A Survey on Software Clone Detection Research*

Chanchal Kumar Roy and James R. Cordy

September 26, 2007

Technical Report No. 2007-541
School of Computing
Queen's University at Kingston
Ontario, Canada

Abstract

Code duplication or copying a code fragment and then reuse by pasting with or without any modifications is a well known code smell in software maintenance. Several studies show that about 5% to 20% of a software systems can contain duplicated code, which is basically the results of copying existing code fragments and using then by pasting with or without minor modifications. One of the major shortcomings of such duplicated fragments is that if a bug is detected in a code fragment, all the other fragments similar to it should be investigated to check the possible existence of the same bug in the similar fragments. Refactoring of the duplicated code is another prime issue in software maintenance although several studies claim that refactoring of certain clones are not desirable and there is a risk of removing them. However, it is also widely agreed that clones should at least be detected.

In this paper, we survey the state of the art in clone detection research. First, we describe the clone terms commonly used in the literature along with their corresponding mappings to the commonly used clone types. Second, we provide a review of the existing clone taxonomies, detection approaches and experimental evaluations of clone detection tools. Applications of clone detection research to other domains of software engineering and in the same time how other domain can assist clone detection research have also been pointed out. Finally, this paper concludes by pointing out several open problems related to clone detection research.

*This document represents our initial findings and a further study is being carried on. Reader's feedback is welcome at croy@cs.queensu.ca.

Contents

List of Figures

List of Tables

1 Introduction

Copying code fragments and then reuse by pasting with or without minor modifications or adaptations are common activities in software development. This type of reuse approach of existing code is called code cloning and the pasted code fragment (with or without modifications) is called a clone of the original. However, in a post-development phase, it is difficult to say which fragment is original and which one is copied and therefore, fragments of code which are exactly the same as or similar to each other are called code clones, i.e., instances of duplicated or similar code fragments are called code clones or just clones. Several studies show that software systems with code clones are more difficult to maintain than the ones without them [118, 18]. The tendency of cloning not only produces code that is difficult to maintain, but may also introduce subtle errors [51, 168, 169]. Code clones are considered as one of the bad smells of a software system [84] and it is widely believed that cloned code has several adverse affects on the maintenance life-cycles of software systems. Therefore, it is beneficial to remove clones and prevent their introduction by constantly monitoring the source code during its evolution [158].

Clones are often the result of copy-paste activities. Such activities are very easy and can significantly reduce programming effort and time as they reuse an existing fragment of code rather than rewriting similar code from scratch. This practice is common, especially in device drivers of operating systems where the algorithms are similar [169]. There are several other factors such as performance enhancement and coding style because of which large systems may contain a significant percentage of duplicated code [31]. There is also "accidental cloning", which is not the result of direct copy and paste activities but by using the same set of APIs to implement similar protocols [6]. The literature on the topic has described many other situations that can lead to the duplication of code within a software system [18, 31, 120, 122, 146, 178] (for details c.f. Section 2).

Code cloning is found to be a more serious problem in industrial software systems [8, 47, 18, 31, 74, 120, 122, 148, 146, 178]. In presence of clones, the normal functioning of the system may not be affected, but without countermeasures by the maintenance team, further development may become prohibitively expensive [200]. Clones are believed to have a negative impact on evolution [92, 91]. Code clones may adversely affect the software systems' quality, especially their maintainability and comprehensibility [87, 88]. For example, cloning increases the probability of update anomalies (inconsistencies in updating) [28]. If a bug is found in a code fragment, all of its similar cloned fragments should be detected to fix the bug in question. Moreover, too much cloning increases the system size and often indicates design problems such as missing inheritance or missing procedural abstraction [74]. Although the cost of maintaining clones over a system's lifetime has not been estimated yet, it is at least agreed that the financial impact on maintenance is very high. The costs of changes carried out after delivery are estimated at 40% - 70% of the total costs during a system's lifetime [100] (for details c.f. Section 3). Existing research shows that a significant amount of code of a software system is cloned code and this amount may vary depending on the domain and origin of the software system [136, 169, 116]. For instance, Baker [18] has found that on large systems between 13% - 20% of source code can be cloned code. Lague et al. [158] have studied only function clones and reported that between 6.4% - 7.5% of code is cloned code whereas Baxter et al. [31] have reported that 12.7% of code being clones of a

software system. Mayrand et al. [178] have also estimated that normal industrial source code contains 5% – 20% of duplicated code. Kapser and Godfrey [123] have experienced that as much as 10% –15% of source code of large system is cloned. For an object-oriented COBOL system, the rate of duplicated code is found even much higher, about 50% [74].

Considering the huge amount of duplicated code and its maintenance cost of large software systems, it is therefore, crucial to detect code clones of large systems for performing the respective maintenance tasks (e.g., refactoring). Fortunately, there are huge research studies to find clones automatically [14, 20, 31]. Again, the question arises of the definition of code clone itself. There is no sound definition of code cloning. In many cases, one cannot be sure that one fragment of code is copied from another. Baxter et al. [31] say that a clone is "a program fragment that [is] identical to another fragment". Krinke [156] uses the term "similar code". Ducasse et al. [74] use the term "duplicated code", Komondoor and Horwitz [141] also use the term "duplicated code" and use "clone" as an instance of duplicated code. Mayrand et al. [178] use metrics to find "an exact copy or a mutant of another function in the system". All these definitions of clones carry some kind of vagueness (e.g., "similar" and "identical") and this imperfect definition of clones makes the clone detection process much harder than the detection approach itself.

Nevertheless, attempts are being undertaken to detect clones [31, 122, 178, 146, 24, 14, 74, 156, 140] and once identified, they can be removed through source code refactoring [21, 84, 107, 78, 142]. Again, refactoring of the detected clones may not be the perfect solution for the software system of interest. While it is widely believed that detecting and refactoring the code clones from the software systems can improve the system's overall code quality, there are some recent works that show that "*refactorings may not always improve the software with respect to clones*" and "*skilled programmers often created and managed code clones with clear intent*" [136]. Cordy is one of the pioneers in this line, who first observed that in some cases, especially for the large financial software systems, changing or refactoring the clones is not advisable from a risk management point of view [57]. It has been also argued that clones are beneficial in certain situations [128, 129]. In the same time, there is no doubt that extensive code duplication is related to problems in maintenance and should be detected at least [84, 120, 178, 62, 176].

There are also many other software engineering tasks such as understanding code quality, aspect mining, plagiarism detection, copyright infringement investigation, software evolution analysis, code compaction (e.g., for mobile devices), virus detection or detecting bugs, do require the extraction of syntactically or semantically similar code fragments which essentially implies clones should be detected [151]. However, the same might not be applicable to XP process software. Nickell and Smith [188] argue that fewer code clones are found in XP process software, claiming that the XP process improves software quality.

The rest of the paper is organized as follows. The different factors that lead to code duplication are listed in Section 2, and the disadvantages caused by code cloning are pointed out in Section 3. Section 4 lists different benefits and applications of detecting clones. Section 5 shows the harmfulness of cloning in software systems. The clone relation terminologies are discussed in Section 6. In Section 7 we provide the different clone terms that commonly used in the literature following a mapping of those terms to the commonly used clone types. Different research groups categorize clones with respect to different contexts. Section 8 reviews all such available categories of clones and presents them in the form of

a taxonomy. In Section 9 we coarsely describe of how clones can be detected, whereas in Section 10 the different clone detection techniques and tools are presented with respect to several properties. Section 11 first provides a higher level comparison of the different detection approaches based on the information available from the corresponding papers and then summarizes tool comparison experiments in comparing the different tools. Visualizations of detected clones are a major concern for clone management and therefore, a list of visualization techniques are discussed in Section 12. Clones can be removed, managed or even avoided for efficient software maintenance. These issues are pointed out in Section 13. There are several studies that use clone detection techniques for observing cloning behavior in software evolution and are reviewed in Section 14. The impact of code clones on software quality is discussed in Section 15. Section 16 discusses some of the related work to clone detection research. Several open issues related to definition of clones, detection approaches and evaluation methods of comparing clone detection tools are pointed out in Section 17, and finally Section 18 concludes the paper.

2 Reasons for Code Duplication

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 accidents. In Figure 1 (the leaf nodes) we provide the various factors for which clones can be introduced in the source code [18, 31, 120, 122, 146, 178, 197, 135] where a short description for some of the factors are discussed below:

2.1 Development Strategy

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

2.1.1 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 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 [135].

Forking: The term *Forking* is used by Kapser and Godfrey [128] 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 hardware 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, functionalities and logic reuse: Functionalities and logic can be reused if there is already similar solution available. For example, often there is a high similarity

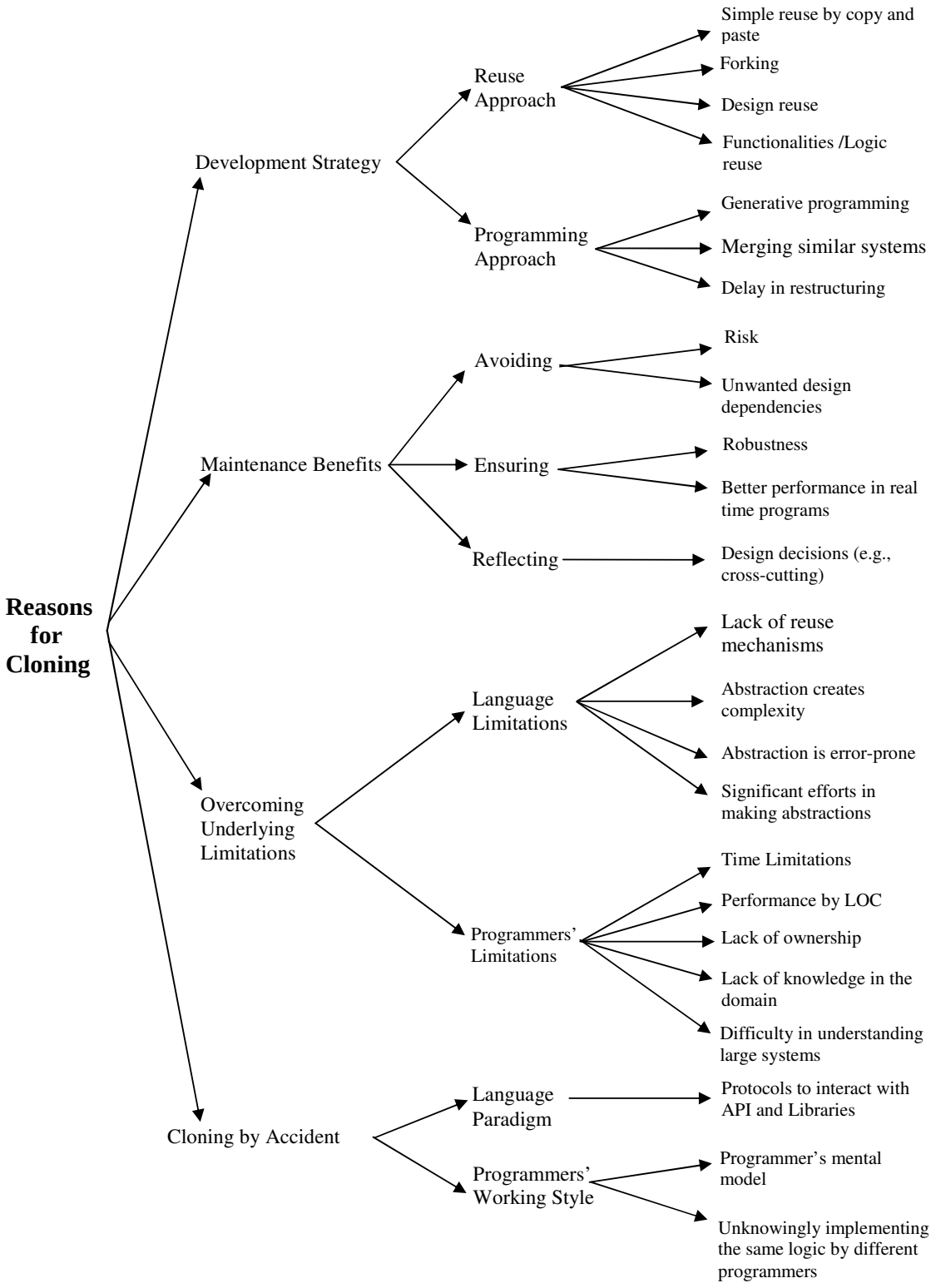

Figure 1: Tree-diagram for the Reasons for Cloning

between the ports of a subsystem. The different ports of a subsystem (especially, for OS's subsystems) are likely similar in their structure and functionality. For example, Linux kernel device drivers contain large rates of duplication [93] because all the drivers have the same interface and most of them implement a simple and similar logic. Moreover, the design of such systems does not allow for more sophisticated forms of reuse.

2.1.2 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 functionalities are merged to produce a new one. Although these systems may have been developed by different teams, clones may produce in the merged system because of the implementations of similar functionalities in both systems.

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

Delay in restructuring: It is also a common practice to the developers that they often delay in restructuring their developed code which may ultimately introduce clones.

2.2 Maintenance Benefits

Clone are also introduced in the systems to obtain several maintenance benefits. Examples are:

Risk in developing new code: Cordy [57] reports that clones do frequently occur in financial software as there are frequent updates/enhancements of the existing system to support similar kinds of new functionalities. Financial products do not change that much from the existing one, especially within the same financial institutions. The developer is often asked to reuse the existing code by copying and adapting to the new product requirements because of the high risk (monetary consequences of software errors can run into the millions in a single day) of software errors in new fragments and because existing code is already well tested (70% of the software effort in the financial domain is spent on testing).

Clean and understandable software architecture: To keep software architecture clean and understandable, sometimes clones are intentionally introduced to the system [128].

Speed up maintenance: As two cloned code fragments are independent of each other both syntactically and semantically, they can evolve at different paces in isolation without affecting the other and testing is only required to the modified fragment. Keeping cloned fragments in the system may thus speed up maintenance, especially when automated regression tests are absent [197].

Ensuring robustness in life-critical systems: Cloning/redundancy is incorporated intentionally while 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 Overcoming Underlying Limitations

Clones can be introduced due to the underlying limitations concerning the programming languages of interest and the developers.

2.3.1 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 reuse mechanism of programming languages: Sometimes programming languages do not have sufficient abstraction mechanisms, e.g., inheritance, generic types (called templates in C++) or parameter passing (missing from, e.g., assembly language and COBOL) and consequently, the developers are required to repeatedly implement these as idioms. Such repeating activities may create possibly small and potentially frequent clones [190, 27].

Significant efforts in writing reusable code: It is hard and time consuming to write reusable code. Perhaps, it is easier to maintain two cloned fragments than the efforts to produce a general but probably 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 the existing code and then reuse it by pasting with or without modification rather than making reusable code. Introduction of new bugs can be avoided in critical system functionality by keeping the critical piece of code untouched [97].

2.3.2 Programmer's Limitations

There are also several limitations associated with the programmers for which clones are introduced in the system. Examples are:

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

Time limit assigned to developers: One of the major causes of code cloning in the system is the 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 to their current needs.

Wrong method of measuring developer's productivity: Sometimes the productivity 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)[58]. In such situations, the only way of reusing the existing code is to copy and paste with required adaptations.

2.4 Cloning By Accident

Clones may be introduced by accidents. Examples are:

Protocols to Interact with APIs and Libraries: The use of a particular API normally needs a series of function calls and/or other ordered sequences of commands. For example, when creating a button using the Java SWING API, a series of commands is to create the button, add it to a container, and assign the action listeners. Similar orderings are common with libraries as well [128]. Thus, the uses 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 kind of problems using the common solution pattern of his/her memory to such similar problems. Therefore, several clones may unknowingly be created to the software systems.

3 Drawbacks of Code Duplication

The factors behind cloning described in Section 2 are reasonable and consequently, clones do occur in large software systems. While it is beneficial to practise cloning, code clones can have severe impacts on the quality, reusability and maintainability of a software system. In the following we list some of the drawbacks of having cloned code in a system.

Increased probability of bug propagation: 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 [118, 169].

Increased probability of introducing a new bug: 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 [117, 31].

Increased probability of bad 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 [185].

Increased difficulty in system improvement/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 becomes difficult to add new functionalities in the system, or even to change existing ones [120, 178].

Increased maintenance cost: If a cloned code segment is found to be contained 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 work to be done [178, 185].

Increased 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 [120] 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.

4 Advantages and Applications of Detecting Code Clones

In addition to the direct benefit of knowing how to improve the quality of the source code by refactoring the cloned code, there are several other benefits and applications of detecting clones. We list some of those as follows:

Detects library candidates: Davey et al. [60] and Burd&Munro [43] 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.

Helps in program understanding: If the functionality of a cloned fragment is comprehended, it is possible to have an overall idea on the other files containing other similar copies of that fragment. For example, when we have a piece of code managing memory we know that all files which contain a copy must implement a data structure with dynamically allocated space [197].

Helps aspect mining research: Detecting similar code is also required in aspect mining for detecting cross-cutting concerns. The code of cross-cutting concerns is typically duplicated over the entire application and could be identified with clone detection tools [40, 41].

Finds usage patterns: If all the cloned fragments of a same source fragment can be detected, the functional usage patterns of that fragment can be discovered [197].

Detects malicious software: Clone detection techniques can be used in finding malicious software. By comparing one malicious software family to another, it is possible to find the evidence where parts of one software system match parts of another [217].

Detects plagiarism and copyright infringement: Finding similar code may also useful in detecting plagiarism and copyright infringement [217, 18, 122].

Helps software evolution research: Clone detection techniques are successfully used in software evolution analysis by looking at the dynamic nature of different clones in different

versions of a system [7, 8, 93, 71, 181].

Helps in code compacting: Clone detection techniques can be used for compact device by reducing the source code size [49, 61].

5 Harmfulness of Cloning: A justification

Clones are generally considered harmful to the quality of source code [8, 18, 31, 47, 74, 120, 125, 146, 178]. One of the main drawbacks (c.f., Section 3 for details) of code clones is that changes to one code segment may need to be propagated to several other similar ones [92]. In order to overcome such difficulties automatic refactoring [21, 31] or aiding developers with manual refactorings [106] of duplicated code has been proposed. However, there are some recent studies [137, 136, 128] 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 [208], decreased program comprehensibility [205] and increased system size [21].

In an industry setting study [135], Kim et al. have observed that "*refactorings may not always improve the software with respect to clones*" and "*skilled programmers often created and managed code clones with clear intent*" [136]. In a large scale case study, Kapser and Godfrey have noticed that code cloning can often be used in a positive way [128, 129]. From several case studies [124, 126, 123], Kapser and Godfrey introduce the notion of categorizing high level patterns of cloning in a similar way to the cataloging of design patterns or anti-patterns. They found eight cloning patterns [128]. For each of the patterns, they studied both the advantages and disadvantages of these patterns of cloning to software development and maintenance. Their study concludes 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 and the developer/manager should be confirmed any action(s) to take about duplications.

Cordy [57] also reports that cloning is frequently used in large financial software for reusing existing design or for separating the dependencies of custom views on data that some modules or applications may have. Existing code is tested extensively and therefore, reusing such code prevents the introduction of bugs to the system, and in the same time limits testing to a subset of the copied code that is modified. He also observes that fixing bugs to all the similar segments may not be expected by the developers as this may break dependent code. Therefore, changing or refactoring the clones is not advisable from a risk management point of view [57].

Geiger et al. [92, 91] 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 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. Their study shows that a reasonable amount of

such relation exists supporting the argument that clones can have a bad impact on software maintenance.

For evaluating the harmfulness of cloning, another change based experiment has been conducted by Lozano et al. [171]. They have developed a prototype tool to justify whether cloning is harmful or not. Rather than working on file level granularity as in Geiger et al. [92] or disregarding parts of the system's history (only considers co-changes) as in Kim et al. [136], they have focused on the method level. Their tool, CloneTracker, focuses on those methods that had clones in the past for some time and then determines the frequencies of both changes and co-changes of such methods both when they have clones and when not. Their study shows that although frequency of co-changes between the cloned methods is less than the methods without clones, the cloned methods change much more frequently than the methods without clones. Their results seem to support the argument that although consistent changes between the cloned fragments are not carried out (developers may unaware of other similar fragments), clones do have a bad impact on software evolution/maintenance. However, their study was based on a small sized piece of software, DNSJava, developed by only two programmers, which limits the generality of their findings.

Another similar study was conducted by Aversano [9]. They considered co-change analysis (Modification Transactions(MTs) extracted from source code repositories) to verify how detected clones in a given release of 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 [155] with five open source Java, C/C++ systems. As usual, 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 groups are inconsistent changes. His study also showed that the occurrences of becoming inconsistently changed clone groups to consistently changed clone groups are very few. Inconsistent changes to clone groups are directly related to the maintenance problems (e.g., bug-fixing or update). Thus, his study shows that clones do have a bad impact on software maintenance. A similar finding was noted by Kim et al. [136] that the number of consistent changes are fewer than anticipated in the evolution versions of a software system.

From the above 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. What we can say with confidence is that awareness of clones in a software system is important.

6 Clone Relation Terminologies

Clone detection tools report clones in the form of *Clone Pairs* (CP) or *Clone Classes* (CC) or both. These two terms speak about the 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) [122]. A clone-relation holds between

two code portions if (and only if) they are the same sequences. Sequences are sometimes original character strings, strings without whitespace, sequences of token type, transformed token sequences and so on. In the following we define *clone pair* and *clone class* in terms of the clone relation:

Clone Pair: A pair of code portions/fragments is called a clone pair if there exists a clone-relation between them, i.e., a clone pair is a pair of code portions/fragments which are identical or similar to each other. For the three code fragments, Fragment 1 (F1), Fragment 2 (F2) and Fragment 3 (F3) of Figure 2, we can get five clone pairs, <F1(a), F2(a)>, <F1(b), F2(b)>, <F2(b), F3(a)>, <F2(c), F3(b)> and <F1(b), F3(a)>. By considering the maximum possible extent of cloned segments, we get basically four clone pairs, <F1(a + b), F2(a + b)>, <F2(b + c), F3(a + b)> and <F1(b), F3(a)>.

Clone Class: A clone class is the maximal set of code portions/fragments in which any two of the code portions/fragments hold a clone-relation, i.e., form a clone pair. For the three code fragments of Figure 2, we get a clone class of <F1(b), F2(b), F3(a)> where the three code portions F1(b), F2(b) and F3(a) form clone pairs with each other, i.e., there are three clone pairs, <F1(b), F2(b)>, <F2(b), F3(a)> and <F1(b), F3(a)> do exist too. A clone class is therefore, the union of all clone pairs which have code portions in common [196]. Clone classes are also called clone communities [178].

Clone Class Family: The group of all clone classes that have the same domain is called a clone class family [196]. Such a clone class family is also termed *super clone* by Jiang et al [114]. In their context: *Multiple clone classes between the same source entities are aggregated into one large super clone.*

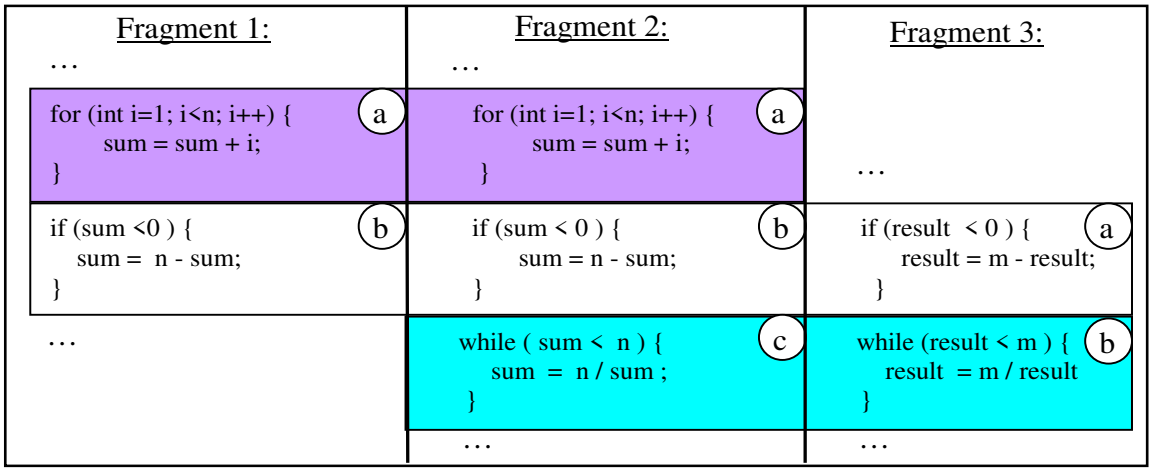

Figure 2: Clone Pair and Clone Class

7 Clone Definitions in the Literature

Clone detection is a research problem where there is no precise 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 depends on the underlying algorithms and thresholds used. 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 section, we provide the different definitions and types of clones from the literature.

7.1 Code Clone and Its Definitional Vagueness

A code fragment that has identical or similar code fragment(s)to it in the source code, in general, terms as code clone. A copied fragment can be used with or without minor modifications in a system by the developer. If there is 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 definitions about code clone [159]. In the following we provide clone definitions from the literature along with their associated vagueness.

Baxter et al. [31] 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 more vaguer definition is provided by Kamiya et al. [122]. They 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 similar vague definition is proposed and used by Burd et al. [44] in their tool evaluation experiment where a code segment is termed as clone if there is/are a second or more 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. [31], detection dependent threshold-based definitions of the terms "similar" or "minor" are attempted by several authors [148, 169, 126]. Attempts of automatically combining multiple detector result sets are also considered to overcome such similarity problems [36, 35, 183, 148]. 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 the clone definition is attempted in the form of taxonomies. For instance, Mayrand et al. [178] provide an ordinal scale of eight different types of clones, of which some have simple, crisp definitions. For example, the category "DistinctName" refers to the clones where only identifiers names can be differed between the cloned segments. However, their ordinal scale is not sufficient towards a sound definition of clone. For instance, they define a category "SimilarExpression" to identify clones with expressions that differ but yet are still "similar". Similarly, Balazinska et al. [22] 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 categories "One long difference" to mean one unit token-sequence difference in an expression or in a statement or in other part of the function body, "Two long differences" to mean changes in two units and "Several long differences" to mean changes in three or more units, all of which involve kind of vagueness.

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 [122, 123, 136]. Other studies argue that measuring clone size with respect to the number of lines could be a better option. However, there is also disagreement among people on the minimum clone size with respect to number of lines. For instance, in Bellon's tool comparison experiment [36, 34], the minimum clone size was set to 6 unprocessed lines of code. On the other hand, Baker [18] has set the minimum threshold to 15 non-commented lines while Johnson [120] to 50 lines. Some studies consider the no.of AST/PDG nodes as the measure of clone size and provide a measure of thresholds [169, 142]. Some studies work with only function clones and limit their clone size to the function body of any size [178, 158].

Human judgment of code clones is also a major issue and varies among experts [215]. In one of their experiments, for more than 60% of automatically detected clones, three experts disagreed whether the fragments are really code clone or not.

Language-specific semantic issues are also problematic in defining code clones. For example, whether there is any relation between the Java *clone()* method and the code clones we are talking about. Java *clone()* method is used for *object* duplication and treated as a reuse mechanism. We are particularly interested in code duplication and thus, even if the Java *clone()* method copies an *object*(actually 'by reference'), there is very little chance that the original *object* (e.g., a class) and the code that uses the *clone()* method would be similar in their text. Thus, we do not think that they should be considered clones. However, it is still questionable whether they are semantically similar. As both the original object and code segment using the *clone()* method are similar in their functionalities, we think that they should be considered as clones when considering semantic similarity. Similar other such situations may arise depending on the programming language of interest and hence, we think that language-specific semantic analysis is required for detecting semantic clones.

As we see from the above discussion, the definition of clone and its minimum size are dependent on the detection approach along with the additional burden of associated vagueness. The first attempt to define a detection independent definition of exact and near-miss clones was considered by Giesecke [87] where he pointed out the following desirable properties of a clone definition:

Independent of a Programming Language: Rather than finding code clones which 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, 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 clones should be independent of the detection approaches. A developer can identify whether two code fragments form a clone or not. Our 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 results of extensive editing two fragments may 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.

7.2 Code Clone Types

There are basically two kinds of similarities between two code fragments. Two code fragments can be similar based on the similarity of their program text or they can be similar in their functionalities without being textually similar. The first kind of clones are often the result of copying a code fragment and then pasting to another location. In this section, we consider clone types based on the kind of similarity two code fragments can have:

- Textual Similarity: Based on the textual similarity we distinguish the following types of clones [35, 34, 153]:

  Type I: Identical code fragments except for variations in whitespace (may be also variations in layout) and comments.

  Type II: Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.

  Type III: Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.

- Functional Similarity: If the functionalities of the two code fragments are identical or similar i.e., they have similar pre and post conditions, we call them semantic clones [142, 156, 184, 60] and referred as *Type IV* clones.

  Type IV: Two or more code fragments that perform the same computation but implemented through different syntactic variants.

These types of clones not only define an increasing level of subtlety from *Type I* through *Type IV* but also the analytical complexity and sophistication in detecting such clones increases from *Type I* through *Type IV* with *Type IV* being the highest. The detection of *Type IV* clones is the hardest even after having a great deal of background knowledge about the program construction and software design. This increasing level of analytical complexity from *Type I* through *Type IV* does not vary whether the process is automatic or not. In the following subsection, we describe each of the types with an example:

7.2.1 Type I Clones

In *Type I* clones, a copied code fragment is the same as the original. However, there might be some variations in whitespace (blanks, new line(s), tabs etc.), comments and/or layouts. *Type I* is widely know as *Exact clones*. Let us consider the following code fragment,

```
if (a >= b) {
    c = d + b; // Comment1
    d = d + 1;}
else
    c = d - a; //Comment2
```

An exact copy clone of this original copy could be as follows:

```
if (a>=b) {
     // Comment1'
    c=d+b;
    d=d+1;}
else  // Comment2'
    c=d-a;
```

We see that these two fragments are textually similar (even line-by-line) after removing the whitespace and comments. However, even after removing the comments and whitespace, the following code fragment is not similar to the previous two on a line-by-line basis as the positions of the "{" and "}" are changed in the code. Nevertheless, this fragment is also *Type I* exact copy clone of the other two.

```
if (a>=b)
   { // Comment1''
    c=d+b;
    d=d+1;
    }
else  // Comment2''
    c=d-a;
```

A typical line-by-line technique may fail to detect such clones that vary in layout.

7.2.2 Type II Clones

A *Type II* clone is a code fragment that is the same as the original except for some possible variations about the corresponding names of user-defined identifiers (name of variables, constants, class, methods and so on), types, layout and comments. The reserved words and the sentence structures are essentially the same as the original one. Let us consider the following code fragment.

```
if (a >= b) {
    c = d + b; // Comment1
```

```
        d = d + 1;}
    else
        c = d - a; //Comment2
```

A *Type II* clone for this fragment can be as follows:

```
if (m >= n)
    { // Comment1'
    y = x + n;
    x = x + 5; //Comment3
    }
else
    y = x - m; //Comment2'
```

We see that the two code segments change a lot in their shape, variable names and value assignments. However, the syntactic structure is still similar in both segments.

7.2.3 Type III Clones

In *Type III* clones, the copied fragment is further modified with statement(s) changed, added and/or deleted. Consider the original code segment,

```
    if (a >= b) {
        c = d + b; // Comment1
        d = d + 1;}
    else
        c = d - a; //Comment2
```

If we now extend this code segment by adding a statement $e = 1$ then we can get,

```
    if (a >= b) {
        c = d + b; // Comment1
        e = 1; // This statement is added
        d = d + 1; }
    else
        c = d - a; //Comment2
```

This copied fragments with one statement inserted is called *Type III* code clone of the original with a gap of one statement inserted.

Another example of *Type III* is shown in Figure 3 whereas the difference between the two code segments are presented in Table 1.

From the two fragments and from the corresponding difference in the table, all the original statements are used directly or after being changed in their identifiers or literals with one insertion (*synchronized*) in the first line, making this code fragment as *Type III* code clone. Without this inserted statement, this copied fragment could be a *Type II* code clone.

```
Fragment 1:
 public int getSoLinger() throws SocketException{
    Object o = impl.getOption( SocketOptions.SO_LINGER);
    if (o instanceof Integer) {
       return((Integer) o).intValue();
     }
    else return -1;
   }

Fragment 2:

   public synchronized int getSoTimeout()
                         throws SocketException{
    Object o = impl.getOption( SocketOptions.SO_TIMEOUT);
   if (o instanceof Integer) {
       return((Integer) o).intValue();
    }
    else return -0;
   }
```

Figure 3: Cloned methods in JDK(from [22])

Table 1: Difference between two code fragments of Figure 3 (from [22])

Fragment1	**Fragment2**	**Status**
<public>	<public>	Exact Copy
ϵ	<synchronized>	Insertion
<int>	<int>	Exact Copy
<getSoLinger>	<getSoTimeout>	Replacement
<() throws...SocketOptions .>	<() throws...SocketOptions .>	Exact Copy
<SO_LINGER>	<SO_TIMEOUT>	Replacement
<) ; ...else return>	<) ; ...else return >	Exact Copy
<-1 >	<-0>	Replacement
<;}>	< ;} >	Exact Copy

7.2.4 Type IV Clones

Type IV clones are the results of semantic similarity between two or more code fragments. In this type of clones, the cloned fragment is not necessarily copied from the original. Two code fragments may be developed by two different programmers to implement the same kind of logic making the code fragments similar in their functionality. Functional similarity reflects the degree to which the components act alike, i.e., captures similar functional properties and similarity assessment methods rely on matching of pre/post-conditions. Let us consider the following code fragment 1, where the final value of 'j' is the factorial value of the variable VALUE.

```
Fragment 1:
```

```
int i, j=1;
  for (i=1; i<=VALUE; i++)
       j=j*i;
```

Now consider the following code fragment 2, which is actually a recursive function that calculates the factorial of its argument n.

Fragment 2:

```
int factorial(int n) {
 if (n == 0) return 1 ;
 else        return n * factorial(n-1) ;
}
```

From the semantics point of view both the code fragments are similar in their functionality and termed as *Type IV* semantic clones although one is a simple code fragment and another is a recursive function with no lexical/syntactic/structural similarities between the statements of the two fragments.

7.3 Code Clone Terms

Although generally there are only four types of clones (c.f., Section 7.2), people use different terms when referring to the clone relation for their experiments. Almost all of them use the same term "Exact Clones" to refer to the identical code fragments. Similarly many of them use the term "Near-Miss Clones" to refer to identical code fragments with statement(s) added, deleted and/or modified. In this section, we list the different clone terms from the literature and map them to the four commonly used types of Section 7.2.

7.3.1 Exact Clones

Two or more code fragments are called exact clones if they are identical to each other with some differences in comments and whitespace or layout. Editing activities like changing the comments, restructuring in layout i.e., changing the positions of $begin - end$ brackets (e.g., "{" and "}") or other language elements through adding/removing tabs, blanks, new lines may have been applied in the copied fragment. Line-based methods may not detect some exact clones those are edited through adding/removing new lines in changing the position of language elements.

Exact clones are essentially *Type I* clones and details can be found in Section 7.2.1 above.

7.3.2 Renamed Clones

People use the term renamed clones when identifier names, literals values, comments or whitespace changes in the copied fragments. Thus, a renamed clone is essentially a *Type II* clone. A parameterized clone is a renamed clone but not vice versa. Consistent renaming is a must in the parameterized clones which is not necessarily required in the case of renamed clones. Consider the following code segment.

```
If (a > b)
    { b++ ;
      a =1;}
```

With consistent renaming, e.g., if a is replaced with i and b is replaced with j, we can get a parameterized clone like the following segment.

```
If (i > j)
    { j++ ;
      i =1;}
```

This is of course, a renamed clone too. However, renamed clones are more general in the sense that there is no need of consistent renaming of identifiers. Therefore, the following code fragment is a renamed clone of the original.

```
If (i > j)
    { i++ ;
      j =1;}
```

7.3.3 Parameterized Clones

A parameterized clone or p-match clone is a renamed clone with systematic renaming. The clone detector looks for consistent name matching rather than normalizing all identifiers and/or literals to a especial symbol. Parameterized clones are thus a subset of *Type II* clones. Consider the code segment,

```
 if (a > b)
    { b++ ;
     a =1;}
```

A parameterized clone of the segment above can be as follows where identifier a is renamed to i and b is renamed to j consistently.

```
If (i > j)
    { j++ ;
      i =1;}
```

On the other hand, the following fragment is not a parameterized clone of the above as consistent renaming is not followed here.

```
If (i > j)
    { i++ ;
      j =1;}
```

However, it is a renamed clone of the original (first one). Any technique with a little normalization on the identifiers can find this code segment as clone.

Let us consider another example, An example of a p-match is given in Figure 4, which contains two code fragments taken from the X Window System source code. The fragments

```
Fragment 1:

 copy_number(&pmin, &pmax,
     pfi->min_bounds.lbearing,
     pfi->max_bounds.lbeaing);
 *pmin++ = *pmax++ = ',' ;
  copy_number(&pmin, &pmax,
      pfi->min_bounds.rbearing,
      pfi->max_bounds.rbearing) ;
    *pmin++ = *pmax++ = ',';


Fragment 2:

 copy_number(&pmin, &pmax,
     pfh->min_bounds.left,
     pfh->max_bounds.left);
 *pmin++ = *pmax++ = ',' ;
  copy_number(&pmin, &pmax,
      pfh->min_bounds.right,
      pfi->max_bounds.right) ;
    *pmin++ = *pmax++ = ',';
```

Figure 4: P-match Clones (from [18])

are identical except for the differing indentation (which is ignored by dup [18]) and the correspondence between the variable names pfi/pfh and the pairs of structure member names lbearing/left and rbearing/right . These fragments are excerpted from two 34-line sections of code that are a p-match with these parameter correspondences.

7.3.4 Near-Miss Clones

Near-miss clones are those clones where the copied fragments are very similar to the original. Editing activities such as changing in comments, layouts, changing the position of the source code elements through blanks and new lines, changing the identifiers, literals, macros may have been applied in such clones which actually implies that all parameterized and renamed clones are near-miss clones. A copied fragment which is not exact copy of the original due to slight changes but the syntactical structure is still the same as the original. Basically, all clones of *Type II* (c.f. Section 7.2.2) are near-miss clones. However, many of the authors also assume that a slight modification within a statement(s) or even addition and deletion of statement(s) in the copied fragment may not make the copied fragment different from the original and hence can be treated as near-miss clones. In this sense, *Type III* clones may also be termed as near-miss clones (c.f. Section 7.2.3).

7.3.5 Gapped Clones

A gap clone code is partly similar to the original segment. In this type of clones, there is some different code portion between the segments. This different code portion is known as a *gap*[210]. Let us consider the code segment,

```
If (a > b) { b++ ; a =1;}
```

If this fragment is copied and then pasted for further reuse, there might be three different types of gap between the code fragments with the original one.

- No renaming/renaming and code insertion: Identifiers can be renamed, values can be changed and some new statements can be inserted into the copied fragment but syntactic structure is same for the existing statements. For example,

```
If (i > j)
 { i = i/2; //inserted
   // Comment
   i++; //variables altered
   j =0; // variables altered and values changed
 }
```

- No renaming/renaming and Code Deletion: Identifiers can be renamed, values can be changed and some statements can be deleted from the copied fragment but syntactic structure is same for the existing statements. For example,

```
If (i > j)
 {
   // one statement is deleted from here
   j =0; // variables altered and values changed
 }
```

- No renaming/renaming and Code Modification: Identifiers can be renamed, values can be changed and some statements can be modified in the copied fragment. As statement(s) is modified there might be different syntactic structure between the two code segments. For example,

```
If (i > j)
 {
   // Comment
   i =  i+ 1 // this statement is modified from the original one
   j =0; // variables altered and values changed
 }
```

7.3.6 Structural Clones

Software components can be compared with various degrees of accuracy. Structural similarity reflects the degree to which the software specifications look alike, i.e., have similar design structures. Structural clones are simple clones within a syntactic boundary following syntactic structure of a particular language. These boundaries can be function boundary, statement boundary, class boundary etc. depending on the programming language of interest. As an example some synopsis of structural clones for java language are as follows:

- Declaration: class { ... } , interface {...}
- Method: method, constructor, static initializer
- Statement: if statement, for statement, while statement, do statement, switch statement, try statement synchronized statement.
- Block range surrounded with '{' and '}'

In the same way, we can define the synopsis of structural clones for other languages of interest. CCShaper [107] is a tool that can extract structural clones from the output of *CCFinder* for java programs. Structural clones are very good candidates for refactoring.

Structural clones can be based with any level of similarity, i.e., a structural cone can be an exact clone, parameterized clone, renamed clones, gapped clones and so on. Structural clones focus on finding similar design structures after identifying the basic similarities like textual, lexical, syntactical and/or semantical similarities. While the types of clones are based on the level of similarity between the code fragments, structural clones are based on the level of clone granularity where the granularity can be any syntactic boundary (e.g., $begin-end$ block) of the language. A structural clone can be any of the four types of clones (c.f., Section 7.2) based on its similarity level. There is also another kind of structural clones [26] and are discussed in Section 7.3.11.

7.3.7 Function Clones

Function clones are simply clones that are restricted to refer to entire functions or procedures[1]. Function clones are therefore, a subset of structural clones (c.f. Section 7.3.6). As with structural clones, function clones can be any of the four types of clones (c.f., Section 7.2) based on it similarity level.

7.3.8 Non-contiguous Clones

Non-contiguous clones are are kind of near-duplication where gaps are allowed between the code fragments and therefore, non-contiguous clones are basically gapped clones. All the editing activities that allowed for gapped clones are also allowed for non-contiguous clones. Let us consider the there fragments of code from the Unix utility bison that contain a group of three clones identified by Komondoor et al. [140]. The clones are indicated by "++" signs. While the clone in Fragment 3 is contiguous, the corresponding clones in Fragments 1 and 2 are non-contiguous. Like gapped clones, non-contiguous clones are *Type III* clones.

[1]The term "function clone" is already an accepted term so we shall adopt the term even if we mean either procedures or functions [22].

```
Fragment 1:
   while (isalpha(c) ||
         c== '_' || c== '-' {
++   if (p == token_buffer + maxtoken)
++       p = grow_token_buffer(p);
     if (c == '-' c = '_' ;
++    *p++ =c;
++    c = getc(finput);
      }

Fragment 2:
 while (isdigit(c)) {
++   if (p == token_buffer + maxtoken)
++       p = grow_token_butter(p);
     numval = numval*20 + c -'0';
++   *p++ =c;
++   c = getc(finput);
     }

Fragment 3:
 while (c != '>') {
    if (c == EOF) fatal();
    if (c == '\n' {
        warn("unterminated type name");
        ungetc(c, finput);
        break;
    }
++   if (p == token_buffer + maxtoken)
++       p = grow_token_buffer(p);
++   *p++ = c;
++    c = getc(finput);
    }
```

Figure 5: Non-contiguous Clones (from [140])

```
Fragment 1:
++ fp1 = LA+i*tokensetsize;
++ fp2 =lookaheadset;
++ while (fp2 < fp3 )
++     *fp2++ |= *fp1++;
```

```
Fragment 2:
++ fp1 = base ;
++ fp2 = F + j * tokensetsize;
++ while (fp1 < fp3)++
++     ++ *fp1++ |=*fp2++;
```

Figure 6: Reordered Clones in bison(from [140])

7.3.9 Reordered Clones

A reordering of some segments may be possible in the copied fragment that do not alter the data or control dependencies of the this fragment compare to the original. Let us consider the two code fragments from bison of Figure 6 identified by Komondoor & Horwitz [140]. Fragment 2 differs from Fragment 1 in two ways: the variables have been renamed (including renaming fp1 to fp2 and vice versa), and the order of the first and second lines has been reversed.

Even though some statements are reordered and variables are renamed, the functionalities of Fragment 1 are the same as the Fragment 2. On the basis of semantic similarity, reordered clones are of *Type IV* clones. However, considering the lexical similarity, there are essentially gaps in the copied fragments and reordered clones can be a sort of gapped clones and therefore, are classified as *Type III* clones.

7.3.10 Intertwined Clones

Sometimes two similar code segments might be intertwined. Rather than making one segment, developer may feel comfortable having them intertwined. Consider the two code fragments of Figure 7 (upper portion). Both segments are similar in its context with few exceptions and hence are considered as clones. These two similar code segments are implemented in one function by intertwining different code lines as in the lower part of this figure. One clone is indicated by "++" signs while the other clone is indicated by "xx" signs. The clones take a character pointer (a/b) and advance the pointer past all blank characters, also setting a temporary variable (tmpa/tmpb) to point to the first non-blank character. The final component of each clone is an "if" predicate that uses the temporary. Komondoor and Horwitz [141] applied backward slicing to find such intertwined clones. From the lexical point of view, such clones are hard to find as they lines are intertwined with each other, consequently a lexical similarity cannot be found. However, both segments are similar with respect to their functionalities and hence intertwined clones are considered as *Type IV* clones.

7.3.11 Design Level Structural Clones

Basit and Jarzabek [26] classify two types of clones. One is simple clones and another is structural clones. By simple clones they mean all the clone terms we discussed above i.e., all clones that fall in any of the four clone types, from *Type I* to *Type IV*. They use these

```
Fragment 1:                                  Fragment 2:
 tmpa = UCHAR(*a);                           tmpb = UCHAR(*b);
 while (blanks[tmpa])                        while (blanks[tmpb])
   tmpa = UCHAR(*++a);                       tmpb = UCHAR(*++b);
   if (tmpa == '-'
 ...                                         else if (tmpb == '-' ..
-----------------------------------------------------------------
-----------------------------------------------------------------
 ++ tmpa = UCHAR(*a);
xx tmpb = UCHAR(*b);
++ while (blanks[tmpa])
++   tmpa = UCHAR(*++a);
xx while (blanks[tmpb])
xx   tmpb = UCHAR(*++b);
++ if (tmpa == '-'
 ...
xx else if (tmpb == '-' ..
```

Figure 7: Intertwined Clones in Unix utility "sort" (from [140])

simple clones to find design level similarities and called those design similarities structural clones. For example, two different clone sets that often occur together in program files are examples of structural clones. They first use *CCFinder* to find simple clones and then apply an itemset data mining algorithm to correlate simple clones for finding design level similarities. As we have already used the term structural clones to refer to other types of clones in Section 7.3.6 and as the clones introduced by Basit and Jarzabek are related to the design of the software system, we call these clones as design level structural clones. PR-Miner [170] also uses frequent itemset mining to detect implicit, high-level programming patterns for specification recovery or bug detection.

7.3.12 Ubiquitous clones

The term ubiquitous clones [136] is used to refer to the short clones that are present in the multiple source files in an application i.e., short clones with high frequency across files of a system are called ubiquitous clones. These clones are usually short methods that perform a specific task, such as returning a new rectangle object, drawing two ovals (including setting their colors) and setting the undo and redo flags for the drawing views.

7.4 Evolving Clones

Some clone terms are used to refer to the clones in the evolution life cycle of a system. In this section, we provide some clone terms that provide hints whether a clone is survived in all the versions of a system or disappeared with the evolution of the system [136, 8].

```
Fragment 1:                          Fragment 2:

return result; }                      return x;}

int foo() {                          int bar(){
          int a;                               int y;
```

Figure 8: Spurious Clones (from [153])

7.4.1 Volatile Clones

There may be clones within a system which disappear with the evolution of the software system due to maintenance activities (e.g., refactoring). These clones are called volatile clones. Any clones of *Type I* to *Type IV* could be under the classification of volatile clones [136].

7.4.2 Long-lived Clones

While volatile clones may disappear in maintenance activities (e.g., refactoring), there might be several clones that are locally unrefactorable and hence remain in all versions of the systems. These clones are called long-lived clones [136].

7.5 Problematic Clones

There are some clones that are detected by the detection methods but are meaningless or uninteresting with respect to the maintenance point of view and therefore, must be filtered out. In this section, we provide some examples of such clones.

7.5.1 Spurious Clones

Sometimes a particular detection method may detect clones that are not really clones in the maintenance perspective. For example, consider the code fragments in Figure 8 where the any technique may find these two code fragments as clones. If we consider token-based approach, we see that both code fragments produce the same token sequence, *return id ; int id ( ) int id;* and consequently returns as clones [153].

Clones with the ending lines of one function and beginning lines of another are not useful to a maintenance programmer even though from a lexical point of view these are in fact rightful clones. There may be several such clones that do not follow a syntactic structure and hence become useless from the maintenance point of view and known as spurious clones.

7.5.2 Frequently false positive clones

There are several clones that are not interesting or considered as false positives in the analysis process. Several of such clones are presented in this section [104].

- Consecutive simple method declarations: Consecutive simple method declarations are found as code clones coincidentally just like the case of consecutive accessor declarations.

```
Fragment 1:
public static boolean isAbstract(int access_flags) {
   return(access_flags & ACC_ABSTRACT) != 0; }

Fragment 2:
public static boolean isPublic(int access_flags) {
   return (access_flags & ACC_PUBLIC) != 0; }

Fragment 3:
public static boolean isStatic(int access_flags) {
    return (access_flags & ACC_STATIC) != 0; }

Fragment 4:
public static boolean isNative(int access_flags) {
    return (access_flags & ACC_NATIVE) != 0; }
```

- Consecutive method invocations are detected as code clones. It is not worthwhile that users see these code clones in the process of code clone analysis because there is nothing they can do about them in a maintenance perspective.

```
Fragment 1:
out.println();
out.println(``---------------------------------------'');
out.println(``ANT_HOME/lib jar listing``);
out.println(``---------------------------------------'');
doReportLibraries(out);

Fragment 2:
out.println();
out.println(``---------------------------------------'');
out.println(``Tasks availability'');
out.println(``---------------------------------------'');
doReportTasksAvailability(out);
```

- Consecutive *if-statements* and *if-else statements* are detected as code clones. These code clones implement verifications of variable states. It is obvious that these code clones are harmless in the context of software maintenance, and users are needless to see them in the process of code clone analysis.

```
Fragment 1:
```

```
 if (null != storepass) {
     cmd.createArg().setValue(‘‘-storepass‘‘);
     cmd.createArg().setValue(storepass);
 }

Fragment 2:
 if (null != storetype) {
     cmd.createArg().setValue(‘‘-storetype‘‘);
     cmd.createArg().setValue(storetype);
 }

Fragment 3:
 if (null != keypass) {
     cmd.createArg().setValue(‘‘-keypass‘‘);
     cmd.createArg().setValue(keypass);
 }
```

- Consecutive case entries are found as code clones coincidentally just like the case of consecutive accessor declarations. Usually, the programmer implements simple instructions in case entries. There are several methods that replace all user-defined names into the same special token. Thus consecutive case entries tend to be detected as code clones, but they are harmless in the context of software maintenance.

```
Fragment 1:
case Project.MSG_ERR:
     msg.insert(0, errColor);
     msg.append(END_COLOR);
     break;

Fragment 2:
case Project.MSG_WARN:
     msg.insert(0, warnColor);
     msg.append(END_COLOR);
     break;

Fragment 3:
case Project.MSG_INFO:
     msg.insert(0, infoColor);
     msg.append(END_COLOR);
     break;
```

- Consecutive variable declarations are found as code clones coincidentally just like the case of consecutive accessor declarations and they should not be detected as code clones.

```
private MenuBar iAntMakeMenuBar = null;
private Menu iFileMenu = null;
private MenuItem iSaveMenuItem = null;
private MenuItem iMenuSeparator = null;
private MenuItem iShowLogMenuItem = null;
private Menu iHelpMenu = null;
private MenuItem iAboutMenuItem = null;
```

- Consecutive assign statements are found as code clones coincidentally just like the case of consecutive accessor declarations and they should not be detected as code clones.

```
src = attributes.getSrcdir();
destDir = attributes.getDestdir();
encoding = attributes.getEncoding();
debug = attributes.getDebug();
optimize = attributes.getOptimize();
deprecation = attributes.getDeprecation();
depend = attributes.getDepend();
verbose = attributes.getVerbose();
```

- Consecutive catch statements are detected as duplicated fragments. Their existence is due to the specification of Java language, and they should not be detected as code clones.

```
catch (final ClassNotFoundException cnfe) {
  throw new BuildException(cnfe);
 }

catch (final InstantiationException ie) {
  throw new BuildException(ie);
}

catch (final IllegalAccessExceptioniae) {
  throw new BuildException(iae);
}
```

- Consecutive while-statements are detected as code clones. In this case, the logics of each while-statement are very simple, and it is no problem to filter out them. But if their logics are complex, they should not be filtered out.

```
e = ccList.elements();
while (e.hasMoreElements()) {
    mailMessage.cc(e.nextElement().toString());
```

```
}

e =bccList.elements();
while (e.hasMoreElements()) {
    mailMessage.bcc(e.nextElement().toString());
}
```

7.6 Clone Types Summary

From Section 7.3 above, we can see that there are several types of clones used in the literature. In Table 2, we provide a summary of most frequently used clone terms in a tabular form where the 1st column shows the clone terms, 2nd column indicates what kind of editing activities normally taken place for such a clone term, the 3rd column shows the citations that introduced and/or used that particular term, the columns 4-6 represent the general types (c.f., Section 7.2) to which this clone term belongs to, and the last column shows the clone granularity of that corresponding term. Clone granularity can be fixed (e.g., method level, class level etc) before the comparison or can be free (fragment without any boundary) and made with atomic units (e.g., lines) after the comparison.

8 Towards a Taxonomy of Clones

Clone taxonomies can be useful for optimization of detection and reengineering techniques. By knowing the frequencies with which different categories of clones occur in source code, we can concentrate our efforts on the most prominent types or on the types which seem most relevant to the reengineering task at hand. In the following, we categorize the different clone taxonomies from the literature based on three attributes, similarities between the clones, location of the clones in the source code and refactoring opportunities with the detected clones.

8.1 Taxonomies Based on Similarity

Starting from the ideal of perfect clones comprised of two exact copies, these taxonomies measure which syntactic elements have been changed by the programmer after copying. For example, high-similarity clones include methods that are the same except for the name, or methods that are the same but for the types of parameters. This kind of information usually very directly suggests a refactoring. Examples are:

8.1.1 Mayrand et al. Taxonomy

Mayrand et al. [178] define an ordinal scale of eight clone levels for functions, going from exact copy, as the most obvious form of duplication, to clones which have differing control flow. Each level is defined as a set of metrics which must have the same value for all the clones in a given category. This simple categorization can only inform about basic refactoring directions, however. They provide three kinds of similarity between two functions based on four points of comparisons. They also define an ordinal scale of cloning. In the following, we provide a short description of their taxonomy:

Table 2: Summary of commonly used clone terms in the literature

Clone Term	**Major Editing Activities**	**Introduced/ Referenced By**	**Possible Types**				**Granularity**	
			Typ I	Typ II	Typ III	Typ IV	Fixed	Free
Exact Clones	Variations in comments and whitespace	almost all research	X	-	-	-	X	X
Parameterized Clones	Systematic renaming of identifiers	[14, 13, 18, 16, 12]	-	X	-	-	X	X
Renamed Clones	Renaming of identifiers	[105, 122, 72]	-	X	-	-	X	X
Near-miss clones	Renaming of identifiers and/or statement(s) modifications, insertions and/or deletions)	[56, 31, 72, 220]	-	X	X	-	X	X
Gapped Clones	Statement modifications, insertions and deletions	[210, 105]	-	-	X	-	X	X
Structural Clones	Any of the above	[56, 146]	X	X	X	X	X	-
Design Level Structural Clones	Any of the above	[29, 26]	X	X	X	X	X	-
Function Clones	Any of the above	[146, 178, 46, 158]	X	X	X	X	X	-
Non-contiguous Clones	Any of the above	[140, 141]	-	-	X	-	X	X
Chained Clones	Any of the above	[220, 223]	-	-	X	-	X	X
Reordered Clones	Reordering of statements and/or renaming of identifiers	[141, 140]	-	-	X	X	X	X
Intertwined Clones	Statements of both segments are intertwined and/or renaming of identifiers	[143, 141]	-	-	-	X	X	X

- Similarity between functions: Three levels of similarity are used in defining the similarity level between two functions. These are,
  - Equal Functions: Two functions are considered equal for a point of comparison if all of its associated metric values are same for both functions.
  - Similar Functions: Two functions are considered similar for a point of comparison if the absolute differences of the associated metrics of the functions are less than or equal to the delta thresholds defined for each of the metrics.
  - Distinct Functions: Two Functions are distinct for a point of comparison if there is at least one metric where the absolute difference of the functions is greater than the delta value defined for that particular metric.

- Points of Comparisons: Four points of comparisons, namely, name of the function, layout of the source code, expressions in the functions and control flow of the functions are used. Each of them is discussed below:

  - Name: Two functions with the same name are considered as clones.
  - Layout: If the layouts i.e., the visual organizations of the source code in terms of comments, indentation, blank lines and variable names of the two functions are similar then they are likely clones for this point of comparison. Several metrics such as the volume of declaration of comments, volume of control comments, number of logical comments, number of non-blank lines and average variable name length are used with some predefined delta thresholds for each of the metrics.
  - Expressions: The number of expressions in a function, their nature and their complexity are considered in defining the metrics for this point of comparison. Normally, total calls to other functions, unique calls to other functions, average complexity of decisions, number of declaration statements and number of executable statements are considered as the metrics for this point of comparison with some predefined delta thresholds.
  - Control Flow: The control flow characteristics of the functions, for example, the number of nodes, number of arcs, information related to decisions and information related to loops in a function are considered for this point of comparison.

- Ordinal Scale of Cloning: For identifying the clones, eight different strategies are used for defining an ordinal scale of cloning. The ordinal scale is: 1. ExactCopy, 2. DistinctName, 3. SimilarLayout, 4. DistinctLayout, 5. SimilarExpression, 6. DistinctExpression, 7. SimilarControlFlow and 8. DistinctControlFlow where bad programming style begins at scale 1 and then gradually ends at good programming style at scale 8.

8.1.2 Balazinska et al. Taxonomy

Balazinska et al. [22] propose a classification scheme for clone methods with 18 different categories. The categories detail what kind of syntax elements have been changed and also how much of the method has been duplicated.

- At the first instances, two categories, based on overall similarity:

  - Identical: strictly identical clones (i.e., exact clones of *Type I*).
  - Superficial changes: differences that do not affect the semantic meaning of the copied method with the original (e.g., names of local variables, names of parameters and name of the method).

- Then at the second level, three categories based on token differences and method attributes:

  - Differences affecting only one lexical token at a time,

  - Differences affecting sequences of tokens, and
  - Differences affecting the list of thrown exceptions and the attributes of methods (public, static, synchronized, list of thrown exceptions, etc.)

- Then at the 3rd level, based on the meaning of the single token differences (type of variable, name of a parameter, etc.) in method body:

  - Called methods: changes in some method calls.
  - Global variables: changes to non-local variables or constants.
  - Return types: changes to return type.
  - Parameter types: changes to parameter types.
  - Local variables: changes to the types of the local variables.
  - Constants: changes to constants hard-coded in the methods.
  - Type usage: changes to types explicitly manipulated in expressions such as "instanceof" or "typecast".
  - Interface changes: changes to called methods and/or global variables and/or parameters types and/or return type.
  - Implementation changes: changes to types of local variables and/or constants used and/or types explicitly manipulated.
  - Interface and implementation changes: changes to any of the previous categories.

- Then at the 4th level, based on token-sequence difference in function body:

  - One long difference: changes in one unit(an expression, a statement or other)
  - Two long differences: changes in two units.
  - Several long differences: changes in three or more units.

8.1.3 Bellon and Koschke Taxonomy

Bellon and Koschke [36, 35, 153] define three different clone types for the sake of a comparison between different detection tools: exact clones, parameterized clones, and clones that have had more extensive edits. This categorization is aimed at testing the detection and categorization capabilities of different tools.

The first three types of clones discussed in Section 7.2 are actually based on Bellon taxonomy of clones and therefore, the reader is referred to subsection 7.2.

8.1.4 Davey et al. Taxonomy

Davey et al. [60] provide a clone topology based on the level of similarity between the code fragments similar to Bellon and Koschke with one additional type as follows:

- Type I: An exactly identical source code, i.e., no changes at all.
- Type II: An exactly identical code clone, but with indentation, comments, or identifier (name) changes.

- Type III: A functionally identical clone, but with small changes made to the code to tailor it to some new function.
- Type IV: A functionally identical clone, developed possibly with the originator unaware that there is a function already available that accomplishes essentially the same function.

The above four types of clones are very similar to the types we provide in Section 7.2 with some minor exceptions. While Davey et al. provide this taxonomy, they work only with the first three types leaving the detection of *Type IV* as future work.

8.1.5 Kontogiannis Taxonomy

Kontogiannis [148] defines four basic types of clones, still based on the operational idea of duplication:

- i) exact clones where an identity function on each nonblank character maps fragment f_1 to fragment f_2,
- ii) clones that are exact except for systematically substituted variable names and data types,
- iii) clones where expression and statements have been modified, and
- iv) clones where statements and expressions have been either deleted or inserted.

These types are quite similar (but not the same as) to the types we have defined earlier in Section 7.2.

8.2 Taxonomies Based on Location and Similarity of Clones

These taxonomies focus on the location differences or the physical distance between clone instance locations. Refactoring opportunities or hindrances are derived from the fact that the source fragments are found in the same function, same file, or in files from different directories. In object-oriented systems, clone instances are located at specific places in the class hierarchy. To derive this kind of categorization for a clone pair, only rudimentary parsing technology suffices. Examples are:

8.2.1 Kapser and Godfrey Taxonomy

Kapser and Godfrey [126] provide a hierarchical classification of clones using attributes such as locations and functionality. Their taxonomy mainly consists of three partitions. First, clones are divided based on their physical location in the source. Clones at this level are classified whether they are within the same region in the file, within the same file, in different files but within the same directory, in different files of different directories. Second, clones are partitioned by the type of region they are found in. There can be clones between functions (Function to Function Clones), between two programming structures (e.g., unions, enumerators and structs), between macros or even between two different regions and

between external variable definition, prototypes and type defines. Third, the Function to Function clones are further subdivided based on their degree of similarity (degree of overlap or containment). Function to Function clones are subdivided as functions that are nearly the same, Function Clones; functions that are very similar, Partial Function Clones; functions where a large portion of one is cloned into another, Cloned Function Bodies; and small segments of code are shared between two functions, Clone Blocks. Clone Blocks are further subdivided based on their location and roles within a function. Clones can be occurred at the beginning or end of functions. There can be clones in the functions between the loops, between switch statements, between if-statements, between several conditional or partially match conditions. In Figure 9, we see the detail taxonomy of Kapser and Godfrey. They also provide why these kind of clones are introduced into the system and common problems caused by them, as well as reengineering scenarios to remove the clones from the system [125, 124, 127].

8.2.2 Moden et al. Taxonomy

Monden et al. [185] define a module-based classification of code clones for clarifying the relation between software quality and code clones considering the module (file) as a basic unit of software. Clones are classified into the following two types:

1. In-module clone: a code fragment is called "in-module clone" if all the equivalent fragments are located in the same module.

2. Inter-module clone: a code fragment is called "inter-module clone" if one of the equivalent fragments located in the different module.

Based on this classification, they also provide a classifications of modules as follows:

1. Non-clone module: a module containing no clones.

2. Clone-included module: A module containing at least one code clone. It is further classified as follows:

    - Closed module: a module containing in-module clones only.
    - Related module: a module containing inter-module clones only.
    - Composite module: a module containing both in-module and inter-module clones.

Monde et al. claim that the two types of clones and the classification of modules based on these two types may have a great influence on software quality, especially on software reliability and maintainability. Inter-module clones may increase the functional coupling between modules, whereas in-module clones do not affect the strength of coupling. Their study shows that modules having clones (clone-included modules) are more reliable than modules having no code clone (non-clone module) in average. On the other hand, clone-included modules are less maintainable than non-clone modules, and modules with larger clone(s) are less maintainable than modules with smaller code clone(s).

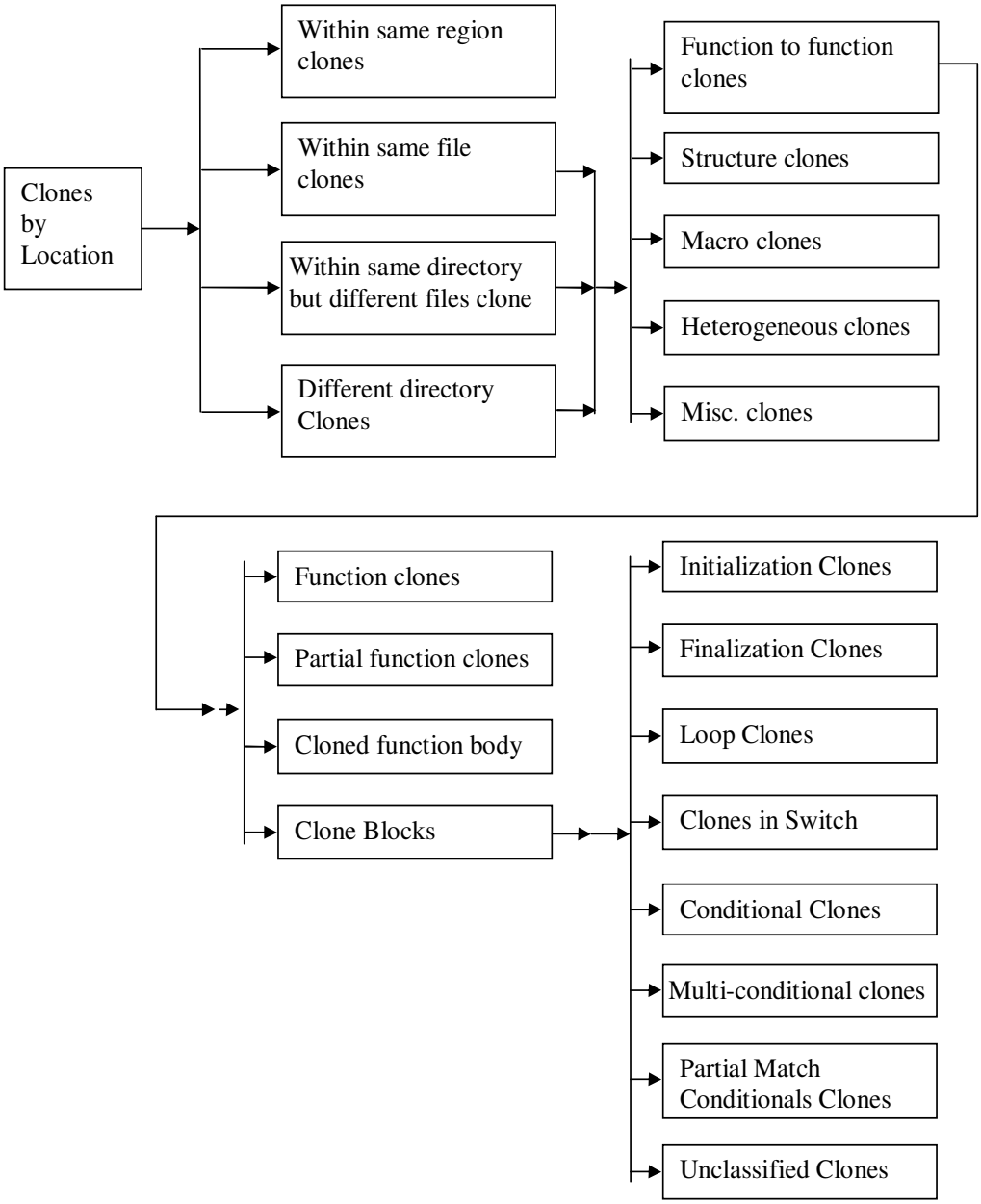

Figure 9: Clone Taxonomy of Kapser and Godfrey (restructured from [126]).

8.3 Taxonomies Based on Refactoring Opportunities

From the refactoring perspective it is useful to know how easy the duplicated code can be extracted from its context (to be put in an unifying function, for example). Taxonomies for these kind of differences are based on the uses of variables and methods defined outside of the copied source fragment. The more such dependencies exist, the harder it will be to perform the refactoring. Sophisticated parsing is required to make this kind of analysis. Examples are:

8.3.1 Balazinska et al. Taxonomy

Balazinska et al. [22] propose context analysis to complete the difference analysis of clones for computer-assisted refactoring.

- Each class is associated with a different risk of clone removal.
- Each class has a different removal strategy (e.g., use of Strategy and Template Method design patterns) [20, 21].

8.3.2 Fanta and Rajlich's Taxonomy

Fanta and Rajlich [78] propose an approach to eliminate clones by reengineering scenarios that are based on automated restructuring tools. As per their study, there are two types of clones in an object-orient system which are subjected to removal. These are as follows:

- Function Clones: Functions clones are further subdivided as follows:
  - Semantically equivalent function clones: Similarity between the functions is based on the semantic similarity of the functions. Two functions with identical functionality may be considered as clones even if they differ in names, have different order and names of arguments, and different names of local variables. This is like the *Type IV* clones that we provide in Section 7.2.
  - Function clones sharing common code: Similarity between the functions is based on the textual similarity of the source code of the functions. This is exactly the same category we provide in Section 7.3.7.
- Class Clones: Two classes can be considered as class clones if they have identical or near identical code. Class clones are further classified in two types:
  - Class clones representing almost identical concepts: The classes share implementation of function and data members between them.
  - Class clones representing separate concepts: The classes share some common code but considering the concept of the classes different from the other one.

In order to remove these clones from the object-oriented system, Fanta and Rajlich provide several scenarios for each of them above.

8.3.3 Golomingi's Taxonomy

Golomingi [145] investigates object-oriented systems (in SMALLTALK) and provides a classification of the clone relationship scenarios based on the class hierarchy relationships of the methods that contained duplicated code fragments. The list of defined scenarios is presented below:

- In the Same Method: Clones are within the same method of a class.
- In the Same Class: Clones are within the same class.
- With a Sibling Class: Clones are in the subclasses of a superclass.
- With the Superclass: Cloning relation between a class and its direct superclass.
- With an Ancestor: Cloning relation between a class and its ancestor class.
- With a First Cousin: Cloning relation between two classes of same hierarchical level with their superclasses being sibling classes with each other.
- In Common Hierarchy: Clones are within the same hierarchy.
- In Unrelated Classes: Cloning relation between two classes with different ancestors.

For each of the above scenarios of cloning relationships, a number of refactorings are proposed. For example, the refactorings "Extract Method" or "Parameterization" can be applied to the clones within the same class. Giesecke proposes similar refactoring scenarios focusing on JAVA [88].

9 Clone Detection Process

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 can be found multiple times. The detector thus essentially has to compare every possible fragment with every other possible fragment. Such comparison is very expensive from a computational point of view and thus, several measures are taken to reduce the domain of comparison before performing the actual comparison. Moreover, after finding the potential cloned fragments, further analysis and/or tool support is required to detect actual clones. In this section, we attempt the to provide an overall summary of the clone detection process. Figure 10 shows the phases that a clone detector may follow in its detection process. In the following, we provide a short description for each of the phases:

1. **Preprocessing** At the beginning of any clone detection approach, the targeted source code is partitioned and the domain of the comparison is determined. There are mainly three objectives of this phase:

   Remove uninteresting parts: All the source code uninteresting to the comparison phase is filtered in this phase. For example, partitioning is applied to embedded code (e.g., SQL embedded in Java code, or Assembler in C code) for separating

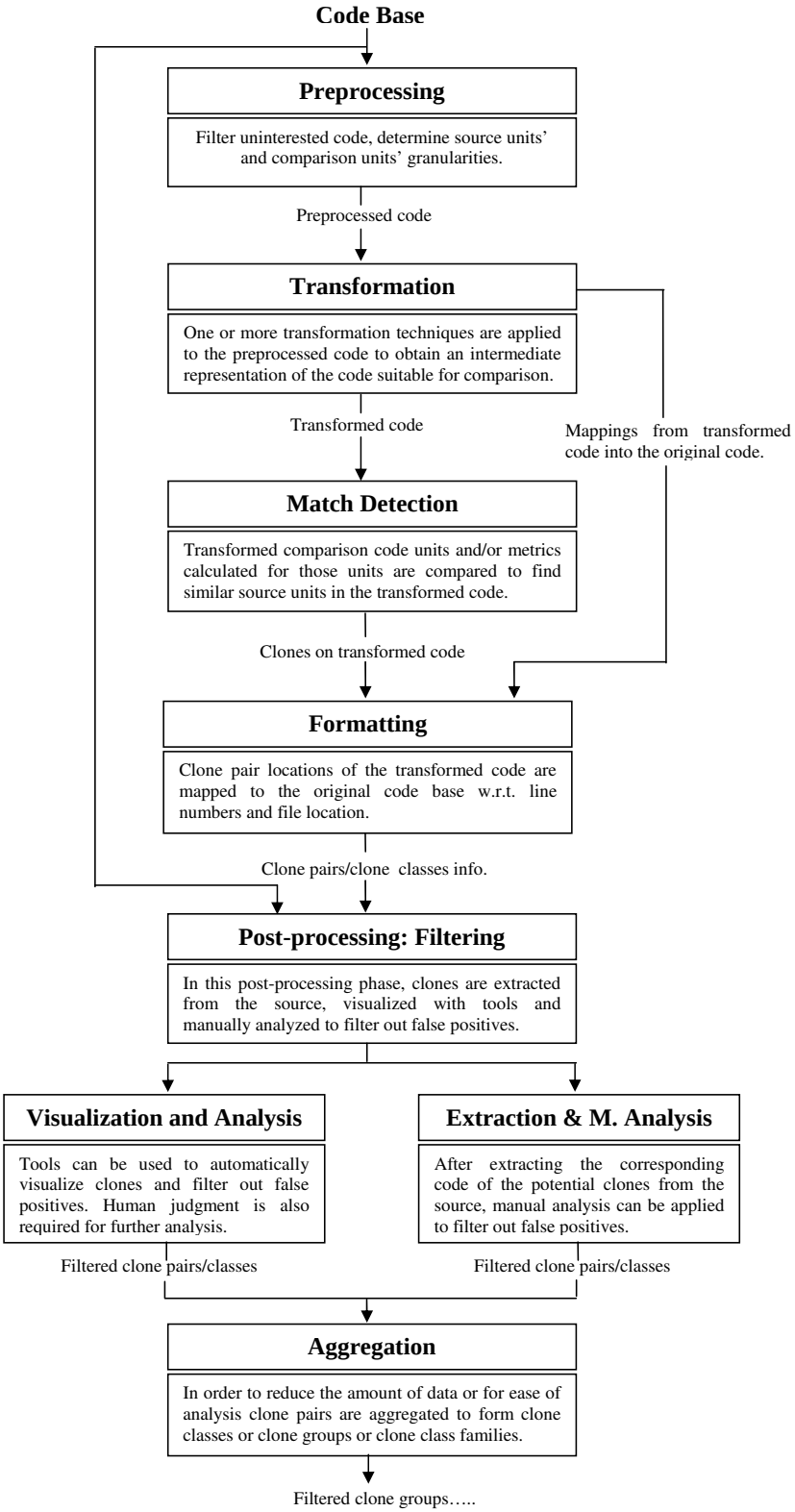

Figure 10: Clone Detection Process

different languages (especially, if the method is not language independent). Similarly, generated code (e.g., LEX- and YACC-generated code) or parts of source code that are likely to produce a lot of false positives (e.g., table initialization) can be removed from the source code before proceeding to the next phase.

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 are involved in direct clone relations to each other. Such units do not maintain any order in the source code and therefore, matching units cannot be aggregated beyond the border of such source units . There can be different levels of granularity for a source unit. For example, files, classes, functions/methods, begin-end blocks, statements or sequences of source lines.

Determine comparison unit/granularity: Source units may need to be further partitioned into smaller units depending on the comparison function of a method. For example, source units can 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. Comparison units are ordered within their corresponding source units. This ordering is important to the comparison function. However, source units may themselves be used as comparison units. For example, in a metric-based method, metrics values can be computed from source units of any granularity and therefore, subdivision of source units is not required in such approaches.

2. **Transformation** The comparison units of the source code are transformed to another intermediate internal representation for ease of comparison or for extracting comparable properties. This transformation can vary from very simple e.g., just removing the whitespace and comments [14] to very complex e.g., generating PDG representation [140, 156] and/or extensive source code transformations [122]. Metrics-based methods usually compute an attribute vector for each comparison unit from such intermediate representations. In the following we list some of the transformation approaches. One or more of the following transformations can be involved in a particular comparison algorithm.

   Pretty printing of source code: Pretty printing is a simple way of reorganizing the source code to a standard form. By applying pretty printing, source code of different layouts can be transformed to a common standard form. Pretty printing is normally used by the text-based clone detection approaches to avoid the false positives that occur due to the different layouts of the similar code segments. Cordy et al. [56] use an *extractor* to generate separate pretty-printed text file for each of the potential clones obtained using an island grammar [70, 186].

   Removal of comments: Most of the approaches (except Marcus & Maletic [177] and Mayrand et al. [178]) ignore/remove comments from the source code before performing the actual comparison. Marcus & Maletic search for similarities of concepts extracted from comments and source code elements. Mayrand et al.,

on the other hand, use metrics to measure the amount of comments and use that metric as a measuring metrics to find clones.

Removal of whitespace: Almost all the approaches (except line-based approaches) disregard whitespace. Line-based approaches remove all whitespace except line breaks. Davey et al. [60] use the indentation pattern of pretty printed source text as one of the features for their attribute vector. Mayrand et al. [178] use layout metrics like *number of non-blank lines*.

Tokenization: In case of token-based approaches, each line of the source is divided into tokens corresponding to a lexical rule of the programming language of interest. Tokens of all lines and/or files are then used to form token sequence(s). All the whitespace (including line breaks and tabs) and comments between tokens are removed from the token sequence. *CCFinder* [122] and *Dup* [14] are the leading tools that use tokenization on the source code.

Parsing: In case of parse tree-based approaches, the entire source code base is parsed to build parse tree or (annotated) abstract syntax tree (AST). In such representation, the source unit and comparison units are represented as subtrees of the parse tree or AST. Comparison algorithm then uses these subtrees to find clones [31, 213, 222]. Metrics-based approaches may also use such representation of code to calculate of the subtrees and find clones based on the metrics values [146, 178].

Generating PDG: Semantics-aware approaches generate program dependence graphs (PDGs) from the source code. Source units or comparison units are the subgraphs of these PDGs. Detection algorithm then looks for isomorphic subgraphs to find clones [140, 156]. Some metrics-based approaches also use these subgraphs to form data and control flow metrics. These metrics can then be used for finding clones [146, 178].

Normalizing identifiers: Most of the approaches apply identifier normalizations before going to the comparison phase. All identifiers of the source are replaced by a single token in such normalizations. However, Baker [14] applies systematic normalizations of the identifiers to find parameterized clones.

Transformation of program elements: In addition to identifier normalizations, several other transformation rules may be applied to the source code elements. In this way, different variants of the same syntactic element may treat as similar to find clones [122, 187].

Calculate metrics values: Metrics-based approaches calculate several metrics from the raw and/or transformed (AST, PDG, etc.) source code and use these metrics values for finding clones [178, 146].

The above transformations just provide an overview of the current transformation techniques used for clone detection. Several other types of transformations with different levels can be applied on the source code before going to the *match detection* phase.

3. **Match Detection** The transformed code is next input to a suitable comparison algorithm where transformed comparison units are compared to each other to find a match. Using the order of the comparison units, adjacent similar units are summed up to form larger units. For fixed granularity clones, all the comparison units that belong to a source unit are aggregated. For free granularity clones, on the other hand, aggregation is continued as long as the aggregated sum is above a given threshold for the number of aggregated comparison units. This makes sure that the aggregation is continued until the largest possible group of comparison units is found.

   The output is a list of matches with respect to the transformed code. These matches are either already in the clone pair candidates or have to aggregate to form clone pair candidates. Each clone pair is normally represented with the location information of the matched fragments in the transformed code. For example, for a token-based approach, a clone pair is represented as a quadruplet (LeftBegin, LeftEnd, RightBegin, RightEnd), where LeftBegin and LeftEnd are the beginning and ending positions (indices in the token sequence) of leading clone, and RightBegin and RightEnd refer to the other cloned fragment that forms clone pair with the first one. Some popular matching algorithms are the suffix-tree [150, 179] algorithm [14, 122], dynamic pattern matching (DPM) [74, 146] and hash-value comparison [31, 178]. Several other algorithms are used in the literature.

4. **Formatting** In this phase, the clone pair list obtained with respect to the transformed code is converted to a clone pair list with respect to the original code base. Normally, each location of the clone pair obtained from the previous phase is converted into line numbers on the original source files. The general format of representing a clone pair can be a nested-tuple, {(FileNameLeft, StartLineLeft, EndLineLeft), (FileNameRight, StartLineRight, EndLineRight)} where FileNameLeft represents the location and name of the file containing the leading fragment with StartLineLeft and and EndLineLeft showing the boundary of the cloned fragment in that file with respect to the line numbers. In a similar way FileNameRight, StartLineRight and EndlineRight represent the other cloned fragment that forms clone pair with the first.

5. **Post-processing** In this phase, false positive clones are filtered out with manual analysis and/or a visualization tool.

   5A. Manual Analysis After extracting the original source code, raw code of the clones of the clone pairs are subject to the manual analysis. In this phase, false positive clones are filtered out.

   5B. Visualization The obtained clone pair list can be used to visualize the clones with a visualization tool. A visualization tool can speed up the process of manual analysis for removing false positives or other associated analysis.

6. **Aggregation** In order to reduce the amount of data or to perform certain analysis, the clone pairs are aggregated to clusters, classes, cliques of clones, or clone groups etc.

The clone detection phases described above are very general and one or more of these may be overlooked in a given detection process.

10 Detection Techniques and Tools

Various clone detection techniques are presented in the literature. While a few of them are commercial, most of them are for research purposes aiming at assisting the development and maintenance processes. Most of the tools also detect different types of clones primarily based on the detection techniques and comparison level of granularity.

In the following subsections, we provide the different clone detection techniques in the form of a taxonomy.

10.1 Taxonomy of Detection Techniques

Each of the clone detection techniques consists of several properties (that we also call dimensions) by which that particular technique can be clarified, for example, how it does, what it does etc. In the following, we provide several such properties.

- Source Transformation/Normalization: Rather than working directly on the raw source code, each of the approaches applies some kind of transformation or normalization or filtering before applying the actual comparison. Some approaches just removes whitespace or comments while other use comprehensive transformation to get an alternative form of code representation suitable for the underlying comparison algorithm and for detecting target clone types for the reengineering purpose. With this property we mean the type of source transformation/normalization is used for a particular method.

- Source Representation: As mentioned above a suitable code representation is obtained by applying different types of transformations/normalizations or filtering to meet the requirements of the target comparison algorithm. With this property we mean the code representation that is used in the comparison phase.

- Comparison Granularity: Different algorithms work on different code representations on different levels of granularity. Some algorithm works on the granularity of one source code line while others work on AST/PDG nodes. With this property we mean the types of clone granularities used for a particular technique in the comparison phase.

- Comparison Algorithm: Choice of the algorithm is also a major concern in detecting clones of different types. As can be seen from the literature, algorithms from different areas are considered in clone detection. For example, some approaches use the sequence matching algorithm which is commonly applied in the biological science for DNA-sequence matching while others apply several data mining/information retrieval algorithms. With this property we mean what kind of comparison algorithm is used for a particular method.

- Computational Complexity: The overall computational complexity of a clone detection technique is a major concern as the technique should be scaled up to detect clones in a large software of millions of lines of code. The complexity of an approach depends on the type of transformations and the comparison algorithm used. With this property we mean the overall computational complexity requires for a particular method.

- Clone Similarity: What kind of clones can be detected by the method of interest. Some detection techniques can find exact match clones while others can detect exact match, parameterized match or near-miss clones. With this property we mean the kind of clone similarity produced by a particular technique.

- Clone Granularity: The granularity of clones can be fixed or free. If there is a predefined syntactic boundary (e.g., function, begin-end brackets etc.) on the returned clones then such clones are called fixed granularity clones. On the other hand, if there is no such syntactic boundary, i.e., clones are similar code fragments without considering any limit or boundary on their structure or size then they are called free granularity clones.

- Language Independency: Language independency is a major concern for a clone detection tool as now a days a software system can be developed with several languages. Also a language independent tool can be applied to any system of interest without any worries. This property will check what kind of language-dependent support is required for the method of interest.

- Output/Groups of Clones: This property implies whether the clone of the systems are returned as clone pairs or clone classes or both. Clone classes are more useful in software maintenance than clone pairs. Therefore, if a technique can provide clone class information directly without going for a post-processing step, then we say that that technique is better than the others that return only clone pairs or that return clone classes after post-processing the clone pair candidates.

- Clone Refactoring: This property implies whether the technique is suitable for clone refactoring or not. There are some techniques that support mechanical clone refactoring while other require human judgements.

- Language Paradigm: This property indicates the language paradigm targeted for the particular method of interest. Some methods can detect clones in both procedure and object-oriented systems while others can detect clones only in procedural or assembly language and so on.

The detection of code clones is mainly a two phase process (practically more, c.f., Section 9) which consists of a transformation and a comparison phase. In the first phase, the source text is transformed into an internal format which allows the use of a more efficient comparison algorithm. During the succeeding comparison phase the actual matches are detected. Due to its central role, it is reasonable to classify detection techniques according to their internal format. Therefore, we use the internal source code representation to classify different detection techniques as follows:

10.1.1 Text-based Techniques

There are several clone detection techniques that are based on pure text-based/string-based methods. In this approach, the target source program is considered as sequence of lines/strings. Two code fragments are compared with each other to find sequences of

same text/strings. Once two or more code fragments are found to be similar in their maximum possible extent (e.g., w.r.t maximum no. of lines) are returned as clone pair or clone class by the detection technique. Because of the purely text-based and/or lexical approach, detected clones do not correspond to structural elements of the language. Little or no transformation/normalization is performed on the source code before starting the actual comparison and most of the cases, the raw source code is directly used in the clone detection process. However, to date the following filtering and/or transformation/normalizations are applied on some approaches:

1. Comments Removal: Ignores all kinds of comments in the source code depending on the language of interest.

2. Whitespace Removal: Removes tabs, and new line(s) and other blanks spaces.

3. Normalization: Some basic normalization can be applied on the source code (c.f., Table 3)

First, some detectors are based on lexical analysis. For instance, Baker's *Dup* [14, 15] uses a sequence of lines as a representation of source code and detects line-by-line clones. Therefore, it uses a lexer and a line-based string matching algorithm on the tokens of the individual lines. *Dup* removes tabs, whitespace and comments; replaces identifiers of functions, variables, and types with a special parameter; concatenates all lines to be analyzed into a single text line; hashes each line for comparison; and extracts a set of pairs of longest matches using a suffix tree algorithm. *Dup* detects parameterized matches and generates reports on the found matches. It can also generate scatter-plots of found matches. This tools does not support exploration and navigation through the duplicated code. Detection accuracy is low e.g., cannot detect code clones written in different coding styles. "{" position of *if-statement* or *while-statement* (see below for details). Cannot detect code clones using different variable names, e.g., we want to identify the same logic code as code clones even if variable names are different. *Dup* compares strings of lexemes rather than strings of characters to combat the problems arise from the changes in comments and whitespace(blanks, tabs, new lines).

Baker [18] [2] also considers the problem of finding exact and near-duplication in software and investigates the notion of parameterized matches (p-matches)[13] that she introduced in *Dup* tool [14]. For two code fragments, some letters of which are designated as parameters (such as identifiers, constants, field names and macro names) can report as clones if there is a renaming of parameters (i.e. following a consistent identifier mapping scheme) that makes the two fragments equal. *Dup* finds all pairs of matching parameterized code fragments. A code fragment matches another if both fragments are contiguous sequences of source lines with some consistent identifier/parameter mapping scheme. For example, the two code fragments of Figure 4 can find similar with *Dup*. A variation of *Dup* is *clones*, provided by Koschke et al. [153] with the difference that it is not based on line but solely on tokens and that it uses nonparameterized suffixes. While the advantages of using nonparameterized suffixes is that *clones* does not depend upon layout (line breaks), the disadvantages is that

[2]Baker actually uses the tokens of each line for a line-by-line comparison and thus, her technique can also be categorized as a token-based technique.

the distinction between exact (*Type I*) and parameterized (*Type II*) cannot be detected and needs a postprocessing step to differentiate them. Another main difference is that *clones* does not currently check whether identifiers in *Type 2* clones are renamed consistently. A further extension of *clones* is *cscope* which finds syntactic clones by splitting cloned token sequences into subsequences with a balanced set of opening and closing scope delimiters in a postprocessing step.

Another pure text-based approach is Johnson's [118] redundancy (exact repetitions of text) finding mechanism using fingerprints on a substring of the source code. In this algorithm, signatures calculated per line are compared in order to identify matched substrings. Karp-Rabin fingerprinting algorithm [130, 131] is used for calculating the fingerprints of all length n substrings of a text. First, a text-to-text transformation is performed on the considered source file for discarding the uninterested characters. Following this the entire text is subdivided to a set of substrings so that every character of the text appears in at least one substring. After that the matching substrings are identified. In that stage, a further transformation is applied on the raw matches to obtain better results. Instead of applying a set of text-to-text transformations, he applies several different transformation scenarios from a combination of basic transformations such as "Remove all whitespace", "Remove all whitespace except line separators", "Remove comments", "Retain only comments" and "Replace each identifier by an identifier marker". For finding near-miss duplication he attempted to find a normalized/transformed text by removing all whitespace characters except line separators and by replacing each maximal sequence of alphanumeric characters with a single letter 'i'. For example, a line "$for(k = 1; k <= n; k++)\{$" is replaced by the line "$i(i = i; i <=; i++\}$" and the line "$\#defineXDEF234$" by "$\#iii$". This kind of transformation produces much more false positives. However, the requirement of keeping at least 50-lines match reduces the huge number of false positives as anticipated.

He also investigated the clones in two versions of GCC using the text-based matching and found that a small part of the source files were clones in each version [120]. He also demonstrated that clone detection could be used to find structural changes at the file level between different releases: if two files from two different versions are actually clones, then the file in the new version may be just a moved or renamed version of the one in old version. As with the line-based techniques, these approaches are sensitive to any minor modifications made in copy-pasted code.

Another line-based technique for detecting near-miss clones in HTML web pages is proposed by Cordy et al. [56]. First, an island grammar is used to identify syntactic constructs in code. These are then extracted and used as smallest comparison unit. The code is pretty-printed to isolate potential differences between clones to as few lines as possible. Extracted code fragments are then compared to each other line-by-line using UNIX's diff utility. This approach is purely lexical and no normalization is applied on the source. An earlier study by them use the similar approach for detection and resolution of exact clones [205].

Another text-based clone detection approach is presented by Ducasse et al. [74, 73]. Their approach does not rely on parsing and therefore, can easily be adapted to any language. This method reads source files, makes sequences of lines, removes whitespace and comments in lines, and detects match by a string-based Dynamic Pattern Matching (DPM) algorithm. The output is the line numbers of clone pairs, possibly with gap (deleted) lines

in them. The computational complexity is $O(n^2)$ for the input size n, which is practically too expensive. The tool uses an optimization technique by a hash function for string, that reduces the computation complexity by a factor B, which is a constant determined by the number of characters in a line. Again, meaningful clone resolution is difficult to achieve in a language-independent manner because it is hard to guarantee that detected clones represent a cohesive unit in the language being analyzed. The process of resolution itself also depends on the language in question.

Marcus [177] applied latent semantic indexing [75] for finding similar code segments (high level concept clones e.g., ADTs) in the source code. This information retrieval approach does not compare the whole source code text; rather limits its comparison domain within the comments and identifier matching. Therefore, it is clear that their method returns code fragments as clones where there is a high similarity between the names (identifiers and comments) of those fragments. One of the main drawbacks of this approach is that it cannot detect two functions with similar structure and functionality if comments do not exist and the identifier names are completely different. Nevertheless, their work shows that semantically equivalence clones can be detected with relatively low cost and can even be automated to a large degree by combining their approach with other existing detection approaches.

Text-based techniques can be further categorized by the granularity of comparison. For example, the approach of Ducasse et al. [74] is line-based. There are several problems that can arise in a line-by-line detection technique. Some of those are as follows:

1. Line Break: Code portions with line break relocation are not detected as clones or detected as shorter clones. In Figure 11, the line break in *Fragment 1* at line 343 is relocated up to the end of the previous line in the later case *Fragment 2, line 785* and the usual line-by-line method may not recognize the two code fragments as clones.

2. Identifier changes: Changes of identifier names may not be handled in line-by-line method. In Figure 11, the identifier, $dwFrameGroupLength$ at line 341 in *Fragment 1* changes to $frameGroupLength$ at line 784 in *Fragment 2*. This kind of changes may not be detected by ordinary tools.

3. Parenthesis removal/adding for a single statement: For example, a single statement can be with or without surrounded by *begin-end* brackets (e.g., "{" and "}") just after *if*, *else* or *for* statements. In line-by-line technique, the presence of "{" and "}" pair in one code segment but not in the other creates a great problem while comparing the two fragments and may detect as distinct fragments even if they might be exact copy clones. In *Fragment 1* of Figure 11, we see that the assignment statement $dwFrameGroupLength* = 2$; at line 347 is surrounded by a "{ ...}" pair which is omitted for the same kind of statement in *Fragment 2* at line 787. Therefore, it is obvious that different kinds of coding style can create problems in line-by-line methods.

4. Transformation: The source code is not suitable for transformation in line-based approaches. However, some kind of normalization may be applied to improve recall with the cost of precision.

```
Fragment 1:
341:    dwFrameGroupLength = 1;
342:    for (dwCnt = 2; dwCnt <= 64; dwCnt *=2)
343:    {
344:            if (( ( ulOutRate / dwCnt) * dwCnt) !=
345:                      ulOutRate)
346:             {
347:                        dwFrameGroupLength *=2;
348:             }
349:    }
-----------------------------------------------------------------
Fragment 2:
784:    frameGroupLength = 1;
785:    for (Cnt = 2; Cnt <= 64; Cnt *=2){
786:            if (( ( Rate / Cnt) * Cnt) != rate)
787:                       frameGroupLength *=2;
788:    }
```

Figure 11: Clone pair between FreeBSD and Linux (from [122])

Although text-based approaches, generally, do not apply normalization or transformation of source code text, the latest text-based approach by Ducasse et al. [72] has used several transformations on the "raw" source code.

They considered several normalizations (c.f., Table 3) on the language elements in their study and investigated how much recall can be achieved for which kind of normalization considering the lower cost of precision.

Table 3: Normalization operations on source code elements (from [72])

Operation	**Language element**	**Example**	**Replacement**
1	Literal string	"Abort"	"..."
2	Literal character	'y'	'.'
3	Literal integer	42	1
4	Literal decimal	0.314159	1.0
5	Identifier	counter	p
6	Basic numerical type	int, short, long, double	num
7	Function name	main	foo()

For instance, in Figure 12, the normalized fragment (bottom part of the figure) is obtained by applying the rules from Table 3 on the upper part code fragment (all identifiers are normalized except function names).

Ducasse et al. [72] also considered to eliminate noise not only on the comments and whitespace levels but also removing uninteresting language elements. For example, in the following code segments, the code segment on the right can be obtained by removing all kinds of comments, whitespace, uninteresting language elements and language delimiters from the left segment. This was investigated to achieve better recall sacrificing precision.

```
Original Fragment:
1.  def manage\_first (self, selected=[]):
2.      options=self.data()
3.      if not selected:
4.          message=‘‘No views to be made first.‘‘
5.      elif len(selected) == len(options):
6.          message=‘‘No making all views first.‘‘
7.      else:
8.          options=self.data()

Normalized Fragment:
1.  def manage\_first (p, p=[]):
2.      p=p.data()
3.      if not p:
4.          p=‘‘...‘‘
5.      elif len(p) == len(p):
6.          p=‘‘...‘‘
7.      else:
8.          p=p.data()
```

Figure 12: Python source code before and after normalization (from [72])

```
Before Filtering:                After Filtering:
#include <stdio.h>
static int stat = 0;
                                  staticintstat=0;
int main (argc, argv)             intmain(argc,argv)
   int argc;                      intargc;
   char **argv;                   char**argv;
{                                 ++argv,--argv;
  /*skip program name */          if(argc>0)
  ++argv, --argv;
  if (argc > 0) {
```

Finally, in Table 4, we provide a brief summary of the text-based detection techniques. In some cases what (?) symbols are used to represent that we were unsure about that particular entry.

10.1.2 Token-based Techniques

In the token-based detection approach, the entire source system is lexed/parsed/transformed to a sequence of tokens. This sequence is then scanned for finding duplicated subsequences of tokens and finally, the original code portions representing the duplicated subsequences returned as clones. Compared to text-based approaches, a token-based approach is usually

Table 4: Summary of the String-based Detection Techniques

Properties	**Baker** [14, 18]	**Johnson** [118, 120]	**Ducasse et al.** [74, 72]	**Marcus & Maletic** [177]
Normalization or Transformations	Removes whitespace and comments	Removes comments and whitespace	Removes comments whitespace & apply transformations	Removes Comment delimiters and Syntactical tokens
Code Representation	Parameterized token string	Fingerprint of substrings	Effective sequence of lines	Text (like natural language)
Comparison Technique	Suffix-tree based token matching	Karp-Rabin Fingerprinting based string matching	Dynamic Pattern Matching (DPM)	LSI/Graph theoretic approach
Complexity	$O(n+m)$, n=input lines, m=number of matches found.	Not Available	$O(n^2)$, n=input lines, then reduces by hashing to B buckets	Not Available
Comparison Granularity	Line (tokens of a line)	substrings	Line	Words, sentences, paragraphs and short essays?
Clone Granularity	Free, threshold-based (minimum of 15 lines), longest match	Free, threshold based (50 lines)	Free, threshold-Based, longest matches	Fixed, functions, files or code segments
Clone Similarity	Exact and parameterized matches	Exact repetitions of strings (near miss also addressed)	Exact matches	High level concept clones e.g., ADT, exact and near miss
Language Independence	At most needs a lexer	No lexer/parser needed	Needs at most a lexer	Only considers source code text (comments and identifiers)
Output Type	Text: Clone Pair & Clone Class	Text: Clone Pair	Text: Clone Pair	?
Clone Refactoring	Needs human hands	Needs human hands	Needs human hands	Needs human hands

more robust against code changes such as formatting and spacing.

One of the leading state of the art token-based techniques is *CCFinder* [122] of Kamiya et al. First, each line of source files is divided into tokens by a lexer and the tokens of all source files are then concatenated into a single token sequence. The token sequence is then transformed, i.e., tokens are added, removed, or changed based on the transformation rules of the language of interest aiming at regularization of identifiers and identification of structures. After that each identifier related to types, variables, and constants is replaced with a special token. This identifier replacement makes code fragments with different variable names clone pairs. A suffix-tree based sub-string matching algorithm is then used to find the similar sub-sequences on the transformed token sequence where the similar sub-sequence pairs are returned as clone pairs/clone classes. Once the clone pair/clone class information is obtained with respect to the token-sequence(s), a mapping is required for obtaining the clone pair/clone class information with respect to the original source code.

Baker's *Dup* [14, 17, 18] is also token-based in the sense that she also uses a lexer to tokenize the source code and then the tokens of each line is compared based on a suffix-tree based algorithm. She did not apply the transformation rules on the token sequence as of *CCFinder*. However, she introduced the notion of parameterized matching by a consistent renaming of the identifiers. RTF [24] uses a suffix array instead of a suffix tree for efficient memory handling and provides flexible tokenization, allowing the user to tailor token strings

for better clone detection. The user can suppress insignificant token classes (e.g., access modifiers of Java) that may cause noise in detection, and there is an option for equating different token classes, for example to assign the same ID to different types *int, short, long, float, double* depending on requirements.

Another state of the art token-based clone detection technique is *CP-Miner* [168, 169] where a frequent subsequence mining technique [3] is used for identifying a similar sequence of tokenized statements. Due to sequential analysis in *CCFinder* and *Dup*, they are generally fragile to statement reordering and code insertion. A reordered or inserted statement can break a token sequence which may otherwise be regarded as duplicate to another sequence. These limitations are overcome in *CP-Miner* by using a frequent subsequence mining technique where a frequent subsequence can be interleaved in its supporting sequences. An extended version of CloSpan [221] is used to support gap constraints in frequent subsequences which allows *CP-Miner* to tolerate one to two statement insertions, deletions, or modifications in copy-pasted code while ignoring arbitrarily long different copy-pasted segment that is unlikely to be copy-pasted.

Token-based techniques are also used in the area of plagiarism detection. Both *Winnowing* [203] and *JPlag* [192] are well know plagiarism detection tools and based on token based techniques. The fragility of code deletions, insertions or modifications can also be partially remedied through fingerprinting of Winnowing. However, it does not fundamentally save token-based algorithms. Another plagiarism detection tool, SIM [90] also compares token sequences using a dynamic programming string alignment technique.

In Table 5, we provide a summary of three (including Baker's) token-based techniques with respect to several properties. In some cases what(?)symbols are used to represent that we were unsure about that particular entry.

10.1.3 Tree-based Techniques

In the tree-based approach a program is pared to a parse tree or an abstract syntax tree (AST) with a parser of the language of interest. Similar subtrees are then searched in the tree with some tree matching techniques and the corresponding source code of the similar subtrees are returned as clones pairs or clone classes. The parse tree or AST contains the complete information about the source code. Although the variable names and literal values of the source are discarded in the tree representation, more sophisticated methods for the detection of clones still can be applied.

One of the pioneers AST-based clone techniques is that of Baxter et al.'s *CloneDR*. [31]. A compiler generator is used to generate an annotated parse tree (AST) and compares its subtrees by characterization metrics based on a hash function through tree matching [4]. Source code of similar subtrees are then returned as clones. The hash function enables one to do parameterized matching, to detect gapped clones and to identify clones of code portions in which some statements are reordered. Bauhaus's [193] clone detection tool named *ccdiml* [35] is a variant of *CloneDR* with some differences like the avoidance of the similarity metric, the handling of sequences and the hashing. *CloneDR* can also work concurrently and check for consistent renaming while *ccdiml* does not do this. In Bauhaus (and hence in *ccdiml*), the ASTs are represented in IML(Intermediate Language) [152] rather than directly using the ASTs in comparison phase as of *CloneDR*.

Table 5: Summary of the Token-based Detection Techniques

Properties	**Baker** [18]	**Kamiya et al.** [122]	**Li et al.** [168] [169]
Normalization or Transformations	Removes whitespace and comments	Removes comments and whitespace + several Transformation and Parameter replacement	Maps the source to collection of sequences with similar statements/identifiers to the same value Token
Code Representation	Parameterized token string	Sequence of normalized, transformed & parameterized tokens	Collection of short sequences (say for each basic block) of numbers)
Comparison Technique	Suffix-tree based Token matching	Suffix-tree based Token matching	Frequent subsequence mining technique
Complexity	$O(n+m)$, n=input lines, m=number of matches found	$O(n)$, n=total length of source file	$O(n^2)$, n=Lines of code
Comparison Granularity	Token sequence of a line	Token	Sequence of Tokens of Basic block
Clone Granularity	Free, threshold-based (minimum of 15 lines)	Free, threshold-based of tokens (30 tokens)	Free, threshold-based (Basic blocks and functions)
Clone Similarity	Exact and parameterized matches	Exact or near miss possibly with gaps	Exact and near miss with gaps
Language Independence	At most needs a lexer	Needs lexer + transformation rules for the language	Needs a full parser
Output Type	Text: Clone Pair & Clone Class	Clone Pair & Clone Class	Clone Pair
Clone Refactoring	Needs human hands	Needs human hands	Needs human hands

Yang [222] has proposed earlier a similar approach for finding the syntactic differences between two versions of the same programs by generating a variant of parse tree for both the versions and then applying dynamic programming approach [108] in searching similar subtrees. Wahler et al. [213] find exact and parameterized clones in a more abstract level than AST where the AST of a program is converted to an XML represented [82] and then a data mining frequent itemset technique [101] is applied in the XML representation of the AST for finding clones. A further abstraction (known as Structural abstraction) of a program's ASTs is proposed by Evans and Fraser [77] for finding exact and near miss clones with gaps. While ASTs are built from lexical abstraction of a program by parameterizing only AST leaves (abstracting identifiers and literal values), structural abstraction is obtained by further parameterizing the arbitrary subtrees of ASTs.

For example, consider the clone

```
a[?] = x
```

occurs twice:

```
a[i] = x ;
```

```
a[i+1] = x ;
```

The argument to the first occurrence is lexical because it includes only a leaf and, perhaps, a unary node that identifies the type of the leaf. The argument to the second occurrence is, however, structural because it includes a binary AST node.

Thus, it is clear that structural abstraction is more general than ASTs and hence, can find gapped clones by abstracting of an AST with the cost of much larger search space. However, there is still no special treatment for identifiers and literal values for detecting clones in ASTs. AST-based approach disregards the information about identifiers (in order to make codes differing on variables names appear the same on ASTs), ignores data flows and therefore, fragile to statement reordering. Moreover, this approach is also fragile to control replacement. PDG-based techniques can overcome such limitations which are discussed in following subsection.

In Table 6, we provide a summary of four tree-based techniques with respect to several properties. In some cases what(?) symbols are used to represent that we were unsure about that particular entry.

Table 6: Summary of the Tree-based Detection Techniques

Properties	**Baxter et al.** [31]	**Yang** [222]	**Wahler et al.** [213]	**Evans** [77]
Normalization or Transformations	Parsed to AST	To a variant of parse tree	Parsed to AST and then AST in XML	Parsed to AST and then to AST in XML
Code Representation	AST	Parse tree	AST in XML	AST in XML
Comparison Technique	Tree matching tech	Tree matching (sequence of token matching) with Dynamic programming scheme [108]	Frequent Itemset	Graph theoretic Approach
Complexity	O(N), N=AST Nodes, after optimizations to B buckets	$O(S_1 S_2)$, S_1=no. of nodes of first tree, S_2=no. of nodes of second tree	$O(k.n^2)$, n=statements containing clones, k=maximal size clones	Not Available
Comparison Granularity	AST Node	Token (Tree node)	one line?	AST Node
Clone Granularity	free, tree similarity based threshold	Free, program/segment	Free, usually 5 statements, threshold-based	Free, threshold-based
Clone Similarity	Exact and near-miss	Exact and near-miss?	Exact and parameterized	Exact and near miss with gap
Language Independence	needs a lexir at least (parsing)	Needs a parser and pretty-printer	Needs a parser?	Needs parser
Output Type	Clone pair	Just displays with pretty-printing	?	HTML document with clone information
Clone Refactoring	Can help mechanical refactoring	Not helpful for refactoring	Semi-automatic (may be)	Needs human hands

10.1.4 PDG-based Techniques

Program Dependency Graph (PDG)-based approaches [141, 156, 165] go one step further in obtaining a source code representation of high abstraction than other approaches by considering the semantic information of the source. PDG [80] contains the control flow and data flow information of a program and hence carries semantic information. Once a set of PDGs are obtained from a subject program, isomorphic subgraph matching algorithm is applied for finding similar subgraphs which are returned as clones.

One of the leading PDG-based clone detection approach is that of Komondoor and Horwitz's *PDG-DUP* [141, 144] which finds isomorphic PDG subgraphs using program slicing [218]. They also propose an approach to group identified clones together while preserving the semantics of the original code [142, 143] for automatic procedure extraction to support software refactoring. Another slicing based clone analysis experiment has been conducted by Gallagher and Lucas [86] where they have attempted to answer the argument "*Are Decomposition Slices Clones?*" by computing program slices on all variables on a system. However, they were unsure about the analysis outcome and just provided the pros and cons in support of the argument.

Krinke uses an iterative approach (k-length patch matching) for detecting maximal similar subgraphs [156]. There is also a PDG-based recent tool, GPLAG [165] for plagiarism detection. Chen at al. [49] also propose a PDG-base technique for code compaction taking into account syntactic structure and data flow. This method has many advantages in embedded systems.

PDG-based approaches are robust to reordered statements, insertion and deletion of code, intertwined code, and non-contiguous code, but they are not scalable to large size programs.

In Table 7, we provide a summary of three PDG-based approaches with respect the several properties. In some cases what(?) symbols are used to represent that we were unsure about that particular entry.

Table 7: Summary of the PDG-based Detection Techniques

Properties	**Komondoor et al.** [141]	**Krinke** [156]	**Liu et al.** [165]
Normalization or Transformations	Use CodeSurfer to get PDG	To PDGs	CodeSurfer to PDG
Code Representation	set of PDGs of procedures	Fine Grained PDGs	set of PDGs without control dependencies
Comparison Technique	Isomorphic PDG subgraph matching using backward slicing	k-length patch matching to find similar subgraphs	Isomorphic subgraph matching
Complexity	Not Available	Non-polynomial	NP-Complete but several considerations to improve complexity (threshold-based filtering)
Comparison Granularity	PGD Node	PGD subgraphs	PDG Node
Clone Granularity	Free, slicing-based	Free, threshold-based, length limited similar path	Fixed, procedures and programs (normally for plagiarism)
Clone Similarity	non-contiguous, reordered, intertwined	Exact and Semantic	Exact and Near-miss
Language Independence	Needs tool to make PDG	Needs a PDG generator	Needs a tool to make PDG
Output Type	Clone pair and Clone Class	Clone Class?	Plagiarized pair of procedures/programs
Clone Refactoring	Mechanical Refactoring	Help semi-automatic refactoring	Not feasible, intention is to see two different versions of programs

10.1.5 Metrics-based Techniques

Metrics-based approaches gather different metrics for code fragments and compare these metrics vectors instead of comparing code directly. There are several clone detection techniques that use various software metrics for detecting similar code. First, a set of software metrics called fingerprinting functions are calculated for one or more syntactic units such as a class, a function, or a method or even statement and then the metrics values are compared to find clones over these syntactic units. In most cases, the source code is parsed to its AST/PDG representation for calculating such metrics.

Mayrand et al. [178] calculate several metrics (e.g., number of lines of source, number of function calls contained, number of CFG edges, etc.) for each function unit of a program. Units with the similar metrics values are identified as code clones. Partly similar units are not detected. It uses a representation of the source code named *Intermediate Representation Language (IRL)* to characterize each function in the source code. Metrics are calculated from names, layout, expression and (simple) control flow of functions. A clone is defined only as pair of whole function bodies that have similar metrics values. This approach does not detect copy-paste at other granularity such as segment-based copy-paste, which occurs more frequently than function-based copy-paste. Very similar kinds of method-level metrics such as number of calls from a method, number of statements, McCabe's cyclomatic complexity, number of use-definition of non-local variables and number of local variables are used for finding similar methods by Patenaude et al. [190]. They define these metrics for Java language and extend the IBM Datrix tool to support Java in software quality assessment.

Kontogiannis et al. [147] build an abstract pattern matching tool to identify probable matches using Markov models. This approach does not find copy-pasted code. Instead, it only measures similarity between two programs. After a while, Kontogiannis [146] proposes two ways of detecting clones. One approach is the direct comparison of the metrics values that classify a code fragment in the granularity of $begin - end$ blocks with the assumption that two code fragments are similar if their corresponding metrics values are proximate. The other uses a dynamic programming technique for comparing $begin - end$ block at a statement-by-statement basis. Modified version of five well known metrics that capture the data and control flow program properties are used in the first approach. The metrics are, 1. the number of functions called (fanout), 2. The ratio of input/output variables to the fanout, 3. McCabe cyclomatic complexity, 4. Modified Albrecht's function point metric and 5. Modified Henry-Kafura's information flow quality metric. After constructing the AST from the source code, metrics values are calculated and stored as annotations in the corresponding nodes of the AST. From the annotated AST a reference table is created that contains source code entities sorted by their corresponding metric values which is then used for selecting the associated source code entities. On the other hand in the dynamic programming (DP) approach, the distance between the pair of $begin - end$ blocks is defined as the least costly sequence of insert, delete and edit steps required to make one block identical line-by-line to the other. The hypothesis is that pairs with a small distance are likely to be clones caused by cut and paste activities. Metrics based approaches have been also proposed by Buss et al. [45] and Dagenais et al. [58].

Metric-based approach has been also applied for finding duplicated web pages or finding

clones in web documents. Di Ducca et al. [66] propose an approach for identifying similar static HTML pages by computing the distance between items in web pages and evaluating their degree of similarity. A string representation is obtained for each of the HTML/ASP pages of a Web Application (WA) by replacing each HTML/ASP control elements with a distinct symbol taken from either of the two distinct set of alphabets, one for HTML tags and the other for ASP objects. For each of the strings the Levenshtein distances are calculated and are used to compare the associated pages from which the strings were extracted [67].

A semiautomated method for detecting cloned script functions is proposed by Lanubile, Calefato and colleagues [161, 46] where the potential function clones are detected with an automated approach first and then a visual inspection is employed in the selected script functions. They first apply their tool *eMetrics* to retrieve the potential function clones and then use the reports from the tool to visually inspect the code of the selected script functions, classify suspect clones, and group discovered function clones according to refactoring opportunities.

Davey et al. [60] detects exact, parameterized and near-miss clones by first computing certain features of code blocks and then using neural networks to find similar blocks based on their features. In Table 8, a summary of four metrics-based techniques are presented with respect to several properties. In some cases what(?) symbols are used to represent that we were unsure about that particular entry.

10.1.6 Hybrid Approaches

There are several other detection approaches that use a hybrid approach (e.g., hybrid code representation and/or techniques) in detecting clones. However, these approaches can also be classified on the previous categories.

In the approach by Koschke et al. [153], the AST nodes are serialized in preorder traversal, a suffix tree is created for these serialized AST nodes, and the resulting maximally long AST node sequences are then cut according to their syntactic region so that only syntactically closed sequences remain. In stead of comparing the AST nodes, their approach compares the tokens of the AST-nodes using a suffix tree-based algorithm and therefore, this approach can find clones in linear time and space, a significant improvement to usual AST-based approaches.

A function-level clone detection technique is proposed for the Microsoft's new Phoenix framework using AST and suffix trees [206]. AST nodes are used to generate a suffix tree, which allows analysis on the nodes to be performed in linear time and space as of Koschke et al. [153]. This approach can find exact matching function clones. Parameterized clones with identifier renaming(not type changes) can also be detected with this approach. A kind of similar approach is proposed by Greenan [98] for finding method level clones on transformed AST using sequence matching algorithm.

A novel approach of detecting similar trees is presented by Jiang et al. [116] where certain characteristic vectors are computed to approximate the structural information within ASTs in the Euclidean space. A Locality Sensitive Hashing (LSH) [59] is then used to cluster similar vectors w.r.t. Euclidean distance metric and thus, code clones.

Balazinska et al. [22] propose a hybrid approach of characterization metrics and DPM

Table 8: Summary of Metrics-based Clone Detection Techniques

Properties	**Kontogiannis** [146]	**Mayrand** [178]	**Di Lucca** [67]	**Lanubile** [161]
Normalization or Transformations	To Feature Vectors	To AST and then Intermediate Representation Language (IRL)	HTML files are parsed to extract the HTML tags and composite tags are substituted with their equivalent one	Use eMetrics to select potential function clones based on their similarity of function names
Code Representation	Feature vectors targeting structural similarity (metrics values with data and control flow properties)	Metrics obtained with Datrix [] tool	HTML/ASP strings	Several metrics (LOC, ELOC, CLOC) obtained with eMetrics of the selected functions
Comparison Technique	Numerical pairwise comparison of selected metrics values evaluating the Euclidean distance and dynamic programming techniques for statement level comparison of feature vectors	21 function metrics with 4 points of comparison	Levenshtein Distance of strings	Only visual inspection of the metrics values
Complexity	Naive approach $O(n^2)$ and DP-model O(mXn) m=model size, n=input size	Polynomial	$O(n^2)$, n=length of the longer string	Not available
Comparison Granularity	Metrics values of a begin-end block at the statement level	Metrics of individual functions	String representation of a full page	Only visual inspection of each function
Clone Granularity	Fixed, begin-end blocks	Fixed, function	whole static page	Fixed, function level
Clone Similarity	Partial and near-miss/allow a measure of dissimilarity between code fragments	exact and near-miss based on the delta values of the metrics (Level 1 to level 8)	Same set of HTML tags/ASP objects but data may be different)	Identical, nearly Identical, similar and distinct (needs human inspection)
Language Independence	Parser and associated tool	Needs a tool (Datrix)	Need to parse the webpage	Needs a tool (eMetrics) to generate the potential clones associated metrics
Output Type	Clone Pair	Clone Class and Clone Pair	Clone Class	set of all homonymy scripted functions with metrics values
Clone Refactoring	Need human judgment	May help for mechanical refactoring of exact copy functions	Not approached	Human hand is a must

(Dynamic Pattern Matching). This paper only discusses detection of whole methods, although the approach would be applied to detect partial code portions also. Characteristic metrics valued are computed for each of the method bodies and compared to find cluster of similar methods following Patenaude et al.'s [190] metrics-based approach. The token sequences for each pair of similar methods are then compared by a Dynamic Pattern Matching Algorithm of Kontogiannis et al. [146] in order to identify cloned methods. Finally, the

found cloned methods are classified into 18 categories (cf. Section 8.1.2). In Table 9 we provide several hybrid approaches to clone detection. In some cases what(?) symbols are used to represent that we were unsure about that particular entry.

Table 9: Summary of the Hybrid Clone Detection Techniques

Properties	**Koschke et al.** [153]	**Jiang et al.** [116]	**Tairas et al.** [206]	**Greenan** [98]	**Balazinska et al.** [22]
Normalization or Transformations	Parsed to AST and then Serialized AST	Parsed to AST and then Characteristics vector metrics of AST	To AST and then Sequence of AST Nodes	Parsed to AST	Annotated AST tokens
Code Representation	Serialized AST	Vectors of fixed dimension	Sequence of annotated AST Nodes	Sequence of annotated AST nodes as strings	Token sequence and annotated AST and then metrics values of methods obtained with the extension of Datrix tool [190]
Comparison Technique	String-based Suffix tree	Tree-matching	Suffix tree matching	Exact-matching, Longest common subsequence and Smith-Waterman	Dynamic Pattern matching for matching token sequence of similar methods
Complexity	O(N), N=input nodes	O($dn^{p+1}logn$)	$O(m^2)$, m=length of the sequence	$O(n^2)$, n=no. of methods	O(nXm), n=no. of tokens of first methods, m=no. of tokens for 2nd method
Comparison Granularity	Serialized AST Node?	Subtrees with numerical vectors in the euclidean space	AST Node of the sequence	AST Node of the Sequence	Metrics values and token sequences of methods
Clone Granularity	Free, threshold based	Free, threshold-based of token of vector count	Fixed, function level	Fixed, method-level	Fixed, method level
Clone Similarity	Exact and parameterized clones(*Type II*)	Exact and Near-miss?	Exact and parameterized	Exat and near-miss	18 different category (c.f., Section 8.1.2)
Language Independence	Needs a parser	Context-free grammer	Needs Phoenix framework to generate AST	Needs a parser	Needs parser, classification algorithm and metrics generator
Output Type	Clone pair	Clone pair (post-processing required)	Filename and function names: Clone Class	Clone Pair	Clone Pair/Clone Class?
Clone Refactoring	Can help mechanical refactoring	Semi-automatic (may be)	Suitable for refactoring	May help in mechanical refatoring	Provide several refactoring scenarios

There is clone detection (and plagiarism) technique for Lisp-like languages too. Leitao [164] provides a hybrid approach that combines syntactic and semantic techniques through a combination of specialized comparison functions, each one exploring different features that compare various aspects (similar call subgraphs, commutative operators, user-defined equivalences, transformations into canonical syntactic forms). Each comparison function

yields an evidence that is summarized in an evidence-factor model yielding a clone likelihood. This approach has the advantage that each of its parts is replaceable. New languages can be added by the specification of their syntaxes. It is also possible to explore other language features by including new specialized comparison functions. Evidences and its combination can also be replaceable with other models. While majority of the clone detection techniques are discussed, there are several other clone and plagiarism detection techniques available [163, 174, 220, 219, 37, 195, 111].

10.2 Overall Taxonomy of the Detection Approaches

In this subsection (Table 10), we provide a reading summary of the clone detection techniques in the form of a taxonomy with respect to the various properties discussed previously. A paper is cited multiple times so as to represent the corresponding properties for that paper. In some cases, we use the asterisk symbol (*) to mean that there may have been some limitations for that particular technique to satisfy the corresponding criteria/sub-criteria. The table may have also missed some citations and will be gathered all (if missing any) on a later version.

Table 10: A Taxonomy of Clone Detection Techniques: A Summary

Properties	Sub-properties	Citations
Language Paradigm	Only Procedural	[8] [7] [158] [178] [120, 174, 177] [24] [222] [141, 156, 165]
	Only Object-oriented	[21] [22] [81] [190] [121] [22, 21, 187]
	Both Procedural and OO	[72, 74] [18, 14] [31] [163] [122]
	Web Applications	[37] [67, 66, 65, 64] [195] [56] [205] [46] [194] [161] [134] [175]
	Lisp-like languages	[164]
	Bytecode	[11]
	Assembly Code	[55] [63] [61] [85]
	Extreme Programming	[188]
Clone Relation	Directly Clone Pair	[74] [22] K[122] [31] [178] [158] [141] [156]
	Directly Clone Class	[120]* [177]* [24] [116] [165]*
	CC in post-processing	[74] [122] [31] [22] [141]
Level of Similarity	Textual	[74] [120, 118] [117] [220] [174] [163] [52]
	Lexical	[122] [173] [14] [18] [24] [26] [169]
	Syntactical	[31] [35] [222] [213] [77] [153] [206] [178] [146] [148] [22] [190] [60] [46] [67]
	Semantical	[140, 141] [156] [165] [164]
	Hybrid	[164]
Clone Granularity	Free	[74] [220] [120] [163] [52] [31] [35] [213] [77] [153] [122] [24] [18] [169] [167] [56]*
	Fixed	[178] [146] [148] [22] [190] [60] [46] [67] [206] [222] [174] [177] [173] [164]
Clone Similarity	Exact Match	All the approaches can detect exact match clones
	Parameterized Match	[14, 18]
	Near Miss	[72] [177] [122] [31] [116] [168][169][141] [56]
	High Level Clone	[177] (ADT)
	Design Level Structural Clones	[26, 25]
Comparison Granularity	Line	[74, 220, 52]
	P-line	[18, 14]
	Substring	[120] (multi-line) [163] (multi-word)
	Identifiers and Comments	[177]
	Tokens	[122, 24] [18]* [153, 206] (tokens of tree nodes)
	Statements	[169] [213]

Continued on Next Page...

Table 10 – Continued from previous page

Properties	Sub-properties	Citations
	Subtree	[31, 35, 222, 77]
	Subgraph	[141, 156, 165]
	Begin-End Blocks	[146]
	Methods	[206] [148, 178, 21, 22] [190]*
	Classes	[202]
	Files	[174] [222]
Language Dependency	Lightweight Parsing	[74, 120, 163, 177, 174] [116] (only grammar)
	Lexer	[122, 18, 24, 56]
	Lexer and Parser	[31, 222, 98] [178, 22] [146]
	PDG/CG Maker	[141, 156, 165] [164] (call graph)
	Transformation Rules	[122] [121]
Text-processing	Pre-processing	[56] [90]*
	Post-processing	[122, 105] [210]
	Pretty-printing	[56] [222]
Basic Normalization Transformation	No Changes	[163]
	Ignore/Remove comments and whitespace	[74] [120] [52] [18] [169] [56] [167] [31] [35] [222] [213] [77] [153] [206] [146][148] [22] [190] [60] [46] [67]
	Above + Normalization	[122, 169, 24]
	Above + Transformation	[122] [187]
	Keep comments and whitespace + Others	[177] [120]* [178] [220]*
	Flexible Normalization	[24] [116]
Code Representation	Strings	[74, 163] [56]
	Substrings	[120, 174]*(fingerprint)
	Normalized Strings	[74] [122, 24](token seq.)
	Parameterized Strings	[18, 14] (p-tokens seq.)
	Word in context	[177]
	Metrics	[178] [146, 148, 22, 190, 46]
	Trees	[31, 222]
	Trees in another form	[153, 206] (Token seq.) [35](IML) R[213](XML)
	Graph	[144, 165] (PDG) [156] (PDG+AST)
	Hybrid	TMP[164] (AST+Metrics+call graph)
Comparison Techniques	String Matching	[120, 174]* (fingerprint) [74](DMP) [14, 122] (suffix-tree) [24](suffix-array) [56] (diff)
	Data Mining Neural Networks	[177](LSI) [169] (F.Sub. Seq) [213] (F.Itemset) [60]
	Discrete Comparison	[178] (metrics)
	Tree-Matching	[222] [31] (hashing)
	Graph-Matching	[156, 165] [141] (slicing)
	Euclidian Distance	[148, 67]
	Sequence Matching	[222]? (dynamic)
	Hybrid	[111]* [**?**]
Overall Complexity	Not Available	[120, 174] [161] [177]
	Linear	[163] [122, 18] [153, 206]
	Quadratic	[74](hash) [31, 35] (hash) [116] [169, 168, 170] (LOC) [146] (Method) [67]
	Non-Polynomial	[156, 141] [165]*
Clone Refactoring	Needs human hands	[14, 18][120][118][74, 72][177][122] [168, 169, 170]
	Semi-automatic(Syntactic Clones)	All metrics-based approaches (low confidence) + and most tree-based approaches
	Automatic	[31]* [142, 140, 141, 143]* [78]*

10.3 Overall Comparison of the Detection Approaches

From the studies in the previous subsections, we notice that clone detection techniques under the same approach share several common characteristics and therefore, in this subsection, we provide an overall comparison of the detection approaches with respect to the same

properties that are used for comparing different techniques in the above subsections. In Table 11 we provide such a brief summary where the first column represents the properties to be compared and the other columns represent the different detection approaches.

Table 11: Overall comparison of the detection approaches w.r.t. different properties

Properties	**String-based**	**Token-based**	**Tree-based**	**PDG-based**	**Metrics-based**	**AST + Suffix Tree**
Transformation	Removes whitespace and comments except Marcus [177] (and sometimes apply normalizations)	lexed to tokens	parsed to AST	parsed to PDG	parsed to AST to generate metrics values	parsed to AST and then another form appropriate for suffix tree
Code Representation	Filtered and/or normalized source code	Sequence of tokens	ASTs of the program based on code text and structure	Set of PDGs for procedures of the system	Set of metrics values	Suffix tree rep. of the AST nodes
Comparison Granularity	line/tokens of line	token	tree node	PGD node	metrics values for each method/block	tokens of the encoded AST in a sequence
Computational Complexity	Depends on algorithm	linear	quadratic	quadratic	linear	linear
Refactoring opportunities	good for exact matches	Need post-processing	finds syntactic clones, good for refactoring	good for refactoring	manual inspection is required	Syntactic clones, good for refactoring
Language Independency	easily adaptable	no syntactic knowledge but needs a lexer	needs parser	Needs syntactic knowledge and PDG generator	mostly needs parser	needs parser

10.4 Clone Detection Tools

For most of the clone detection approach discussed earlier, a corresponding tool name is proposed. In this section, we list the the different clone detection tools available in the literature in a tabular form (however, there are several others). Table 12 shows the tool details where the first column represents the tool name, 2nd column refers the citations for that tool, the 3rd column indicates the languages currently supported by the tool, the 4th column shows whether the tool is a clone detection tools or plagiarism detection tool or designed for other reengineering task, the 5th column represents the approach used in developing the tool, the 6th column indicates whether the tool is for commercial or academic use, the 7th column shows the maximum input size used in validating the tool and the last and 8th column tells us whether the tool was empirically validated or not.

Table 12: List of Clone Detection Tools

Tool	Citations	Sup. Lang.	Domain	Approach	B.Ground	L.Input	Validated?
Dup	Baker [14, 18]	c, c++, Java	CD/Unix	line-based/text-based	academic	1.1M LOC	With two systems
JPlag	Prechelt et al. [192]	Java, c, c++, Scheme, NL text	PD/Online	Token/Greedy String	Academic	236 LOC	Student assignments/artificial data
CloneDr	Baxter et al. [31, 30]	c, c++, Java, COBOL, Others with DMS domain	CD WindownNT	AST/Tree Matching	Commercial	400K LOC, C Code	Process Control System
DupLoc	Ducasse et al. [74, 72]	Language Independent/ 45 Mins to adapt	CD/Visual Works2.5	Line/Exact string matching	Academic	46K LOC	With 4 systems of different languages
CCFinder	Kamiya et al. [122]	C, C++, Java, COBOL and other with lexical analyzer and transformation rules	CD/ Windows/ NT	Transformation /Token comp. with suffix tree	Academic	10M LOC	With 4 systems
CP-Miner	Li et al. [168, 169]	C, C++	CD & Copy-pasted bugs detection /Windows/ Linux	Sequence Database/Frequent subsequent mining	Academic	4365K LOC	Several systems
Sim	Gitchell et al. [90]	C	PD/Linux	Parse tree to string / String alignment	Academic	3.5K bytes	With 65 student assignments
Covet/CLAN	Mayrand [178]	C, Others supported by Datrix	CD	Metrics from Datrix, 4 Points of comp., Ordinal scale of 8 cloning level	Academic	507K LOC	With two telecommunication systems
DiLucca Pro.	Di Lucca et al. [67, 66]	HTML client & ASP server pages	Duplicated web-pages/PD	Sequence of tags/ Levenshtein distance	Academic	331 files	With 3 web applications
eMetrics	Fabio et al. [46, 161]	HTML & scripting languages	Visual inspection of potential function clones	Gets potential function clones from eMetrics	Academic	403 files	Validated with 4 applications

Continued on Next Page. . .

Table 12 – Continued from previous page

Tool	Citations	Sup. Lang.	Domain	Approach	B.Ground	L.Input	Validated?
Konto's Tool	Kontogiannis et al. [146]	C, (other possible?)	System clustering, redocumentation & program understanding	Direct comparison of metrics values and Dynamic programming approach of begin-end block	Academic	>300K LOC	With several systems
ccdiml (Bauhaus)	Bellon [35, 35]	C, C++	CD/Linux	AST/Tree matching	Academic	235K LOC?	With several systems?
Coogle	Sager et al. [202]	Java	Finding similar java class	AST to FAMIX & Tree matching	Academic	?	Eclipse plug-in and test cases
Deckard	Jiang et al. [116]	C, Java and others with formally specified grammar, YACC grammar	CD/Linux	Tree, Characteristics vectors, Euclidean space	Academic	5,287K LOC	With Linux kernel and JDK
cpdetector	Koschke et al. [153, 154, 152]	C, C++	CD/Linux?	AST to sequence of AST tokens, suffix tree	Academic	235K LOC	With 4 systems and 8 tools
Duplix	Krinke et al. [156, 157]	C?	CD	PDG, graph matching	Academic	25K LOC	several test cases
clones	Koschke [153]	C, C++, Java, COBOL, VB	CD	Tokens/Suffiex tree	Academic	235K LOC	With 4 systems and 8 tools
cscope	Frenzel [153]	C, C++	CD	Tokens/Suffix Tree/Post processing for syntactic clones	Academic	235K LOC	With 4 systems and 8 tools
PDG-DUP	Komondor & Horwitz [140, 141, 142]	C, C++?	CD / Clone Refactoring	PDG/ Slicing	Academic	?	With some systems
Tairas's Tool	Tairas et al. [206]	C, C++, C#, Phoenix framwork languages?	CD/Exact function clones	AST/Suffix tree	Academic	1500K LOC	With several systems
RTF	Basit et al. [24]	C	CD	Token/Suffix-array	Academic	3025K LOC	With Linux parts

10.5 Frequently Used Software Systems

Table 13 shows some frequently used software systems used in clone detection research. Although many of the studies use several versions of a particular system, we just list the name of the system and provide approximate size in lines of code (LOC). We see that Linux

Kernel has been used most covering different areas, starting from validation of techniques to refactoring. Similarly, we see that some systems (e.g., ArgoUML and DNSJava) are particularly used in analyzing the maintenance issues of clones (e.g., harmfulness of clones, ratio of consistent and inconsistent changes etc.).

Table 13: Frequently Used Software Systems in Clone Detection Research

Software	Size(Approx.)	Language	Purpose	Citations
Linux Kernel or part	3MLOC	C	Technique Validation	[169] [24] [116] [122]
			Visualization	[114] [113] (SCSI)
			Bug-detection	[115]
			Clone Evolution	[8] [166]
			Clone Coverage	[47]
			Taxonomy	[125]
			Software Evolution	[181]
JDK or part	570KLOC	Java	Technique Validation	[116] [122] [163] [213] [190]
			Taxonomy	[22]
			Refactoring	[20] [21] [25, 112] (buffer library)
ArgoUML	118KLOC	Java	Maintenance Analysis	[9] [155] [23]
DNSJava	25KLOC	Java	Maintenance Analysis	[9] [137] [136] [171]
Mozilla FireFox	1356KLOC	C/C++	Clone Evolution	[19] [91]
			Change couplings	[92] [91]
Ant	141KLOC	Java	Tracking Clones	[71]
			Maintenance Analysis	[23] [105] [104]
			Refactoring	[106] [104]
ANTLR	44KLOC	Java	Technique Validation	[190]
			Taxonomy	[22]
			Refactoring	[105] [106]
Postgresql	235KLOC	C	Technique Validation	[153] [169] [35] [36] [10]
			Clone Analysis/Visualization	[123]
FreeBSD	3MLOC	C	Technique Validation	[167] [169] [122]
Apache httpd or part	261KLOC	C	Technique Validation	[163] [169]
			Taxonomy	[127]
			Maintenance Analysis	[123]
Carol	9KLOC	Java	Clone Evolution	[137]
			Maintenance Analysis	[155] [136]
SNNS	105KLOC	C	Technique Validation	[153] [10] [36]
			Reference Data	[215]

11 Evaluation of Clone Detection Techniques

As we see in the previous sections, there are plenty of clone detection techniques and their corresponding tools, and therefore, a comparison of these techniques/tools is worth much in order to pick the right technique for a particular purpose of interest. There are several parameters with which the tools can be compared. These parameters are also know as clone detection challenges. In the following we list some of the parameters we use for comparing the different tools/techniques:

- Portability: The tool should be portable in terms of multiple Languages and dialects. Having thousands of programming languages in use with several dialects for many of them, a clone detection tool is expected to be portable and easily configurable

for different types of languages and dialects tackling the syntactic variations of those languages.

- Precision: The tool should be sound enough so that it detect less number of false positives i.e., the tool should find duplicated code with higher precision.
- Recall: The tool should be capable of finding most (or even all) of the clones of a system of interest. Often, duplicated fragments are not textually similar. Although editing activities on the copied fragments may disguise the similarity with the original, a cloning relationship may exist between them. A good clone detection tool will be robust enough in identifying such hidden cloning relationship so that it can detect most or even all the clones of the subject system.
- Scalability: The tool should be capable of finding clones from large code bases as duplication is the most problematic in large and complex system. The tool should handle large and complex systems with efficient use of memory.
- Robustness: A good tool should be robust in terms of the different editing activities (c.f., Section 11.2) that might be applied on the copied fragment so that it can detect different types of clones with higher precision and recall.

11.1 Higher Level Evaluation of the Detection Approaches

Before going to the actual experimental evaluation of the different tools, in Table 14 we provide a higher level comparison of the detection approaches in terms of portability, precision, recall and scalability based on our study in Section 10.

We see that text-based techniques (line-based and parameterized line-based in the table) are easily adaptable to different languages of interest as such techniques need at most lexers of these languages. Precision of this approach is also very good as clones are determined by textual similarity. However, this approach cannot detect many clones of the system that are produced with some of the editing activities (c.f. Section 11.2). The scalability of this approach on the other hand depends on the comparison algorithm and optimizations used.

Token-based techniques are bit language dependent as they need at least a lexer (and sometimes transformation rules) for the language of interest. Although, their precision is low (return many false positives due to the normalization and application of transformation rules) the recall and scalability of these techniques are high. They can detect most clones and using suffix tree algorithm they are scalable enough to tackle token sequences of large systems.

Although parse-tree based techniques are not good while considering the portability, recall and scalability, they are good at detecting real clones with high precision. As these techniques focus on the structural property of the source code, very few false positives are returned. Similarly, PDG-based techniques focus on both the structural, data and control flow properties of the source code fragments and therefore, return very few false positives. However, they are mostly language-dependent (needs PDG generator) and not scalable (Graph matching is too costly). Metrics-based techniques, on the other hand, are good enough both in terms of precision and recall with high scalability. However, these

techniques also need to generate AST/PDG from the source and therefore, heavily language dependent.

The hybrid approach of using suffix trees on ASTs is as scalable as the token-based techniques while providing high precision. However, its portability and recall are still the same as the usual tree-based techniques.

Table 14: Higher level comparison of the detection approaches

Approach	Portability	Precision	Recall	Scalability
Line-based	High, needs lexer at most	100%, No false positives as checks for exact copies	Low, only finds exact copies	Depends on Comparison algorithms
Parameterized line based	High, needs lexer at most	Medium, may return false +ves	Medium, can detect only exact and parameterized clones	Disagreement among researchers
Token-based	Medium, Needs lexer transformation rules	Low, due to normalization and/or transformation returns many false +ves	High, can detect most clones	High with suffix-tree algorithm
Parse-tree based	Low, needs parser	High, parse-tree considers structural info also	Low, cannot detect all types of clones, however, several approaches taken to overcome this	Depends, how comparison is made
PDG-based	Low, needs PDG generator	High, considers structural and semantic info too	Medium, cannot detect all clones	Low, graph matching is costly
Metrics-based	Low, needs parser/PDG generator to get metrics values	Medium, two code blocks with similar metrics values may not be same	Medium, cannot detect many clones	High, after getting metrics values, needs only to compare the metrics values for each begin-end block
AST + Suffix tree	Low, needs parser	High, considers structural info	Low, cannot detect all clones	High, applies suffix tree comparison on AST nodes

11.2 Higher Level Robustness of the Detection Approaches

As mentioned before, a good tool should be robust enough in dealing with the different editing activities that can be applied on the copied fragments. In the following, we list them in the most general form:

- Variation in Layout(VOfLayout): When a code segment is copied, it is very usual that there might be some sort of variations in the visual organization of the source code, i.e., how the source code is organized in terms of comments, indentation, blank lines and the position of the different program elements. In the following we list them in details:
  - Variations in Whitespace (VWSpace): In the copied code fragment, there might be an editing activity in terms of inserting or removing whitespace (tabs, new

lines etc.). A good tool should ignore such whitespace in its detecting algorithm as insertion/deletion of whitespace does not affect on the functionality of the copied fragment. Copied fragments with such whitespace alternations are definitely exact clone and hence falls in the category of *Typ I* clone.

- Variations in Comments (VCom): Comments can be added or removed or modified in the copied fragment. A good tool should be robust enough in dealing with comments either by ignoring them in both fragments or by considering them in detection algorithm. Whatever the option is, proper treatment should be taken care of. While most of the available techniques ignore comments, Marcus and Maletic [177] consider comments as a significant hint in detecting high level semantic clones. There is also some metric-based approaches (e.g., [178]) that consider the volume and layout of comments in calculating the metrics. However, any sort of alternation of comments should not affect the detection techniques as the copied fragments with comments alternations are definitely exact clone and hence falls in the category of *Typ I* clones.
- Variations in Code Elements' Positions (VCEPos): Sometimes the copied fragment is codified (i.e., proper indentation) to a well organized structure by changing the position of some code elements (e.g., begin/end symbols of a block/loop) within the same line (intra-line) or between different lines (inter-lines). Intra-line position changing is basically the alternations of whitespace (with no line breaks) between the program elements within a line and thus, may not be problematic for a clone detector. Whereas inter-line position changing may affect a detection approach (e.g., line-by-line approach is affected). However, even if there is an inter-line position changing, the copied fragment is still the same as the original with respect to functionality and are definitely exact clone and therefore, falls in the category of *Typ I* clone. A good clone detector should detect such clones as Type I clones.

- Renaming of Identifiers (RenamOfIden): Consistent renaming of the identifiers (variable names, method names, literal values, types etc.) can easily be made in the copied code fragment without changing the program structure by keeping the syntactic, corrective and semantic similarity. Clones of this type are classified as *Typ II* clones. *Typ II* clones has the same potential as of the exact clones in terms of the applicable reengineering opportunities such as refactoring. A clone detection tool is expected to have especial treatments for detecting *Typ II* clones without leaving any kind of manual preprocessing and/or postprocessing for the user.

- Code Modifications/Insertions/Deletions (Code M/I/D): A copied fragment may be modified by changing its existing statement(s) or completely new statement(s) may be added to it or even existing statement(s) may be deleted from it as per the requirements of the developer. As a result of such activities, the copied fragment may loose the textual, lexical, syntactical and/or even semantical similarities with the original. Clones produced with such activities are classified as *Typ III* clones. A good clone detection tool should be robust enough to deal with such editing activities so as to detect and report *Typ III* clones (along with the editing activities made).

- Reordering of Statements (ReordOfStat): Some statements can be reordered in the copied fragment without changing the program behavior and correctness (i.e., no loss of semantics) while losing textual, lexical and structural (e.g., syntactical) similarities compare to the original. A reordering of one source line/statement can be seen as a deletion followed by an insertion of that particular line/statement and therefore, clones produced with such activities can be classified as *Typ III* clones. On the other hand, as the semantics of the code segments is unchanged, these types of clones can also be treated as *Type IV* clones. A good clone detection tools should efficiently smells such reordered statements and report such fragments as clones.

- Variations in Code Structure (VOfCoStruct): A copied fragment can be modified extensively by changing the structure of the fragment. For example, a code segment with a *while loop* can easily be changed to a code segment with a *for loop* without violating the semantic meaning of the copied segment compare to the original. A copied fragment can be made completely different from the original with respect to textual and/or syntactical similarity by changing its various control statements to their alternative ones supported by that particular language. As these kind of alternations of code structures are related to the semantic similarity, the clones produced by these activities are definitely *Typ IV* clones. On the other hand, a minor such changes within a copied fragment may produce *Typ III* clones. A good clone detection tool is expected to be sound enough to detect such semantic clones. However, detecting semantic clones is undecidable in general.

In Table 15 we provide an overall robustness summary of different approaches. However, this overall generic ranking of different approaches w.r.t. robustness is not always true. For example, making AST after normalization of code on various levels might increase the robustness on the different changes mentioned above.

11.3 Tool Evaluation Experiments from the Literature

Literature is not only enriched with several clone detection techniques but also with several experiments of tool comparison/evaluation in terms of portability, precision, recall and scalability. In this section, we provide a summary of the available tool comparison experiments from the literature.

One of the first experiments is conducted by Bailey and Burd [44] who compared three clone detection tools, *CCFinder* [122], *CloneDR* [31] and *Covet* (re-implementation of [178] by Bailey and Mayrand) and two plagiarism detectors *JPlag* [192] and Moss [5].

At first they validated all the clone candidates of the subject application obtained with all the techniques of their experiment and made a human oracle which was then used for comparing the different techniques in terms of several metrics to measure various aspects of the found clones, such as scope (i.e., within the same file or across file boundaries) including the recall and precision of the tools. For the manual validation phase and the clone candidates, they considered several attributes such as similar or identical control flow and layout (e.g., two functions both contained the same number of if-statements testing similar conditions), similar or identical method names (e.g., saveGraph() and saveGINGraph()), similar or identical variables and similar or identical comment blocks. In Table 16 we

Table 15: Higher level robustness of the detection approaches w.r.t. editing activities

Approach	VOfLayout	RenamOfIden	ReordOfStat	Code M/I/D	VOfCoStruct
String-baed	High, doesn't affect	Low, look for exact matching unless normalization applied, however, with parameterized match its robustness is high as well	Low, look for line by line matching	Low, matching fails	Low, matching completely fails
Token-based	High, doesn't affect	High, normalization of identifiers make them same	Medium, especial treatment and/or post processing required	Low, especial treatment and/or postprocessing required	Low, mismatches of tokens
Tree-based	High, automatically ignores by the parser	High, ignores identifier information	Low, structure changes	Low, structure changes	Low, structure changes
PDG-based	High, ignores by the PDG generator	High, ignores	High, considers data and control flow info	Medium	High, again considers the semantic info
Metrics-based	Medium/High?, ignores before metrics generation	Depends, if metrics are generated from AST/PDG then ignores, if directly from source code, then may affect	Medium, metrics values might be similar	Medium, metrics values may not change a lot	Medium, metrics values might still closer
AST + Suffix tree	High, ignores	High, ignores	Low, structure changes	Low, structure changes	Low, structure changes

provide the precision and recall of the five tools from Burd and Bailey's tool comparison experiment.

Table 16: Precision and recall from Burd and Bailey's Experiment (from [44])

	CCFinder	CloneDr	Covet	JPlag	Moss
Precision	72%	100%	63%	82%	73%
Recall	72%	9%	19%	12%	10%

As can be noticed, the AST-based tool *CloneDR* shows 100% precision indicating that this tool does not produce any false positives in its detected clones. However, the recall of *CloneDR* is the lowest (9%) which essentially implies that this tool is not good for finding the clones from a system where the concern is to find all or majority of the clones of that system. Although, the metrics-based tool *Covet* is reasonable in terms of recall (19%) compare to the other tools (except *CCFinder*), it shows the lowest precision (63%) than the others. It is interesting to note that the plagiarism detection tools *JPlag* and Moss show kind of similar precisions and recalls compare to the clone detection tools although these tools detect clones across different files only. The token-based tool *CCFinder* seems to be the most appropriate in finding clones. It shows the highest recall (72%) with a huge difference than other tools and in the same time the precision is also reasonable (72%), only produces 28% false positives of the found candidate clones while detecting most of the clones (only missing 28%) of the system. However, their case study is based on only one medium size system

(16K LOC). Although they were able to verify all the clone candidates, the limitation of the case study in terms of system quantity and size makes their finding questionable. Moreover, the intention of their analysis was to assist the preventative maintenance tasks, which may also had an influence in validating the candidate clones.

Probably, considering the limitations of Burd and Bailey's study, Bellon conducted a larger tool comparison experiment [36, 34] with the same three clone detection tools that used in Burd and Bailey's study (Mayrand et al.'s CLAN is directly used in Bellon's study where not only the metrics but also the tokens and their textual images are compared to identify *Typ I* and *Typ II* clones) and with three more additional tools, one more token-based tool, *Dup* [18], one PDG-based tool, *Duplix* [156] and one text-based tool, *Duploc* [74]. They also applied diversity in choosing the software systems, 4 Java and 4 C systems in the range of totalling almost 850KLOC.

In Table 17, a summary of the Bellon's experiment is presented in terms of precision, recall, speed, RAM and the types of clones detected for each of the tools. The data is reported at an ordinal scale –, -, +, ++ where – is worst and ++ is best (exact measures can be found in the corresponding papers). In some cases what(?) symbols are used to represent that they were unsure about that particular measure. As in the study of Burd and Bailey, a human oracle was formed with the candidate clones from all the tools and then manual verifying. However, Bellon was able to verify only 2% of the candidate clones due to his time limit.

Table 17: Summary of Bellon's tool comparison experiment (adapted from [35])

Tool		**Precision**	**Recall**	**RAM**	**Speed**	**Clone Types**		
Based On	Name					Typ I	Typ II	Typ III
AST	CloneDr	+	-	-	-	+	–	X
Token	Dup	-	+	+	++	++	+	–
	CCFinder	-	+	+	+	++	++	-
PDG	Duplix	-	-	+	–	–	–	+
Metrics	CLAN	+	-	++	++	+	–	–
Line	Duploc	-	+	?	?	+	-	–

From Table 17, it is clear that no tool is good enough both in terms of precision and recall. In fact, precision and recall is complementary for each of the tool except PDG-based *Duplix* where both reported as worst. While *CloneDR* and CLAN (with token and textual comparison) shows high precision, their recall is very low and vice versa for the other tools. RAM and Speed are closely related and found very high for the metrics-base tool CLAN, and high for token-based tools, *Dup* and *CCFinder*. PDG-based tool *Duplix* is reported as very bad in terms of speed.

Interesting results can be observed in the case of produced types of clones also. It is observed that most of the tools (except *Duplix*) can detect *Typ I* and *Typ II* clones. Additionally, *CCFinder*, CLAN and *Duploc* can also detect *Typ III* clones whereas *Duplix* can only detect *Typ III* clones. In this experiment, the metrics-based tool CLAN shows better precision in the sense that Bellon also compared the tokens and textual images of the functions for which metrics values were compared.

Bellon's experiment [34, 35] has been reused by Koschke et al. [153] for evaluating their new AST-suffix tree based tool, *cpdetector*. To see how good their new tool is, not only in terms of precision and recall but also in runtime, they developed some variants of the

existing tools. For example, *ccdiml* is a variant of Baxter's *CloneDR*, *clones* is a variant of Baker's *Dup* and *cscope* is a variant of *clones* with an addition post-processing step of finding syntactic clones. While they reused the tools of Bellon's experiment they worked only with 4 C systems as their developed tools were only suitable for C systems and provided detail results for only one system, SNNS 4.2, a neural network simulator.

In Table 18, a summary of the Koschke et al.'s experiment for SNNS is presented in terms of precision, recall, number of candidates found, running time (second) and the types of clones detected for each of the tools. The data is reported at an ordinal scale of –, -, +-,+, ++, +++ where – is worst and +++ is best (exact measures can be found in the corresponding paper). In some cases what(?) symbols are used to represent that they were unsure about that particular measure.

Table 18: Summary of Koschke et al.'s experiment (based on [153] and [35])

Tool		**Precision**	**Recall**	**No. of Candidates**	**Running time**	**Clone Types**		
Based On	Name					Typ I	Typ II	Typ III
AST	CloneDr	++	–	1434	–	++	-	X
	ccdiml	+	+	18245	+-	++	++	-
Token	Dup	+	+-	8978	++	++	+	–
	CCFinder	+	++	18961	+	++	++	-
	clones	–	+-	32975	++	++	+	-
	cscope	-	+-	17758	++	++	+	-
ASST	cpdetector	+	+-	4852	+	++	+	–
PDG	Duplix	-	–	12181	–	–	–	++
Metrics	CLAN	+++	–	318	++	++	+	-
Line	Duploc	+	-	5212	?	++	-	–

From Table 18 we see more or less the same statistics as of Bellon's experiment for the common tools. We see that the AST-based tool *ccdiml* shows a good recall contrary to our previous beliefs for AST-based tools. However, still the average recall for the token-based tools is almost double than the AST-based tools (while no token-based tools get a minus, AST-based *CloneDR* gets double minus). On the other hand, in terms of precision, the token-based tools are not that good compare to the AST-based or metrics-based tools. Although, *clones* is a variant of *Dup*, it shows very low precision while *Dup* shows good precision. Unlike *Dup*, *clones* is pure token-based and does not depend on layout (e.g., line breaks). However, the detection of *Typ I* and *Typ II* clones must be performed in a post-processing steps.

With respect to the number of candidates found for SNNS, *clones* found 71% more clones than *CCFinder*. It is worth to note that *CCFinder* was the best tool in Bellon's experiment in terms of finding candidate clones. While the metrics-based tool *CLAN* is the worst in finding candidate clones, token-based tools can find almost doubles than AST-based tools. In terms of running time, metrics-based tool *CLAN* is very good and so the token-based tools. PDG-based *Duplix* is found to be worst in terms of running time.

As of Bellon's experiment, we see that all the tools except PDG-based *Duplix* are good at detecting *Typ I* clones and *ccdiml* and *CCFinder* are very good in detecting both *Typ I* and *Typ II* clones. Although *Duplix* is not good for detecting Typ I and *Typ II*, it is good at detecting *Typ III* clones. Almost all the remaining tools are not good in detecting *Typ III* clones. We also see that *CloneDR* was unable to find any *Typ III* clones in their experiment.

Rysselberghe and Demeyer [200] evaluate three representative detection techniques, string matching [74], token matching [18, 122] and metric fingerprints [178, 146]. After having reference implementations for each of the approach in Java, they use five small to medium (under 10KLOC) sized cases for evaluating the techniques in terms of portability, kinds of duplication reported, scalability, number of false matches, number of useless matches (matches which are not worth to be removed by means of refactoring) and number of recognizable matches (matches that are easily recognizable). In Table 19 a summary of their results is provided where the number of symbols indicates the comparative degree of satisfaction of the property studied and where positive properties are marked with + and negative with −. The additional symbols placed between brackets denote the additional impact when using block-granularity.

Table 19: Summary of Rysselberghe and Demeyer's experiment (from [200])

Approach	**Portability**	**Duplication**	**Scalability**	**Matches: No. of**		
				False	Useless	Recognizable
Simple Line	+++++	general lines	+		− − − −	+
Parm. Line	+++	general line block	+++	−		++++
Token suffix tree	++	general token block	++			++++
Metrics finger print		functional entity	+++	− − (− −)	− − (− −)	+++++

As can be seen their findings are more or less similar to the previous studies with some exceptions. 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 (e.g., refactoring). They also evaluate the different techniques with respect to 1. suitability (can a candidate be manipulated by a refactoring tool?), 2. relevance (is there a priority which of the matches should be refactored first?), 3. confidence (can one solely rely on the results of the code cloning tool, or is manual inspection necessary?), and 4. focus (does one have to concentrate on a single class or is it also possible to assess an entire project?) [201]. In Table 20 we provide a summary of their findings.

From Table 20, it is clear that metric fingerprint techniques are best suitable to work with a refactoring tool as these techniques identify duplicated code or scope blocks (e.g., function body) which can readily be refactored with straightforward refactorings like "*pull up method*" and "*move method*". No significant difference is noticed between the approaches with respect to *Relevance* and *Focus*. All the techniques provide only clone length and filename information of the returned clones. This information is not sufficient for a real assessment of the relevant clones (No difference w.r.t. Relevance). On the other hand, all the techniques can analyze entire projects as well as more fine grained entities like packages or classes (No difference w.r.t. Focus). Simple line matching only targets the exact matches and hence does not suffer from the *Confidence* problem whereas all other techniques may return false matches and a manual inspection is required to find the real clones. Although metric fingerprint is best suitable for a refactoring tool and simple line matching finds clones with high confidence, they are not good in providing more refactoring opportunities. They both return limited number of clones compare to the other two. Parameterized line

Table 20: Evaluation of the techniques from a refactoring perspective

Approach	Suitability	Relevance	Confidence	Focus	Refac. Opportunities
Simple Line	Low, no block boundaries, so in-depth manual inspection is required	No difference	High, only exact matches are targeted	No Difference	Medium, finds only exact matches
Parm. Line	Low, no block boundaries	No. difference	Low, may return false matches	No difference	High, finds more matches with more false matches
Token suffix tree	Low, no block boundaries	No difference	Medium, targets only exact & one-to-one matching	No difference	High, finds exact and parameterized matches
Metrics finger print	High, identifies duplicated code or scope block (e.g., function body) which can readily be refactored	No. difference	Low, returns false matches, manual inspection is required	No difference	Low, finds only limited matches

matching returns more clones but with more false positives than parameterized matching with suffix trees. Parameterized matching with suffix trees returns both the exact clones and parameterized clones following a one-to-one mapping of the identifiers and literals and hence returns very few false positives but more matches compare to other classical line-by-line matching. Therefore, with respect to refactoring activities, parameterized matching with suffix trees approach is rated best in this study.

Another interesting study has been conducted by Bruntink et al. [41] where several clone detection techniques are evaluated in terms of finding cross-cutting concerns [133] in C programs with homogeneous implementations. Three clone detection techniques namely token-based *CCFinder* [122], AST-based *ccdiml* [35] (variant of Baxter's *CloneDR*. [31]) and PDG-based *PGD-DUP* [140] are used in this study in order to find cross-cutting concerns. Some well known cross-cutting concerns such as error handling, tracing, pre and post condition checking, and memory error handling were targeted to find in their study. Their study showed that both *ccdiml* and *CCFinder* can find null-pointer and error handling concerns while *ccdiml* can find the range checking concern too. On the other hand, *PDG-DUP* finds cross-cutting concerns that could not be found with *ccdiml* or *CCFinder*. *PDG-DUP* can efficiently finds tracing and memory handling concerns.

Unfortunately, there are no comparisons that cover the most recent tools such as *CP-Miner* or *CCFinderX*, *DCCFinder* or *Deckard*. However, Jiang et al. [116] claim that their tool *Deckard* is more scalable and accurate than *CloneDR* and *CP-Miner*. They apply these three tools to JDK and Linux kernel and compare *Deckard* with *CloneDR* and *CP-Miner* considering several parameters. Their studies show that *Deckard* finds more clones than the other two. It is also more scalable than *CloneDR*, which is also tree-based and as scalable as the token-based *CP-Miner*. Although an independent comparison study is not available for the recent tools, it is commonly agreed by all the above mentioned studies that there is no approach or tool overtopping all others. All approaches have their distinct advantages and

drawbacks and further improvement or more hybrid approach is required for overcoming the limitations of the techniques while preserving the the strengths.

12 Visualization of Clones

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, line number, starting position, ending position 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 amount of clones in large systems (For example, *CCFinder* resulted 13,062 clone pairs for Apache httpd [127]). Because of the insufficient information on the returned clones, their various contexts and huge amount, the visualization of clones becomes difficult. For the proper use of the detected clones, especially for clone management, the aid of a sound 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 [52] visualization of clones in the form of two-dimensional charts where all software units are listed on both axes [211, 74, 18, 196]. 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. [104]. 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 [18, 74, 196] 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.

Johnson [119] has applied Hasse diagrams for visualizing cloning relationships (textual similarity) between files. A Hasse diagram is consisted of nodes and edges. 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 relation between 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 has proposed to navigate the web of files and clone classes via hyper-linked web pages [117]. 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 hyperlink properties are very nice to navigate users, 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.

In addition to scatter-plots, Gemini [105], that uses the output of *CCFinder*, also provides visualization through metrics graphs and file similarity tables. It allows one to browse the code clones either pair by pair or using clone classes. Aries [106] (which is also based on the output of *CCFinder*), is a refactoring support environment for clones using metric

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.

Lanza and Ducasse's polymetric views [162] have been successfully used in visualizing clones by Rieger et al. [196]. They also focus on to visualize cloning relationships in order to find out the parts of the systems that are in a cloning relationships and the parts of the system that contain many clones. 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 clone comprehension tool, CLICS is developed by Kapser et al. [127]. CLICS uses the output of *CCFinder* and a taxonomy of clone types [126] 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 in CLICS because of it limited scalability.

Tairas et al. [207] provides an Eclipse plug-in for displaying the results of *CloneDR*. Their approach extends AJDT visualizer [3] and provides a different approach than scatter plots in visualizing clones. The integration of *CloneDR* with Eclipse allows the tools to take advantage of the rich environment of the IDE, which offers frameworks for wizards, views and editor connections.

Extending GUESS [1] Adar and Kim [2] provide a code clone exploration tool, SoftGUESS. 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. [114] has extended the concept of coupling and cohesion to code cloning by visualizing the clone relations in the architecture level. Their framework is useful in generating data to investigate and manage cloning activities within and across subsystems. Jiang and Hassan [113] 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 Table 21, a summary of the different visualization approaches are shown. As we see from the table each of the approach has different types of source code granularities, i.e., code segment(CC), File(FE) and subsystems(SS), and clone relations i.e.,clone pair (CP), clone class (CC) & super clone(SC).

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

Table 21: Different Types of Clone Visualizations (adapted from [114])

Citations	Visualization Type	Granularity			Clone Relation		
		CS	FE	SS	CP	CC	SC
Rieger et al. [196]	Scatter Plot	+	-	-	+	-	-
	Duplication Web	-	+	-	+	-	-
	Tree Map	-	-	+	-	+	-
	System Model View	-	+	+	-	+	-
	CC Family Enumeration	-	+	-	-	+	-
Ueda et al. [211]	Scatter Plot	+	-	-	+	-	-
	File Similarity Graph	-	+	-	-	-	-
	Metric Graph	+	-	-	-	+	-
Jiang et al. [114]	Cohesion and Coupling	-	-	+	-	+	+
Johnson [119]	Hass Diagram	-	+	-	-	+	-
Helfman [103]	Scatter Plot	+	-	-	+	-	-
Johnson [117]	Hyper-linked Web	-	+	-	-	+	-
Kapser & Godfrey [123]	Dependency Graph	-	-	+	+	-	-

13 Removal, Avoidance and Management of Code Clones

A primary purpose of clone detection is to remove them from the system via refactoring for improving the system's quality. However, it may also be possible to avoid clones right from the beginning in the development process. Rather than removing, existing clones can also be managed in the evolution of a system. In the following subsections we discuss these issues in short:

13.1 Removal of Code Clones

One of the major objectives of detecting clones is to remove them from the software system through (automatic) refactoring. 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 *Exact Method* refactoring [78, 107, 212, 121, 142, 31, 145] that replaces the cloned code by a function call to a newly created function created from the shared code of the clone fragments. *Type I* and *Type II* clones, where clones are exact copies or differ only in identifiers are suitable for such simple functional abstraction. Fanta and Rajlich [78] remove function and class clones from industrial object-oriented code by reengineering scenarios obtained with the aid of automated restructuring tools. Higo et al.'s CCShaper [107] filters the output of *CCFinder* to find good candidates for the *Extract Method* and *Pull Up Method* refactoring patterns. Higo et al. [106] 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 [143, 142]. 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 [121] 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 [201].

However, all the above refactorings impose some kind of restricted preconditions. For example, the *Extract Method* refactoring is applicable only to the clones of consecutive statements. Although, in some cases, it is possible to reorder the statements [142], limitations of the programming language, such as inability of Java to pass back multiple values, can make such a refactoring impossible. Similarly, for *Pull Up Method* refactoring, the shared code must have method granularity and that they have a common superclass.

It may be possible to use macros to replace cloned code provided that the programming language in question has a preprocessor [31]. 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. As compiler is not able to check for certain common mistakes in presence of macros and as macros may obfuscate the source code, developers normally avoid introducing macros [76]. Alternative approaches of solving cloning problem are therefore proposed in the design level. One may use the design patterns to avoid clones by better design [21, 20]. Balazinska et al. have developed a clone reengineering tool, called CloRT [22, 21]. CloRT 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 also been proposed by Synytskyy et al. [205]. 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., [21]) 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 [57]. As an alternative approach optimization of clone refactoring with constraints and conditions can applied [39].

13.2 Avoidance of Code Clones

Clones are considered harmful in software maintenance (for a counter argument and details c.f., Section 5) and should be removed or detected at least. However, it would have been much better if there is no clone at all in the developed system so that we would not have to think about neither removal nor detection of clones. The idea is to use a clone detection tool in the normal development process to avoid cloning in the software right from the beginning. There are two ways of how to use a clone detection tool in the development process [158] for avoiding clones. One way is the *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 the *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 [38].

13.3 Management of Code Clones

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 (c.f., Section 5). 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 system [182]. 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. [208] 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 [71] 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 groups. 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. [50]. 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 [54]. 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 [89]. Based on this motivation, Balint et al. [23] 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. [19] that introduces some 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 the 12 versions of Mozilla Firefox internet browser.

14 Evolution Analysis of Clones

Clones are hindrances to software evolution and its maintenance. There are also several studies that look at how clones evolve in different versions of a software system. These studies focus on the presence and dynamic behaviors of clones in the evolving versions of software systems.

The analysis of clone evolution was first performed by Laguë et al. [158] for evaluating whether a clone detection tool helps when integrated with 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 experiment, 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 a time series by analyzing the clones of several versions of a medium scale software system. The proposed model is validated with 27 subsequent versions of mSQL and found that it can predict the clone percentage of subsequent release with an average error rate below 4%.

In a follow-up study, Antoniol et al. [8] analyzed the clones of different releases of Linux Kernel and observed that cloning scope is mainly limited to subsystems. While most of the clone groups are scattered within a subsystem, very few of them scattered across subsystems. Moreover, it is noticed that a newly introduced architecture 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 [96, 95] show similar results and conclude that cloning is a common and steady practice in the Linux kernel.

A different strategy is applied in studying clone evolution by Kim et al. [136]. 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 groups over different versions of a software system. They studied for example, how each element of a group of clones are 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 have concluded that there are many volatile clones (c.f., Section 7.4.1) 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 (c.f., Section 7.4.2) in the system and a refactoring of such clones may not be possible because of the limitations of programming languages.

Li et al. [169] also studied the clone evolution in Linux and FreeBSD while evaluating their tool, *CP-Miner* [168]. Their study shows that 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.

15 Quality Analysis Based on Code Clones

Although it is generally agreed (not all cases, c.f., Section 5) that clones have a bad impact on software maintenance, there are very few studies clarifying the impact of clones on software quality attributes. To date only one study by Monden et al. [185] has been conducted for evaluating the relation between clones and software quality attributes. They have studied the impact of code clones on an industrial legacy system and attempted to derive relations between clones and two quality attributes, namely, *reliability* and *maintainability*.

In order to evaluate the relation between code clones and *reliability*, they have used the number of faults per line as a measure. On the other hand, for *maintainability*, they have used revision numbers from the version system as a measure considering that modules with higher revision numbers on average to be more difficult to maintain than others.

Their study shows some contradictory results for software *reliability* quality attribute. Modules/files with code clones are on average 1.7 times more reliable than files/modules without code clones. However, files/modules with larger clones are less reliable than others. In the case of *maintainability*, the results support the usual hypothesis that clones have a bad impact on software maintenance. Files/modules with code clones are less maintainable than files/modules without code clones. The maintenance overhead increases linearly with the increasing size of code clones. They explained this phenomenon with the fact that files/modules with code clones are to be changed/updated frequently than files/modules without clones and hence, less maintainable.

Mayrand et al. [178], as well as Lague et al. [158], document the cloning phenomenon for the purpose of evaluating the quality of software systems; Lague et al. [158] have also evaluated the benefits in terms of maintenance of the detection of cloned methods.

16 Applications and Related Research for Clone Detection

In addition to the immediate applications of clone detection techniques to clone refactoring, avoidance and management (c.f., Section 13), there are several other domains in which clone detection techniques seem helpful. There are also some other areas related to clone detection from which clone detection techniques themselves can get benefited. In this Section, we provide a list of applications and related works of clone detection research. Some applications of clone detection are also pointed out in Section 4.

16.1 Plagiarism Detection

One of the closely related areas of clone detection is the plagiarism detection [203, 192, 83, 174, 111, 172, 33, 53]. In clone detection existing code is copied and then reused with or without modifications or adaptations for various reasons (c.f., Section 2). 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. However, such normalization may produce lots of false positives. Clone detection tools

such as token-based *CCFinder* [122] and metrics-based CLAN [180] 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 [44] evaluate three clone detection tools and two plagiarism detection tools, *JPlag* [192] and Moss [5]. 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 even 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 [169].

16.2 Origin Analysis, Merging and Software Evolution

Clone detection techniques can be applied to origin analysis, merging and software evolution analysis. All of them are based on a comparison analysis between two software variants (merging) or between two different versions of a system (origin analysis and evolution analysis). In this Section, we provide a brief introduction of how each of them are related to clone detection research.

16.2.1 Origin Analysis

A closely related area of clone detection is the origin analysis [209] in which two versions of a system are analyzed for deriving a model of where, how and why structural changes have occurred. Clone detection techniques may help origin analysis research [225, 94] and in the same time origin analysis techniques may assist clone detection research. As of plagiarism detection, the critical difference between the scope of detection approaches makes them infeasible to assist one another directly.

16.2.2 Merging

Another related research area to clone detection is the problem of merging [109] two existing similar systems for producing 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 [94]. Again, for establishing the relation between the pair of systems, clones from both the systems are compared and analyzed. Again, the comparison is only between 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 be benefited from other by sharing their comparison algorithms and analysis approaches.

16.2.3 Software Evolution

A very similar problem to origin analysis and merging is the software evolution analysis. As of origin analysis, two or more different versions of a software systems are mapped for finding a relation between them with a view to observe their evolution behavior. By detecting clones from each of the versions and then mapping the similar clone groups such a relation can be established [95, 209, 96, 199, 198].

16.3 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 other version of that program. For such mapping, a matching between the program elements of the two versions should be established. Clone detection techniques can be used for establishing such matching relation, and in the same time, other matching techniques [138] can be used for clone detection research.

16.4 Bug Detection

There is also a close relation between clone detection and software bug detection. Especially, copy-pasted software bugs can be successfully detected by clone detection tools [169, 115, 110]. However, it is not yet clear how bug detection techniques can help clone detection research.

16.5 Aspect Mining

Clone detection can also be applied to aspect mining research [132, 224] and vice versa. On a study by Bruntink et al. [41] clone detection techniques are evaluated for finding cross-cutting concerns [133]. Their analysis shows that even though there is a fundamental difference between clone detection and aspect mining, one can get benefited from the other. Classical clone detection techniques are designed in finding similar code fragments based on the program texts. On the other hand, a cross-cutting concern is scattered (or tangled) in different places and their implementation might not be textually similar. Rather, cross-cutting concerns tend to preserve semantic similarity between the scattered or tanged 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.

16.6 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 [118] visualizes the redundant substrings to ease the task of comprehending large legacy systems. Program

comprehension techniques, such as search-based techniques [102, 189, 204] or concept analysis [191] may greatly help clone detection research. However, empirical evidence on the effectiveness of such approaches to clone detection is required.

16.7 Code compacting

Clone detection techniques can be used for compact device (e.g., mobile devices) by reducing the source code size [49, 61].

16.8 Malicious software Detection

The possibility of detecting malicious software with clone detection techniques has also been investigated. By comparing one malicious software family to another, it is possible to find the evidence where parts of one software system match parts of another [217]. Detecting self-mutating malware (a particular form of code obfuscation) with clone detection techniques has also been attempted by Bruschi et al. [42]. However, it is not yet clear how malicious detection techniques can be used in finding clones.

16.9 Copyright infringement

The problem of detecting source code copyright infringement is viewed as a code similarity measuring problem between software systems. Clone detection tools can, therefore, be applied or can easily be adapted in detecting copyright infringement [160, 18, 122, 99].

16.10 Product Line Analysis

Clone detection techniques may be used in the area of software product line analysis (PLA) [48] by reengineering existing systems. Software product line (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 technique can be used in obtaining a SPL from existing systems [139] or managing a product line [32]. Koleilat and Shaft [139] 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. 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, 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 [32] use clone detection technique for enhancing a product line development. Clones indicate the presence of a useful problem domain concept, and in the same time provide an example implementation. Differences between the cloned copies identify parameters or point 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.

17 Open Problems in Clone Detection Research

There are several open problems concerning the various issues discussed in this paper. Many of the open problems have already been raised in the Dagstuhl seminar 2006 by Koschke [154]. In this section, we point out some of the open problems for future research:

17.1 List of Open Questions and Current Solvable Status

A list of 57 open questions on clone detection research was raised in the brainstorming session of the Second International Workshop on Detection of Software Clones (IWDSC'03) in November 2003 [214]. For each of the questions two decisions were made, 1. whether the problem concerning the question had been already solved, partially solved or unsolved until November 2003, and 2. a classification of this problem to clone detection research. In this section, we list the same 57 questions with an additional information of current solvable status for each of the questions. In Table 22, the 1st and 2nd columns show the index and the open question respectively, the 3nd column shows the classification of that question, the 4rd column indicates the then (November 2003) solvable status of that question and the 5th column shows current (August 2007) solvable status. While the 4rd column shows the solvable status based on the opinions from a group of experts present in the workshop, the solvable status at 5th column is exclusively based on our opinions. However, in support of our opinions we provide corresponding citations where applicable. From the table, we see that although several studies have been undertaken over the last 45 months (November, 2003 to August, 2007), most of the open problems are still unsolved or partially solved which definitely indicates that further studies are required in clone detection research.

17.2 Types and Taxonomies of Clones

Although different types and taxonomies of clones are provided in Section 7 and Section 8 respectively, there are several unclear issues to be taken care of. In the following, we point out some of these:

Formal definition of clones: To date there is no crisp definition of "clone". All the available definitions are either vague to a certain extent or incomplete (c.f., Section 7.1). A formal clone definition should be established overcoming the current limitations and associated vagueness. The definition should take into account different types of similarities such as representation similarity (textual, syntactic and structural), semantic or behavioral similarity, execution similarity, metrics similarity and feature-based similarity. While it is easy to define similarity for some types, it is still an open problem to define and measure semantic similarity.

Clone length by tokens or lines: It is still not clear what could be the minimum clone length and with which unit of measurement. Some people measure clone length by

Table 22: List of open questions and current solvable status (adapted from [214])

#	Issues (Asked as questions about knowledge)	Classification	Then Solved?	Now Solved?	Citations
1	Can we perform basic clone detection on procedural systems?	detection	yes	yes	[178]
2	Can we find clones in huge systems?	detection	no	yes	[167]
3	Can we detect clones well in generated xml/Hhml?	detection	no	no	
4	Do we have IDE-specific clone detection tools?	detection	maybe	partly	[206]
5	Can clones be traced through system evolution?	detection	no	partly	[71]
6	Can we detect higher-level clones well?	detection	no	maybe	[25, 29, 27]
7	Should we be detecting clones on abstracting of the source code?	detection	no	partly	[87]
8	Can we clones in abstract levels while no clones in the implementation?	detection / definition	yes	yes	-
9	How can we identify clone requirements from the code?	detection / definition	no	partly	[128, 129]
10	Do we understand clones for other programming paradigms?	detection / definition	no	partly	[56, 164, 46]
11	What kinds of clones are in functional languages?	detection / definition	no	no	X
12	What is the definition of a "clone"?	definition	yes	maybe	[126, 127]
13	For what levels of source code do clear definitions for 'clone' exist?	definition	no	no	
14	Are clones defined objectively, subjectively, or inter-subjectively?	definition	no	no	
15	Are clone relations symmetric or not?	definition	no	no	X
16	Does the size of the system affect the rate of clone occurrence?	definition	no	no	X
17	What are clone types?	definition	no	partly	[36, 127]
18	Do we have standard difference measures between similar code fragments?	detection taxonomic	partly	partly	[127, 22, 178]
19	Do we have a framework to classify/evaluate clones?	taxonomic	no, partly	no, partly	[36, 10]
20	Do we understand "other" level clones?	taxonomic	no	no	X
21	Can we describe the differences between the similar code fragments?	taxonomic	no	no	X
22	Do we understand context-sensitive clones?	taxonomic	no	no	X
23	Which kinds of clones are bad? Which are OK?	ranking	no	partly	[136, 129, 128]
24	Do we understand the costs/benefits of CDs or removal?	ranking	no	no, partly	[79]
25	Do we know when detected clones are too complex to be beneficial?	ranking	no	no	X
26	Should we care about finding or removing generated clones?	ranking	no	no	X
27	Can we deal with generated code?	ranking	no	no	X
28	Are "accidental" clones widespread and important to manage?	ranking	no	no, partly	[6]
29	Can we measure clone rates in systems?	ranking	no	no	X
30	Can we construct clone derivation histories?	ranking	no	no	X
31	What are the purposes of CD?	purpose	maybe	maybe	Sec. 2 & 4
32	What categories of goals are to be achieved by CD?	purpose	no	no	X
33	Do we know purposes (contexts) of CD in industry?	purpose	no	no	X
34	Which domain is CD useful for?	purpose	no	partly	X
35	Should we find all clones in huge systems?	purpose	no	no	X
36	How should we prevent clones?	purpose	no	no	X
37	Can copyright violations be found by current CD software?	purpose	yes	yes	[122]
38	What do we do with clones?	clone-based actions	no	no	X
39	Must clones be removed?	clone-based actions	no	no, partly	[136, 129, 171]
40	Which clones should be refactored, kept, or encouraged?	clone-based actions	no	no, partly	[136, 129]
41	Can we automate refactoring well?	clone-based actions	no	no	X
42	How do we evaluate CD techniques?	evaluation	no	no, partly	[36, 10]
43	Which CD techniques are better?	evaluation	no	no, partly	[36, 10]
44	Do we have multiple-project CD benchmarks?	evaluation	partly	partly	[36, 10]
45	Are sampled clone rates generalizable?	evaluation	no	no	X
46	Does XP affect CD?	process	no	no, partly	X
47	Does XP produce fewer clones?	process	no	no, partly	[188]
48	How can CDs be used to recommend human knowledge sources?	process	no	no	X
49	Can case knowledge (e.g., about programming solutions) be indexed by clones?	process	no	no	X
50	Do we know the market value of clone detection?	economics	no	no	X
51	Will clone detection matter in 15 years? 30 years? 100 years	economics	yes	yes	-
52	What is the end-user value of CD tools? Can we quantify them	economics	no	no	X
53	Do we have adequate economic models of clone costs?	economics	no	no	X
54	Do we know the reasons for clones?	cause	no	no, partly	Sec. 2
55	Do we understand the psychology of clones?	cause	no	no	X
56	Can language support prevent clones?	programming language	no	no	X
57	What are the implications of clones for language design?	programming language	no	no	X

the number of tokens while others by number of lines or even with some detection-dependent units (such as number of AST/PDG nodes). Once the unit of measurement is chosen, it should be decided what could be the appropriate minimum threshold for clone length based on this unit.

Meta model of clone taxonomy: Several attempts have been made to overcome the associated vagueness of clone definitions by providing different types of clones in the form of a taxonomy. However, to date, it is not clear whether the available taxonomies are sufficient enough either for detecting clones or for software maintenance. A language-independent meta model of the clone taxonomy is crucial in overcoming the current limitations. Similar to the normal forms in databases, a theory of duplication will be very helpful in clone detection research. Moreover, each kind of similarity measure should be closely related to a specific purpose in software maintenance.

Clone types based on origins and/or risks: There are still no categorizations of clones based on origins or risks. It is also not yet known the statistical distribution of such types in larger systems. Empirical studies can be undertaken for finding such distribution of clone types.

Removal, avoidance and risks: For each of the clone types, the strategies and techniques of removal and avoidance, and the risks, costs and benefits of such removals should be studied.

Relevance ranking: Once the clones are identified, a relevance ranking should be established for removal or other maintenance activities.

17.3 Evaluation of Clone Detection Techniques

There are several clone detection techniques available today. There are also several studies that provide an empirical evaluation of several tools. In these empirical evaluations, several limitations of the current benchmarks are pointed out.

Established human oracle for refactoring: A human oracle for different removal strategies of the candidate clones should be established. Current oracles cannot provide unique decision (e.g., differences among different human raters for clone candidates) on clone candidates when clones ought to be removed [215].

Benchmark on gapped clones: Current benchmarks focus on contiguous lines or tokens for clones. However, it is agreed that most clones in large systems are non-contiguous lines/tokens(gapped clones), and such gapped clones might be good candidates for refactoring. Therefore, it is reasonable to establish a benchmark focusing on gapped clones.

Standard benchmark: Current benchmarks are based only on *clone pairs*. However, there are several clone detection tools that report clones as *clone classes*. The benchmark should be sound enough to evaluate detection techniques with respect to clone pairs or clone classes or even both. Current benchmarks provide only a *yes/no* decision on the candidate clones. Rather than the *yes/no* decision, a degree of confidence

and thus, a relevance ranking of the clone candidates (with respect to the context of use) would be helpful [216]. Once a benchmark is established, it should become the standard procedure of the community. Any new technique is subject to comparison with this benchmark prior to publication [154].

17.4 Better Clone Detection Techniques

Although there are several clone detection techniques available today, there is always a need to have a better one. The open issues concerning a new technique are:

Higher precision and recall: To date no clone detection technique is found sound enough in terms of precision, recall, robustness and scalability (c.f., Section 11). Most tools show complementary behaviors for precision and recall. Scalability and robustness are also challenging in almost all cases. Therefore, there is a crucial need to develop a new clone detection technique that can overcome the existing limitations.

Especial treatment for detecting *Type III* clones: No tool is reported to do well detecting *Type III* clones. Moreover, attempts in detecting semantic clones (*Type IV*) are very few or none. Both *Type III* and *Type IV* clones are important from a maintenance point of view. When developing a new technique especial treatments should be considered for detecting *Type III* clones. Again, a classification or taxonomy of such clones is crucial.

Semantic clone detection tool: For detecting *Type IV* clones, the necessity of a semantic clone detection technique is evident. Clone definition and detection by semantic similarity are undecidable problems in general. Nevertheless, attempts can be made to detect semantic clones by applying extensive and intelligent normalizations to the code. In such normalizations different variants of similar syntactic constructs are transformed to the same normal form and comparison is performed on the normalized code to find clones.

17.5 Empirical Studies in Clone Detection Research

More empirical studies with large scale industrial and open source software systems are required. There are several open issues that can be attempted by means of empirical studies. These are as follows:

Classification and statistics of *Type III* and *Type IV* clones: Deriving the classifications/taxonomies for *Type III* and *Type IV* clones is challenging. With multilingual large scale case studies (industrial and open source systems), it may be possible to derive such classifications along with their frequencies of occurrences. Such classification and statistics then can be used for deriving refactoring strategies, evaluating clone impacts on software quality and developing clone detection tools.

Statistics of intentional clones: In Section 2, several intentions behind cloning have been listed. It will be helpful to know the rate of clones produced by each intention. Cloning statistics based on intentions may help to develop a new technique focusing on the higher rate intentions.

Effect of normalizations on precision and recall: Little or no statistics are available concerning the effect of normalizations and/or transformations of code to precision and recall. Empirical studies are required to show the effect of normalizations on precision and recall.

Study of extensive normalizations in finding semantic clones: The issue of detecting semantic clones is a major concern to the research community [151]. Unfortunately, no proper way of detecting semantic clones is available yet. An alternative approach of detecting such clones could be to apply extensive and intelligent normalizations and transformations to source code. However, it is not clear how such normalizations and transformations can assist in detecting semantic clones. Large scale empirical studies are required to verify this approach.

Relation between extensive normalizations and plagiarism detection: Instead of simple copy and then reuse by adaptations in case of code cloning, copied code is intentionally disguised in case of plagiarism. Replacements of control statements with their semantically equivalent variants is one of the major disguising activities for camouflaging copied code [165]. It might be interesting to see how extensive code normalizations and transformations are beneficial to detecting camouflaged fragments. Although it is generally agreed that more normalizations can help finding plagiarized code, there is no large scale empirical study to validate such argument.

Strategies for refactoring *Type III* clones: While refactoring of *Type I* and *Type II* clones are possible with basic refactoring methods, it is still unclear of how to refactor *Type III* clones. Having a classification of such clones , refactoring strategies for each of types should be studied in a large code base.

Harmfulness of clones: There is little information available concerning the impacts of code clones on software quality. Recent studies show that clones may not be always harmful to software systems (c.f., Section 5). More empirical studies are required to find the relation between clones and different quality attributes (such as maintainability, *reliability*, *modifiability*, security etc.) of software systems. This relation then should be fine tuned for different types of clones to the different quality attributes.

Patterns of uninteresting clones: Most clone detection techniques produce a high number of clones that are uninteresting from a maintenance perspective. Such clones are normally filtered out in a post-processing step. However, it is not yet clear what patterns of clones are surely uninteresting in which context. It would be beneficial to find the patterns of such uninteresting clones and employ strategies of filtering out the code that follows such patters in pre-processing step. In this way, manual analysis of the produced clones will be reduced significantly, and consequently, we can have a clone detection with higher precision.

Visualization of Clones and Filtering Automation: Over the last decades huge research has been undertaken on information visualization in general and software visualization in particular. As yet many of these visualization techniques have not been explored for visualizing clones. Empirical studies can be conducted to see how visualization techniques of other domains can assist clone visualization. There is also a

debate which clone visualization approach is better for which purpose. More systematic empirical research is required to resolve this issue.

One of the objectives of the visualization tools is to automate the post-processing tasks. Considering the availability of the state of the art clone detection techniques, one can focus on automating the filtering process after obtaining the clone candidates from a tool. Again, such filtering needs a proper definition of code clone.

Studies of clone coverage: Although there are several studies that show that a significant amount of code is cloned code of a software system, a solid argument is still missing. This is because comparing the results of these studies is difficult and error-prone. Different studies use different clone detection tools where most tools use their own detection-dependent definitions of a clone. Many detection algorithms also take adjustable parameters. A large scale case study with systems of different languages can be conducted with a common clone definition.

Evolution analysis of clones: Although clone evolution analysis of large open source software has already been conducted there is still little or no study concerning the evolution of clones in large industrial software. The following issues [154] should be resolved with large empirical studies:

- How do clones evolve in industrial systems?
- What does their evolution tell about the development organization?
- What are reliable predictors?

Relation to other domains: Empirical studies are required to see how clone detection techniques can be benefited from the other areas and at the same time how other areas can be assisted by clone detection research.

18 Conclusion

Clone detection is an active research area and the literature is overwhelmed with plenty of work in detecting and removing clones from software systems. Research is also done in maintaining clones in software systems in its evolution life cycle. In this paper, a comprehensive survey on the area of software clone detection research is made putting emphasis on the types of clones used, their detection mechanism and empirical evaluation of the corresponding tools. Several open issues are also pointed out for further research. The report by Koschke [154] is also a good source for an overview of clone detection research.

The results of this study may serve as a roadmap to potential users of clone detection techniques, to help them in selecting the right tool or technique for their interests. We hope it may also assist in identifying remaining open research questions, possible avenues for future research, and interesting combinations of existing techniques.

19 Acknowledgements

The authors would like to thank Tom Dean, Jenny Zou and Mohammad Zulkernine for their help in providing useful comments and suggestions on an earlier draft. Thanks are also due to Giuliano Antoniol for a fruitful discussion during his visit at Queen's. The authors also acknowledge some of the tool owners for providing useful feedback on our queries. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Eytan Adar. GUESS: a language and interface for graph exploration. In *Proceedings of the 2006 Conference on Human Factors in Computing Systems (CHI'06)*, pp. 791-800, Montréal, Québec, Canada, April 2006. (PDF)

[2] Eytan Adar and Miryung Kim. SoftGUESS: Visualization and Exploration of Code Clones in Context. In *the proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, Tool Demo, pp.762-766, Minneapolis, MN, USA, May 2007 .

[3] R. Agrawal and R. Srikant. Mining Sequencial Patterns. In *Proceddings of the 11th Infernation Conference of Data Engineering (ICDE'95)*, pp. 3-14, Taipei, Taiwan, March 1995.

[4] Alfred Aho, Ravi Sethi and Jeffrey Ullman. *Compilers, Principles, Techniques and Tools*. Addition-Wesley, 1986.

[5] A. Aiken. A system for detecting software plagiarism (moss homepage). URL `http://www.cs.berkeley.edu/aiken/moss.html`. 2002.

[6] Raihan Al-Ekram, Cory Kapser, Michael Godfrey. Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems. *International Symposium on Empirical Software Engineering (ISESE'05)*, pp. 376-385, Noosa Heads, Australia, November 2005.

[7] Giuliano Antoniol, Gerardo Casazza, Massimiliano Di Penta, Ettore Merlo. Modeling Clones Evolution through Time Series. In *Proceedings of the 17th IEEE International Conference on Software Maintenance (ICSM'01)*, pp. 273-280, Florence, Italy, November 2001. (PDF)

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

[9] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How Clones are Maintained: An Empirical Study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pp. 81-90, Amsterdam, the Netherlands, March 2007.

[10] Brenda S. Baker. Finding Clones with Dup: Analysis of an Experiment. *IEEE Transactions on Software Engineering*, Vol. 33(9): 608-621, September 2007.

[11] Brenda S. Baker and Uni Manber. Deducing similarities in Java sources from bytecodes. In *Proceedings of the USENIX Annual Technical Conference*, pp. 179-190, New Orleans, Louisiana, USA, June 1998.

[12] Brenda Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26 (5):13431362, October 1997.

[13] Brenda S. Baker. A theory of parameterized pattern matching: Algorithms and applications (extended abstract). In *Proceedings of the 25th ACM Symposium on Theory of Computing (STOC'93))*, pp. 7180, San Diego, California, USA, May 1993.

[14] Brenda S. Baker. A Program for Identifying Duplicated Code. In *Proceedings of Computing Science and Statistics: 24th Symposium on the Interface*, Vol. 24:4957, March 1992.

[15] Brenda S. Baker. Parameterized diff. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, pp. 854-855, Baltimore, Maryland, USA, January 1999.

[16] Brenda S. Baker. Parameterized Pattern Matching: Algorithms and Applications. In *Journal Computer System Science*, Vol. 52(1):2842, February 1996. URL http://citeseer.nj.nec.com/baker94parameterized.html.

[17] Brenda S. Baker. On Finding Duplication in Strings and Software. *Journal of Algorithms*, 1993.

[18] Brenda Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95)*, pp. 86-95, Toronto, Ontario, Canada, July 1995.

[19] Tibor Bakota, Rudolf Ference and Tibor Gyimothy. Clone Smells in Software Evolution. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM'07)*, 10pp., Paris, France, October 2007.

[20] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, Kostas Kontogiannis. Advanced Clone-analysis to Support Object-oriented System Refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, pp. 98-107, Brisbane, Qld., Australia, November 2000.

[21] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, Kostas Kontogiannis. Partial Redesign of Java Software Systems Based on Clone Analysis. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99)*, pp. 326-336, Atlanta, GA, USA, October 1999.

[22] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, Kostas Kontogiannis. Measuring Clone Based Reengineering Opportunities. In *Proceedings of the 6th International Software Metrics Symposium (METRICS'99)*, pp. 292-303, Boca Raton, Florida, USA, November 1999.

[23] Mihai Balint, Tudor Girba, Radu Marinescu. How Developers Copy. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 56- 68, Athens, Greece, June 2006.

[24] Hamid Basit, Simon Pugliesi, William Smyth, Andrei Turpin, and Stan Jarzabek. Efficient Token Based Clone Detection with Flexible Tokenization. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, pp. 513-515, Dubrovnik, Croatia, September 2007.

[25] Hamid Basit, Damith Rajapakse, Stan Jarzabek. An Empirical Study on Limits of Clone Unification Using Generics. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, pp. 109-114, Taipei, Taiwan, Republic of China, July 2005.

[26] Hamid Basit, Stan Jarzabek. Detecting Higher-level Similarity Patterns in Programs. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE'05)*, pp. 156-165, Lisbon, Portugal, September 2005.

[27] Hamid Basit, Damith Rajapakse, Stan Jarzabek. Beyond Templates: a Study of Clones in the STL and Some General Implications. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pp. 15-21, St. Louis, Missouri, USA, May 2005.

[28] Hamid Basit, Damith Rajapakse, Stan Jarzabek. An Investigation of Cloning in Web Applications. In *Proceedings of the Special Interest Tracks and Posters of the 14th International Conference on World Wide Web (WWW'05)*, pp. 924-925, Chiba, Japan, May 2005.

[29] Hamid Abdul Basit, Damith C. Rajapakse and Stan Jarzabek. Structural Clones Higher Level Similarity Patterns in Programs. Department of Computer Science School of Computing, National University of Singapore, 2007 (not published yet).

[30] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering (ICSE04)*, pp. 625-634, Scotland, UK, May 2004.

[31] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the 14th International Conference on Software Maintenance (ICSM'98)*, pp. 368-377, Bethesda, Maryland, November 1998.

[32] Ira Baxter, and Dale Churchett. Using Clone Detection to Manage a Product Line. Semantic Designs, Inc (Not published).

[33] Boumediene Belkhouche, Anastasia Nix, Johnette Hassell. Plagiarism detection in software designs. *ACM Southeast 42nd Regional Conference*, pp. 207-211, Huntsville, Alabama, USA, April 2004.

[34] Stefan Bellon. Detection of Software Clones Tool Comparison Experiment. *Tool Comparison Experiment presented at the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, Montreal, Canada, October 2002.

[35] Stefan Bellon. Vergleich von techniken zur erkennung duplizierten quellcodes. Diploma Thesis, No. 1998, University of Stuttgart (Germany), Institute for Software Technology, September 2002.

[36] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. In *IEEE Transactions on Software Engineering*, Vol. 33(9): 577-591, September 2007.

[37] C. Boldyreff, and R. Kewish. Reverse Engineering to Achieve Maintainable WWW Sites. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pp. 249-257, Stuttgart, Germany, October 2001.

[38] Udo Borkowski. C^4D oder wie ich lernte, mit code clones zu leben. Softwaretechnik-Trends, 24(2), 2004. In *Proceedings of the 6th Workshop on Software Reengineering (WSR'04)*, Bad Honnef, Germany, 2004.

[39] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, Markus Neteler. A novel approach to optimize clone refactoring activity. In *Proceedings of the 8th Annual Genetic and Evolutionary Computation Conference (GECCO'06)*, pp. 1885-1892, Seattle, WA, USA, July 2006.

[40] Magiel Bruntink. Aspect Mining using Clone Class Metrics. In *Proceedings of the 1st Workshop on Aspect Reverse Engineering*, 2004.

[41] Magiel Bruntink, Arie van Deursen, Remco van Engelen, Tom Tourwe. On the Use of Clone Detection for Identifying Crosscutting Concern Code. *Transactions on Software Engineering*, Volume 31(10):804-818, October 2005.

[42] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings od the International Symposium of Secure Software Engineering (ISSSE'06)*, Arlington, VA, USA,March 2006.

[43] Elizabeth Burd and Malcolm Munro. Investigating the maintenance implications of the replication of code. In *Proceedings of the 13th International Conference on Software Maintenance (ICSM'97)*, Bari, Italy, September 1997.

[44] Elizabeth Burd, John Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, pp. 36 - 43, Montreal, Canada, October 2002.

[45] E. Buss, R. De Mori, M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. Mller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. Tilley, J. Troster, K. Wong. Investigating reverse engineering technologies for the cas program understanding project. *IBM Systems Journal*, 33(3):477-500, 1994.

[46] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. Function Clone Detection in Web Applications: A Semiautomated Approach. *Journal of Web Engineering*, Vol. 3(1): 003-021, 2004.

[47] Gerardo Casazza, Giuliano Antoniol, Umberto Villano, Ettore Merlo, Massimiliano Di Penta. Identifying Clones in the Linux Kernel. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'01)*, pp, 90-97, Florence, Italy, November 2001.

[48] Gary Chastek, Patrick Donohoe, Kyo Chul Kang, and Steffen Thiel. Product Line Analysis: A Practical Introduction (Technical Report No. CMU/SEI-2001-TR-001, ADA396137). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001.

[49] W-K. Chen, B. Li, and R. Gupta. Code Compaction of Matching Single-Entry Multiple-Exit Regions. In *Proceedings of the 10th Annual International Static Analysis Symposium (SAS'03)*, pp. 401-417, San Diego, CA, USA, June 2003.

[50] A. Chiu, and D. Hirtle. Beyond Clone Detection. *CS846: Software Evolution project report*. Cheriton School of Computer Science, University of Waterloo, April 2007.

[51] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM symposium on Operating systems principles (SOSP'01)*, pp. 7388, Banff, Alberta, Canada, October 2001.

[52] K. W. Church and J. I. Helfman. Dotplot: A program for exploring self-similarity in millions of lines for text and code. *Journal of American Statistical Association, Institute for Mathematical Statistics and Interface Foundations of North America*, 2(2):153174, June 1993.

[53] Paul Clough. Old and new challenges in automatic plagiarism detection. *National Plagiarism Advisory Service*, 2003; `http://ir.shef.ac.uk/cloughie/index.html`

[54] M. E. Conway. How do committees invent? *Datamation*, 14(4):2831, Apr. 1968.

[55] K. Cooper and N. McIntosh. Enhanced code compression for embedded risc processors. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pp. 139149, Atlanta, Georgia, USA, May 1999.

[56] James Cordy, Thomas Dean, Nikita Synytskyy. Practical Language-Independent Detection of Near-Miss. In *Proceedings of the 14th IBM Centre for Advanced Studies Conference (CASCON'04)*, pp. 1 - 12, Toronto, Ontario, Canada, October 2004.

[57] J.R. Cordy. Comprehending reality: Practical challenges to software maintenance automation. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pp. 196206, Portland, Oregon, USA, May 2003.

[58] Michel Dagenais, Ettore Merlo, Bruno Laguë, and Daniel Proulx. Clones occurrence in large object oriented software packages. In *Proceedings of the 8th IBM Centre for*

Advanced Studies Conference (CASCON'98), pp. 192200, Toronto, Ontario, Canada, October 1998.

[59] M. Datar, N. Immorlica, P. Indyk and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th annual symposium on Computational geometry (SoGG'04)*, pp. 253-262, Brooklyn, New York, USA, June 2004.

[60] Neil Davey, Paul Barson, Simon Field, Ray J Frank. The Development of a Software Clone Detector. *International Journal of Applied Software Technology*, Vol. 1(3/4):219-236, 1995

[61] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS'00)*, Vol. 22(2):378-415, March 2000.

[62] S. Demeyer, S. Ducasse, and O. Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2002.

[63] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Sifting out the mud: Low level C++ code reuse. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pp. 275-291, Seattle, Washington, USA, November 2002.

[64] Andrea De Lucia, Rita Francese, Giuseppe Scanniello, and Genoveffa Tortora. Understanding Cloned Patterns in Web Applications. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*, pp. 333-336, St. Louis, MO, USA, May 2005.

[65] Andrea De Lucia, Rita Francese, Giuseppe Scanniello and Genoveffa Tortora. Reengineering Web Applications Based on Cloned Pattern Analysis. In *Proceedings of 12th International Workshop on Program Comprehension (IWPC'04)*, pp. 132-141 Bari, Italy, June 2004.

[66] G.A. Di Lucca, M. Di Penta, and A.R. Fasolino and P. Granato. Clone Analysis in the Web Era: an Approach to Identify Cloned Web Pages. In *Proceedings of the 7th IEEE Workshop on Empirical Studies of Software Maintenance (WESS'99)*, pp. 107-113, Florence, Italy, November 2001.

[67] G.A. Di Lucca, M. Di Penta, and A.R. Fasolino. An approach to identify duplicated web pages. In *Proceedings of the 26th International Computer Software and Applications Conference (COMPSAC'02)*, pp. 481486, Oxford, England, August 2002.

[68] G. A. Di Lucca, A. R. Fasolino, P. Tramontana, U. De Carlini. Identifying Reusable Components in Web Applications. In *Proceedings of the IASTED International Conference on Software Engineering*, Innsbruck, Austria, February 2004.

[69] Giuseppe Antonio Di Lucca, Damiano Distante, and Mario Luca Bernardi. Recovering Conceptual Models from Web Applications. In *Proceedings of the 24th Annual Conference on Design of communication (SIGDOC'06)*, pp. 113-120, Myrtle Beach, SC, USA, October 2006.

[70] A.van Deursen, T. Kuipers. Building Documentation Generators. In *Proceedings of International Conference on Software Maintenance (ICSM'99)*, Oxford, England, UK, August 1999.

[71] Ekwa Duala-Ekoko, Martin Robillard. Tracking Code Clones in Evolving Software. In *Proceedings of the International Conference on Software Engineering (ICSE'07)*, pp. 158-167, Minneapolis, Minnesota, USA, May 2007.

[72] Stéphane Ducasse, Oscar Nierstrasz, Matthias Rieger. On the Effectiveness of Clone Detection by String Matching. *International Journal on Software Maintenance and Evolution: Research and Practice*, Volume 18(1): 37-58, January 2006.

[73] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. Lightweight detection of duplicated codea language-independent approach. Technical report, University of Bern, Institute of Computer Science and Applied Mathematics, Bern, Switzerland, February 2004.

[74] Stéphane Ducasse, Matthias Rieger, Serge Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the 15th International Conference on Software Maintenance (ICSM'99)*, pp. 109-118, Oxford, England, September 1999.

[75] Susan T. Dumais. Latent Semantic Indexing (LSI) and TREC-2. In *Proceedings of the 2nd Text Retrieval Conference (TREC'94)*, pp. 105-115, Gaithersburg, Maryland, March 1994.

[76] M. Ernst, G.J. Badros, and D. Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28(12): 1146-1170, December 2002.

[77] Williams Evans, and Christopher Fraser. Clone Detection via Structural Abstraction. In Proceedings of the 14th Conference on Reverse Engineering (WCRE'07), Vancouver, BC, Canada, October 2007(to appear, available as Technical Report since August 2005).

[78] Richard Fanta, Vclav Rajlich. Removing Clones from the Code. *Journal of Software Maintenance: Research and Practice*, Volume 11(4):223-243, August 1999.

[79] Karl-Filip Faxén. The Costs and Benefits of Cloning in a Lazy Functional Language. *Trends in Functional Programming*, Volume 2:1-12, 2001 (Selected papers from the 2nd Scottish Functional Programming Workshop (SFP00).

[80] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319349, 1987.

[81] F. Fioravanti, G. Migliarase, and P. Nesi. Reengineering Analysis of Object-Oriented Systems via Duplication Analysis. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pp. 577-590, Toronto, Ontario, Canada, May 2001.

[82] G. Fischer, J. Wolff v. Gudenberg. Simplifying Source Code Analysis by an XML Representation. *Softwaretechnik Trends*, vol. 23(2), 2003.

[83] G. Flammia. On the internet, software should be milked, not brewed. *IEEE Expert*, 11(6):8788, December 1996.

[84] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[85] Christopher W. Fraser, Eugene W. Myers, and Alan L. Wendt. Analyzing and compressing assembly code. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, pp. 117-121, June 1984.

[86] Keith Gallagher, Lucas Layman. Are Decomposition Slices Clones? In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pp. 251-256 Portland, Oregon, USA, May 2003.

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

[88] Simon Giesecke. *Clonebased Reengineering für Java auf der EclipsePlattform*. Masters thesis, Carl von Ossietzky Universität Oldenburg, Germany, September 2003.

[89] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE)*, pp. 113122. Lisbon, Portugal, September 2005.

[90] David Gitchell and Nicholas Tran. Sim: a utility for detecting similarity in computer programs. *ACM SIGCSE Bulletin*, 31(1): 266-270, March 1999.

[91] Reto Geiger. *Evolution Impact of Code Clones*. Diploma Thesis, University of Zurich, October 2005.

[92] Reto Geiger, Beat Fluri, Harald C. Gall and Martin Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering (FASE'06)*, pp. 411-425, Vienna, Austria, March 2006

[93] M.W. Godfrey, D. Svetinovic, and Q. Tu. Evolution, growth, and cloning in Linux: A case study. In *CASCON workshop on Detecting duplicated and near duplicated structures in large software systems: Methods and applications*, October 2000.

[94] M. Godfry, and L. Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. In *IEEE Transactions on Software Engineering* 31(2): 166-181, February 2005).

[95] M. Godfrey and Q. Tu. Growth, evolution and structural change in open source software. In *Proceedings of the 4th International Workshop on Principles of Software Evolution* , pp. 103-106, Vienna, Austria, September 2001.

[96] M. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the 16th International Conference on Software Maintenance*, pp. 131- San Jose, California, USA, October 2000.

[97] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey P. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineerings*, 26(7): 653-661, 2000.

[98] Kevin Greenan. *Method-Level Code Clone Detection on Transformed Abstract Syntax Trees using Sequence Matching Algorithms*. Student Report, University of California - Santa Cruz, Winter 2005,

[99] Sam Grier. A tool that detects plagiarism in pascal programs. In *Proceedings of the 12th SIGCSE Technical Symposium on Computer Science Education* ,pp. 15-20, St. Louis, Missouri, USA, 1981.

[100] Penny Grubb, and Armstrong A Takang. *Software Maintenance Concepts and Practice*. 2nd edn. World Scientific (2003).

[101] Jiawei Han, and Micheline Kamber. Data Mining-Concepts and Techniques. Korgan Kaufmann, 2001.

[102] Mark Harman. Search Based Software Engineering for Program Comprehension. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*, pp. 3-13, Banff, Canada, June 2007.

[103] Jonathan I. Helfman. Dotplot Patterns: a Literal Look at Pattern Languages. In *Theory and Practice of Object Systems (TAPOS'95)*, pp. 3141, 1995.

[104] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. Code Clone Analysis Methods for Efficient Software Maintenance. Graduate School of Information Science and Technology, Osaka University, 2006 (published as a report (PhD Thesis?) and paper version unpublished).

[105] Yoshiki Higo, Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. On Software Maintenance Process Improvement Based on Code Clone Analysis. In *Proceedings of the 4th International Conference on Product Focused Software Process Improvement (PROFES'02)*, pp. 185-197, Rovaniemi, Finland, November 2002.

[106] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. ARIES: Refactoring Support Environment based on Code Clone Analysis. In *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications*, Cambridge, MA, USA, November 2004.

[107] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. Refactoring Support Based on Code Clone Analysis. In *Proceedings of the 5th International Conference on Product Focused Software Process Improvement(PROFES'04)*, pp. 220-233, Kansai Science City, Japan, April 2004.

[108] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications ACM*, 18(6):341-343, June 1975.

[109] James J. Hunt, and Walter F. Tichy.Extensible Language Aware Merging. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pp. 511-520, Montréal, Canada, October 2002.

[110] P. Jablonski, and D. Hou. CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE. In *Proceedings of Eclipse Technology Exchange Workshop at OOPSLA 2007(ETX'07)*, 5pp., Montréal, Québec, Canada October 2007.

[111] Hugo T. Jankowitz. Detecting Plagiarism in Student PASCAL Programs. *Computer Journal*, 31(1):18, February 1988.

[112] Stan Jarzabek, and Shubiao Li. Unifying clones with a generative programming technique: a case study. *Journal of Software Maintenance and Evolution: Research and Practice, John Wiley & Sons*, Volume 18(4):267-292,July/August 2006 (Extended version of ESEC-FSE03 paper)

[113] Zhenming Jiang, and Ahmed Hassan. A Framework for Studying Clones in Large Software Systems. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07)*, Paris, France, October 2007.

[114] Zhen Ming Jiang, Ahmed E. Hassan, and Richard C. Holt. Visualizing Clone Cohesion and Coupling. In *Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pp. 467-476, Bangalore, India, December 2006.

[115] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-Based Detection of Clone-Related Bugs. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, 10pp., Dubrovnik, Croatia, September 2007 (to appear).

[116] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pp. 96-105, Minnesota, USA, May 2007.

[117] J Howard Johnson. Navigating the textual redundancy Web in legacy source. In *Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'96)*, pp. 7-16, Toronto, Canada, October 1996.

[118] J Howard Johnson. Identifying Redundancy in Source Code Using Fingerprints. In *Proceeding of the 1993 Conference of the Centre for Advanced Studies Conference (CASCON'93)*, pp. 171-183, Toronto, Canada, October 1993.

[119] J. Howard Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative research (CASCON'94)*, pp. 171-183, Toronto, Canada, 1994.

[120] John Johnson. Substring Matching for Clone Detection and Change Tracking. In *Proceedings of the 10th International Conference on Software Maintenance*, pp. 120-126, Victoria, British Columbia, Canada, September 1994.

[121] Nicolas Juillerat, and Béat Hirsbrunner. An Algorithm for Detecting and Removing Clones in Java Code. In *Proceedings of the 3rd Workshop on Software Evolution through Transformations: Embracing the Change (SeTra'06)*, pp. 63-74, Rio Grande do Norte, Brazil, September 2006.

[122] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *Transactions on Software Engineering*, Vol. 28(7): 654- 670, July 2002.

[123] Cory J. Kapser and Michael W. Godfrey. Supporting the Analysis of Clones in Software Systems: A Case Study. *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 18(2): 61-82, March 2006.

[124] Cory Kapser and Michael Godfrey. A Taxonomy of Clones in Source Code: The Re-Engineers Most Wanted List. In *Proceedings of the 2nd International Workshop on Detection of Software Clones (IWDSC'03)*, 2pp., Victoria, BC, November 2003.

[125] Cory Kapser, and Michael Godfrey. Toward a taxonomy of clones in source code: A case study. In *Proceedings of the Conference on Evolution of Large Scale Industrial Software Architectures (ELISA '03)*, pp. 67-78, Amsterdam, The Netherlands, September 2003.

[126] Cory Kapser, and Michael Godfrey . Aiding Comprehension of Cloning Throu gh Categorization. In *Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE'04)*, pp. 85-94, Kyoto, Japan, September 2004.

[127] Cory Kapser, Michael Godfrey. Improved Tool Support for the Investigation of Duplication in Software. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, pp. 305-314, Budapest, Hungary, September 2005.

[128] Cory Kapser and Michael W. Godfrey. "clones considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pp. 19-28, Benevento, Italy, October 2006.

[129] Cory Kapser and Michael W. Godfrey. "Cloning Considered Harmful" Considered Harmful: A case study of the positive and negative effects. Empirical Software Engineering (invited for publication), 2007.

[130] R. M. Karp. Combinatorics, complexity, and randomness. *Communications of the ACM*, 29(2):98109, February 1986.

[131] R. M. Karp and M. 0. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal Research and Development*, 31(2):249260, March 1987.

[132] Andy Kellens, Kim Mens, and Paolo Tonella. A Survey of Automated Code-Level Aspect Mining Techniques. In *Transactions on Aspect Oriented Software Development*, Vol. 4 (LNCS 4640), pp. 145-164, 2007.

[133] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, Springer-Verlag LNCS 1241, June 1997.

[134] Holger M. Kienle, Hausi A. Müller and Anke Weber. In the Web of Generated "Clones". In *Proceedings of 2nd International Workshop on Detection of Software Clones (IWDSC'03)*, 22pp., Victoria, British Columbia, Canada, November 2003.

[135] Miryung Kim, Lawrence Bergman, Tessa Lau, David Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of 3rd International ACM-IEEE Symposium on Empirical Software Engineering (ISESE'04)*, pp. 83- 92, Redondo Beach, CA, USA, August 2004.

[136] Miryung Kim, Gail Murphy. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/SIGSOFT FSE 2005 '05)*, pp. 187-196, Lisbon, Portugal, September 2005.

[137] Miryung Kim, David Notkin. Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones. In *Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR'05)*, pp. 1-5, Saint Louis, Missouri, USA, May 2005.

[138] Miryung Kim and David Notkin. Program Element Matching for Multi-Version Program Analyzes. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR'06)*, pp. 58-64, Shanghai, China, May 2006.

[139] Walid Koleilat, and Niv Shaft. Extracting Executable Skeletons. *CS846: Software Evolution project report*. Cheriton School of Computer Science, University of Waterloo, April 2007.

[140] Raghavan Komondoor and Susan Horwitz. Tool demonstration: Finding duplicated code using program dependences. In *Proceedings of the European Symposium on Programming (ESOP'01)*, Vol. LNCS 2028, pp. 383386, Genova, Italy, April 2001.

[141] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS'01)*, Vol. LNCS 2126, pp. 40-56, Paris, France, July 2001.

[142] Raghavan Komondoor and Susan Horwitz. Effective, Automatic Procedure Extraction. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pp. 33-42, Portland, Oregon, USA, May 2003.

[143] Raghavan Komondoor and Susan Horwitz. Semantics-Preserving Procedure Extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL'00)*, pp. 155-169, Boston, MA, USA, January 2000.

[144] Raghavan Komondoor. *Automated Duplicated-Code Detection and Procedure Extraction*. Ph.D. Thesis, 2003.

[145] Georges Golomingi Koni-N'sapu. *A scenario based approach for refactoring duplicated code in object oriented systems*. Diploma Thesis, University of Bern, June 2001.

[146] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M.Bernstein. Pattern Matching for Clone and Concept Detection. In Automated Software Engineering, Vol. 3(1-2):77-108, June 1996.

[147] K. Kontogiannis, M. Galler, and R. DeMori. Detecting code similarity using patterns. In *Working Notes of 3rd Workshop on AI and Software Engineering*, 6pp., Montreal, Canada, August 1995.

[148] Kostas Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns using Software Metrics. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'97)*, pp. 44-54, Amsterdam, The Netherlands, October 1997.

[149] Rainer Koppler. A systematic approach to fuzzy parsing. In *Software: Practice and Experience*, 27(6):637649, 1997.

[150] S. Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees. In *In Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS95)*, pp. 631638, October 1995.

[151] R. Koschke, E. Merlo, A. Walenstein (Eds.). Dagstuhl Seminar Proceedings 06301. In *Proceedings of Duplication, Redundancy, and Similarity in Software*,ISSN 16824405, Dagstuhl, Germany, July 2006.

[152] Rainer Koschke, J.-F. Girard, M. Wrthner. An Intermediate Representation for Reverse Engineering Analyzes. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE'98)*, pp. 241-250, Honolulu, Hawai, USA, October 1998.

[153] Rainer Koschke, Raimar Falke and Pierre Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pp. 253-262, Benevento, Italy, October 2006.

[154] Rainer Koschke. Survey of Research on Software Clones. In *Proceedings of Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software*, 24pp., Dagstuhl, Germany, July 2006.

[155] Jens Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE'07)*, 9pp., Vancouver, Canada, October 2007 (to appear).

[156] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pp. 301-309, Stuttgart, Germany, October 2001.

[157] Jens Krinke, Silvia Breu. Control-Flow-Graph-Based Aspect Mining. In *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE'04)*, 5pp., Delft University of Technology, the Netherlands, November 2004.

[158] Bruno Laguë, Daniel Proulx, Jean Mayrand, Ettore M. Merlo and John Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proceedings of the 13th International Conference on Software Maintenance (ICSM'97)*, pp. 314-321, Bari, Italy, October 1997.

[159] Arun Lakhotia, Junwei Li, Andrew Walenstein, Yun Yang. Towards a Clone Detection Benchmark Suite and Results Archive. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pp. 285- 286, Portland, Oregon, USA, May 2003.

[160] Thomas Lancaster, a nd Culwin Finta. A Comparison of Source Code Plagiarism Detection Engines. In *Computer Science Education*, Vol. 14(2):101-112, June 2004.

[161] Filippo Lanubile, and Teresa Mallardo. Finding Function Clones in Web Applications. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, pp. 379-386, Benevento, Italy, March 2003.

[162] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, Vol. 29(9):782-795, Semptember 2003.

[163] Seunghak Lee, Iryoung Jeong. SDD: High Performance Code Clone Detection System for Large Scale Source Code. In *Proceedings of the Object Oriented Programming Systems Languages and Applications Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA Companion'05)*, pp. 140-141, San Diego, CA, USA, October 2005.

[164] A.M. Leitao. Detection of redundant code using R^2D^2. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pp. 183-192, Amsterdam, The Netherlands, September 2003.

[165] Chao Liu, Chen Chen, Jiawei Han and Philip S. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)*, pp. 872-881, Philadelphia, USA, August 2006

[166] Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue. Analysis of the Linux Kernel Evolution Using Code Clone Coverage. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR'07)*, Minneapolis, USA, May 2007.

[167] Simone Livieri, Yoshiki Higo, Makoto Matsushita, Katsuro Inoue: Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. In *Proceedings of 29th International Conference on Software Engineering (ICSE'07)*, pp. 106-115, Minneapolis, MN, USA, May 2007.

[168] Zhenmin Li, Shan Lu, Suvda Myagmar, Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI'04)*, pp. 289-302, San Francisco, CA, USA, December 2004.

[169] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. In *IEEE Transactions on Software Engineering*, Vol. 32(3): 176-192, March 2006.

[170] Zhenmin Li, and Yuanyuan Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 5th joint meeting of the European Software Engineering Conference and the Foundations of Software Engineering Conference (ESEC/FSE'05)*, pp. 306-315, Lisbon, Portugal, September 2005.

[171] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the Harmfulness of Cloning: A Change Based Experiment. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR'07)*, 4 pp., Minneapolis, USA, May 2007.

[172] Nazim H. Madhavji. Compare: a collusion detector for pascal. *Techniques et Sciences Informatiques*, 4(6):489497, December 1985.

[173] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pp. 103112, Toronto, Ontario, Canada, May 2001.

[174] Udi Manber. Finding similar files in a large file system. In *Proceedings of the Winter 1994 Usenix Technical Conference*, pp. 110, San Francisco, USA, January 1994.

[175] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, pp. 141-150 Banff, Alberta, Canada, May 2007

[176] Zoltan Mann. Three Public Enemies: Cut, Copy, and Paste. *IEEE Computer*, Vol. 39(7): 31-35, July 2006.

[177] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code.In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pp. 107-114, San Diego, CA, USA, November 2001.

[178] Jean Mayrand, Claude Leblanc, Ettore Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of*

the 12th International Conference on Software Maintenance (ICSM'96), pp. 244-253, Monterey, CA, USA, November 1996.

[179] E. McCreight. A space-economical suffix tree construction algorithm. In *Journal of the ACM*, 32(2):262272, April 1976.

[180] Ettore Merlo. Detection of Plagiarism in University Projects Using Metrics-based Spectral Similarity. In *Proceedings of Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software*, 10pp., Dagstuhl, Germany, Dagstuhl, July 2006.

[181] E. Merlo, M. Dagenais, P. Bachand, J.S. Sormani, S. Gradara, and G. Antoniol. Investigating large software system evolution: the linux kernel. In *Proceedings of the 26th International Computer Software and Applications Conference (COMPSAC'02)*, pp. 421426, Oxford, England, August 2002.

[182] R.C. Miller and B. A. Myers. Interactive Simultaneous Editing of Multiple Text Regions. In *Proceedings of the USENIX 2001 Annual Technical Conference*, pp. 161-174, Boston, Massachusetts, June 2001.

[183] Brian S. Mitchell and Spiros Mancoridis. CRAFT: A Framework for Evaluating Software Clustering Results in the Absence of Benchmark Decompositions. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pp. 93-102, Stuttgart, Germany, October 2001.

[184] Gilad Mishne and Maarten de Rijke. Source Code Retrieval Using Conceptual Similarity. In *Proceeding of the 2004 Conference on Computer Assisted Information Retrieval (RIAO'04)*, pp. 539-554, Avignon (Vaucluse), France, April 2004.

[185] Akito Monden, Daikai Nakae,Toshihiro Kamiya,Shin-ichi Sato,Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of 8th IEEE International Symposium on Software Metrics (METRICS'02)*, pp. 87-94, Ottawa, Canada, June 2002.

[186] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pp. 1322, Stuttgart, Germany, October 2001.

[187] S.M. Nasehi, G.R. Sotudeh, and M. Gomrokchi. Source Code Enhancement using Reduction of Duplicated Code. In *Proceedings of the 25th IASTED International Multi-Conference*, pp. 192-197, Innsbruck, Austria, February 2007.

[188] Eric Nickell and Ian Smith. Extreme programming and software clones.In *Proceedings of the 2nd International Workshop on Detection of Software Clones (IWDSC'03)*, 2pp., Victoria, BC, November 2003.

[189] Santanu Paul, and Atul Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions Software Engineering* 20(6): 463-475, June 1994.

[190] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Lague. Extending software quality assessment techniques to java systems. In *Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99)*, pp. 4956, Pittsburgh, PA, USA, May 1999.

[191] Denys Poshyvanyk, and Andrian Marcus: Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*, pp. 37-48, Banff, Canada, June 2007.

[192] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. In *Journal of Universal Computer Science*, 8(11):10161038, November 2002.

[193] Aoun Raza, Gunther Vogel, Erhard Plödereder. Bauhaus–A Tool Suite for Program Analysis and Reverse Engineering. In *Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies* , LNCS 4006, pp. 71-82, Porto, Portugal, June 2006.

[194] Damith C. Rajapakse, and Stan Jarzabek. Using Server Pages to Unify Clones in Web Applications: A Trade-off Analysis. In *Proceedings of the 29th International Conference of Software Engineering (ICSE'07)*, pp. 116-126, Minneapolis, USA, May 2007.

[195] F. Ricca and P. Tonella. Using Clustering to Support the Migration from Static to Dynamic Web Pages. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC'03)*, pp. 207-216, Portland, USA, May 2003.

[196] Matthias Rieger, Stephane Ducasse, Michele Lanza. Insights into System–Wide Code Duplication. In *Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE'04)*, pp. 100-109, Delft University of Technology, the Netherlands, November 2004.

[197] Matthias Rieger. *Effective Clone Detection Without Language Barriers*. Ph.D. Thesis, University of Bern, Switzerland, June 2005.

[198] Filip Van Rysselberghe, Serge Demeyer. Studying Software Evolution Using Clone Detection. In *Proceedings of the 4th ECOOP'03 International Workshop on Object-Oriented Reengineering (WOOR'03)*, pp. 71-75, Darmstadt, Germany, July 2003.

[199] Filip Van Rysselberghe, Serge Demeyer. Reconstruction of successful software evolution using clone detection. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE'03)*, pp. 126130, Helsinki, Finland, September 2003.

[200] Filip Van Rysselberghe, Serge Demeyer. Evaluating Clone Detection Techniques. In *Proceedings of the International Workshop on Evolution of Large Scale Industrial Applications (ELISA'03)*, 12pp., Amsterdam, The Netherlands, September 2003.

[201] Filip Van Rysselberghe, Serge Demeyer. Evaluating Clone Detection Techniques from a Refactoring Perspective. In *Proceedings of the 9th IEEE International Conf. Automated Software Eng. (ASE'04)*, pp. 336-339, Linz, Austria, September 2004.

[202] T. Sager, A. Bernstein, M. Pinzger, C. Keifer. Detecting Similar Java Classes Using Tree Algorithms. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR'06)*, pp. 65-71, Shanghai, China, May 2006.

[203] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, pp. 7685, San Diego, California, June 2003.

[204] Andrew Sutton, Huzefa Kagdi, Jonathan I. Maletic, L. Gwenn Volkert. Hybridizing Evolutionary Algorithms and Clustering Algorithms to Find Source-Code Clones. In *Proceedings of the 2005 Genetic and Evolutionary Computation Conference (GECCO'05)*, pp. 1079-1080, Washington, DC, USA, June 2005.

[205] Nikita Synytskyy, James R. Cordy, Thomas Dean. Resolution of Static Clones in Dynamic Web Pages. In *Proceedings of the 5th IEEE International Workshop on Web Site Evolution (WSE'03)*, pp. 49-58, Amsterdam, September 2003.

[206] Robert Tairas, Jeff Gray. Phoenix-Based Clone Detection Using Suffix Trees. In *Proceedings of the 44th annual Southeast regional conference (ACM-SE'06)*, pp. 679-684, Melbourne, Florida, USA, March 2006.

[207] Robert Tairas, Jeff Gray and Ira Baxter. Visualization of clone detection results. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, pp. 50-54, Portland, Oregon, October 2006.

[208] Michael Toomim, Andrew Begel and Susan L. Graham. Managing Duplicated Code with Linked Editing. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'04)*, pp. 173-180, Rome, Italy, September 2004.

[209] Qiang Tu and Michael W. Godfrey. An Integrated Approach for Studying Architectural Evolution. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*, pp. 127-136, Paris, France, June 2002.

[210] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *Proceedings 9th Asia-Pacific Software Engineering Conference (APSEC'02)*, pp. 327336, Gold Coast, Queensland, Australia, December 2002.

[211] Yasushi Ueda, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Code clone analysis tool. In *International Symposium on Empirical Software Engineering (ISESE'02)*, Vol. 2, pp. 3132, Nara, Japan, October 2002.

[212] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the*

8th IEEE Symposium on Software Metrics (METRICS'02), pp. 6776, Ottawa, Canada, June 2002.

[213] V. Wahler, D. Seipel, Jurgen Wolff von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation (SCAM'04)*, pp. 128135, Chicago, IL, USA, September 2004.

[214] A. Walenstein, A. Lakhotia and R. Koschke. The Second International Workshop on Detection of Software Clones (IWDSC'03). Workshop report. In *ACM SIGSOFT Software Engineering Notes 29(2)*, pp. 1-5, March 2004.

[215] Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, Arun Lakhotia. Problems Creating Task-relevant Clone Detection Reference Data. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, pp. 285-295, Victoria, BC, Canada, November 2003.

[216] Andrew Walenstein and Arun Lakhotia. Clone Detector Evaluation Can Be Improved: Ideas from Information Retrieval. In *Proceedings of the 2nd International Workshop on Detection of Software Clones (IWDSC'03)*, 2pp., Victoria, BC, November 2003..

[217] Andrew Walenstein and Arun Lakhotia. The Software Similarity Problem in Malware Analysis. In *Proceedings Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software*, 10 pp., Dagstuhl, Germany, July 2006.

[218] M. Weiser. Program Slicing. In *IEEE Transactions on Software Engineering*, Vol. 10(4):352-357, July 1984.

[219] Richard Wettel. Automated Detection Of Code Duplication Clusters. Diploma Thesis, Politehnica University of Timisoara, June 2004.

[220] Richard Wettel, Radu Marinescu. Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments. *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, 8pp., Timisoara, Romania, September 2005.

[221] X. Yan, J. Han, and R. Afshar. CloSpan: Mining Closed Sequential Patterns in Large Datasets. In *Proceedings of the 3rd SIAM International Conference on Data Mining (SDM'03)*, pp. San Francisco, CA, USA, May, 2003.

[222] Wuu Yang. Identifying syntactic differences between two programs. In *SoftwarePractice and Experience*, 21(7):739755, July 1991.

[223] Norihiro Yoshida, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. On Refactoring Support Based on Code Clone Dependency Relation. newblock In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pp. 16-25, Como, Italy, September 2005.

[224] J. Zhang, J. Gray, Y. Lin, and R. Tairas. Aspect Mining from a Modeling Perspective. In *International Journal of Computer Applications in Technology*, Special Issue on Concern-Oriented Software, 9pp., Fall 2007.

[225] Lijie Zou and Michael W. Godfrey. Detecting Merging and Splitting Using Origin Analysis. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, pp. 146-154, Victoria, BC, Canada, November 2003.