation operators for cloning

<table>
<thead>
<tr>
<th>Name</th>
<th>Random Editing Activities</th>
<th>Ref1</th>
<th>Ref2</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mCW</td>
<td>Changes in whitespace and</td>
<td>a ⇒ b</td>
<td>S1(a)</td>
<td>Type 1</td>
</tr>
<tr>
<td>mCC</td>
<td>Changes in comments</td>
<td>a ⇒ b</td>
<td>S1(b)</td>
<td></td>
</tr>
<tr>
<td>mCF</td>
<td>Changes in formatting</td>
<td>a ⇒ c</td>
<td>S1(c)</td>
<td></td>
</tr>
<tr>
<td>mSRI</td>
<td>Systematic renaming of identifiers*</td>
<td>c ⇒ d</td>
<td>S2(a)</td>
<td>Type 2</td>
</tr>
<tr>
<td>mARI</td>
<td>Arbitrary renaming of identifiers*</td>
<td>c ⇒ e</td>
<td>S2(bc)</td>
<td></td>
</tr>
<tr>
<td>mRPE</td>
<td>Replacement of identifiers with expressions (systematically or non-systematically)</td>
<td>d ⇒ f</td>
<td>S2(d)</td>
<td></td>
</tr>
<tr>
<td>mSIL</td>
<td>Small insertions within a line</td>
<td>f ⇒ g</td>
<td>S3(a)</td>
<td>Type 3</td>
</tr>
<tr>
<td>mSDL</td>
<td>Small deletions within a line</td>
<td>f ⇒ h</td>
<td>S3(b)</td>
<td></td>
</tr>
<tr>
<td>mLs</td>
<td>Insertions of one or more lines</td>
<td>g ⇒ i</td>
<td>S3(c)</td>
<td></td>
</tr>
<tr>
<td>mDLs</td>
<td>Deletions of one or more lines</td>
<td>g ⇒ j</td>
<td>S3(d)</td>
<td></td>
</tr>
<tr>
<td>mMLs</td>
<td>Modifications of whole line(s)</td>
<td>g ⇒ k</td>
<td>S3(e)</td>
<td></td>
</tr>
<tr>
<td>mRDS</td>
<td>Reordering of declaration statements</td>
<td>i ⇒ l</td>
<td>S4(a)</td>
<td>Type 4</td>
</tr>
<tr>
<td>mROS</td>
<td>Reordering of other statements+</td>
<td>i ⇒ m</td>
<td>S4(bc)</td>
<td></td>
</tr>
<tr>
<td>mCR</td>
<td>Replacing one type of control by another</td>
<td>i ⇒ n</td>
<td>S4(d)</td>
<td></td>
</tr>
</tbody>
</table>

Ref1-Refers to the editing taxonomy (Figure 4.1) for the corresponding mutator
Ref2-Refers to the editing scenarios (Figure 5.1) for the corresponding mutator
*Function names, variables, data types and literal values
+Data-dependent or independent statements

on which to test clone detectors. As in all mutation-based analysis, we cannot be certain that the created mutants actually reflect a developer’s editing activities, but we know that they are similar.

Almost all of our mutation operators are language independent, and can be applied to fragments of any granularity (e.g., functions, begin-end blocks, statement sequences, and so on). Only the keywords of the target language need be provided, so that they are not mistaken for identifiers by the mutator. The operator mCR is of course language-specific, and currently supports C, Java and C# for a small subset of controls, although support for other controls or adaptation to other languages is straightforward.
6.3 The Evaluation Framework

In this section, we provide the details of our evaluation framework. We begin with the components of the framework and then discuss evaluation strategies, language dependency, scalability and adaptability to other tools.

Figure 6.1 shows a conceptual diagram of our framework. The framework has two main phases, the Generation Phase, in which randomly mutated clone fragments are generated from the original code base and randomly injected into the code base to get mutated code bases, and the Evaluation Phase, in which the mutated code bases are used to evaluate and compare clone detection tools. The random mutation and injection steps allow for thousands of randomly placed clones to be generated.

6.3.1 Clone Generation Phase

In the first phase (Figure 6.1(a)), any number of mutated versions of the code base containing injected mutant clones are created. Beginning with a subject system (code base) in a language supported by the tools under evaluation, the generation phase has the following steps.

Random Fragment Selection: Once the subject code base is selected, any desired number of existing code fragments from the code base are automatically and randomly selected for clone mutation. Code fragments can be of any chosen granularity (e.g., function definition, begin-end block, statement sequence) and can be selected by hand if desired. Selected code fragments are stored as fragment files in a directory for further processing.

Mutant Clone Generation: A TXL-based mutation process then uses the clone mutation operators (Section 6.2) to mutate the selected code fragment files either randomly or sequentially to create a set of mutant clone fragments, which are stored as files in a second directory. Any number of mutant clone fragments can be generated from each selected
CHAPTER 6. MUTATION / INJECTION-BASED FRAMEWORK

(a) The Framework: Generation Phase

(b) The Framework: Evaluation Phase

Figure 6.1: The proposed evaluation framework
fragment, using any fixed or random combination of the clone mutation operators.

Random Clone Injection: For each of the mutated code fragments, a mutated version of the code base is created by injecting the mutated code fragment at a random position in a randomly selected source file of the code base. (This contrasts with our original framework plan [189] in which we proposed to mutate known clones in place.) Random file and location injection mimics a developer's possible copy/paste scenario, yielding a set of realistic new code bases each with exactly one new copy/pasted clone. Information for each new code base with the location of its injected mutant clone, the mutation operators that were used to generate it, and the location of the original code fragment from which it was mutated is stored in a database for use in the evaluation phase (Figure 6.1(b)).

6.3.2 Tool Evaluation Phase

In the evaluation phase (Figure 6.1(b)) each of the mutant code bases is fed to each of the subject clone detection tools for evaluation and comparison. If the tool detects and pairs the injected mutant clone with its original fragment, then it is considered to have “killed” the mutant. This phase consists of the following major components:

Tool Runs: Each of the subject tools is run with each of the mutant code bases as input. Depending on the mutation operators used, each tool is run with its tunable parameters (if any) set to target the clone types of the injected mutants.

Detection Evaluation: For each tool with each mutant code base, the generated clone report is analyzed for unit recall (Section 6.3.3) and precision (Section 6.3.4) of the tool in detecting the injected mutant fragment and its original as a clone pair. Location and type of the mutant clone is provided by the database generated in the generation phase.

Evaluation Database: As the tools are run, unit recall and precision information for each mutant is added to an evaluation database for summary analysis. Information from
the clone validator (see Section 6.3.6) is added to the database before analysis.

Statistical Analysis and Reporting: Once the study is completed, the evaluation database is used to calculate the observed summary recall and precision of each subject tool for each fine-grained clone type (i.e., individual mutation operator), each clone type (types 1 through 4), and overall recall and precision. Accurately summarizing precision may involve a small amount of manual analysis for some clone pairs (see Section 6.3.6).

6.3.3 Measurement of Recall

One of the primary objectives of our work is to automatically measure the recall of clone detection tools. Our framework reports recall for each fine-grained clone type (i.e., individual mutation operator), each clone type (types 1 through 4), and overall for each tool. Our recall definition is the usual one in IR research, that is, the number of items detected divided by the total number of detectable items. In our case the detectable items are our injected mutant clones, and our mutation-based technique makes each individual recall decision simple — if the mutant clone $moCF$ of original code fragment $oCF$ injected into mutant code base $mioCB$ of code base $oCB$ is “killed” (i.e., $(oCF, moCF)$ is detected as a clone pair) by the detector, then its recall for that clone is 1, otherwise it is 0. We can denote this decision by:

$$R_T^{(oCF,moCF)} = \begin{cases} 
1, & \text{if } (oCF, moCF) \text{ is detected by } T \text{ in } mioCB; \\
0, & \text{otherwise.}
\end{cases}$$

The same mutated code fragment, $moCF$ can be randomly injected to the original code base, $oCB$ any number of times, producing $n$ different mutated / injected versions of $oCB$, say $mioCB1, mioCB2 ... mioCBn$. (The framework randomly injects the same mutant clone several times in order to check sensitivity of tools to clone location). The random fragment
CHAPTER 6. MUTATION / INJECTION-BASED FRAMEWORK

selector chooses \( m \) code fragments (say \( oCF_1, oCF_2 \ldots oCF_m \)) from the code base, and each of them will be mutated by each mutation operator \( dmOP \) producing mutated code fragments \( moCF_1, moCF_2 \ldots moCF_m \). Thus, the recall for mutation operator \( dmOP \) for tool \( T \) is given by:

\[
R_{dmOP}^{T} = \sum_{i=1}^{m \times n} R\left(oCF_i, moCF_i\right)_{T} / m \times n
\]

Similarly, for Type 1 clones, three mutation operators \((mCW, mCC, mCF)\) and their combinations \((mCW+mCC+mCF), (mCW+mCC), (mCW+ mCF), (mCC+ mCF)\) can be applied to the \( m \) code fragments (although note that if we allow operator repetition we could theoretically generate any number of combinations), each of which will be injected \( n \) times to the code base. Thus, recall of tool \( T \) for Type 1 clones can be defined as:

\[
R_{Typ1}^{T} = \sum_{i=1}^{m \times n \times (3+4)} R\left(oCF_i, moCF_i\right)_{T} / m \times n \times (3 + 4)
\]

The overall recall for tool \( T \) is the summary of recall for the \( l \) clone mutation operators and \( c \) combinations applied \( n \) times to \( m \) selected code fragments, given by:

\[
R_{overall}^{T} = \sum_{i=1}^{m \times n \times (l+c)} R\left(oCF_i, moCF_i\right)_{T} / m \times n \times (l + c)
\]

For tools that give textual results as clone classes rather than clone pairs, we simply check whether the original and mutated / injected code fragments are in the same class.

6.3.4 Measurement of Precision

In information retrieval, precision measures the noise in the results, such as irrelevant items appearing in the results of a query [134]. The relationship between recall and precision is
an indication of how well a matching engine performs. Ideally, precision should remain high as recall increases, but in practice this is difficult to achieve. The more the constraints of a query are relaxed to retrieve more relevant items (increasing recall), the more noise is present in the results (decreasing precision). Similarly, in clone detection the tunable parameters of a tool might be relaxed to detect a reference clone pair, but this can decrease the precision of the results, both for the reference pair in particular and for the tool in general.

Our precision definition is the usual one of IR research, that is, the number of correctly detected items divided by the total number of items detected. Applying precision measurement in the context of mutation-based analysis, we consider the total set of items to be all clone pairs (if any) reported for the injected mutant clone (the mutation-related clone pairs). We then automatically evaluate how many of the mutation-related clone pairs are valid clone pairs (Section 6.3.6) and use this as the number of correct items. However, unlike recall measurement, precision measurement is bit tricky since there might be already some other existing fragments that form clone pairs with the original fragment selected for mutation. Validation of such existing clone pairs is challenging and cannot be done automatically. We have thus used the above heuristics, only considering those that form clone pairs with the mutated code block and apply intelligent pair comparison to validate (Section 6.3.6) these mutation-related clone pairs.

Using the notation of the previous subsection, let us say that for a mutated code fragment $moCF$ created by mutation operator $dmOP$, a tool $T$ reports $k$ clone pairs, $(moCF, CF1)$, $(moCF, CF2)$ ... $(moCF, CFk)$ in mutant code base $mioCB$. If automatic validation reports that $v$ of these are valid, then the unit precision of the tool $T$ for the single injection of $moCF$ for clone type/mutation operator $dmOP$ is as follows:
As before, $moCF$ may be injected $n$ different times, and there may be $m$ different code fragments selected for mutation using mutation operator $dmOP$. Thus, the precision of tool $T$ for mutation operator $dmOP$ is given by:

$$P_{T}^{dmOP} \text{ w.r.t. single injection of } moCF = \frac{v}{k}$$

For Type 1 clones, the 3 mutation operators and 4 combinations may be applied $n$ times to the $m$ code fragments. Thus, precision for Type 1 clones can be calculated as:

$$P_{T}^{Typ1} = \frac{\sum_{i=1}^{n \times m} v_i}{\sum_{i=1}^{n \times m} k_i}$$

For overall precision of tool $T$ over $l$ mutation operators and $c$ combinations applied $n$ times to $m$ code fragments, we have:

$$P_{T}^{overall} = \frac{\sum_{i=1}^{n \times m \times (l+c)} v_i}{\sum_{i=1}^{n \times m \times (l+c)} k_i}$$

Again, for tools that return clone classes rather than clone pairs, we consider all the reported classes associated with the mutated / injected code fragment and form clone pairs from the member fragments of the clone classes.

### 6.3.5 Mapping of Code Fragments

In order to measure recall (Section 6.3.3), we need to be able to accurately tell if a tool has actually paired a mutant clone with its original, and in order to measure precision (Section
6.3.4), we additionally need to know which other reported pairs include the mutant clone. If all tools reported with exactly the same granularity and exactly accurate source locations, then this would be easy — but that is not the case. Thus the framework must be able to determine if a known code fragment is matched or subsumed by fragments in the reported clone pairs.

**Definition 4: Fragment containment.** We say that a code fragment $CF1$ is contained by another fragment $CF2$ if both are in the same file, and and the range of line numbers of $CF1$ is within the range of line numbers of $CF2$. In algorithmic form,

```java
boolean isContained(CF CF1, CF CF2) {
    return ((CF1.FileName == CF2.FileName)
            AND (CF1.EndLine <= CF2.EndLine))
}
```

Unlike Bellon et al. [32] and Baker [11], we do not consider partial overlapping of clone pairs in this study, because our mutant clones are exactly like the original except for the intended editing mutation. Thus, in a partial match, either the non-matching part of the clone pair is the mutated part, in which case the subject tool was unable to match the mutated code lines, or the original part, in which case it failed an exact match. However, if a detected clone pair subsumes the mutant-original clone pair by containment, then we consider that the mutant clone has been detected because similar surrounding code of the original and injected context may also match and be included by the tool. This strict non-overlap binary definition of detection is consistent with the mutation testing tradition of “killed” mutants.

The following algorithm implements our definition of detection for a mutant pair (MP) consisting of clone mutant $moCF$ of original fragment $oCF$ and a tool $T$’s clone candidate
set (CSet) of detected clone pairs $C$:

```java
boolean isDetected(MP (oCF, moCF), CSet C) {
    for each clone pair (CF1, CF2) in C {
        if ((isContained(oCF, CF1)
            AND isContained(moCF, CF2))
            OR (isContained(moCF, CF1)
                AND isContained(oCF, CF2)))
            return True;
    }
    return False;
}
```

To measure precision, we need to find all pairs in $C$ for which one of the fragments is the mutant clone $moCF$:

```java
CSet validateUs(CF moCF, CSet C){
    CSet ValidateMe = {};
    for each clone pair (CF1, CF2) in C {
        if (isContained(moCF, CF1)
            OR isContained(moCF, CF2))
            ValidateMe = ValidateMe + (CF1, CF2);
    }
    return ValidateMe;
}
```

If the function call `validateUs(moCF, C)` returns the empty set for a run of a particular tool, we do not need to validate any clone pairs. For all non-empty results from `validateUs(moCF, C)` we may need to validate all clone pairs it returns (Section 6.3.6) except the mutant clone pair itself.
6.3.6 Validation of Clone Pairs

While recall measurement is completely automatic and no validation of clone pairs is required, to accurately measure precision we need to validate those few clone pairs that are associated with the mutant code fragment. To partially automate this, we have developed a clone pair validator based on our NICAD [185, 186] tool, using standard pretty-printing (to remove *Type 1* formatting differences), flexible code normalization (to equalize *Type 2* variation in code fragments) and a dissimilarity threshold (to relax the comparison to allow for *Type 3* line insertions/deletions) to validate clone pairs. However, unlike NICAD, the validator is not itself a clone detector, both because it works on a specific given clone pair and because it is aware of the mutation operator that was used to produce the mutant clone. Using this information, it can tailor its normalization to make the validation task relatively straightforward.

For example, when only mutation operators of *Type 1* clones are used, the validator only applies standard pretty-printing and compares the pair text line-wise. Similarly, when only mutation operators of *Type 3* are applied, the validator allows a corresponding dissimilarity threshold [185] in addition to standard pretty-printing of the fragments to relax the similarity between the code fragments of the subject clone pair, and so on. Thus because the validator is aware of the possible differences between the two code fragments, it can accurately measure their real similarity. However, for mutation operators of *Type 4* clones, it is not always possible to automatically validate a clone pair. Fortunately, in these few cases the validator can accurately identify those clone pairs for which manual validation is required based on their similarity values.

As part of our first experiment (Section 6.4), we hand validated more than 500 clone pairs (of all types) as a secondary check. The clone validator neither falsely accepted nor
falsely rejected any clone pairs, and correctly tagged the small number of Type 4 pairs requiring manual validation.

6.3.7 Other Issues

Measurement of Time and Memory Requirements: Measuring time and space requirements of tools seems easy since one needs only to run the tools on large systems, and most tool authors have extensively evaluated these properties of their tools. Similarly, memory requirements are generally not a problem since most state-of-the-art tools use linear space. However, our framework can report fine-grained time and memory requirements of subject tools for different clone types, something others have not reported.

Scalability of the Framework: The framework can work with subject systems of any size, depending only on the scalability of the subject tools. While a mutation-based technique inherently implies a large number of tool runs, the framework is built to take advantage of multi-processor machines to balance the load of runs over as many processors as are available. In future we plan to extend this to networks of distributed processors as well.

Adapting Tools to the Framework: The only requirement for adapting a third party tool to the framework is that the tool should be run from the command line (most tools usually do or at least have such an option in addition to their graphical output) and that it should provide a textual report of the detected clones, either in XML format or the usual column-oriented textual format of full file name and begin/end line numbers of the code fragments of the candidate clone pairs. The framework can run subject tools with varying tunable parameters suitable for identifying different types of clones. The framework is language-specific, and currently supports only C, Java and C#. However, since the majority of the framework is language independent, it is simple to add new languages.
Table 6.2: Overview of the subject code bases

<table>
<thead>
<tr>
<th>Language</th>
<th>Code Base</th>
<th>LOC</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Gzip-1.2.4 [90]</td>
<td>8K</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>Apache-httpd-2.2.8 [9]</td>
<td>275K</td>
<td>4301</td>
</tr>
<tr>
<td></td>
<td>Weltab [31]</td>
<td>11K</td>
<td>123</td>
</tr>
<tr>
<td>Java</td>
<td>Netbeans-Javadoc [31]</td>
<td>14K</td>
<td>972</td>
</tr>
<tr>
<td></td>
<td>Eclipse-jdtcore [31]</td>
<td>148K</td>
<td>7383</td>
</tr>
<tr>
<td></td>
<td>JHotDraw 5.4b1 [101]</td>
<td>40K</td>
<td>2399</td>
</tr>
</tbody>
</table>

6.4 An Example Use of the Framework

In order to test the framework, both in evaluating a single tool and in comparing a set of tools with different subject systems in a variety of languages, we have conducted several studies of our own NICAD [185] clone detector and its variants in detecting function-level clones. In the following we briefly report our findings.

**Subject Code Bases:** To test the framework with different code bases and languages, we have chosen several open source subject systems written in C and Java. Table 6.2 provides a statistical overview of these subject systems (only C and Java files are counted in the calculations).

**Single Tool Evaluation:** One of the main objectives of the framework is to evaluate the performance of single tools on different types of clones. To examine whether the framework works well with single tools we have used our Basic NICAD variant (see below) and obtained recall and precision values for different types of clones as shown in the Basic NICAD column of Table 6.3.

**Multiple Tool Comparison:** Next we wanted to see how the framework would handle multiple tools at a time on the same set of mutants. For this test we used three different versions of NICAD. NICAD has three variants. The first is Basic NICAD, which is largely
Table 6.3: Recall and precision of the Tools (%)

<table>
<thead>
<tr>
<th>Type</th>
<th>Mutator</th>
<th>Basic NICAD</th>
<th></th>
<th></th>
<th>FlexP NICAD</th>
<th></th>
<th></th>
<th>Full NICAD</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Recall</td>
<td>Precision</td>
<td>Recall</td>
<td>Precision</td>
<td>Recall</td>
<td>Precision</td>
<td>Recall</td>
<td>Precision</td>
<td></td>
</tr>
<tr>
<td>Type 1</td>
<td>mCW</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mCC</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mCF</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Type 1 overall</td>
<td></td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 2</td>
<td>mSRI</td>
<td>29</td>
<td>96</td>
<td>32</td>
<td>92</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mARI</td>
<td>31</td>
<td>95</td>
<td>28</td>
<td>91</td>
<td>100</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mRPE</td>
<td>28</td>
<td>94</td>
<td>31</td>
<td>95</td>
<td>100</td>
<td>97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 2 overall</td>
<td></td>
<td>29</td>
<td>94</td>
<td>27</td>
<td>94</td>
<td>100</td>
<td>97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 3</td>
<td>mSIL</td>
<td>96</td>
<td>97</td>
<td>95</td>
<td>97</td>
<td>100</td>
<td>98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mSDL</td>
<td>96</td>
<td>97</td>
<td>95</td>
<td>97</td>
<td>100</td>
<td>98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mILs</td>
<td>96</td>
<td>84</td>
<td>91</td>
<td>76</td>
<td>100</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mDLs</td>
<td>96</td>
<td>82</td>
<td>93</td>
<td>78</td>
<td>100</td>
<td>94</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mMLs</td>
<td>94</td>
<td>84</td>
<td>92</td>
<td>75</td>
<td>100</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 3 overall</td>
<td></td>
<td>95</td>
<td>85</td>
<td>94</td>
<td>81</td>
<td>100</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 4</td>
<td>mRDS</td>
<td>71</td>
<td>82</td>
<td>66</td>
<td>82</td>
<td>76</td>
<td>90</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mROS</td>
<td>69</td>
<td>83</td>
<td>67</td>
<td>79</td>
<td>78</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type 4 overall</td>
<td></td>
<td>67</td>
<td>81</td>
<td>67</td>
<td>79</td>
<td>77</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Overall</td>
<td></td>
<td>87</td>
<td>90</td>
<td>84</td>
<td>89</td>
<td>96</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

language independent, applies standard pretty-printing to the source and uses a sequence matching algorithm with dissimilarity thresholds for detecting clones. (This variant was used in a recent large empirical study [186] with good results.) The second variant, Flexible Pretty-Printed NICAD or FlexP NICAD [185], adds “flexible” pretty-printing, which reformats to isolate potential changes to different lines, and then uses the same sequence matching algorithm with dissimilarity thresholds. The third variant, Full NICAD, utilizes the entire range of NICAD capabilities, including flexible pretty-printing, flexible code normalization, clone filtering and the sequence matching algorithm with dissimilarity thresholds. Our objective was to evaluate and compare the relative capabilities of these three variants using the proposed framework.

Although we have run the experiment with several subject systems, we only have room for a sample of the results. Table 6.3 shows the results that we obtained using the framework.
using Weltab [31] as the subject system with each of the three NICAD variants as subject tools. In this experiment, 20 functions were randomly selected from Weltab and each of the mutation operators was applied 500 times on those 20 functions, yielding 10,000 mutant code bases. Combinations of the mutation operators were also used. These results are typical of those obtained for NICAD and its variants on other systems. In these results we can see that Basic NICAD has relatively poorer recall for clones generated by mutation operators of Type 2 clones (which is not surprising since it was not designed to detect them). FlexP NICAD is also not a good choice for these, and thus full NICAD is the best option, although there is a cost to its language-specific transformation rules.

6.5 Related Work

To our knowledge, ours is the first attempt to provide a fully automated means for evaluating and comparing clone detection tools. However, several tool authors have done partial evaluation of the precision of their tools, and a number of experiments have attempted to compare tools in terms of precision, recall, time and space. In this section, we outline their evaluation strategies and compare them with ours.

**Tool Comparison Experiments:** Bailey and Burd [41] compared three state-of-the-art clone detection and two plagiarism detection tools using hand validation of clone candidates. Although they were able to verify all of the clone candidates, the modest size of the subject system, focus on preventive maintenance tasks and reliance on unguided subjective judgement limits the generality of their conclusions.

In an attempt to overcome the limitations of Burd and Bailey’s study, Bellon et al. [31, 32] conducted a larger tool comparison experiment on the three clone detection tools used in the Burd and Bailey study as well as three additional tools, and with a much larger
CHAPTER 6. MUTATION / INJECTION-BASED FRAMEWORK

and more diverse set of subject systems in C and Java and totalling 850 KLOC. While their study is the most extensive to date, only a small proportion of the clone candidates (about 2%) were oracled, and a number of other factors may have influenced the results [11]. Bellon’s framework has since been reused by Koschke et al. [71, 137] and Ducasse et al. [67], with similar results.

Rysselberghe and Demeyer [207] evaluated and compared prototypes of three representative clone detection techniques with respect to portability, kinds of duplication reported, scalability, number of false matches, and number of useless matches. However, their evaluation is qualitative rather than quantitative, focussing on refactoring opportunities rather than general clones. It is also based on small systems and is limited by subjectivity in validation.

Finally, a recent study by Bruntink et al. [36], evaluates several clone detection techniques in identifying known cross-cutting concerns in C programs with homogeneous implementations.

Our work differs from all these experiments in its full automation, its formal clone definition based on an editing taxonomy, its use of mutation and injection to provide large numbers of known and pre-validated clones, its quantitative statistical evaluation based on accepted mutation analysis techniques, and its independence from the target task.

Single Tool Evaluation Strategies: Most text-based techniques [68, 109, 161] have been published with only example-based evaluation, reporting neither precision nor recall. Our own recent text-based tool, NICAD [185] was initially evaluated for precision and recall by manually analyzing and injecting different types of clones into two small C systems. While this evaluation did yield defensible results, the small size of the subject systems and heavy use of manual work does not allow for generalization.
Among the token-based tools [15, 113, 156], only \textit{CP-Miner} [156] was evaluated for precision, by manually examining 100 randomly selected copy/paste segments. Recall was only qualitatively estimated, on the basis of finding more clone pairs than \textit{CCFinder} [113].

The abstract syntax tree (AST) based tool \textit{cpdetector} [71, 137] has been evaluated in terms of both precision and recall for clone Types 1 and 2 by adapting Bellon’s framework (and thus inheriting the same limitations). Of the other AST-based tools [26, 70, 105], only \textit{Deckard} [105] was evaluated for precision, by manually validating 100 randomly selected clone groups. While recall was not measured, it reports of detecting more cloned lines than \textit{CloneDr} [26] and \textit{CP-Miner} [156].

Among the metrics-based techniques [133, 134, 162], only Kontogiannis [134] has evaluated the effectiveness of different metrics in his IR-based approach in terms of precision and recall, using subject systems that were manually tagged with known clones by the developers.

The new graph-based approach for semantic clones of Gabel et al. [76] is aimed at the scalability problems of previous graph-based techniques. To estimate precision they randomly sampled 30 clone groups per experiment and manually verified each of them as clones. While they report detecting more clones than \textit{Deckard}, they did not study recall. Previous graph-based methods [129, 143] similarly used random validation for precision but did not report recall.

Our proposed framework provides both recall and precision in an automated, statistically sound way for a range of different types of fine-grained clones. It allows for complete experiments free of subjective opinions and hand analysis, yielding comparable evaluations that overcome the limitations of previous attempts at both validating individual tools and comparing sets of tools.
CHAPTER 6. MUTATION / INJECTION-BASED FRAMEWORK

6.6 Summary

Existing methods for evaluating (and comparing) clone detection tools suffer from several limitations. We have presented an evaluation framework that uses code fragment mutation to artificially create and inject known code clones that can be used to accurately measure recall and precision of clone detection tools. The framework is based on an editing taxonomy that is used to synthesize artificial clones by mimicking developers’ typical editing activities in clone creation. The framework is capable of evaluating and comparing recall of clone detection tools for various languages and clone types with no need for manual intervention, and can evaluate precision either automatically using language-dependent validation rules, or semi-automatically using an interface with minimal manual intervention. An example study of evaluating basic NICAD (for single tool evaluation) and three variants of NICAD (for tool comparison) has been successfully conducted using the framework.

As future work, we are already in the planning stages of a large scale tool comparison experiment based on the framework by inviting participation from all current tool authors.
Chapter 7

An Empirical Study of Near-miss Function Clones in Open Source Software

Although NICAD gave good results in our first study (Chapter 3), it was only evaluated with relatively two small C systems. NICAD also gave good results (high precision and recall) when evaluated with the mutation-based framework (Chapter 6). Again, the systems we used in the mutation analysis were relatively small-medium sized. Furthermore, the framework is only for automatically evaluating tools and not for detecting actual clones of the subject systems. On the other hand, there is a marked lack of in-depth comparative studies of cloning, particularly in a variety of systems. In this chapter, we have thus conducted a large scale empirical study with large open source systems of three different languages, C, Java and C# in order to both evaluate NICAD for large systems of different languages and to study the cloning status of those systems in several different dimensions. Experimental results shows that NICAD is capable of detecting clones from large systems.
including Linux kernel and from systems of different languages and that there seem to be a large amount of copy/pasted function clones in those systems.

The chapter is organized as follows. After providing a general motivation in Section 7.1, we provide the experimental setup of our study in Section 7.2. Section 7.3 presents and analyzes our findings, and Section 7.4 considers other empirical studies and their relation to ours. Finally, Section 7.5 concludes the chapter with directions of future research.

### 7.1 Motivation

Over the past decade several techniques and tools for detecting code clones have been proposed [182, 184, 188]. Despite a decade of active research in the detection and analysis of clones, there has been a marked lack of in-depth comparative studies of cloning, particularly over a variety of systems. There have been many empirical studies on cloning. Every new technique comes with some sort of empirical validation [184], and empirical studies are used when comparing clone detection tools [31, 32, 71, 41, 206, 207]. However, in both cases the focus is on validating or comparing the techniques rather than the clone properties of the subject systems themselves. Particular subject systems have also been analyzed with respect to maintenance issues such as the harmfulness or usefulness of clones [121, 10, 125, 142], taxonomies of clones [117, 115] and evolution of clones [8], and there has been one in-depth study [178] on exact clones in web applications.

In this chapter we present an in-depth study of near-miss function clones in open source software using NICAD. We examine more than twenty open source C, Java, and C# systems, including the entire Linux Kernel, Apache httpd, and J2SDK-Swing and compare their use of cloned code in several different dimensions, including language, clone size, clone similarity (with varying thresholds), clone location and clone density by proportion
of cloned functions. In particular, we focus on the following four research questions in this study:

1. Is NICAD capable of detecting clones from large systems? Can it detect clones from systems of different languages?
2. What is the cloning status of open source systems? Are there many clones? Are there more near-miss clones than exact clones?
3. Are there significant differences in cloning between the different language paradigms used in open source systems?
4. Are there significant differences in cloning in large systems compared to medium and small-sized open source systems?

We manually verify all detected clones and provide a complete catalogue of different clones in an online repository in a variety of formats. These validated results can be used as a cloning reference for these systems and as a benchmark for evaluating other clone detection tools.

NICAD [185] has previously been found to be effective in detecting near-miss function clones (Chapter 3). However, in that work was applied to only two small C systems, focusing on its efficacy in detecting copy/pasted near-miss clones by using flexible pretty-printing, code normalization and code filtering. In this chapter, we exploit further improvements to NICAD to deal with large systems, using a dynamic clustering technique and distributing the comparison load across multiple processors, and an adaptation to its use on Java and C# systems.

Although NICAD is designed to allow for flexible pretty-printing, code normalization and filtering, in this study we focus on detecting only exact and near-miss function clones, using only the basic NICAD technique, consisting of standard pretty-printing of code fragments to encode structure, followed by text line comparison at a variety of dissimilarity thresholds.
Our study demonstrates that NICAD is capable of detecting clones in very large systems in many different languages, and that there is a significant proportion of code in these systems that has been reused by copy/paste.

7.2 Experimental Setup

In this experiment we have applied NICAD to find function clones in a number of open source systems. We have then used a set of metrics to analyze the results. This section introduces the systems we have studied and the metrics used, including a brief overview of our definition and methodology for manual verification of the detected clones.

7.2.1 Subject Systems

In this study we have analyzed ten C, seven Java and six C# systems varying in size from 4K LOC to 6265K lines of code (LOC) and including the entire Linux Kernel. In Table 7.1 we provide a statistical overview of these subject systems (only .c, .java and .cs files were considered in the calculations).

Because Bellon’s experiment [31] is the most extensive to date, we have chosen all the C and Java systems of his experiment including the systems used in his test run. In addition, we have studied Apache httpd [9], JHotDraw [101], the entire Linux Kernel [152] and a number of smaller systems. We have also added six C# systems of different sizes and kinds, including db4o, a popular and widely used production commercial and open source object database. Although it is partly adapted from its Java version, it differs significantly from the original and is the largest C# system ever studied for clones.

Since the Linux kernel is almost two orders of magnitude larger than any of the other
systems, we have treated it as an outlier. We provide statistical results both including and not including Linux in Section 7.3.1, and in later subsections we have dropped Linux from averages and provided Linux results separately to avoid any bias due to its exceptionally large size.
CHAPTER 7. AN EMPIRICAL STUDY OF FUNCTION CLONES

7.2.2 Clone Definition

In this study we have considered all non-empty functions of at least 3 LOC in pretty-printed format (function header and opening bracket on the first line, at least one code line, and ending bracket on the third line). Empty functions, which are common in some systems, have intentionally not been considered. We then use different UPI (difference) thresholds (Chapter 3) to find exact and near-miss (copy/paste/modify) function clones. For example, if the UPI threshold is 0.0, we detect only exact clones; if the UPI threshold 0.10, we detect two functions as clones if at least 90% of the pretty-printed text lines are the same (i.e., if they are at most 10% different – see Chapter 3). In this thesis we present our results for the representative set of UPI thresholds 0.0, 0.10, 0.20 and 0.30, although we have also tested 0.05, 0.15, 0.25 and 0.35 in our work.

7.2.3 Validation of Clones

All clones detected in this study were validated by hand. To validate detected clones we use a two-step process. First, we use NICAD’s interactive HTML web page output to give an overall view of the original source of the clone classes. Second, we use the XML output to pairwise compare the original source of the functions in each clone class using Unix diff to determine the textual similarity of the original source. We then manually check all code clone pairs that have lower similarity values w.r.t the UPI threshold chosen. Because of our concise interactive HTML view and tool support for comparing original source, manual validation is not time-consuming, and the total time to manually validate all clones in this experiment was less than one person-month.

We should note that it was not our intention to measure the recall of our tool NICAD in this study, since we did not know the clone status of the systems before we began. However,
NICAD has previously been shown to give high recall in both our first experiment (Chapter 3) and using our mutation-based evaluation framework (Chapter 6).

In terms of precision, our hand validation of all detected clones did not find any false positives at UPI threshold 0.2 or lower. However, at UPI threshold 0.3 and higher, we noticed that some of our reported clones might be considered false positives, even though they meet our clone definition above. This issue arises specially because of NICAD’s handling of preprocessor directives in C systems.

For example, the two code fragments (of 45 LOC and 5 LOC respectively) shown in Figure 7.1 are detected as exact clones in our study because during if-def resolution, we
comment out the *else* part of the *if-def*. It is not clear whether or not these two are truly clones. By the number of lines and the contents of the *else* part of the first function, they are obviously not clones. On the other hand, it seems quite possible that the bottom fragment was copied and the *if-def* directives were added later on (or vice-versa). However, the number of such cases was very few in our study. We thus say that NICAD gives about 100% precision in this study. Of course, we are using the basic version of NICAD in this study which is without any significant source transformation and that we are dealing with only function clones and therefore, we expect about 100% precision too.

### 7.2.4 Time and Space Complexities of NICAD

In Table 7.2 we provide the timing statistics for running *NICAD* on several of the example systems. The timing statistics are taken when we ran the systems on an Intel(R) Core 2 Quad processor @2.66GHz with 4GB RAM running Ubuntu 7.10. The third column shows the number of potential clones (i.e., functions of 3 or more pretty-printed lines) for the subject system shown in the second column. The fourth column shows the number of comparisons that may be required in the worst case, and fifth and sixth columns show the number of actual comparisons made and the time in minutes taken to find exact clones (i.e., UPI threshold 0.0). Finally, the seventh and eighth columns show the number of actual comparisons and time in minutes taken to find near-miss clones, with UPI threshold 0.30.

From the table we see that even though *NICAD* does not find clones from large systems in seconds, it does find in minutes requiring no post-processing or manual analysis of the results. Moreover, when the same experiments were conducted on single CPU Intel(R) Pentium(R) 4 @2.40GHz, 1.5GB RAM Linux desktop machine or single CPU Intel(R) Pentium(R) M processor 1.73GHz, 512MB RAM Linux laptop, the timing requirements
were mostly similar for small/medium sized systems. For comparatively large systems (e.g., here jdtcore or swing), high-speed multi-processor machine performs much faster. As we cache only those cloned fragments that need to be compared with the exemplar in question, memory requirements was not a problem at all. However, for the largest system Linux kernel, it took more than three days for finding exact clones and four days for finding near-miss clones (with UPI threshold 0.3).

### 7.2.5 Metrics and Visualizations

This subsection describes the different metrics and visualizations that we have used in this experiment. These metrics are either adapted or reused from previous studies of cloning [178, 5, 117, 204, 180].

**Total Cloned Methods (TCM):** In this study we focus on function clones, and thus
our first metric is related to the number of methods. By $TCM$ we mean the total number of cloned methods of a system for a given UPI threshold (after manual verification). $TCMp$ is the percentage $TCM$ of the total number of methods in a system. A higher value of $TCMp$ corresponds to a higher percentage of cloning in the system w.r.t the number of methods. For example, if the $TCMp$ of a system is greater than 50% with UPI threshold 0.0, we can say that the system has more exact cloned methods than non-cloned methods. Such systems have a high update anomaly risk; every update to the system has a higher chance of involving a clone than not.

Since methods can be of different sizes and there may be many clones that are quite small, we also consider similar metrics w.r.t the number of lines in the systems. We define $TCLOC$ as the total number of cloned lines of a system for a given UPI threshold and $TCLOCp$ as the percentage of total number of lines of the system for a given UPI threshold. Since we apply standard pretty-printing before clone detection, which eliminates formatting and layout differences, resolves #ifdefs (in C systems) and ignores comments, we can get an accurate percentage of cloned lines. We thus define the similar metrics w.r.t standard pretty-printed lines of code as $TCSppLOC$ and $TCSppLOCp$ respectively. In practice, there is not much difference between $TCLOCp$ and $TCSppLOCp$, but at the same time $TCSppLOCp$ gives a more accurate measure. We thus provide our findings w.r.t $TCSppLOCp$ rather than both.

**File Associated with Clones (FAWC):** While the above metrics provide the overall cloning statistics for a subject system, they cannot provide any clue as to whether the clones are from some specific files or scattered all over the system among many files. With FAWC we provide these statistics for each system at each UPI threshold. We consider that a file is associated with clones if it has at least one method that forms a clone pair with another
method in the same file or in a different file. We define \( FAWCp \) as the percentage of files associated with clones of a system for a given threshold. For example, “\( FAWCp \) of a systems \( x \) with UPI threshold 0.0 (exact clones) is 50%” means that 50% of the files of \( x \) contain at least one exact cloned method. From a software maintenance point of view, a lower value of \( FAWCp \) is desirable, as in this case clones are localized to certain specific files and thus may be easier to maintain.

**Cloned Ratio of File for Methods (CRFM):** While \( TCM \) related metrics provide a good indication of the overall cloning level and \( FAWCp \) hints at the overall localization of the clones, still one cannot say which files contain the majority of the clones in the system. With CRFM we attempt to discover the highly cloned files. In particular, for a file \( f \), \( CRFM(f) \) is defined as follows:

\[
CRFM(f) = \frac{\text{Total number of cloned methods in file } f}{\text{Total number of methods in file } f} \times 100
\]

Where a method is considered to be a *cloned method* if it forms a clone pair/clone class with another method(s) of the same file (e.g., for near-miss clones) or another file (within the same directory or a different directory) and total number of methods in file \( f \) denotes the number of methods of \( f \) that are 3 LOC or more in standard pretty-printed format. Similar metrics are defined w.r.t the lines of code (\( CRFLOC \)) and standard pretty-printed lines of code (\( CRFSppLOC \)). These metrics are similar to (but not same as) the FSA metric of Rajapakse and Jarzabek [178] and the RSA metric of Ueda et al. [204], although they ignore clones that form clone pairs within the same file and we do not.

With CRFM we can determine the highly cloned files of a system and possibly can also predict the maintenance difficulty based on the metric values. For example, consider two systems \( x \) and \( y \) of similar size, both having the same values for the \( TCM \) related metrics. In \( x \), clones are scattered across the system in such a way that no two files are substantially
similar. But in $y$, clones are well concentrated into a certain set of files. From a clone treatment perspective, system $y$ is more interesting than $x$ because the clones in $y$ might be more easily treatable than those of $x$.

**Qualifying File Count for Methods (QFCM):** As in Rajapakse and Jarzabek [178] we define $QFCM(v)$ for $CRFM$ value $v$ as the number of files for which $CRFM$ is not less than $v$. For example, $QFCM(20\%)$ gives the number of files in the system having a $CRFM$ value not less than $20\%$. $QFCMp$ is $QFCM$ expressed as a percentage of the total number of files in the system. For example, “$QFCMp(30\%+) = 28\%$ for a system $x$ with UPI threshold 0.0” means that $28\%$ of the files of $x$ have $30\%$ or more exact cloned methods. As usual we define similar metrics for source lines of code ($QFCLOC$ and $QFCLOCp$) and for standard pretty-printed lines of code ($QFCSppLOC$ and $QFCSppLOCp$).

**Profiles of Cloning Locality w.r.t Methods (PCLM):** Kapser and Godfrey [117] provide three types of function clones based on their location — clone pairs in the same file (category 1), in the same directory (category 2) and in a different directory (category 3). They also provide the reasons, usefulness / harmfulness for each of these categories [117]. In this study we define three metrics, $PCLM(1)$ for category 1, $PCLM(2)$ for category 2 and $PCLM(3)$ for category 3 where $PCLM(i)$ gives the total number of clone pairs for category $i$. Furthermore, $PCLMp(i)$, the percentage clone pairs for category $i$ is defined as follows:

$$ PCLMp(i) = \frac{PCLM(i) \times 100}{\text{Total number of clone pairs in the system}} $$

As usual similar metrics are defined with respect to lines of code ($PCLLOC$ and $PCLLOCp$). Furthermore, the metrics values are calculated for a range of different UPI thresholds.

**Profiles of Remote Cloning Locality w.r.t Methods (PRCLM):** In order to study the further insights of cloning locality, we define three more metrics for the remote clone pairs
(those are in different directories). Two fragments of a clone pair that are neither within the same file nor under the same directory (i.e., do not have the same parent), might have the same grandparent directory (category 1), or they might be under the same subsystem (category 2) (note that category 1 is essentially a subset of category 2) or in the worst case, they might be under different subsystems (category 3). Similarly to the above definition, we define three metrics for the three categories, \( PRCLM(1) \) for category 1, \( PRCLM(2) \) for category 2 and \( PRCLM(3) \) for category 3 where \( PRCLM(i) \) gives the total number of clone pairs for category \( i \). Furthermore, \( PRCLMp(i) \), the percentage clone pairs for category \( i \) is defined as follows:

\[
PRCLMp(i) = \frac{PRCLM(i)}{\text{Total number of different directory clone pairs in the system}} \times 100
\]

As usual similar metrics are defined with respect to lines of code (\( PRCLLOC \) and \( PRCLLO Cp \)). Furthermore, the metrics values are calculated for a range of different UPI thresholds.

### 7.3 Experimental Results

In this section we provide the experimental results of this study starting from overall cloning level in C, Java, and C# systems and then for each individual system in a variety of measures based on the metrics described in Section 7.2.5. While we provide here only the overall findings and statistical measures, the detailed results and the raw data for each systems for different UPI thresholds can be found in an online repository [183] as XML databases and HTML website.
7.3.1 Overall Cloning Level

In this section we provide the overall cloning level, both in terms of number of methods and in terms of number of pretty-printed LOC (i.e., the values of the TCM-related metrics of subsection 7.2.5). Figure 7.2a summarizes our results for the C, Java and C# systems by the proportion of functions (or methods in the case of Java) that are cloned (i.e., $TCM_p$ over languages). The corresponding values for the $TCS_{ppLOC_p}$ metric (i.e., the proportion of clones by number of pretty-printed LOC for each language) can be found in the %Total rows of Table 7.3.

The first thing we can notice is that there is significantly more function cloning in our open source Java and C# systems than in C. On average, about 15% (7.2% w.r.t LOC) of the methods in open source Java programs are exact clones — those with no changes at all (except changes in formatting, whitespace and comments), whereas only about 2.5% (1.1% w.r.t LOC) of C functions are exact clones. After detecting clones by setting a size range for the function size (see [183] for detailed results) we noticed that this is possibly in large part due to the large number of small accessor, iterator and utility methods in Java and C# programs that are not present in C.

When we plot the percentage of exact clones (with UPI threshold 0.0) for different clone sizes (number of pretty-printed source lines) (Figure 7.4), we can see that both Java and C# show similar percentages of clones for similar clone sizes, both higher than in C systems. While it is difficult to provide the exact statistics for the types of smaller methods for all the systems, we manually examined the small clones of the systems and found that there are in fact many accessor methods in Java and utility / enumerator methods in C# systems. For example, the two methods (Fragments 1 and 2) in Figure 7.3 both appear 34 times in 34 different files in the C# system Nant.
Figure 7.2: Proportion of cloned functions/methods in the systems
Chapter 7. An Empirical Study of Function Clones

Figure 7.3: Example of small functions/methods in C# and Java

Similarly, in Java system eclipse-ant, Fragment 3 appears 9 times in 9 different files and Fragment 4 appears 10 times in Swing file plaf/basic/BasicTreeUI.java. For C systems we did not find that many small methods except httpd and Linux.

The second thing we can notice in Figure 7.2a (and in the % Total rows of Table 7.3 for LOC) is that the effect of increasing the UPI threshold is almost identical in the languages. We can interpret this as meaning that the numbers of small changes made to cloned functions in each of these languages seem to be roughly the same in these systems. This is in some ways surprising - there is no particular reason why the pattern of changes to copied code should be similar across languages. This is even more surprising for C# systems at UPI threshold 0.3, with much more clones than C and Java systems.

Figures 7.2b, 7.2c and 7.2d (also columns 3 to 6 of Table 7.3) refine Figure 7.2a to show a detailed view of the same information for the individual open source C, Java and C# systems respectively.
CHAPTER 7. AN EMPIRICAL STUDY OF FUNCTION CLONES

Figure 7.4: Percentage of exact clones by clone size

As expected, the overall trends for each language are much like the summaries in Figure 7.2a (or the % Total rows of Table 7.3), with lower levels of cloning in C than Java and C#. However, we can now see more. For example, we can see that several of the C systems have very little exact function cloning at all — less than 10%, and to a large extent independently of the system size (e.g., Postgresql and Apache httpd are very large, whereas snns, cook and Weltab are quite small). Considering the fact that Linux system is larger than any other subject systems, we also provide the average for C language systems without Linux. It is interesting to see that there is not much difference whether we include or drop Linux in the calculations showing the fact that the cloning characteristics of the Linux Kernel is not significantly different from any other kind of C systems.

Figure 7.2c (also Table 7.3 for LOC) is even more interesting, because while the C and C# systems vary, we can see that the Java systems are remarkably consistent in their cloning characteristics. All begin with a relatively high level of exact method clones (between 8 and 22 percent), and in all cases allowing for changes increases the proportion only modestly.
What’s interesting is that this seems to be completely independent of system size, and appears to be a characteristic of the language. The only exception to this consistency seems to be JDTcore, which has about twice as many clones at the 0.30 dissimilarity (UPI threshold) level than exact clones. While Java and C# systems share some cloning behaviour, they are not consistent. More research will be needed to investigate this phenomenon and compare to other object-oriented languages.

In Table 7.3 we also provide the number of clone pairs and clone classes for each of

### Table 7.3: Percentage cloned LOC, clone pairs and clone classes

<table>
<thead>
<tr>
<th>Lang</th>
<th>System</th>
<th>% Cloned LOC</th>
<th>No. of Clone Pairs</th>
<th>No. of Clone Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>T=0.0</td>
<td>T=0.1</td>
<td>T=0.2</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cook</td>
<td></td>
<td>0.3</td>
<td>2.0</td>
<td>7.7</td>
</tr>
<tr>
<td>Httpd</td>
<td></td>
<td>2.1</td>
<td>4.1</td>
<td>6.2</td>
</tr>
<tr>
<td>Postgresql</td>
<td></td>
<td>0.1</td>
<td>1.0</td>
<td>4.3</td>
</tr>
<tr>
<td>Sns</td>
<td></td>
<td>3.2</td>
<td>6.2</td>
<td>13.3</td>
</tr>
<tr>
<td>Wetlab</td>
<td></td>
<td>21.0</td>
<td>55.2</td>
<td>62.7</td>
</tr>
<tr>
<td>Wget</td>
<td></td>
<td>0.0</td>
<td>1.3</td>
<td>1.7</td>
</tr>
<tr>
<td>Linux</td>
<td></td>
<td>1.0</td>
<td>2.6</td>
<td>8.3</td>
</tr>
<tr>
<td>% Total C+</td>
<td></td>
<td>1.1</td>
<td>2.8</td>
<td>8.4</td>
</tr>
<tr>
<td>% Total C*</td>
<td></td>
<td>2.0</td>
<td>4.7</td>
<td>8.6</td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ant</td>
<td></td>
<td>5.1</td>
<td>5.4</td>
<td>6.3</td>
</tr>
<tr>
<td>EIRC</td>
<td></td>
<td>7.2</td>
<td>7.2</td>
<td>7.7</td>
</tr>
<tr>
<td>Javadoc</td>
<td></td>
<td>10.8</td>
<td>12.6</td>
<td>18.6</td>
</tr>
<tr>
<td>Jdtcore</td>
<td></td>
<td>5.1</td>
<td>8.8</td>
<td>16.2</td>
</tr>
<tr>
<td>JHotDraw</td>
<td></td>
<td>7.6</td>
<td>8.28</td>
<td>12.0</td>
</tr>
<tr>
<td>Spule</td>
<td></td>
<td>2.0</td>
<td>2.7</td>
<td>3.1</td>
</tr>
<tr>
<td>Swing</td>
<td></td>
<td>9.4</td>
<td>11.0</td>
<td>15.0</td>
</tr>
<tr>
<td>% Total Java</td>
<td></td>
<td>7.2</td>
<td>9.4</td>
<td>14.4</td>
</tr>
<tr>
<td>C#</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VMulti</td>
<td></td>
<td>3.9</td>
<td>15.2</td>
<td>16.1</td>
</tr>
<tr>
<td>Linq</td>
<td></td>
<td>3.4</td>
<td>3.8</td>
<td>8.3</td>
</tr>
<tr>
<td>Nant</td>
<td></td>
<td>3.7</td>
<td>5.8</td>
<td>12.8</td>
</tr>
<tr>
<td>RSS</td>
<td></td>
<td>9.8</td>
<td>11.7</td>
<td>15.3</td>
</tr>
<tr>
<td>Castle</td>
<td></td>
<td>5.4</td>
<td>7.6</td>
<td>15.5</td>
</tr>
<tr>
<td>db4o</td>
<td></td>
<td>4.8</td>
<td>5.4</td>
<td>10.6</td>
</tr>
<tr>
<td>% Total C#</td>
<td></td>
<td>6.0</td>
<td>7.6</td>
<td>13.3</td>
</tr>
<tr>
<td>% Total Overall*</td>
<td></td>
<td>4.9</td>
<td>7.1</td>
<td>11.9</td>
</tr>
</tbody>
</table>

+ means with Linux, and * means without Linux
the systems for varying UPI thresholds. It is interesting to notice that most systems have significantly fewer clone classes than clone pairs, indicating the fact that there are many pairs of functions in the systems that are similar to each other with higher numbers for Java and C# systems. It is also interesting to see that while average number of clone pairs per clone class (see the average row for each language) is more or less consistent for C and Java systems for different UPI thresholds, there is a surprising number (39.7 clone pairs per clone class) for C# systems with UPI threshold 0.3, indicating the fact cloning of the same functions/methods was frequent in the C# systems with significant amount of editing/adapting in the copy/pasted code.

7.3.2 Clone Associated Files

The $FAWC$ and $FAWCp$ metrics of Section 7.2 address the issue of what proportion of the files in a system is associated with clones, that is, contains at least one cloned method. A system with more clones but associated with only a few files is in some sense better than a system with fewer clones scattered over many files from a software maintenance point of view. In this section, we examine the $FAWCp$ metrics for each of the systems with varying UPI thresholds. Figure 7.5a shows the average values of $FAWCp$ by language with varying UPI threshold.

We see that on average 15% of the files in the C systems, 46% of the files in the Java systems and 29% of files in the C# systems are associated with exact clones (i.e., with UPI threshold 0.0). The higher percentage of Java systems can be explained by the fact that in Java systems there are many small similar accessor methods (Figure 7.4). Figures 7.5b, 7.5c and 7.5d refine Figure 7.5a to show a detailed view of the same information for the individual open source C, Java and C# systems respectively. From Figure 7.5b we see that
Figure 7.5: Percentage of files that are associated with cloned code
while most C systems have lower percentages at UPIT=0.0, a relatively small C system Weltab has about 51%, showing that this is a highly cloned system where 51% files are associated with at least one exact cloned method. In case of Java (Figure 7.5c), Swing shows the highest percentage, 65.9% for exact clones.

When we increase the UPI threshold to detect near-miss clones, we see that C and C# systems show a faster growing ratio than the Java systems, indicating the fact that there might be more near-miss clones in the C and C# systems than the Java systems and that the clones are scattered across different files. For example, even for the largest C system, the Linux Kernel, it increases from 14% (for UPI threshold 0.0) to 50% (for UPI threshold 0.3), whereas for the largest Java system, the Swing, it increases from 66% (for UPI threshold 0.0) to 76% (for UPI threshold 0.3). Most C# systems have a very high growing rate for UPI threshold 0.3 compare to other thresholds. For example, for one of the largest C# systems, the db4o, it only increases from 28% to 39% for UPI thresholds 0.0 to 0.2. When UPI threshold 0.3 is used, it increases to 77%, about double than with threshold 0.2.

### 7.3.3 Profiles of Cloning Density

While the subsection above provides an overall view of cloning over the files in a system, one cannot immediately see which files are highly cloned or which files contain the majority of clones. In this section we provide the values for the CRFM and QFCM related metrics. Tables 7.4, 7.5 and 7.6 provide the data for the C, Java and C# systems respectively. The first and second columns give the subject systems and different UPI thresholds, while the remaining columns show the corresponding QFCMp(v) (indicated with column Meth) and QFCLOCp(v) (indicated with column LOC) values. The last row of each table shows the average values of the metrics for each of the languages of the systems. When v=0+% or
Table 7.4: Percentage of files that have clones over a certain percentage for C systems

<table>
<thead>
<tr>
<th>System Name</th>
<th>v=0+ % Both</th>
<th>v=10+ % Meth LOC</th>
<th>v=20+ % Meth LOC</th>
<th>v=30+ % Meth LOC</th>
<th>v=40+ % Meth LOC</th>
<th>v=50+ % Meth LOC</th>
<th>v=100% Both</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cook</td>
<td>0.0</td>
<td>3.8</td>
<td>2.8</td>
<td>1.7</td>
<td>1.4</td>
<td>1.4</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>8.4</td>
<td>7.7</td>
<td>6.3</td>
<td>5.7</td>
<td>4.5</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>26.8</td>
<td>25.4</td>
<td>21.6</td>
<td>17.1</td>
<td>12.9</td>
<td>6.3</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>43.6</td>
<td>42.2</td>
<td>35.9</td>
<td>31.4</td>
<td>25.4</td>
<td>5.7</td>
</tr>
<tr>
<td>Hitpd</td>
<td>0.0</td>
<td>18.3</td>
<td>16.7</td>
<td>10.9</td>
<td>12.5</td>
<td>8.3</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>22.0</td>
<td>21.6</td>
<td>17.4</td>
<td>17.1</td>
<td>11.3</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>31.0</td>
<td>28.4</td>
<td>22.4</td>
<td>23.3</td>
<td>19.1</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>41.7</td>
<td>39.3</td>
<td>31.9</td>
<td>33.5</td>
<td>23.3</td>
<td>1.4</td>
</tr>
<tr>
<td>Postgresql</td>
<td>0.0</td>
<td>0.1</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>4.5</td>
<td>2.2</td>
<td>0.6</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>16.9</td>
<td>9.6</td>
<td>8.3</td>
<td>4.1</td>
<td>2.5</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>30.3</td>
<td>21.3</td>
<td>18.2</td>
<td>12.4</td>
<td>8.3</td>
<td>1.7</td>
</tr>
<tr>
<td>Snsns</td>
<td>0.0</td>
<td>13.0</td>
<td>8.7</td>
<td>5.8</td>
<td>8.7</td>
<td>5.1</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>20.3</td>
<td>13.0</td>
<td>8.7</td>
<td>10.1</td>
<td>6.5</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>36.2</td>
<td>26.1</td>
<td>19.6</td>
<td>19.6</td>
<td>16.7</td>
<td>2.8</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>46.4</td>
<td>39.1</td>
<td>29.0</td>
<td>29.0</td>
<td>22.5</td>
<td>2.8</td>
</tr>
<tr>
<td>Weltab</td>
<td>0.0</td>
<td>15.1</td>
<td>51.3</td>
<td>41.0</td>
<td>51.3</td>
<td>28.7</td>
<td>43.6</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>26.7</td>
<td>51.3</td>
<td>46.2</td>
<td>51.3</td>
<td>31.0</td>
<td>43.6</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>66.7</td>
<td>51.3</td>
<td>51.3</td>
<td>41.0</td>
<td>26.0</td>
<td>43.6</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>71.9</td>
<td>69.2</td>
<td>71.8</td>
<td>69.2</td>
<td>64.1</td>
<td>43.6</td>
</tr>
<tr>
<td>Avg. w/o</td>
<td>0.0</td>
<td>15.3</td>
<td>51.3</td>
<td>41.0</td>
<td>51.3</td>
<td>28.7</td>
<td>43.6</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>18.4</td>
<td>51.3</td>
<td>46.2</td>
<td>51.3</td>
<td>31.0</td>
<td>43.6</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>31.1</td>
<td>51.3</td>
<td>51.3</td>
<td>41.0</td>
<td>26.0</td>
<td>43.6</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>42.2</td>
<td>69.2</td>
<td>71.8</td>
<td>69.2</td>
<td>64.1</td>
<td>43.6</td>
</tr>
<tr>
<td>Linux</td>
<td>0.0</td>
<td>14.0</td>
<td>9.6</td>
<td>6.0</td>
<td>12.5</td>
<td>6.0</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>19.7</td>
<td>13.1</td>
<td>10.3</td>
<td>12.5</td>
<td>6.0</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>34.9</td>
<td>24.0</td>
<td>19.4</td>
<td>16.1</td>
<td>11.0</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>49.6</td>
<td>38.1</td>
<td>31.0</td>
<td>32.8</td>
<td>28.0</td>
<td>3.1</td>
</tr>
</tbody>
</table>

From the average rows of the tables with v=10+ we can see that on average 14.1% (9.9% LOC) of the files for C systems have 10% or more of their content as exact (UPI threshold 0.0) cloned methods. For Java systems this is even higher, at 37.8% (26.2% LOC). C# systems, on the other hand, are in the middle, at 25.1% (20.1% LOC) with respect to the number of cloned methods/functions. When we increase the UPI threshold (say to 0.3) the C# systems become the winners, at 62.8% (55.3% LOC), keeping Java systems at the second, at 54.3% (44.9% LOC) compared to 38.1% (31.0% LOC) for the C systems both in terms of methods and LOC. For higher values of v, say 50+ with UPI threshold
Table 7.5: Percentage of files that have clones over a certain percentage for Java systems

<table>
<thead>
<tr>
<th>System Name</th>
<th>UPIT v=0+%</th>
<th>v = 10+%</th>
<th>v = 20+%</th>
<th>v = 30+%</th>
<th>v = 40+%</th>
<th>v = 50+%</th>
<th>v = 100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both Meth LOC</td>
<td>Both Meth LOC</td>
<td>Both Meth LOC</td>
<td>Both Meth LOC</td>
<td>Both Meth LOC</td>
<td>Both Meth LOC</td>
<td>Both Meth LOC</td>
<td>Both Meth LOC</td>
</tr>
<tr>
<td>Ant</td>
<td>0.0</td>
<td>45.3</td>
<td>37.3</td>
<td>17.4</td>
<td>26.1</td>
<td>5.5</td>
<td>14.3</td>
</tr>
<tr>
<td>0.1</td>
<td>47.2</td>
<td>37.9</td>
<td>19.3</td>
<td>26.1</td>
<td>6.2</td>
<td>14.3</td>
<td>3.1</td>
</tr>
<tr>
<td>0.2</td>
<td>49.7</td>
<td>39.1</td>
<td>20.5</td>
<td>27.2</td>
<td>7.5</td>
<td>15.3</td>
<td>5.0</td>
</tr>
<tr>
<td>0.3</td>
<td>57.8</td>
<td>50.9</td>
<td>31.1</td>
<td>37.9</td>
<td>17.4</td>
<td>20.5</td>
<td>10.6</td>
</tr>
<tr>
<td>Javadoc</td>
<td>0.0</td>
<td>61.9</td>
<td>52.6</td>
<td>41.2</td>
<td>43.3</td>
<td>30.9</td>
<td>26.1</td>
</tr>
<tr>
<td>0.1</td>
<td>61.9</td>
<td>52.6</td>
<td>42.3</td>
<td>44.3</td>
<td>32.0</td>
<td>36.1</td>
<td>23.7</td>
</tr>
<tr>
<td>0.2</td>
<td>66.0</td>
<td>57.7</td>
<td>51.6</td>
<td>50.5</td>
<td>34.0</td>
<td>44.3</td>
<td>29.9</td>
</tr>
<tr>
<td>0.3</td>
<td>68.0</td>
<td>61.9</td>
<td>56.7</td>
<td>54.6</td>
<td>45.4</td>
<td>48.5</td>
<td>38.1</td>
</tr>
<tr>
<td>Jdtcore</td>
<td>0.0</td>
<td>66.7</td>
<td>59.3</td>
<td>48.6</td>
<td>49.3</td>
<td>38.7</td>
<td>36.3</td>
</tr>
<tr>
<td>0.1</td>
<td>74.2</td>
<td>68.2</td>
<td>58.4</td>
<td>57.2</td>
<td>45.4</td>
<td>46.2</td>
<td>37.3</td>
</tr>
<tr>
<td>0.2</td>
<td>51.5</td>
<td>45.9</td>
<td>42.9</td>
<td>36.9</td>
<td>29.6</td>
<td>27.9</td>
<td>22.7</td>
</tr>
<tr>
<td>0.3</td>
<td>57.5</td>
<td>53.6</td>
<td>51.1</td>
<td>45.1</td>
<td>39.5</td>
<td>36.9</td>
<td>30.9</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>0.0</td>
<td>44.6</td>
<td>36.9</td>
<td>32.2</td>
<td>27.5</td>
<td>18.9</td>
<td>19.3</td>
</tr>
<tr>
<td>0.1</td>
<td>45.5</td>
<td>38.2</td>
<td>33.5</td>
<td>28.3</td>
<td>21.0</td>
<td>21.0</td>
<td>15.0</td>
</tr>
<tr>
<td>0.2</td>
<td>51.5</td>
<td>45.9</td>
<td>42.9</td>
<td>36.9</td>
<td>29.6</td>
<td>27.9</td>
<td>22.7</td>
</tr>
<tr>
<td>0.3</td>
<td>57.5</td>
<td>53.6</td>
<td>51.1</td>
<td>45.1</td>
<td>39.5</td>
<td>36.9</td>
<td>30.9</td>
</tr>
<tr>
<td>Spule</td>
<td>0.0</td>
<td>6.0</td>
<td>4.0</td>
<td>2.0</td>
<td>2.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.1</td>
<td>16.0</td>
<td>14.0</td>
<td>12.0</td>
<td>12.0</td>
<td>6.0</td>
<td>12.0</td>
<td>4.0</td>
</tr>
<tr>
<td>0.2</td>
<td>16.0</td>
<td>14.0</td>
<td>12.0</td>
<td>12.0</td>
<td>6.0</td>
<td>12.0</td>
<td>4.0</td>
</tr>
<tr>
<td>0.3</td>
<td>30.0</td>
<td>28.0</td>
<td>24.0</td>
<td>24.0</td>
<td>20.0</td>
<td>20.0</td>
<td>14.0</td>
</tr>
<tr>
<td>Swing</td>
<td>0.0</td>
<td>65.9</td>
<td>47.3</td>
<td>32.6</td>
<td>32.1</td>
<td>23.4</td>
<td>24.9</td>
</tr>
<tr>
<td>0.1</td>
<td>67.6</td>
<td>49.8</td>
<td>35.5</td>
<td>33.3</td>
<td>25.1</td>
<td>25.6</td>
<td>19.3</td>
</tr>
<tr>
<td>0.2</td>
<td>71.5</td>
<td>56.8</td>
<td>43.0</td>
<td>40.1</td>
<td>32.1</td>
<td>29.0</td>
<td>24.6</td>
</tr>
<tr>
<td>0.3</td>
<td>76.1</td>
<td>65.9</td>
<td>54.1</td>
<td>51.0</td>
<td>40.3</td>
<td>33.8</td>
<td>31.6</td>
</tr>
<tr>
<td>Avg.</td>
<td>0.0</td>
<td>45.7</td>
<td>37.8</td>
<td>26.2</td>
<td>27.4</td>
<td>17.4</td>
<td>20.4</td>
</tr>
<tr>
<td>0.1</td>
<td>48.6</td>
<td>40.7</td>
<td>29.5</td>
<td>30.0</td>
<td>19.9</td>
<td>22.9</td>
<td>14.3</td>
</tr>
<tr>
<td>0.2</td>
<td>52.5</td>
<td>45.3</td>
<td>35.5</td>
<td>35.1</td>
<td>24.6</td>
<td>27.0</td>
<td>18.7</td>
</tr>
<tr>
<td>0.3</td>
<td>59.4</td>
<td>54.3</td>
<td>44.9</td>
<td>44.1</td>
<td>33.9</td>
<td>34.2</td>
<td>26.1</td>
</tr>
</tbody>
</table>

0.3, C# systems are still the winners, at 38.5% (28.5% LOC). However, both C and Java systems tend to have about the same percentage, at 22.9% (16.6% LOC) for C systems and 22.2% (16.7% LOC) for Java systems. One C system, Weltab, even has 64.1% (56.4% LOC) when v=50+% and 56.4% (still 56.4% LOC) when v=100%, once again indicating its high density of cloned code across different files. Similarly, when we look at the last columns of the tables at v=100% with UPI threshold 0.3, we see that C# systems still have higher percentages, at 14.7% (14.7% LOC), while C systems become the second, at 12.5% (12.5% LOC) and Java systems the third, at 5.1% (5.1% LOC). It is interesting to note that although both Java and C# are object-oriented languages and share many of the earlier
cloning characteristics, they are different in this case. It is actually also surprising to us that 14.7% of C# systems’ files are 100% cloned code (with UPI threshold 0.3), hinting for further research on this issue and compare to other object-oriented and procedural languages. These tables show only the overall percentage for the systems. Detailed results for each file of each system can be found in our online repository [183] as an XML database.

We have also looked at the copy/paste patterns of the detected clones. Figure 7.6 shows examples of copy/paste changes from Weltab, snns and Postgresql. Assuming that cloned methods in high density cloned files have been intentionally copy/pasted, we have also compared copy/paste patterns between the high density files of the systems.

From the above discussion it is obvious that one of the interesting systems is Weltab.

<table>
<thead>
<tr>
<th>System Name</th>
<th>UPIT</th>
<th>v=0+ %</th>
<th>v = 10+ %</th>
<th>v = 20+ %</th>
<th>v = 30+ %</th>
<th>v = 40+ %</th>
<th>v = 50+ %</th>
<th>v=100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vmukti</td>
<td>0</td>
<td>33.3</td>
<td>33.3</td>
<td>33.3</td>
<td>33.3</td>
<td>33.3</td>
<td>33.3</td>
<td>25.0</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>41.7</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>58.3</td>
<td>58.3</td>
<td>58.3</td>
<td>58.3</td>
<td>58.3</td>
<td>58.3</td>
<td>41.7</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>83.3</td>
<td>83.3</td>
<td>83.3</td>
<td>75</td>
<td>66.7</td>
<td>66.7</td>
<td>41.7</td>
</tr>
<tr>
<td>Linq</td>
<td>0</td>
<td>11.5</td>
<td>6.6</td>
<td>4.9</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>14.8</td>
<td>6.6</td>
<td>4.9</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>26.2</td>
<td>18</td>
<td>14.8</td>
<td>13.1</td>
<td>4.9</td>
<td>3.3</td>
<td>4.9</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>42.6</td>
<td>37.7</td>
<td>32.8</td>
<td>31.1</td>
<td>23</td>
<td>19.7</td>
<td>16.4</td>
</tr>
<tr>
<td>Nant</td>
<td>0</td>
<td>26.6</td>
<td>24.6</td>
<td>19.6</td>
<td>13.2</td>
<td>16.1</td>
<td>6.4</td>
<td>6.1</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>30.4</td>
<td>28.1</td>
<td>19.9</td>
<td>22.5</td>
<td>18.7</td>
<td>9.6</td>
<td>7.6</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>40.9</td>
<td>38.3</td>
<td>29.2</td>
<td>31.6</td>
<td>24.3</td>
<td>27.5</td>
<td>21.3</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>56.4</td>
<td>54.7</td>
<td>40.9</td>
<td>33</td>
<td>40.4</td>
<td>29.5</td>
<td>29.8</td>
</tr>
<tr>
<td>Castle</td>
<td>0</td>
<td>27.9</td>
<td>25.6</td>
<td>21.4</td>
<td>22.6</td>
<td>17.1</td>
<td>19.2</td>
<td>14.4</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>28.8</td>
<td>26.4</td>
<td>22.1</td>
<td>23.3</td>
<td>17.8</td>
<td>19.8</td>
<td>15.2</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>39.1</td>
<td>36.2</td>
<td>32.6</td>
<td>32.3</td>
<td>26.9</td>
<td>27.9</td>
<td>22.7</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>61.6</td>
<td>59.7</td>
<td>54.6</td>
<td>55.4</td>
<td>48.4</td>
<td>50.2</td>
<td>43.3</td>
</tr>
<tr>
<td>db4o</td>
<td>0</td>
<td>27.8</td>
<td>23.5</td>
<td>18.2</td>
<td>16.5</td>
<td>12.1</td>
<td>11.1</td>
<td>8.2</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>28.4</td>
<td>24.1</td>
<td>19.4</td>
<td>17.2</td>
<td>13</td>
<td>11.6</td>
<td>8.8</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>38.8</td>
<td>34</td>
<td>30.1</td>
<td>26.6</td>
<td>21.4</td>
<td>19.1</td>
<td>16.1</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>76.6</td>
<td>74.5</td>
<td>68</td>
<td>66.6</td>
<td>55.5</td>
<td>57.2</td>
<td>46.4</td>
</tr>
<tr>
<td>Avg.</td>
<td>0.0</td>
<td>28.6</td>
<td>25.1</td>
<td>20.1</td>
<td>21.6</td>
<td>16.7</td>
<td>18.2</td>
<td>14.0</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>33.5</td>
<td>29.2</td>
<td>24.4</td>
<td>25.5</td>
<td>21.0</td>
<td>21.9</td>
<td>18.1</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>43.2</td>
<td>38.5</td>
<td>33.4</td>
<td>33.6</td>
<td>28.9</td>
<td>28.5</td>
<td>23.5</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>65.4</td>
<td>62.8</td>
<td>55.3</td>
<td>56.1</td>
<td>46.3</td>
<td>48.4</td>
<td>40.2</td>
</tr>
</tbody>
</table>
Using the NICAD interface we noticed that although there is no file in *Weltab* that is exactly similar to another, 14 files of its 39 are clones of each other even with UPI threshold 0.05, that is, they are the same but for very minor edits. As an example, in the three files *vedt.c*, *vfix.c* and *xfix.c* (each of which has two functions including one large *main* function) there are only minor differences in the *main* function, and no differences at all in the other *(acknowledge)* function. Most of the changes are in *printf* statements and the parameters of the *acknowledge* function when it is called, and there are also lines added/deleted in the *main* function. For other high density files we have noticed similar changes, including changes in *if-statements*, the names of functions, and so on.

Similarly, in the files *canv.c* *canv1.c* and *canv1a.c*, the same method `void canv (lines, ipage, ppage, index, iward, iprec, isplt, lavcb, date, time)` is reused but the files differ slightly (e.g., changes in *printf* statements and some other additions/deletions of lines) in their main methods. *canv1.c* and *canv1a.c* have been detected as similar even with UPI threshold 0.05. When UPI threshold 0.20 is used all the functions of the three files are detected as clones of each other. Similarly, files *rsum.c*, *rsumxx.c*, *r51tmp.c*, *r26tmp.c*, *r11tmp.c*, *r101tmp.c* and *r01tmp.c* are not same but highly similar to each other (detected with UPI threshold 0.05), and the files *lans.c* and *lansxx.c* are highly similar.

By contrast, although *postgresql* is a substantial medium-sized system, there are very few clones. There is however, one file *scansup.c* that is 100% cloned with UPI threshold 0.20. Interestingly, it has only one method (`backend/parser/scansup.c (original lines 30-92)`) that forms clone pair with another method in another file (`backend/utils/misc/guc-file.c (original lines 1369-1427)`). The differences are in the name of the functions, in *if statements* and couple of additions and deletions. Even with UPI threshold 0.3, no more files in this system found to be 100% cloned.
For **snns**, two files, `tools/sources/lex.yyy.c` (17 methods) and `tools/sources/lex.yyy.c` (17 methods) are found to be cloned to each other with UPI threshold 0.15. Interestingly, when we use Linux `diff` they do not seem to be similar at all. But each of the extracted pretty-printed methods of the two files is either exactly similar or very similar. In fact, there are only two methods that differ between them. One pair is found cloned with UPI threshold 0.10 and the other with 0.15. Once again, the differences are minor and mostly in the `printf` statements. There is also another file, `kernel/sources/kr_pat_scan.c` that is highly similar (77.27% w.r.t methods and 92.22% w.r.t LOC) to the first two. Again, the differences are in the `if statements` and in `assignment statements`. Although in **Bison**, there are not much clones, 6 files are found completely cloned with UPI threshold 0.3. However, most files are
smaller files with only one method. For the *cook* system, two files *cook/src/c_incl/os.c* and *cook/src/find_libs/os.c* (each 50 LOC) are found to be identical (detected with UPI threshold 0.0). However, only one commented line is found to be different between the files. There are 81 files in the Linux kernel that have 100% of their contents cloned either in the same files or with other files with UPI threshold 0.3. Similarly, for the other C systems we have examined the highly similar files and found the similar types of copy/paste change patterns. Java and C# systems show similar behaviour but the methods are most cases smaller in size.

### 7.3.4 Profiles of Cloning Localization

In this section we provide the $P_{CLM_p}$ and $P_{RCLM_p}$ related metrics. The location of a clone pair is a factor in software maintainence [117, 204]. A code fragment can form a clone pair with another fragment within the same file, or it can form a clone pair with another fragment of a different file located in the same directory or with a code fragment that is located in a different file in a different directory. Kapser and Godfrey [117, 115] provide a categorization of function clones based on such location differences and analyze the causes, usefulness, harmfulness and possible solutions for each kind of cloning. For example, clone pairs that appear in the same file may not be harmful as they are not physically apart and might be easily maintainable. On the other hand, clone pairs that appear in different directories, might be harmful to software maintenance as the similar fragments are hard to find and thus might not easily maintainable. They also provide cloning statistics of such clone types on the Linux Kernel *file-system subsystem version 2.4.19* and *Postgresql 7.4.2*. We further extend the similar study with more than 20 C, Java and C# systems including entire Linux Kernel with our new hybrid clone detection tool.

In Tables 7.7, 7.8 and 7.9 we provide the percentage clone pairs for each of the different
Table 7.7: Percentage localization of clone pairs for C Systems

<table>
<thead>
<tr>
<th>Lan</th>
<th>System</th>
<th>Same File, Same Dir</th>
<th>Same Dir, Different Files</th>
<th>Different Dirs, Different Files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>UPI</td>
<td>0.0</td>
<td>0.10</td>
</tr>
<tr>
<td>Cook</td>
<td>Meth</td>
<td>0.0</td>
<td>16.7</td>
<td>40.2</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>0.0</td>
<td>32.3</td>
<td>51.3</td>
</tr>
<tr>
<td>Httpd</td>
<td>Meth</td>
<td>0.0</td>
<td>2.7</td>
<td>15.2</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>7.3</td>
<td>18.6</td>
<td>34.6</td>
</tr>
<tr>
<td>Postgresql</td>
<td>Meth</td>
<td>0.0</td>
<td>62.5</td>
<td>89.7</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>81.3</td>
<td>89.5</td>
<td>86.1</td>
</tr>
<tr>
<td>Snns</td>
<td>Meth</td>
<td>0.0</td>
<td>15.9</td>
<td>45.2</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>33.8</td>
<td>50.4</td>
<td>59.8</td>
</tr>
<tr>
<td>Weltab</td>
<td>Meth</td>
<td>0.0</td>
<td>0.0</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>0.0</td>
<td>0.0</td>
<td>0.4</td>
</tr>
<tr>
<td>Avg. w/o Meth</td>
<td>0.1</td>
<td>24.6</td>
<td>44.7</td>
<td>49.8</td>
</tr>
<tr>
<td>Linux</td>
<td>LOC</td>
<td>0.3</td>
<td>8.6</td>
<td>20.2</td>
</tr>
</tbody>
</table>

categories for the C, Java and C# systems respectively. For each of the systems the first row represents $PCLMp$ metric (i.e., w.r.t no. of methods) and the second row represents $PCLLOCp$ metric (i.e., w.r.t LOC). For each of the metrics four different values are shown corresponding to UPI thresholds ranging from 0.0 (exact clone) to 0.30 (relaxed near-miss clone).

From the tables we see that while there are no exact clones (UPI threshold 0.0) within the same file for C systems (except in the Linux Kernel), there are on average 18.7% (17.6% LOC) exact clone pairs within the same files for Java systems and 19.3% (18.6% LOC) for C# systems. This is particularly surprising in the Java system, Spule. Out of 60 exact clone pairs, 96.7% (97.2% LOC) of clone pairs occur within the same files, more specifically in file spule/src/common/Messages.java. Cloned methods occur between the static classes in the files. For example, two write methods (Figure 7.7) of 11 LOC and 7 LOC appear two times (original lines 1659-1669 and 1612-1622) and seven times (original...
public void write (OutputStream out) throws IOException
{
    Write_int (out, Template_ID);
    Write_int (out, Template_FatherID);
    Write_int (out, Template_LeftBrotherID);
    Write_int (out, Template_Type);
    Write_String (out, Template_MetaText);
    Write_int (out, Template_Person_ID);
    Write_String (out, Answer_MetaAnswer);
}

Figure 7.7: Two exact clones in Java system in Spule

protected virtual void SendPropertyChanged(String pName)
{
    if ((this.PropertyChanged != null))
    {
        this.PropertyChanged(this, new PropertyChangedEventArgs(pName));
    }
}

protected virtual void SendPropertyChanging()
{
    if ((this.PropertyChanging != null))
    {
        this.PropertyChanging(this, emptyChangingEventArgs);
    }
}

Figure 7.8: Two exact clones in C# system Linq
Table 7.8: Percentage localization of clone pairs for Java Systems

<table>
<thead>
<tr>
<th>Lan Name</th>
<th>System Name</th>
<th>Same File, Same Dir</th>
<th>Same Dir, Different Files</th>
<th>Different Dirs, Different Files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>UPTT 0.0 0.10 0.20 0.30</td>
<td>UPTT 0.0 0.10 0.20 0.30</td>
<td>UPTT 0.0 0.10 0.20 0.30</td>
</tr>
<tr>
<td>Java</td>
<td>Ant</td>
<td>Meth 4.4 4.7 5.1 6.6</td>
<td>Meth 19.7 20.8 19.6 22.7</td>
<td>Meth 4.9 5.2 6.1 6.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOC 4.2 5.2 6.1 11.6</td>
<td>LOC 14.3 22.5 25.8 25.8</td>
<td>LOC 0.9 3.1 11.8 41.9</td>
</tr>
<tr>
<td></td>
<td>javadoc</td>
<td>Meth 19.7 20.8 19.6 22.7</td>
<td>Meth 14.3 22.5 25.8 25.8</td>
<td>Meth 0.9 3.1 11.8 41.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOC 4.2 5.2 6.1 11.6</td>
<td>LOC 14.3 22.5 25.8 25.8</td>
<td>LOC 0.9 3.1 11.8 41.9</td>
</tr>
<tr>
<td></td>
<td>Jdtcore</td>
<td>Meth 19.7 20.8 19.6 22.7</td>
<td>Meth 14.3 22.5 25.8 25.8</td>
<td>Meth 0.9 3.1 11.8 41.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOC 4.2 5.2 6.1 11.6</td>
<td>LOC 14.3 22.5 25.8 25.8</td>
<td>LOC 0.9 3.1 11.8 41.9</td>
</tr>
<tr>
<td></td>
<td>JHotDraw</td>
<td>Meth 5.8 5.8 6.6 13.7</td>
<td>Meth 5.8 5.8 6.6 13.7</td>
<td>Meth 5.8 5.8 6.6 13.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOC 4.5 4.2 7.5 18.7</td>
<td>LOC 4.5 4.2 7.5 18.7</td>
<td>LOC 4.5 4.2 7.5 18.7</td>
</tr>
<tr>
<td></td>
<td>Spule</td>
<td>Meth 96.7 90.6 91.2 77.0</td>
<td>Meth 97.2 86.1 87.3 64.6</td>
<td>Meth 96.7 90.6 91.2 77.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOC 2.4 3.2 8.3 18.5</td>
<td>LOC 2.4 3.2 8.3 18.5</td>
<td>LOC 2.4 3.2 8.3 18.5</td>
</tr>
<tr>
<td></td>
<td>Swing</td>
<td>Meth 96.7 90.6 91.2 77.0</td>
<td>Meth 97.2 86.1 87.3 64.6</td>
<td>Meth 96.7 90.6 91.2 77.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOC 5.8 5.8 6.6 13.7</td>
<td>LOC 5.8 5.8 6.6 13.7</td>
<td>LOC 5.8 5.8 6.6 13.7</td>
</tr>
<tr>
<td></td>
<td>Avg</td>
<td>Meth 18.7 18.4 20.1 25.3</td>
<td>Meth 19.6 19.2 24.5 29.7</td>
<td>Meth 18.7 18.4 20.1 25.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOC 17.6 19.2 24.5 29.7</td>
<td>LOC 17.6 19.2 24.5 29.7</td>
<td>LOC 17.6 19.2 24.5 29.7</td>
</tr>
</tbody>
</table>

lines 1796-1802, 1687-1693, 1533-1539, 1432-1438, 1391-1397, 1349-1355, and 1268-1274) respectively in file Messages.java. Similarly, exact clone pairs appear within the same file in other Java systems between original and abstract classes (e.g., ant/BuildEvent.java lines 163-165 and 93-95), as inlined functions (e.g., jdtcore/src/internal/compiler/Compiler.java lines 148-151 and 84-87) and so on. In C# system, LINQ, there are also many exact clone pairs within the same file. For example, two functions SendPropertyChanged and SendPropertyChanging (Figure 7.8) appear 21 times each in file Linq/Source1712/Source/Entities/DB/DB.designer.cs as protected virtual functions. While most of the clone pairs are smaller in size, exact cloning within the same file might be interesting both from software maintenance and language design points of view.

As an exception to the C systems, the Linux Kernel has one file (/drivers/net/fec.c) that has six exact clone pairs in it. In fact there are two exact clone classes in this file. In the
first class, function \texttt{fec\_get\_mac} (32 LOC) appears three times (lines 1744-1775, 1595-1626 and 1464-1495) and in the second class, function \texttt{fec\_set\_mii} (16 LOC) also appears three times (lines 1727-1742, 1577-1593 and 1446-1462) as static inline functions.

However, when we detect near-miss clones by allowing a higher UPI threshold, we see that the metrics values grow at a higher rate for the C systems than the Java and C# systems. For example, when UPI threshold is 0.3, on average 49.8\% (52.5\% LOC) of clone pairs of the C systems occur within the same file compared to only 25.3\% (29.7\% LOC) of the clone pairs in the Java systems. C# systems show 33.0\% (36.2\%), a little bit higher percentages than Java systems. If we have a close look at the individual systems we also see higher ratios for most of the C systems than the Java and C# systems. In the case of the C system \texttt{Wget} it actually reaches to 90.9\% (92.7\% LOC). However, \texttt{Wget} is a small system, has only 11 clone pairs in 2 classes with UPI threshold 0.3, and only 3 of its 21 files are associated with clones. Taking a closer look, we found that of the 2 clone classes, one contains 5
cloned methods (original lines 503-539, 463-499, 389-425, 350-385, and 428-460) and all are from the same file wget/src/ftp-basic.c.

However, a reasonably large system, Postgresql (530 clone pairs in 203 classes with UPI threshold 0.3), also shows a higher percentage (87.6% methods and 86.1% LOC) of clone pairs within the same files with UPI threshold 0.3. As with Wget, it has also higher frequency of clone classes in the same file. For example, in file postgresql/src/backend/utils/adt/float.c, there are six similar methods (original lines 952-965, 935-948, 900-913, 865-878, 848-861, and 831-844) of 14 LOC differing only in their function names and built-in function calls (e.g., tan changes to sin, cos to acos, and so on).

Both C and Java systems tend to have a higher percentage of exact clones (UPI threshold 0.0) within the same directory but in different files than the C# systems. Even the largest C system, the Linux Kernel, has 55.5% (58.9% LOC) of its exact clone pairs in the same directory (but different files). The largest Java system, the Java 2 SDK Swing, has even more, 87.5% (90.0% LOC). Among the C# systems, the Castle, the biggest C# system in this study, shows the highest, 56.5% (58.4% LOC). C# systems are the winners for exact clone pairs of different directories though, 62.9% (63.2% LOC), compare to 45.5% (50.5% LOC) for C systems and 20.1% (19.5% LOC) for Java systems.

When we look for near-miss clones by increasing the UPI threshold, we see interesting phenomena for different language paradigms. While these percentages of different directory clone pairs tend decrease significantly for procedural C systems, from 45.6% (52.1% LOC) to 15.0% (14.2% LOC), they tend to remain constant or even increase for the object-oriented Java and C# systems. This phenomenon for the C systems indicates that clone pairs either form within the same file or between different files of the same directories. After a close look on the same file and same directory percentages, we see that clone pairs
for the C systems actually tend to form within the same files with increasing UPI thresholds, from 0.0% (0.0% LOC) to 49.8% (52.5% LOC), confirming the fact that there are significant amount of near-miss clone pairs within the same files in the C systems.

In order to study the further insights of cloning locality, we also provide $PRCLM_p$ related metrics, i.e., the locality of the remote clone pairs (i.e., those are in different directories). Two fragments of a different directory clone pair (i.e., the fragments have different parent directories) might share the same grandparent directory or they might be under the same subsystem or in the worst case, they might be from different subsystems. As discussed earlier, the more closer the fragments of clone pairs are, the more easily maintainable they might be. Tables 7.10, 7.11 and 7.12 show the data for the C, Java and C# systems of this study that have different directory clone pairs.

<table>
<thead>
<tr>
<th>Language System</th>
<th>With Respect To</th>
<th>UPIT=0.0</th>
<th>UPIT=0.1</th>
<th>UPIT=0.2</th>
<th>UPIT=0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Same Grandparent</td>
<td>Same Subsystem</td>
<td>Different Subsystem</td>
<td>Same Grandparent</td>
<td>Same Subsystem</td>
</tr>
<tr>
<td>Cook</td>
<td>Meth LOC</td>
<td>100.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>C</td>
<td>Meth LOC</td>
<td>76.6</td>
<td>22.8</td>
<td>0.6</td>
<td>74.3</td>
</tr>
<tr>
<td>Postgresql</td>
<td>Meth LOC</td>
<td>0.0</td>
<td>100.0</td>
<td>0.0</td>
<td>25.0</td>
</tr>
<tr>
<td>snns</td>
<td>Meth LOC</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Avg. w/o Method</td>
<td>Meth LOC</td>
<td>44.2</td>
<td>30.1</td>
<td>25.7</td>
<td>49.8</td>
</tr>
<tr>
<td>Linux</td>
<td>Meth LOC</td>
<td>31.1</td>
<td>64.2</td>
<td>3.5</td>
<td>31.1</td>
</tr>
<tr>
<td>Linux</td>
<td>Meth LOC</td>
<td>30.7</td>
<td>65.7</td>
<td>4.3</td>
<td>33.2</td>
</tr>
</tbody>
</table>
Table 7.11: Percentage localization of remote clone pairs for Java systems

<table>
<thead>
<tr>
<th>Language</th>
<th>System</th>
<th>With Respect To</th>
<th>UPIT=0.0</th>
<th>UPIT=0.1</th>
<th>UPIT=0.2</th>
<th>UPIT=0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Same Grandparent</td>
<td>Same Subsystem</td>
<td>Different Subsystem</td>
<td>Same Grandparent</td>
<td>Same Subsystem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Same Grandparent</td>
<td>Same Subsystem</td>
<td>Different Subsystem</td>
<td>Same Grandparent</td>
<td>Same Subsystem</td>
</tr>
<tr>
<td>Ant</td>
<td>Meth</td>
<td>47.6</td>
<td>45.1</td>
<td>7.3</td>
<td>47.0</td>
<td>45.8</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>48.1</td>
<td>44.8</td>
<td>7.1</td>
<td>45.1</td>
<td>48.3</td>
</tr>
<tr>
<td>javadoc</td>
<td>Meth</td>
<td>68.8</td>
<td>31.3</td>
<td>0.0</td>
<td>68.8</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>58.9</td>
<td>0.0</td>
<td>41.1</td>
<td>58.9</td>
<td>0.0</td>
</tr>
<tr>
<td>Jfcro</td>
<td>Meth</td>
<td>20.1</td>
<td>71.5</td>
<td>8.4</td>
<td>20.7</td>
<td>71.5</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>16.8</td>
<td>74.5</td>
<td>8.7</td>
<td>16.9</td>
<td>77.9</td>
</tr>
<tr>
<td>JhotDra</td>
<td>Meth</td>
<td>74.2</td>
<td>0.0</td>
<td>25.8</td>
<td>74.2</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>75.9</td>
<td>0.0</td>
<td>24.1</td>
<td>75.9</td>
<td>0.0</td>
</tr>
<tr>
<td>Swing</td>
<td>Meth</td>
<td>13.3</td>
<td>74.2</td>
<td>12.5</td>
<td>13.2</td>
<td>73.6</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>9.3</td>
<td>75.8</td>
<td>14.8</td>
<td>8.8</td>
<td>72.2</td>
</tr>
<tr>
<td>Avg.</td>
<td>Meth</td>
<td>38.1</td>
<td>20.2</td>
<td>41.7</td>
<td>38.1</td>
<td>20.0</td>
</tr>
<tr>
<td></td>
<td>LOC</td>
<td>35.5</td>
<td>22.1</td>
<td>42.4</td>
<td>34.9</td>
<td>20.8</td>
</tr>
</tbody>
</table>

From the Avg. rows of the tables, we see that exact clone pairs (UPI threshold 0.0) those are not within the same file or in different files under the same directory, tend to have same grandparents or subsystems. For example, 36.6% (38.5% LOC) of different directory exact clone pairs of C# systems have the same grandparents, while only 14.7% (14.3% LOC) are under different subsystems. C and Java systems tend to have similar percentages to C#. One should also note that clone pairs those have the same grandparents, essentially are under the same subsystems.

When we increase the UPI threshold for detecting near-miss clones, we see that both for C and C# systems, the percentages of same grandparent clone pairs tend to remain constant or even increase while Java systems tend to decrease (from 37% to 31%). Percentages of clone pairs for the other two metrics (same subsystem and different subsystems) also tend to...
remain constant for C and C# systems and with increasing ratios for Java systems. Figures 7.9a and 7.9b show an overview about the localization of clone pairs for UPI threshold 0.3. Since we consider all functions with 3 LOC or more in pretty-printed format, one might argue that the findings are biased on the size of the functions. However, it does not seem so when we look the values for LOC. In almost all cases, both the method and LOC metrics values are very close, showing that a significant proportion of files in each system have a significant proportion of similar code in the files themselves. The commonalities and variabilities of cloning properties of C# with the C and Java systems might be exacerbated by the fact that C# a multi-paradigm programming language that encompasses functional, imperative, generic, object-oriented (class-based), and component-oriented programming disciplines. When we look the source of the C# systems, we also see the usage of different programming paradigms.
CHAPTER 7. AN EMPIRICAL STUDY OF FUNCTION CLONES

7.4 Related Work

Empirical study of clones in open source systems is not a new topic. When a new clone detection technique is published, it normally comes with an empirical study (at least in part). However, these studies focus on validating the proposed methods [182] rather than on the subject systems.

Several tool comparison studies have used open source systems for comparing different tools [182]. Of them, the Bellon et al. experiment [32, 31] is the most extensive to date, using four C and four Java systems to compare several state-of-the-art tools. Although we have used the subject systems from Bellon’s experiment as a part of our study, our study differs in using a new hybrid clone detection tool, in the size of the subject systems analyzed (e.g., the entire Linux Kernel) and in providing the cloning status of the subject systems themselves in several different dimensions.

Kapser and Godfrey have conducted extensive empirical studies with Apache httpd,
CHAPTER 7. AN EMPIRICAL STUDY OF FUNCTION CLONES

the Linux file system and several other open source systems. They provide a detailed categorization of code clones in the form of a taxonomy [117], propose a new analysis framework [115] and give an in-depth study on the harmfulness / usefulness of cloning [121]. Our study differs in that we focus on the comprehensive cloning status of a wide variety of different systems in different languages, whereas they focus on the maintenance implications of cloning.

Empirical studies of cloning in the Linux Kernel have also been carried out by several other researchers. Of them, Casazza et al. [43] and Antoniol et al. [8] provide interesting findings, but they focus on clone evolution, whereas we focus on the occurrence of copy/paste clones in several dimensions. Kim et al. [125] also studied the evolution of code clones in several systems and concluded that programmers often intentionally practice code cloning. Jiang and Hassan [102] also used the Linux Kernel as an example for their framework for understanding cloning in large systems.

Al-Ekram et al. [5] have also conducted a promising empirical study on cloning, focussing on C/C++ systems from two different domains. They examined different clone types (e.g., accidental clones) by analyzing clones across systems in the same domain, whereas we have studied a wide variety of systems and concentrated on copy/paste function clones of individual system. Krinke [142] has conducted an empirical study with five C/C++/Java systems, focussing on consistent and inconsistent changes to exact code clones in different versions of the subject systems. The most closely related work to ours is the work of Rajapakse and Jarzabek [178] which was also one of the motivations of our study. However, they studied cloning in a different domain, web applications, and have looked at only exact clones.
7.5 Summary

In this chapter we have provided an empirical study of function clones in several C, Java and C# open source software systems of varying size, including Apache httpd and the entire Linux Kernel, using the new hybrid clone detection method NICAD. The study has demonstrated that NICAD is capable of accurately finding both exact and near-miss function clones even in large systems and different languages, and that there seem to be a large number of copy/paste function clones in those systems. We have provided cloning statistics for these systems in several different dimensions and made the detailed results available in an online repository. These results can potentially be used as a benchmark for evaluating other clone detection tools.

Threats to Validity: One of the major threats to the results of this study is the lack of a sound definition of code clones. While one can precisely define exact clones, there is no agreed upon definition of near-miss clones. In this study we have used a dissimilarity threshold on the standard pretty-printed code as a measure of near-miss clones. While this gives good results, we cannot be sure that these are definitively the right set for software maintenance activities such as refactoring.
Chapter 8

Conclusion

This chapter concludes this thesis. While Section 8.1 summarizes the contributions, Section 8.2 outlines future research directions.

8.1 Summary

Clones are considered harmful in software maintenance and evolution [110]. For example, if a bug is detected in a code fragment, all the fragments similar to it should be investigated to check for the same bug. In response, over the past decade many tools for detecting code clones have been proposed (Chapter 5).

However, a recent study [32] shows that none of the tools can detect clones (especially near-miss syntactic clones) with high precision and recall. It also shows that lightweight text-based techniques can find clones with high accuracy and confidence, but the detected clones often do not correspond to appropriate syntactic units. Abstract syntax tree- / Parser-based syntactic techniques, on the other hand, find syntactically meaningful clones but tend to be more heavyweight, requiring a full parser and sub-tree comparison algorithm.
Furthermore, none of the techniques can detect near-miss clones well.

In this thesis, we have developed a hybrid language-specific but lightweight clone detection method called NICAD (Chapter 3) that combines the strengths and overcomes the limitations of both text-based and parser-based clone detection techniques to yield highly accurate (with high precision and recall) identification of exact and near-miss syntactic code clones in software systems. The novel use of pretty-printing, flexible code normalization and filtering, extraction of syntactic potential clones before the comparison, and threshold-based line-wise comparison using a dynamic clustering algorithm makes NICAD demonstrably superior to other existing tools both in terms of precision and recall and for different types of exact and near-miss clones. We have evaluated NICAD in three phases: first, with an early evaluation with two small to medium-size systems (Chapter 3), second, with a mutation-based automatic framework (Chapter 6) and third, with large scale open source systems of different languages (Chapter 7). These studies show that NICAD can efficiently detect both exact and near-miss clones with high precision and recall. NICAD was also placed one of the best methods available in our scenario-based comparison and evaluation (Chapter 5) study that organizes all the available techniques and tools in a coherent conceptual framework.

The definition of code clone in the literature [182] is inherently vague. We have thus proposed an editing taxonomy for clone creation (Chapter 4) by studying the literature [182]. We have also validated the editing taxonomy by studying the copy/paste patterns of the near-miss function clones [183] obtained from an empirical study (Chapter 7). We have then enhanced NICAD to detect the fine-grained clone types of the editing taxonomy. This taxonomy has also been used as a basis for the scenario-based qualitative comparison and evaluation of existing clone detection techniques and tools (Chapter 5). It has been used
in building the model-driven clone mutation operators in the mutation-based framework (Chapter 6) that automatically measure (and compare) the precision and recall of clone detection tools. A subset of this taxonomy has also been used in conducting a large scale empirical study with open source systems of different languages (Chapter 7).

Clone detection techniques are often inadequately evaluated. The general lack of evaluation is exacerbated by the fact that there are no agreed-upon evaluation criteria or representative benchmarks [11]. In particular, accuracy measures such as precision and recall have only been roughly estimated, due both to problems in creating a validated clone benchmark against which tools can be compared, and to the enormous manual effort required to hand check large numbers of candidate clones. In an attempt to compare all clone detection techniques more uniformly, independent of tool availability, implementation limitations or language, we have conducted a predictive, scenario-based approach (Chapter 5) and organize the large number of techniques and tools in a coherent conceptual framework. Using this framework, one can efficiently and instantly choose the right set of tools for her purpose or can develop a hybrid clone detection method. The hybrid architecture of NICAD is in fact derived from this framework. Consequently, NICAD was placed one of the best tools available in the framework. This study also shows that existing methods are not sound enough to detect all the different types of fine-grained clones and thus poses further research in the area.

Furthermore, in order to evaluate and compare the available tools in the realistic setting, we have also developed a mutation / injection-based framework that automatically and efficiently measures (and compares) the recall and precision of clone detection tools for different fine-grained clone types of the proposed unified model (Chapter 6). We have
evaluated NICAD and two of its variants. We experienced that full NICAD is capable of detecting different fine-grained clone types of the editing taxonomy. Although the framework was initially targeted at evaluating NICAD and its variants, the objective was to design a generic framework for evaluating and comparing clone detection tools by overcoming the known challenges to objective tool comparison experiments. The resulting framework is flexible and adaptable enough to evaluate third party clone detection tools provided that the tools can be run from the command line and that they provide textual report of detected clones with full file name, and begin and end line numbers of the code fragments of found clone pairs.

Finally, we have conducted a large scale empirical study, both to evaluate NICAD and to study the cloning status of large open source systems in a variety of different dimensions, including language, clone size, clone similarity, clone location and clone density both by proportion of cloned functions and cloned lines of code (Chapter 7). The study shows that NICAD is capable of accurately finding both exact and near-miss function clones even in large systems and different languages. We have also found that there are a large number of exact function clones in these open source systems. We have also seen much higher percentages of near-miss clones, indicating significantly higher numbers of near-miss clones than exact clones in these systems. We have also experienced that there are more exact function clones in object-oriented Java and C# systems than in C systems. However, the effect of increasing the UPI threshold for near-miss clones was almost identical regardless of language paradigm. We observed no significant differences in cloning related to the size of the systems. Even though Linux is huge, its cloning characteristics seem to be typical of other C systems. Similarly, the largest Java system Swing and the largest C# system db4o seem to be representative of the cloning characteristics of other systems written in their
corresponding languages. The detailed results are available in an online repository [183] in a variety of formats and can be used as a benchmark.

8.2 Future Research Directions

**Further Improvement of NICAD:** Given the fact that NICAD needs a quadratic time complexity in the worst case, our proposed approach is slower than many others, especially those based on metrics or those based on suffix-trees. As our intention is to compare program text rather than metrics, we plan to adopt a suffix-tree based comparison algorithm in our detection process. Until now only exact and parameterized matching suffix-tree (or suffix-array) algorithms have been used in clone detection research [15, 113, 137, 199, 22]. As we can apply different flexible options (e.g., flexible code abstraction/normalization and code filtering), direct application of suffix-trees would give better results than others those merely based on suffix-tree without such options. However, we want to do more. We want to allow unanticipated gaps (those that cannot be captured even with normalization and littering) and thus we want to apply approximate matching suffix-trees. An exact or parameterized matching suffix-tree/suffix-array algorithm is thus not suitable for our purpose. We therefore plan to use an approximate matching suffix-tree/suffix-array (k-difference suffix-tree/suffix-array) [172] in our comparison phase. This adaptation would be easy in our detection method, since different phases of our method work independently in the detection process.

NICAD still cannot accurately detect *Type 4* semantic clones. We thus plan to further hybridize NICAD with other techniques (e.g., Latent Semantic Indexing, Independent Component Analysis or Program Dependency Analysis) so that we can efficiently deal with *Type 4* clones.
**Further Empirical Study:** In the large empirical study conducted in this thesis (Chapter 7), we have only used the basic NICAD method, consisting of extraction of potential clones, standard pretty-printing and dissimilarity thresholds in comparison with dynamic clustering. In near future we also plan to use flexible pretty-printing, code normalization and filtering with all the open source systems of our current experiment (and many more).

In our current study we have studied function clones only. In future we also plan to perform a case study to detect clones of smaller granularities such as \textit{begin-end} blocks and/or any structured blocks with standard/flexible pretty-printing, dissimilarity thresholds, code normalization and filtering options.

**Mega Tool Comparison Experiment with the Mutation-based Framework:** Thus far we have only evaluated NICAD and two of its variants with our mutation- / injection-based framework (Chapter 6). In future, we plan to conduct a mega tool comparison experiment with third party tools using this framework. Our evaluation framework is generalized enough and third party clone detection tools can be evaluated with our framework provided that their tools can give textual output (the traditional one with file name and begin-end line number of the cloned block or XML output as we do) and that their tools can be run in command mode.

**Forward Engineering:** We plan to exploit NICAD in the development process. For example, one immediate plan is to build an Eclipse plug-in on top of NICAD that will assist developers in making clones (e.g., efficient and error-free copy/paste and renaming) and in finding the desired code block to clone if one is already in the system or in other repositories.

A clone in the model causes a clone in the generated source code and thus poses similar
maintenance and quality threats. We thus plan to adapt NICAD to detect clones in models. While model-driven development is becoming increasingly popular, up to now, little work (or only one noteworthy work [59]) has been done on clone detection and analysis in models.
Bibliography


[65] Tool Dupman http://sourceforge.net/projects/dupman


[85] M.W. Godfrey, D. Svetinovic and Q. Tu. Evolution, growth, and cloning in Linux:


Detection of Software Clones (IWDSC’03), 2 pp., Victoria, BC, November 2003.


[124] M. Kim, L. Bergman, T. Lau and D. Notkin. An Ethnographic Study of Copy and


