A Survey on Clone Refactoring and Tracking

MANISHANKAR MONDAL, University of Saskatchewan
CHANCHAL K. ROY, University of Saskatchewan
KEVIN A. SCHNEIDER, University of Saskatchewan

Code clones, identical or nearly similar code fragments in a software system's code-base, have mixed impacts on software evolution and maintenance. Focusing on the issues of clones researchers suggest managing them through refactoring, and tracking. In this paper we present a survey on the state-of-the-art of clone refactoring and tracking techniques, and identify future research possibilities in these areas. We define the quality assessment features for the clone refactoring and tracking tools, and make a comparison among these tools considering these features. To the best of our knowledge, our survey is the first comprehensive study on clone refactoring and tracking. According to our survey on clone refactoring we realize that automatic refactoring cannot eradicate the necessity of manual effort regarding finding refactoring opportunities, and post refactoring testing of system behaviour. Post refactoring testing can require a significant amount of time and effort from the quality assurance engineers. There is a marked lack of research on the effect of clone refactoring on system performance. Future investigations in this direction will add much value to clone refactoring research. We also feel the necessity of future research towards real-time detection, and tracking of code clones in a big-data environment.

CCS Concepts: •**Software and its engineering** → **Software design engineering;**

Additional Key Words and Phrases: Code Clones, Clone-Types, Clone Refactoring, Clone Tracking

1. INTRODUCTION

Changes are inevitable during software maintenance and evolution. Code cloning is a common practice which is often employed by the programmers while implementing changes during both the development and maintenance phases of a software system. Code cloning involves copying a code fragment from one place of a software system and pasting it to several other places of that system with or without modifications. Code cloning results the existence of identical or nearly similar code fragments in the code-base. These identical or similar code fragments are known as code clones.

Code clones are of great importance from the perspectives of software maintenance and evolution. A great many studies [108; 1; 42; 55; 67; 6; 7; 85; 44; 94; 110; 115; 124; 112; 114; 113; 111] have been conducted on discovering and analyzing the impacts of code clones on software maintenance. While a number of studies [1; 42; 55; 67; 71; 72; 73] identify some positive impacts of code clones, there is strong empirical evidence [80; 6; 7; 85; 44; 84; 93; 94; 78; 24; 60; 17; 56; 144] of negative impacts too. Focusing on the issues related to code clones researchers suggest managing them through refactoring [11; 3] and tracking [31; 58] for minimizing their negative impacts, and getting benefited from their positive sides.

Clone refactoring refers to the task of merging several clone fragments from a clone class (i.e., a group of code fragments that are similar to one another) into a single one if possible. However, refactoring of all clone fragments in a software system is impractical [69]. There can be situations where refactoring of clone fragments in a particular class is impossible but the fragments need to be updated together consistently. Clone tracking is important in such situations.

The clone fragments in a particular clone class may remain scattered at different source code files and folders of a software system's code-base. Clone tracking [31; 58] means remembering all the clone fragments in a clone class as the software system evolves through changes so that when a programmer makes some changes to a particular clone fragment in that class, the clone tracking system can automatically notify her about the existence of the other clone fragments in the class. The programmer can then decide whether she needs to implement similar changes to these other clone fragments in order to ensure consistency of the software system's code-base.

A number of studies have been conducted on clone refactoring [151; 4; 147; 123; 122; 11; 5; 130; 131; 5; 49; 50; 15; 13; 26; 74] and tracking [92; 137; 30; 31; 32; 58; 48] resulting a number of related techniques and tools [11; 5; 92; 137; 30; 31; 32; 58; 70; 49; 53; 76; 16; 133; 38; 90; 138]. The goal of our survey is to investigate the state-of-the-art in clone refactoring and tracking, and pointing out possibilities of future research. We answer the research questions listed in Table I and make the following contributions:

— Classifying and discussing the existing studies on clone refactoring and tracking
— Defining quality assessment features for the clone refactoring and tracking tools, and performing a comparative analysis of the tools with respect to these features.
— Identifying future research possibilities in the area of clone refactoring and tracking.

Our survey can be useful for the researchers in the field of clone refactoring and tracking because it can help us quickly identify the research directions that have already been explored, find the directions that are yet to explore, identify the existing tools and techniques in the field, and make a comparative scenario among these tools on the basis of their features. Our analysis indicates that future research can be conducted on enhancing the existing refactoring and tracking tools to support more programming languages as well as clone-types. We also realize that in order to keep pace with the rapid advancement of technologies, it is important to support clone management (i.e., clone detection, tracking, and refactoring) in a big-data environment empowered by Hadoop-MapReduce framework. Future research in this direction can make a significant contribution in software maintenance.

The rest of the paper is organized as follows. Section 2 defines code clones, Section 3 describes our survey procedure, Section 4 discusses the studies on clone refactoring by separating those into eight categories, Section 5 presents a qualitative analysis of the clone refactoring tools, Section 6 discusses the existing studies on clone tracking, Section 7 shows a qualitative analysis of the clone tracking tools, Section 8 discusses future research possibilities in clone refactoring and tracking, Section 9 presents answers to the research questions, and Section 10 concludes the paper with final remarks.

2. CODE CLONE

According to the literature [110; 115], *if two or more code fragments in a code-base are identical or nearly similar to one another, we call them code clones*.

Clone-Pair: Two code fragments that are similar to each other form a clone pair.

Clone Class: A group of similar code fragments forms a clone class or a clone group.

2.1. Types of Code Clones

There are four types of code clones as discussed below.

— **Type 1 Clones.** Exactly similar (i.e., identical) code fragments disregarding their comments and indentations are known as Type 1 clones.
— **Type 2 Clones.** Type 2 clones are syntactically similar code fragments. These are mainly created from Type 1 clones because of renaming identifiers and changing data-types.
— **Type 3 Clones.** Type 3 clones are created from Type 1 and Type 2 clones because of addition, deletion, or modification of source code lines. Type 3 clones are also known as gapped clones.
— **Type 4 Clones.** Semantically similar code fragments are known as Type 4 clones. If two or more code fragments perform the same task but are implemented in different ways, these code fragments are called Type 4 clones. Type 4 clones are also known as semantic clones.

Table I. Research Questions

RQ 1	Can we categorize the existing studies on clone refactoring and tracking? If so, how much has each category been explored?
RQ 2	What are the features of the existing clone refactoring and tracking tools? Can we draw a comparative scenario among these tools on the basis of these features?
RQ 3	What are the possible future research directions in clone refactoring and tracking?

Table II. Results from Preliminary Search

	Clone Refactoring	Clone Tracking	Clone Synchronization
http://ieeexplore.ieee.org/	119	139	53
https://dl.acm.org/	77	52	12
https://www.sciencedirect.com/	183	1,115	47
https://link.springer.com	101	213	101
http://onlinelibrary.wiley.com	2	3	0
www.worldscientific.com/	8	211	138
http://digital-library.theiet.org	9	109	31

Code clones can be of different granularities such as: file clones, class clones, method clones, or arbitrary block clones. Researchers have also investigated on detecting duplications (i.e., clones) in higher level code structures [8; 9; 87], formal models [27; 28], UML sequence diagrams [126; 83], software requirements specifications [61; 62], and Matlab/Simulink models [106].

3. SURVEY PROCEDURE

We have performed a literature survey on refactoring, tracking, and synchronization of code clones. The list of websites that we explored for searching studies relevant to our survey include: *www.dl.acm.org* (ACM Digital Library), *ieeexplore.ieee.org* (IEEE Xplore Digital Library), *www.sciencedirect.com*, *link.springer.com*, *onlinelibrary.wiley.com*, *www.worldscientific.com*, and *digital-library.theiet.org* (IET Software). The keywords that we used for searching are: 'clone refactoring', 'clone tracking', and 'clone synchronization'. We got a lot of papers as our search result from the websites. The numbers of papers that we obtained from different websites for different keywords are listed in Table II. When searching papers, we used advanced search options and selected appropriate fields of research to narrow down our search results. The search results from different websites as well as from different keywords contain duplicates. These duplicates might significantly increase our checking time of the search results. For this reason, we downloaded the CSV files of the search results and then developed a program to automatically determine the distinct results comparing all the search results from all the websites. We should note that some of the websites do not support exporting search results. For these cases we checked the search results in the websites one by one. We obtained 1037 distinct results in total. We checked each of these distinct results in the following way.

First we checked the title of the paper and tried to decide whether the paper is relevant to our survey. When we could not decide by just looking at the title, we read the abstract. For most of the cases, we could decide after reading the abstract. For only a few cases, we had to read the experimental details to come to a final decision. We finally selected 97 papers (77 papers on clone refactoring and 20 papers on clone tracking and synchronization) for our survey. After selecting the papers for our survey, we thoroughly studied each of those papers to answer the research questions in Table I. Such questions were not answered before. In the following sections we present our survey. We will answer the research questions in Section 9.

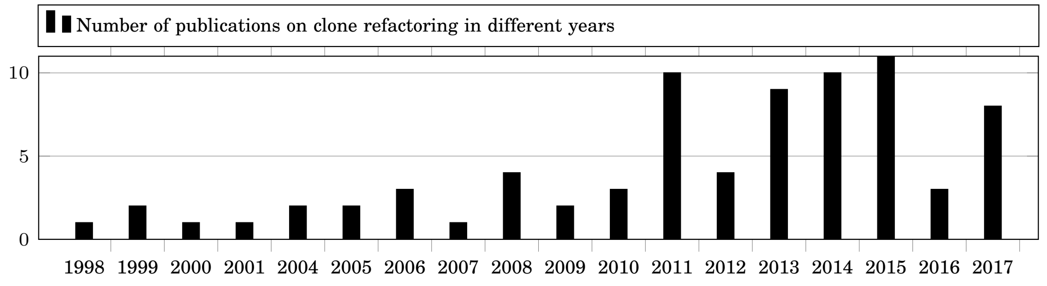

Fig. 1. Number of publications on clone refactoring in different years

4. CLONE REFACTORING

Focusing on the impacts of code clones researchers suggest to properly manage code clones so that we can get rid of their negative impacts as well as can be benefited from their positive sides. Clone refactoring is a possible way of clone management. According to the literature [39], *refactoring refers to changing a software system's code-base with the goal of enhancing its internal structure so that its external behaviour does not change. Clone refactoring refers to the task of merging (i.e., unifying) two or more clone fragments from the same clone class.* Software researchers suggest clone refactoring in order to improve the maintainability of the source code. A great many studies [151; 4; 147; 123; 122; 11; 5; 130; 131; 5; 49; 50; 15; 13; 26; 74; 100; 135] have already been done on clone refactoring. The graph in Fig. 1 reports the number of publications on code clone refactoring in different years beginning from the year of 1998. We see that there is an increasing trend in the number of publications on clone refactoring. After reading and analyzing the existing studies on clone refactoring, we classify those into the following research directions:

— Clone categorization from refactoring perspectives
— Automatic refactoring of code clones
— Semi-automatic refactoring of code clones
— Integrating clone detection and refactoring
— Scheduling for clone refactoring
— Comparing clone detection techniques from refactoring perspectives
— Analyzing the effect of clone refactoring
— Investigating how developers refactor clones
— Identifying code clones that are important for refactoring

In the following subsections, we discuss the clone refactoring studies by separating those into the research directions mentioned above.

4.1. Clone Categorization from Refactoring Perspective

A number of studies [4; 147; 123] have categorized code clones from the perspectives of refactoring. Different refactoring techniques have been proposed for different categories. We discuss these studies in the following paragraphs.

Balazinska et al. [4] proposed 18 categories of clone classes from their manual investigation on 800 clones from six open-source subject systems written in Java. They implemented a tool called SMC (Similar Method Classifier) for automatically classifying clones into these categories. However, they considered only the method clones in their study. They did not propose any particular refactoring mechanism for any of the 18 categories they proposed.

Yu and Ramaswamy [147] detected code clones from Linux using CCFinderX clone detector and categorized these clones into three categories: (1) singular concern clones, (2) cross-cutting concern clones, and (3) partial concern clones. They found that respectively 39%, 24%, and 37% of all the code clones belong to these three categories. They involved a domain expert to refactor these clones. The domain expert could refactor code clones in the first two categories. However, clones in the third category were not suitable for refactoring because such code clones were parts of other concerns.

Schulze et al. [123] categorized code clones for refactoring on the basis of two things: (1) type of code in the clone fragments, and (2) location of the clone fragments. On the basis of the location of code clones in the directory tree, the authors decide whether OOR (object oriented refactoring) or AOR (aspect oriented refactoring) is suitable for refactoring those. In their categorization, they have shown which type of refactoring is applicable to which type of clone fragments on the basis of the location information of the clone fragments. According to their analysis when clone fragments of a particular clone class are scattered throughout the code-base, AOR is more appropriate for them compared to OOR.

4.2. Automatic Refactoring of Code Clones

A number of studies investigate fully automatic refactoring. Automatic refactoring refers to the task of refactoring code clones without interaction from programmers. This task is challenging because the refactoring tool might often need to select the most appropriate refactoring technique among a number of alternatives for a particular refactoring case. There are only a few studies on automatic clone refactoring. We discuss these studies below.

Balazinska et al. [5] refactored method clones in JDK 1.1.5 using the strategy design pattern proposed by Gamma et al. [40]. The clones were detected by matching sub-trees in the ASTs (abstract syntax trees) of the source code. Their refactoring tool, CloRT (Clone Reengineering Tool), capable of automatically refactoring three categories of method clones: identical clones, clones having superficial differences (such as differences in variable names), and clones having differences in the use of non-local variables. Balazinska et al. identified these three categories of method clones by using their previously implemented tool SMC [4]. They selected 28 method clones for refactoring grouped into 11 clone classes belonging to those three categories. Their refactoring activity increased the size of the source code. For merging 28 method clones their tool created 84 new methods which were easier to manage.

Meng et al. [90] implemented a fully automatic clone removal tool, RASE, which is capable of extracting common code guided by systematic edits, creating new types and methods, parameterizing differences in types, methods, variables, and expressions, and inserting return objects and exit labels on the basis of control and data flow. With systematic editing scope RASE could refactor clones from 54% of the investigated method pairs and 67% of the investigated method groups. RASE can apply combinations of six refactoring operations: extract method, add parameter, parameterize type, form template method, introduce return object, and introduce exit label.

Mazinanian et al. [88] introduced a clone refactoring tool called JDeodorant. This tool can be used in batch processing mode for automatically reading clone detection results, analyzing refactorability of code clones, and finally performing the refactoring action if possible. JDeodorant [88] is the updated version of the clone refactoring tool implemented in a study performed by Tsantalis et al. [138]. We will discuss this study in detail in Section 4.4. In this study [138], Tsantalis et al. did not perform automatic clone refactoring. However, in a later study [139], they used their updated tool, JDeodorant [88], for the purpose of automatic refactoring of code clones.

4.3. Integrating Clone Detection and Refactoring

Clone detection and refactoring are two different tasks. Seamless integration of these two tasks is necessary in order to ensure proper management of code clones. Different clone detection tools are currently existing with different capabilities. Developing a clone refactoring tool aiming to work on the output of a subset of these clone detectors might be challenging. Most of the clone refactoring tools work on top of a single clone detector. Only JDeodorant [88] works on more than one clone detector. In the following paragraphs, we discuss the studies on integrating clone detection and refactoring.

Tairas [130; 131] proposed a seamless integration between the detection and refactoring of code clones. There are existing detection techniques as well as refactoring tools. Tairas's idea was to customize these existing techniques and tools to build a complete system for clone maintenance. In a later study, Tairas and Gray [133] developed an Eclipse plug-in called CeDAR (Clone Detection, Analysis, and Refactoring) that bridges the gap between clone detection and refactoring. CeDAR works on top of DECKARD [59] clone detector. The code clones detected by DECKARD are passed to the refactoring engine of Eclipse. The refactoring engine then analyzes which code clones are suitable for refactoring. CeDAR was evaluated on eight open source software systems. The authors report that CeDAR can considerably increase the possibilities of clone refactoring by providing higher number of refactorable clone classes to the programmers compared to other clone refactoring tools ARIES [53] and SUPREMO [70]. CeDAR is capable of detecting and refactoring only Type 1 and Type 2 code clones.

The clone refactoring tool JDeodorant [88] which was introduced by Mazinanian et al. [88] integrates clone detection and refactoring. The tool was implemented as a plug-in for the Eclipse IDE. It can automatically import clone detection results from five different clone detectors (CCFinder [65], DECKARD [59], CloneDR [11], NiCad [23], and ConQAT [63]), analyze the code clones for their refactorability, and apply refactoring if possible. While importing clone detection results from a clone detector, JDeodorant checks and corrects the syntactic inconsistencies in the clone fragments. It disregards clone fragments that extend beyond method boundaries.

4.4. Semi-automatic Refactoring of Code Clones

Most of the existing clone refactoring techniques [5; 49; 50; 70; 64; 76; 16; 21; 141; 52; 138; 38; 36; 134; 37; 127] are semi-automatic (i.e., they require user interactions for the actual implementation of refactoring). We discuss these studies and techniques in the following paragraphs.

The first ever study on clone removal was done by Baxter et al. [11]. They detected code clones by generating ASTs (abstract syntax trees) of the source code, and then by finding matches between sub-trees. They applied their technique on a software system written in C. Their implemented clone detection tool could produce macro bodies and macro invocations for removing clones. Their refactoring approach is restricted to the programming languages that support macro generation. They considered arbitrary block clones in their study.

The clone refactoring tool CloRT [5] implemented by Balazinska et al. [5] using strategy design pattern was capable of automatic refactoring (i.e., refactoring without user interaction). However, in another study [3] Balazinska et al. proposed a semi-automated clone refactoring technique using template design pattern with an aim of providing a higher level of refactoring flexibility to the programmers by showing them refactoring possibilities. In this technique they first identify the differences between the method clones, present these differences to the programmers in an easily understandable way (showing the differences in ASTs), and provide descriptions about the categories of differences (such as differences in signature, in variable names) so that they can have enough knowledge for refactoring.

Koni-N'Shapu et al. [70] performed scenario based refactoring of code clones using the clone detector DUPLOC [33]. They considered Type 1 clones and Type 3 clones that get created by additions, deletions, or modifications in Type 1 clones for refactoring. Type 2 clones (i.e., created from Type 1 clones because of renaming variables or changing data-types) were not considered in the study, because DUPLOC cannot detect Type 2 clones. Koni-N'Shapu et al. implemented a tool called SUPREMO to assist programmers in semi-automatic refactoring of code clones. SUPREMO helps us apply a number of refactoring patterns such as: Extract Method, Pull Up Method, Create Template Method, Parameterization, Insert Method Calls, and Insert Super Calls. Koni-N'Shapu et al. identified different cloning scenarios, and suggested particular refactoring patterns for each scenario. For example, if a clone-pair remains in the same method, then the possible patterns for refactoring these clones are Extract Method, and/or Parameterize Method. SUPREMO is language independent. It was used for refactoring clones in SMALL TALK, C++, and Java systems.

Higo et al. [49] performed a study on clone refactoring considering one open source object oriented subject system called ANTLR. They first detect clone-pairs and clone classes from ANTLR using Gemini which is a clone analysis and visualization tool. Gemini uses CCFinder for detecting clones. They implement a tool called CCShaper that analyzes code clones detected by CCFinder. CCShaper automatically identifies structural blocks (such as: loops, if-else blocks, blocks enclosed by " and ") in the clone fragments. Such structural blocks are suitable for refactoring. According to CCShaper output they divide the detected clones into two groups. The clone classes in one group were refactored using Extract Method Refactoring. The clone classes in the other group were refactored using 'Pull Up Method Refactoring'.

Higo et al. [50; 53] implemented the clone refactoring tool ARIES on top of their previous tool CCShaper [49]. CCShaper could identify language constructs that are suitable for extracting from a clone pair detected by CCFinder. ARIES determines whether such language constructs (e.g., loop, method, if-else) contains variables beyond their scopes or not. Such information is necessary for Extract Method refactoring. ARIES also identifies the container classes as well as the parent classes of the clone fragments in a clone-pair. Such class level information is necessary for Pull Up Method refactoring. ARIS can also help us take decision regarding the following refactoring patterns: Extract Class, Form Template Method, Move Method, Parameterize Method, and Pull Up Constructor. Higo et al. [50; 53] detected clones using CCFinder from a subject system called Apache Ant and found 154 clone classes. By applying ARIES they found 52 clone classes that can be refactored using 'Extract Method' refactoring and 12 clone classes that can be refactored using 'Pull Up Method' refactoring.

Juillerat and Hirsbrunner [64] proposed an algorithm for 'Extract Method Refactoring' on the code clones in Java source code. The first step of the algorithm is to construct the AST of the Java source code. In the second step, a token list is generated through a post order traversal of the tree. Then a loss less data compression technique is applied to the token list to identify code clones. Code clones are similar sub-lists of tokens. The final step is to identify those code clones that obey certain constraints necessary for 'Extract Method Refactoring'. The authors properly describe the constraints in the paper. However, investigations regarding the application of the proposed algorithm on any subject system was not reported.

Li and Thompson [76] proposed a hybrid approach by combining token based and AST based techniques for detecting code clones in Erlang/OTP programs, and for refactoring those code clones under user control. The code clones were detected using token matching because this approach is faster. An AST based technique was applied to identify the syntactically well-formed clones. The detection and refactoring mechanisms were implemented as a tool called Wrangler. Wrangler was integrated with Emacs and Eclipse. In a later study [77] Li and Thompson updated Wrangler to detect clones incrementally.

Brown and Thompson [16] proposed an AST based clone detection and refactoring technique for Haskell source code. While the clone detection part is automatic, the refactoring part is semi-automatic and requires user analysis and interactions at different steps of clone removal. The detection and refactoring mechanisms were combined into a tool called HaRe (Haskell Refactorer).

Choi et al. [21] identified clone refactoring opportunities by combining different clone metrics. They identified and analyzed code clones using Gemini [140] which is a GUI front-end of the clone detector CCFinder [65]. Gemini reports three metircs: LEN(S), RNR(S), and POP(S) for the code clones detected by CCFinder. LEN(S) is the average length of the clone fragments in the clone set S. RNR(S) is the ratio of non-repeated token sequences of the clone fragments in S. Finally, POP(S) is the number of clone fragments in the clone set S. They showed that combinations of these metrics can report refactorable clone sets (i.e., clone classes) with higher precision compared to each individual metric.

Tokunaga et al. [136] proposed a methodology for defining a collection of refactoring patterns for code clones. They showed two categorizations of code clones from refactoring perspectives: (1) categorization on the basis of the class relationships of code clones, and (2) categorization on the basis of the granularity of clone fragments. They presented the refactoring pattern for Pull Up Method. It seems that the refactoring pattern that they describe is trivial. A number of studies have used pull up method refactoring before. Also, this refactoring technique has been clearly defined by Fowler [39]. Fowler has defined a long list of refactoring patterns (93 patterns in total). In presence of such patterns there is no necessity of redefining those. Tokunaga et al. have shown two possible categorizations of code clones. However, more sophisticated categorizations were proposed by the previous studies.

Volanschi [141] identified many domain specific refactoring opportunities which are called stereotypes. These opportunities are not generally targeted by the standard refactoring tools that apply some common refactoring mechanisms such as: extract method, pull-up method etc. Volanschi proposed to refactor C and COBOL stereotypes by using code generators (macros in C, and COPY in COBOL) through a semi-automatic way. The proposed approach was applied on 10 software systems. Three systems were written in C, and the remaining seven were written in COBOL. In case of C programs, the reduction through refactoring was up to 46%. In case of COBOL programs, this reduction was up to 26%.

Higo and Kusumoto [52] identified code clones for refactoring by investigating their past evolution historoy. According to their consideration, the clone fragments that experienced the same changes together in the past can be the most promising candidates for refactoring to the programmers. They conducted a small experiment on a subject system ArgoUML written in Java. Using their approach they found 13 clone sets from the last revision of ArgoUML. Nine of these clone sets were suitable for refactoring. Their approach only considers clone fragments that together experienced the same changes previously. They did not consider late propagation in code clones. In late propagation the changes that are made to one clone fragment are propagated to the other clone fragment in the same clone-pair at a later time. Before change propagation there is a divergence period when the updated code fragment is not regarded a clone fragment of the other fragment that was not updated. The two fragments converge again after change propagation. Such clone fragments that experience late propagation are also important for refactoring because they have a tendency of preserving their similarity. Higo and Kusumoto [48] did not consider such clones in their study.

Sarala and Deepika [121] proposed an approach that unifies clone detection, analysis, and refactoring for C# applications. They first generate the abstract syntax tree for a C# program, and then, construct an abstract semantic graph from it. The abstract semantic graph is generated by extracting the type information of the program entities such as identifiers, and methods. Code clones are detected from this semantic graph. After detecting clones they can refactor those on the basis of some preconditions and post conditions. However, the authors did not mention the pre and post conditions for refactoring. Also, they did not report which type of clones they can refactor. Moreover, their proposed technique was not compared with any preexisting detection and refactoring techniques.

Yoshida et al. [145] proposed dynamic support for clone refactoring. According to their observations developers are not generally interested in refactoring code clones that were created long ago. Refactoring of such clones will require testing previous functionalities, and this might require much time and effort. Developers are mainly interested to refactor code fragments that are clones of a fragment which he/she is currently working on. From such observations Yoshida et al.[145] propose that when a developer will be working on a particular code fragment, the IDE should be able to proactively detect clones of that particular code fragment. The idea of detecting refactoring candidates proactively is promising. However, the proactive detection proposal was not implemented by the authors.

Fontana et al. [38] investigated clone refactoring in Java software systems, and implemented a tool called DCRA (Duplicated code refactoring advisor) that can identify clone refactoring opportunities. Their tool was build on top of NiCad [23] clone detector. DCRA includes a CloneDetailer module that determines necessary information for refactoring clone-pairs. Seven clone refactoring techniques could be suggested by DCRA. Fontana et al. evaluated DCRA on four Java software systems.

Hauptmann et al. [47] proposed an approach for extracting overlapping clones to reusable components. Their approach is suitable for removing clones from automated system tests. They use a grammatical inference algorithm called Sequitur [103] for the purpose of refactoring. Sequitur creates a grammar by replacing recurring parts using rules. Using Sequitur they propose a decomposition of the test suite where all reusable components are efficiently extracted. The test engineers can visualize the decomposed test suite, and can manually apply refactoring on the basis of the extracted reusable components. Hauptmann et al. validated their approach in an industrial setting and experienced that the approach is in fact beneficial for refactoring clones from test suites.

Tsantalis et al. [138] proposed a promising approach for automatically assessing the refactorability of a clone-pair by extending their previous work [74] on clone refactoring. Their approach analyzes whether the differences that exist between the two clone fragments in a clone-pair can be safely parameterized without affecting the behaviour of the software system. Their approach is based on matching the PDGs of the candidate clone fragments. They applied their approach on thousands of clone-pairs detected from 9 subject systems using four clone detectors: CCFinder [65], DECKARD [59], CloneDR [11] and NiCad [23]. According to their investigation, the clone-pairs detected as refactorable by their approach can indeed be refactored without causing any side effects. They also found that clones in a close relative location are generally more refactorable compared to clones in distant locations. Type 1 clones appear to be more refactorable than Type 2 and Type 3 clones. For refactoring Type 3 clones their approach tries to move the unmatched statements at the beginning or at the end of the clone fragments in such a way that the movements do not affect program behaviour.

They compared their clone refactoring approach with CeDAR [133] and found that their approach is capable of finding 83% more refactorable clones. An updated version (JDeodorant [88]) of their clone refactoring tool has already been discussed in Section 4.2. Although the proposed refactoring technique is a very promising one, it can only assess the refactorability of a clone pair. It cannot assess the refactorability of a clone group consisting of more than two clone fragments.

Narasimhan and Reichenbach [102] investigated refactoring of near-miss clones in software systems written in C++. While most of the clone refactoring studies were done on Java systems, the study of Narasimhan and Reichenbach indicates the necessity of clone refactoring facilities in C++ systems too. Narasimhan [101] also developed a tool for merging method clones in C++. Further study on refactoring block clones in C++ subject systems can be an important contribution. Basit et al. [10] developed a tool for unifying method clones using meta-programming based reuse technique of VCL. They discussed different patterns of cloning between methods and suggested VCL based techniques for unifying them.

Hotta et al. [54] investigated clone refactoring using a particular refactoring technique called 'Form Template Method' and developed a tool named 'Creios' that can semi-automatically refactor code clones existing in the sub-classes derieved from a common base class. Their tool works on top of Scropio [51] clone detector and refactors clones in software systems written in Java only. Also, the proposed approach of clone refactoring is only applicable to software systems developed using object oriented programming languages. The authors showed the applicability of 'Creios' by successfully applying it on an open-source Java system called 'Apache-Synapse'.

Ettinger et al. [35; 34] proposed an algorithm for automatically eliminating / refactoring Type 3 clones through method extraction. The algorithm was proposed being inspired by a similar algorithm introduced by Komondoor and Horwitz. The authors identified and resolved a number of optimization problems in Komondoor's algorithm and showed that solutions to these optimization problems contribute to refactoring of Type 3 clones.

Bian et al. [12] investigated semi-automatic refactoring of near-miss clones using Extract Method refactoring technique. They proposed a clone refactoring approach called SPAPE which is capable of merging two Type 3 clone fragments such that the structures of the clone fragments are syntactically dissimilar but semantically similar. Their approach involves building PDGs for the two near-miss clone fragments to be merged and transforming those PDGs if possible for merging. The authors applied their approach on ten open source subject systems and found that SPAPE can effectively apply Extract Method refactoring on near-miss clones.

In a recent study, Tsantalis et al. [139] investigated the use of Lambda Expressions (introduced in Java 8) for the purpose of refactoring code clones having behavioral differences (i.e., Type 2 and Type 3 clones). They implemented their refactoring approach as a part of their previously introduced clone refactoring tool JDeodorant [88] and tested the applicability of their approach on a clone dataset consisting of 46,765 Type 2 and Type 3 clone pairs which were considered non-refactorable by the pre-existing clone refactoring techniques. However, the authors could refactor 27,218 clone pairs using their tool capable of using Lambda Expressions, where 16,173 clone pairs were of Type 2 and the remaining 11,045 clone pairs were of Type 3. They performed an extensive evaluation of the clone pairs identified as refactorable and established the applicability of Lambda Expressions for refactoring code clones.

Hatano and Matsuo [46] investigated refactoring Type 2 clones in industrial software systems developed in COBOL programming language. Their approach of refactoring involves making compiler directives for common code fragments. They also suggest a mechanism for refactoring nested clones by ordering their refactorings. The authors show that refactoring can result in 27% reduction of COBOL source code. Futher investigations on refactoring Type 3 clones in COBOL systems can add value to clone refactoring research.

Fense et al. [37] investigated clone refactoring in the context of migrating cloned product variants to a software product line. They experimented on five cloned product variants and could reduce about 25% of the code clones (common code in product variants). Thaller et al. [135] investigated clone detection and refactoring in a programmable logic controller (PLC) software developed using IEC 61131-3 Structured Text and C/C++. Such a study indicates the necessity of clone management in PLC software systems as well. Su et al. [127] proposed a mechanism for refactoring functionally equivalent code clones in C systems. Their mechanism incorporates both static and dynamic analysis for identifying functionally equivalent code fragments. Li et al. [79] conducted a study on identifying and refactoring code clones that have semantically similar structures through matching PDGs of the target clones. While most of the existing studies investigate refactoring Type 1, Type 2, or Type 3 clones, Su et al. [127] and Li et al. [79] investigated functionally equivalent (Type 4) clones. Further studies on refactoring Type 4 clones can add value to clone refactoring research.

4.5. Scheduling for Clone Refactoring

After identifying the code clones for refactoring we can refactor them in different orders. Different refactoring orders will result different extents of gains in terms of system performance, and maintainability. The studies [15; 75; 152; 149] regarding refactoring scheduling propose different schedules (i.e., orders) for refactoring code clones with the goal of achiving the maximum gain while minimizing the refactoring effort. We discuss these studies in the following paragraphs.

Bouktif et al. [15] proposed a clone refactoring effort model in order to optimize the refactoring task. They represented the clone refactoring task as a multi-objective problem where the objectives are: minimizing the refactoring effort, and maximizing the gain (i.e., maximizing the refactoring). They performed their case study on a geographical information system called GRASS.

Yoshida et al. [146] investigated whether dependency among code clones can help us find a better schedule for clone refactoring. They introduced the concept of chained methods (methods that are related together) and chained clones (clones belonging to the chained methods) and proposed a suitable refactoring pattern for them.

Lee et al. [75] proposed a technique for automatically identifying code clones that are suitable for refactoring, and for scheduling (i.e., ordering) the refactoring activities of those clones. Their technique aims to identify the most beneficial refactoring schedule using genetic algorithm (GA). They compared their scheduling algorithm with manual scheduling, greedy algorithm based scheduling, and exhaustive scheduling approaches for four open source software systems and found that their proposed alrogithm performs better than the other algorithms.

Zibran et al. [152; 150] proposed an effort estimation model for estimating the effort required for refactoring code clones. They also proposed an algorithm for scheduling the clone refactoring activities using constraint programming approach. Their effort estimation model can be used for estimating clone refactoring effort both in object oriented and procedural subject systems. They also show that their scheduling algorithm outperforms the other scheduling algorithms that use genetic algorithm, or greedy algorithm.

4.6. Comparing clone detection techniques from refactoring perspective

Rysselberghe and Demeyer [116] investigate three clone detection techniques (string matching based, token matching based, and metric fingerprint based technique) on five subject systems from a refactoring perspective. They found that the code clones reported by metric fingerprint based technique are more suitable for refactoring. However, the other two techniques reveal more refactoring opportunities than the former technique. The code clones detected using token based technique require more effort to be refactored compared to the other two techniques. Also, refactoring of such code clones is less obvious compared to the code clones detected using the other two techniques.

Tsantalis et al. [138] investigated refactoring of code clones detected from four clone detection tools: CCFinder [65], DECKARD [59], CloneDR [11] and NiCad [23]. They also made a compaison of these clone detectors from refactoring perspectives and found that AST based clone detection tools, CloneDR and DECKARD, have a tendency of detecting more refactorable clones (particularly, more refactorable clones of Type 2 and Type 3) in the production code compared to the other two clone detectors. The token-based clone detector, CCFinder, appears to detect more refactorable clones in the test code. Moreover, while CloneDR tends to detect smaller refactorable clones mostly located in the same method, DECKARD appears to detect larger and more uniformly distributed (in term of clone size) refactorable clones. For the hybrid clone detector, NiCad, Tsantalis et al. found that the consistent renaming option provides us with more refactorable clones.

4.7. Analyzing the effect of clone refactoring

Rajapakse and Jarzabek [109] investigated the effects of unifying code clones in an web application. They first implemented the web application using PHP without any controlling on the cloning process. Then, they refactored the code clones in the PHP server pages. They compared the performance of the web applications before and after the application refactorings. According to their findings, clone unification may negatively affect system performance. It might greatly increase the testing effort and time. Two or more modules might utilize the same method. A change in that method should require all modules to be tested. However, if different copies of that method exist for different modules, then the modules can evolve independently with independent evolution of the method copies.

The findings of Rajapakse and Jarzabek [109] are very important from the perspective of software maintenance. Their findings imply that clone refactoring is not always beneficial for the performance and maintenance of web applications. We think that similar studies should also be conducted considering desktop applications. Mahmoud and Niu [86] found that removing code clones through refactoring can negatively affect requirements to code traceability. However, Mourad et al. [100] found that after clone refactoring, the source code attributes of the refactored classes get significantly improved.

4.8. Investigating how Developers Refactor Clones

Researchers think that before developing clone refactoring tools we should also investigate how developers generally perform clone refactoring during development and maintenance. A number of studies [41; 132; 20; 153; 143; 142; 66; 18] have identified and reported different patterns of refactoring applied by the programmers. We discuss these studies in the following paragraphs.

Göde [41] identified removal of code clones by analyzing the clone evolution history of four subject systems using an incremental clone detection tool [43]. According to his investigation, 'Extract Method Refactoring' is the refactoring technique which is most frequently used by the programmers. Also, the clones that are removed by refactoring are mostly located in the same source code file. According to his manual observation there was a notable discrepancy between the code clones detected by the clone detector and the clones removed by the developers through refactoring.

Tairas and Gray [132] identified and analyzed the clone refactoring activities of the programmers from the evolution histories of two open source software systems: JBoss, and ArgoUML. They detected clones using the DECKARD [59] clone detector. For identifying the changes to the code clones they used UNIX *diff* command. According to their observation, refactoring can be done on parts of code clones (i.e., sub-clones). They suggest that clone refactoring tools should be able to analyze and suggest sub-clone refactoring opportunities.

Choi et al. [20] performed a study on three open source software system in order to understand how clones are refactored during software development by the developers. They detected clones using CCFinder [65] and identified clone refactoring using Ref-Finder [68; 107]. Although CCFinder detects only Type 1 and Type 2 clones, they also detected Type 3 clones using undirected similarity [89]. They targeted to mine one of the following seven refactoring patterns: Extract Method, Pull-up-Method, Replace Method with Method Object, Extract Class, Parameterize Method, Extract Super Class, and Form Template Method. They found that two refactoring patterns: Extract Method, and Replace Method with Method Object are mostly used by the programmers during development.

Zibran et al. [153] investigated the effects of seven different factors on clone removal. These factors include: clone group size (number of clone fragments in a clone class or group), size of clone fragment, distribution of clone fragments in the file system, change types experienced by the clones, frequency of experiencing changes, clone granularity, and extent of textual similarity among the clone fragments. They performed their investigation on 9 subject systems using a NiCad-based clone genealogy extractor called gCad. They found that clone fragment size and textual similarity of clone fragments significantly affect the removal of clone fragments. Clone fragments with larger size have a significantly higher tendency of being removed from the system. Also, clone fragments with variable name differences were found to be more promising candidates for removal. Zibran et al. found that there are many promising candidates for removal through refactoring. However, those were not refactored possibly because of the lack of tool support. Zibran et al. did not investigate any clone removal or refactoring pattern. As we have already discussed in the previous subsections, there are different patterns for clone refactoring such as: extract method refactoring, pull up method refactoring. A study on which clone refactoring pattern is mostly used by the developers during programming could be important. In a recent study [148] proposed developing a tool for visualizing and analyzing different instances of clone refactoring.

Wang and Godfrey [143] performed a study on recommending code clones for refactoring considering design, context, and evolution history of the code clones. They detect code clones from different revisions of a software system using the clone detector called iCones [43]. Then they automatically analyze the clone evolution history and identify instances of clone refactoring by applying the approach proposed by Demeyer [29]. They collected 15 features by observing the clone refactoring instances. Using these features they build a classifier for recommending refactoring candidates. They evaluated their clone refactoring recommendation technique on three subject systems. In a within-project-testing environment their technique can suggest refactoring candidates with a precision of 77.3% to 87.9%. In case of cross-project-testing their technique's precision was between 73.2% to 88.5% percent.

Wang and Godfrey [142] also investigated intentional clone refactoring in Linux kernel, and found that only a very little proportion of the code clones are intentionally refactored by the programmers during evolution. Chen et al. [18] developed a tool for identifying and visualizing clone refactoring instances during software evolution. Their tool also supports detecting inconsistent refactoring instances.

4.9. Identifying code clones that are important for refactoring

Mondal et al. [96] investigated identifying code clones that can be important for refactoring. They automatically analyzed clone evolution history from thousands of commit operations of software systems downloaded from on-line SVN repositories, and discovered a particular clone change pattern, *Similarity Preserving Change Pattern*, such that code clones that evolve following this pattern can be important for refactoring. They showed that the number of SPCP clones (i.e., the number of code clones that can be important for refactoring) is generally very low in a software system. Also, refactoring of SPCP clones can help us minimize late propagations [6] in code clones.

In another study [95] Mondal et al. investigated the cross-boundary evolutionary coupling of SPCP clones and found that the SPCP clones having such couplings should not be considered for removal through refactoring. Such SPCP clones (i.e., the cross-boundary SPCP clones) should be considered for tracking along with their relationships across their class boundaries. The non-cross-boundary SPCP clones can be considered important for refactoring. Mondal et al. [99] also developed a tool called SPCP-Miner for identifying the important code clones for refactoring and tracking.

Mondal et al. [98] also compared the bug-proneness of three different types (Type 1, Type 2, and Type 3) of code clones and found that Type 3 clones have the highest possibility of experiencing bug-fixes during evolution. According to their findings, they suggested prioritizing Type 3 clones when making clone management (such as clone refactoring or tracking) decisions.

5. QUALITATIVE ANALYSIS OF THE CLONE REFACTORING TOOLS

According to the literature, eleven clone refactoring tools are currently available. We have listed these tools in Table III. We have also performed a qualitative analysis of the tools on the basis of the following features.

- Which type(s) of clones we can refactor using the tool
- Which refactoring patterns can be applied using the tool
- Whether the tool refactors code clones automatically or semi-automatically
- Whether the tool can automatically assess the refactorability of code clones
- Whether the tool depends on a clone detector or it detects clones by itself.
- The tool's capability of refactoring clones from different programming languages.

5.1. Clone-type centric analysis of the clone refactoring tools

We build Table IV considering the types of clones the tools can refactor. From the table it is clear that most of the tools can refactor Type 1 and Type 2 block clones. Only three tools: SUPREMO [70], DCRA [38] and JDeodorant [88] can help us refactor Type 3 clones. Although DCRA considers Type 3 clones, it cannot refactor Type 2 clones. The Type 3 clones that it can refactor get generated from Type 1 clones. Thus, DCRA primarily considers exact similarity of code fragments while refactoring. The same is also true for SUPREMO. Tsantalis et al.'s tool can be used to refactor all three types of clones. From Table IV we see that only one tool, CLoRT [5], considers method clone for refactoring. However, the other tools that consider block clones can also refactor method clones, because block clones include method clones. From our clone-type centric analysis we see that only JDeodorant [88] supports refactoring of all three major clone-types.

We also observe that no existing clone refactoring tool can refactor Type 4 clones. The primary reason is that the detection technology of Type 4 clones is not much improved.

Table III. Clone Refactoring Tools

Tool	**Authors**	**Description**
Baxter et al.'s Tool	Baxter et al. [11]	It detects arbitrary block clones in C programs and generates macros for replacing groups of clone fragments.
CLoRT	Balazinska et al. [5]	CLoRT can automatically refactor Type 1 and Type 2 method clones.
SUPREMO	Koni-N'Shapu et al. [70]	SUPREMO helps us refactor Type 1 and Type 3 clones on the basis of different cloning scenarios. It detects code clones using the clone detector DUPLOC.
CCShaper	Higo et al. [49]	CCShaper analyzes code clones detected by CCFinder [65] and automatically finds structural blocks that are suitable for refactoring. It supports refactoring of Type 1 and Type 2 clones.
ARIES	Higo et al. [53]	ARIES works on top of CCShaper and generates different metrics that are suitable for determining clone refactoring possibilities. It supports refactoring of Type 1 and Type 2 clones.
Wrangler	Li and Thompson [76]	Wrangler performs AST based detection and semi-automatic refactoring of code clones in Erlang/OTP programs. It is integrated with Emacs and Eclipse. The clone detection mechanism of Wrangler is incremental. It helps us refactor Type 1 and Type 2 clones in Erlang/OTP code.
HaRe	Brown and Thompson [16]	HaRe can perform AST based detection and semi-automatic refactoriing of code clones from Haskell source code. It helps us refactor Type 1 and Type 2 clones in Haskel programs.
CeDAR	Tairas and Gray [133]	CeDAR is an Eclipse plug-in that brides the gap between clone detection and refactoring. CeDAR detects code clones using DECKARD [59] and passes the clones to the Eclipse refactoring engine which is responsible for automatic refactoring of code clones. It currently supports refactoring of Type 1 and Type 2 clones.
DCRA	Fontana et al. [38]	DCRA was built on top of NiCad [23] clone detector. It can identify clone refactoring opportunities in Java source code. It supports refactoring of Type 1 and Type 3 clones.
RASE	Meng et al. [90]	RASE is a fully automatic clone refactoring tool that applies combinations of six refactoring operations: extract method, add parameter, parameterize type, form template method, introduce return object, and introduce exit label. It can help us refactor Type 1 and Type 2 clones.
JDeodorant	Tsantalis et al. [138], Mazinanian et al. [88]	The tool called JDeodoant [88] can automatically assess the refactorability of a clone pair. It supports semi-automatic refactoring of the code clones of all three major clone-types: Type 1, Type 2, and Type 3.

Table IV. Types of clones that the tools can refactor

Clone Refactoring Tool	Type 1	Type 2	Type 3	Type 4	Clone Granularity
Baxter et al.'s Tool [11]	Yes	Yes			Block Clones
CLoRT [5]	Yes	Yes			Method Clones
SUPREMO [70]	Yes		Yes		Block Clones
CCShaper [49]	Yes	Yes			Block Clones
ARIES [53]	Yes	Yes			Block Clones
Wrangler [76]	Yes	Yes			Block Clones
HaRe [16]	Yes	Yes			Block Clones
CeDAR [133]	Yes	Yes			Block Clones
DCRA [38]	Yes		Yes		Block Clones
RASE [90]	Yes	Yes			Block Clones
JDeodorant [88]	Yes	Yes	Yes		Block Clones

Table V. Language capabilities of the refactoring tools

Clone Refactoring Tool	**Language Capability**
Baxter et al.'s Tool [11]	It can be used to semi-automatically refactor code clones in **C** systems only.
CLoRT [5]	It automatically refactors code clones in **Java** systems only.
SUPREMO [70]	Koni-N'Shapu et al. used this tool to refactor clones in subject systems written in **SMALL-TALK**, **C**, and **Java**. However, SUPREMO only shows the clones and reports the cloning scenarios. It cannot suggest any refactoring patterns to the programmers. Taking refactoring decisions and applying particular refactoring can only be done manually by the programmer.
CCShaper [49]	This tool can be used to refactor code clones only in **Java** systems using the metrics that it generates. The generated metrics are specific for Java language.
ARIES [53]	This tool can be used to refactor code clones only in **Java** systems using the metrics that it generates. The generated metrics are specific for Java language.
Wrangler [76]	This tool can be used to semi-automatically refactor code clones in **Erlang/OTP** programs only.
HaRe [16]	This tool can be used to semi-automatically refactor code clones in **Haskel** programs only.
CeDAR [133]	It automatically refactors code clones in **Java** systems only
DCRA [38]	This tool can be used to semi-automatically refactor code clones in **Java** systems only.
RASE [90]	It automatically refactors code clones in **Java** systems only
JDeodorant [88]	This tool can be used to semi-automatically refactor code clones in **Java** systems only.

Also, according to the literature [110; 115], Type 4 clones are semantically similar but syntactically dissimilar. Thus, the traditional refactoring techniques cannot be used for refactoring Type 4 clones. Future research on refactoring Type 4 clones can be an important contribution to clone management.

5.2. Programming language centric analysis of the clone refactoring tools

From Table V it seems that each of the clone refactoring tools except SUPREMO [70] was designed to refactor clones of a particular programming language. Seven refactoring tools were implemented for Java systems only. Three (i.e., Baxter et al.'s tool [11], Wrangler [76], and HaRe [16]) of the remaining four tools were developed for three different languages (C, Erlang/OTP, and Haskel). The tool SUPREMO [70] was used for refactoring clones in SMALL-TALK, C, and Java programs. However, this tool can only show the code clones and report the cloning scenarios. The programmers manually check the scenarios, decide particular refactoring patterns, and then apply the refactoring. Thus, SUPREMO does not provide any language specific support for taking clone refactoring decisions. We feel the necessity of clone refactoring tool supports for other programming languages (such as: C++, C#, Python) too.

From Table IV and Table V we realize that the three refactoring tools (i.e., Baxter et al.'s tool [11], Wrangler [76], and HaRe [16]) provide semi-automatic support for refactoring Type 1 and Type 2 clones in C, Erlang/OTP, and Haskel programs. It seems that we need further investigations towards developing more sophisticated clone refactoring tools with the capabilities of refactoring Type 3 clones.

5.3. Analysis regarding automatically assessing the refactorability of code clones

Given two clone fragments, if a clone refactoring tool can automatically decide whether these clone fragments are refactorable or not, then we say that the tool is capable of assessing refactorability of clone fragments. From Table VI we see that five tools (Baxter et al.'s tool [11], CLoRT [5], CeDAR [133], RASE [90], and JDeodorant [88]) can automatically assess the refactorability of code clones. The only tool DCRA [38] provides

Table VI. Capabilities of the refactoring tools in assessing the refactorability of code clones

Clone Refactoring Tool	Capability in Assessing Refactorability of Code Clones
Baxter et al.'s Tool [11]	**It can automatically assess refactorability of code clones** and generates replacement macros for the detected clone-pairs.
CLoRT [5]	**It can automatically assess refactorability of code clones** using the strategy design pattern.
SUPREMO [70]	**It cannot assess refactorability of code clones.** It can determine the cloning scenario of a clone-pair. Programmers need to take decision manually on the basis of the cloning scenario.
CCShaper [49]	**It cannot assess refactorability of code clones.** Programmers need to take decision manually on the basis of the structural blocks that it extracts from the code clones.
ARIES [53]	**It cannot assess refactorability of code clones.** Programmers need to take decision manually on the basis of the structural blocks and variable scoping information that it extracts from the code clones.
Wrangler [76]	**It cannot assess refactorability of code clones.** Programmers need to do it manually.
HaRe [16]	**It cannot assess refactorability of code clones.** Programmers need to do it manually.
CeDAR [133]	**It can automatically assess refactorability of code clones** with the help of Eclipse refactoring engine.
DCRA [38]	**It can semi-automatically assess refactorability of code clones.** It suggests a ranked list of possible refactoring patterns for a clone-pair. The programmer can then select the most suitable refactoring patterns.
RASE [90]	**It can automatically assess refactorability of code clones.**
JDeodorant [88]	**It can automatically assess refactorability of code clones** by evaluating eight preconditions of refactorability.

semi-automatic support for assessing refactorability. The remaining five tools cannot provide any support for determining whether clone fragments are really refactorable using any refactoring patterns.

5.4. Analysis regarding automaticity in clone refactoring

From Table VII we see that most of the tools perform semi-automatic refactoring of code clones. Semi-automatic refactoring refers to refactoring under programmer control. The tool suggests a particular refactoring for the clone fragments, and the programmer takes the decision about whether to apply that refactoring. Automatic refactoring means refactoring code clones without programmer interactions. Automating the task of clone refactoring is challenging. The automatic tool might often need to select the most suitable refactoring technique among a number of alternatives such as extract method, extract super class, pull-up clone, introduce template method etc. After selecting a particular refactoring technique the tool might need to select a non-conflicting name for an extracted method or an introduced class. The few tools that automate refactoring by addressing these challenges include CLoRT, RASE, and JDeodorant. Among these three tools, JDeodorant supports a new refactoring technique that involves introducing Lambda Expressions. Such a refactoring technique significantly increases the number of refactorable clones having behavioral differences. No other existing clone refactoring tools support this technique. We should note that automatic refactoring cannot obviate the necessity of manual checking after refactoring as well as manual refactoring in particular situations. This is the reason why most of the tools facilitate semi-automatic refactoring. Although RASE performs refactoring automatically, Meng et al. [90], the authors of this tool, report that there were cases where automatic refactoring was impossible. Only the experienced programmers can perform refactoring in such cases. Thus, we think that refactoring of code clones should be done semi-automatically. JDeodorant supports semi-automatic refactoring as well. We see that the tool SUPREMO [70] cannot provide tool support for refactoring. It

Table VII. Capabilities of the tools in refactoring code clones

Clone Refactoring Tool	**Capability in Refactoring Code Clones**
Baxter et al.'s Tool [11]	**It can semi-automatically refactor code clones.** It generates replacement macros for clone-pairs detected from C systems. The programmers then decide whether they will use the macros for refactoring clones.
CLoRT [5]	**It can automatically refactor code clones.** It uses strategy design pattern to automatically factorize the common parts of the cloned methods, and parameterize their differences.
SUPREMO [70]	**It does not provide automatic or semi-automatic support for refactoring code clones.** Programmers can see the code clones in SUPREMO, and then take their own decisions on how to refactor clones on the basis of the cloning scenario.
CCShaper [49]	**It provides semi-automatic support for refactoring code clones.** It automatically identifies structural blocks from clone fragments. These blocks are suitable for refactoring. Programmers can then take refactoring decisions on the basis of these structural blocks.
ARIES [53]	**It provides semi-automatic support for refactoring code clones.** ARIES was built on top of CCShaper. It can identify the structural blocks suitable for refactoring. It additionally determines whether such blocks contain variables beyond their scopes or not. Programmers can take refactoring decisions from these information.
Wrangler [76]	**It provides semi-automatic support for refactoring clones.** The programmer highlights the code clones in the IDE. Wrangler then checks whether the clone fragments can be safely refactored using the refactoring patterns mentioned in Table IX.
HaRe [16]	**It provides semi-automatic support for refactoring code clones.**
CeDAR [133]	**It provides semi-automatic support for refactoring code clones** through Eclipse refactoring engine.
DCRA [38]	**It provides semi-automatic support for refactoring clones** using a module called Refactoring Advisor. This module analyzes a clone-pair, and provides a ranked list of possible refactoring patterns (such as Extract Method or Pull Up Method) for the pair. Programmer can then select the most suitable patterns for refactoring the clone-pair.
RASE [90]	**It can automatically refactor code clones** by applying one or more of six refactoring patterns mentioned in Table IX.
JDeodorant [88]	**It provides semi-automatic support for refactoring code clones.** If a programmer selects two clone fragments for refactoring, the tool automatically checks eight preconditions to determine whether the fragments are really refactorable. If the fragments are refactorable, then the tool automatically shows a preview to the programmer containing all the changes that will occur to the code-base after refactoring. The programmer can then select the particular refactoring.

only helps programmers look at the code clones. The programmers then manually decide which refactoring pattern should be applied for refactoring the code clones, and then perform the refactoring.

5.5. Analysis regarding tool's dependency on clone detectors

If we look at Table VIII we see that five refactoring tools (e.g., RASE) detect clones by themselves. The remaining six tools (e.g., ARIES) use clone results from clone detectors. Among these six tools, each of the five tools: SUPREMO, CeDAR, CCShaper, ARIES, and DCRA can refactor clones detected by a single clone detector. The remaining tool, JDeodorant [88], can work on clone detection results from five clone detectors: CCFinder, DECKARD, CloneDR, NiCad, and ConQAT.

Integration of detection and refactoring capabilities in the same tool is important. It eliminates the dependency on a separate clone detector. However, clone detection technologies have been improved a lot in the past decades. Using a clone detector that can detect clones from multiple programming languages can open the possibilities of refactoring clones from these languages. Thus, making use of clones detected by existing clone detectors is also very important. We see that JDeodorant [88] can refactor clones detected by more than one clone detector. Further investigations on this tool

Table VIII. Dependency of the refactoring tools on clone detectors

Refactoring Tool	Dependency on Clone Detector
Baxter et al.'s Tool [11]	**Independent**
CLoRT [5]	**Independent**
SUPREMO [70]	Refactors clones detected by **DUPLOC**
CCShaper [49]	Refactors clones detected by **CCFinder**
ARIES [53]	Refactors clones detected by **CCFinder**
Wrangler [76]	**Independent**
HaRe [16]	**Independent**
CeDAR [133]	Refactors clones detected by **DECKARD**
DCRA [38]	Refactors clones detected by **NiCad**
RASE [90]	**Independent**
JDeodorant [88]	Refactors clones detected by **CCFinder**, **DECKARD**, **CloneDR**, **NiCad**, **ConQAT**

Independent = The tool both detects and refactors clones.

Table IX. Refactoring patterns that can be applied by the clone refactoring tools

Clone Refactoring Tool	Refactoring patterns that it helps us to apply
Baxter et al.'s Tool [11]	The tool produces macro bodies for clone removal, and generates macro invocations for replacing the clones.
CLoRT [5]	This tool applies strategy design pattern to parameterize clone differences, and to decouple clones from their contexts.
SUPREMO [70]	The authors manually applied the following refactoring patterns: Extract Method, Pull Up Method, Parameterize Method, Create Template Method, Insert Method Calls, and Insert Super Calls by analyzing the code clones showed by SUPREMO.
CCShaper [49]	The tool helps us refactor clones using Extract Method and Pull Up Method refactoring techniques.
ARIES [53]	This tool can help us apply the following refactoring patterns: Extract Method, Pull Up Method, Extract Class, Form Template Method, Move Method, Parameterize Method, and Pull Up Constructor.
Wrangler [76]	The tool can help us generalize a function definition, extract function, and fold expressions against a function definition.
HaRe [16]	It helps us in function folding, As-pattern folding, and merging.
CeDAR [133]	It helps us apply the following refactoring patterns: Extract Method, Pull Up Method, and Introduce Utility Method.
DCRA [38]	It advises a number of refactoring techniques including: Extract Method, Replace Method with Method Object, Merge Method, Pull Up Method, Pull Up Method Object, Form Template Method, and Leave Unchanged.
RASE [90]	This tool helps us apply six refactoring patterns: Extract Method, Add Parameter, Parameterize Type, Form Template Method, Introduce Return Object, and Introduce Exit Label.
JDeodorant [88]	This tool helps us apply the following refactoring patterns: Extract Method, Pull Up Method, Introduce Template Method, and Introduce Utility Method. It can also generalize types within the clone fragments, if necessary, and parameterize the differences within the clone fragments (PM), either with regular parameters [138] or Lambda expressions [139].

may enable it to refactor clones from all the programming languages supported by the clone detectors.

5.6. Clone refactoring pattern centric analysis

From Table IX we see that three refactoring tools: Baxter et al.'s tool [11], Wrangler [76], and Hare [16] can help us apply language specific refactoring patterns for refactoring code clones in C, Erlang/OTP, and Haskel programs respectively. The other eight tools can help us apply different refactoring patterns for clone refactoring in Java systems. CCShaper [49] helps us apply only two patterns: Extract Method refactoring, and Pull Up Method refactoring. Each of the five tools: CeDAR [133], RASE [90], ARIES

Table X. All features of the clone refactoring tools

Clone Refactoring Tool	**Publication Year**	**Clone-type Capability**	**Lang. Capability**	**Assessing Refactorability**	**Application of Refactoring**	**Dependency on Clone Detector**	**Applicable Refactoring Patterns**
Baxter et al.'s Tool [11]	1998	Type 1 Type 2	C	Automatic	Semi-automatic	Independent	Macro generation
CLoRT [5]	1999	Type 1 Type 2	Java	Automatic	Automatic	Independent	Strategy design pattern
SUPREMO [70]	2001	Type 1 Type 3	Small-Talk, C, Java	Manual	Manual	DUPLOC	EM, PUM, PM, CTM
CCShaper [49]	2004	Type 1 Type 2	Java	Semi-automatic	Semi-automatic	CCFinder	EM, PUM
ARIES [53]	2008	Type 1 Type 2	Java	Semi-automatic	Semi-automatic	CCFinder	EM, PUM, EC, CTM, MM, PM, PUC
Wrangler [76]	2009	Type 1 Type 2	Erlang /OTP	Semi-automatic	Semi-automatic	Independent	GFD, EF, FEFD
HaRe [16]	2010	Type 1 Type 2	Haskel	Semi-automatic	Semi-automatic	Independent	FF, APF, Mrg
CeDAR [133]	2012	Type 1 Type 2	Java	Semi-automatic	Semi-automatic	DECKARD	EM, PUM, IUM
DCRA [38]	2015	Type 1 Type 3	Java	Semi-automatic	Semi-automatic	NiCad	EM, RMMO, MrgM, PUM, PUMO, CTM
RASE [90]	2015	Type 1 Type 2	Java	Automatic	Automatic	Independent	EM, AP, PT, CTM, IRO, IEL
JDeodorant [88]	2015	Type 1 Type 2 Type 3	Java	Automatic	Semi-automatic	CCFinder, DECKARD, CloneDR, NiCad, ConQAT	EM, PUM, CTM, IUM, ILE, PM

EM = Extract Method **PUM** = Pull Up Method **PM** = Parameterize Method **EC** = Extract Class
GFD = Generalize a Function Definition **EF** = Extract Function **MrgM** = Merge Method
FEFD = Fold Expressions against a Function Definition **MM** = Move Method **Mrg** = Merging
FF = Function Folding **APF** = As-Pattern Folding **RMMO** = Replace Method with Method Object
PUMO = Pull Up Method Object **CTM** = Create Template Method **PUC** = Pull Up Constructor
IRO = Introduce Return Object **AP** = Add Parameter **PT** = Parameterize Type
IUM = Introduce Utility Method **ILE** = Introduce Lambda Expression **IEL** = Introduce Exit Label

[53], DCRA [38], and JDeodorant [88] can help us apply a number of refactoring patterns. JDeodorant additionally supports clone refactoring using Lambda Expressions. The tool CLoRT [5] applies strategy design patterns for factorizing common parts and parameterizing the differences between two cloned methods. We see that these six tools (CeDAR, RASE, ARIES, DCRA, CLoRT, and JDeodorant) are promising in terms of the refactoring patterns that they help us apply for refactoring clones. The authors of the tool SUPREMO [70] applied a number of refactoring patterns for refactoring clones. However, the application was done manually. SUPREMO cannot help us apply any refactoring pattern.

5.7. Overall analysis considering all the features

We have accumulated all the features of all the clone refactoring tools in Table X. From our previous discussions we realize that the tool called JDeodorant [88] can be used to refactor code clones of all three major clone-types (Type 1, Type 2, and Type 3). None of the other existing clone refactoring tools can refactor all these three clone-types. JDeodorant can work on clone detection results from five clone detectors: CCFinder, DECKARD, CloneDR, NiCad, ConQAT. The other clone refactoring tools provide support only for a single clone detector. JDeodorant supports both automatic and semi-automatic refactoring of code clones.

RASE [90] is an automatic clone refactoring tool for Java systems. However, it refactors Type 1 and Type 2 clones only. CLoRT [5] is also an automatic refactoring tool for the same programming language. However, it cannot refactor block clones. Thus, JDeodorant [88] seems to be a promising clone refactoring tool for Java systems.

5.8. Analyzing clone refactoring tools on the basis of different refactoring scenarios

In this section we provide a comparative analysis of the clone refactoring tools on the basis of their capabilities of in refactoring code clones in different refactoring scenarios. Refactoring scenarios are language specific. There are different refactoring tools for different programming languages. However, most of these tools can be used for refactoring code clones in Java systems. From Table X we see that eight tools (excluding Baxter et al.'s tool, Wrangler, and HaRe) can be used for refactoring code clones in Java systems. We provide a scenario-based comparison of these eight refactoring tools considering different scenarios that are specific to Java programming language.

5.8.1. Scenario-based comparison of the refactoring tools considering Java language. We have already mentioned that eight tools can be used for clone refactoring in Java systems. These tools are: CLoRT [5], SUPREMO [70], CCShaper [49], ARIES [53], CeDAR [133], DCRA [38], RASE [90], and JDeodorant [88] Tool. We compare capabilities of these eight tools considering the following scenarios.

— **Scenario 1:** The two clone fragments that need to be refactored reside in the same method.
— **Scenario 2:** The two clone fragments that we want to refactor reside in two different methods of the same Java class.
— **Scenario 3:** The two clone fragments that we want to refactor reside in two subclasses of the same immediate super class.
— **Scenario 4:** One of the two clone fragments that are candidates for refactoring resides in a subclass, and the other one resides in the immediate superclass.
— **Scenario 5:** One of the two clone fragments that are candidates for refactoring resides in a subclass, and the other one resides in a superclass which is not the immediate superclass of the subclass.
— **Scenario 6:** The two clone fragments that we want to refactor reside in two different subclasses: C1 and C2. The immediate superclasses of these two subclasses are different.
— **Scenario 7:** The two clone fragments that we want to refactor reside in two unrelated classes.

We show these seven scenarios in Fig. 2. In each of these scenarios we show two clone fragments: CF1 and CF2 which are candidates for refactoring. We discuss whether a tool facilitates refactoring in these scenarios, and what type of facilities it provides for refactoring. Table XI shows the tool capabilities with respect to the scenarios. The following paragraphs describe how a tool facilitates refactoring in a particular scenario.

We first discuss the clone refactoring tool CLoRT [5]. From Table XI we see that it cannot facilitate refactoring in the simplest scenario, Scenario 1 (S1). The reason is that it can only refactor method clones. Scenario 1 indicates that the clone fragments to be refactored are block clones in the same method. However, CLoRT can be used for refactoring in the other six scenarios if the two refactoring candidates are method clones. CLoRT applies strategy design pattern for refactoring. The tool SUPREMO [70] facilitates refactoring in each of the seven scenarios. It automatically determines each of these scenarios on the basis of the positions of the clone fragments to be refac-

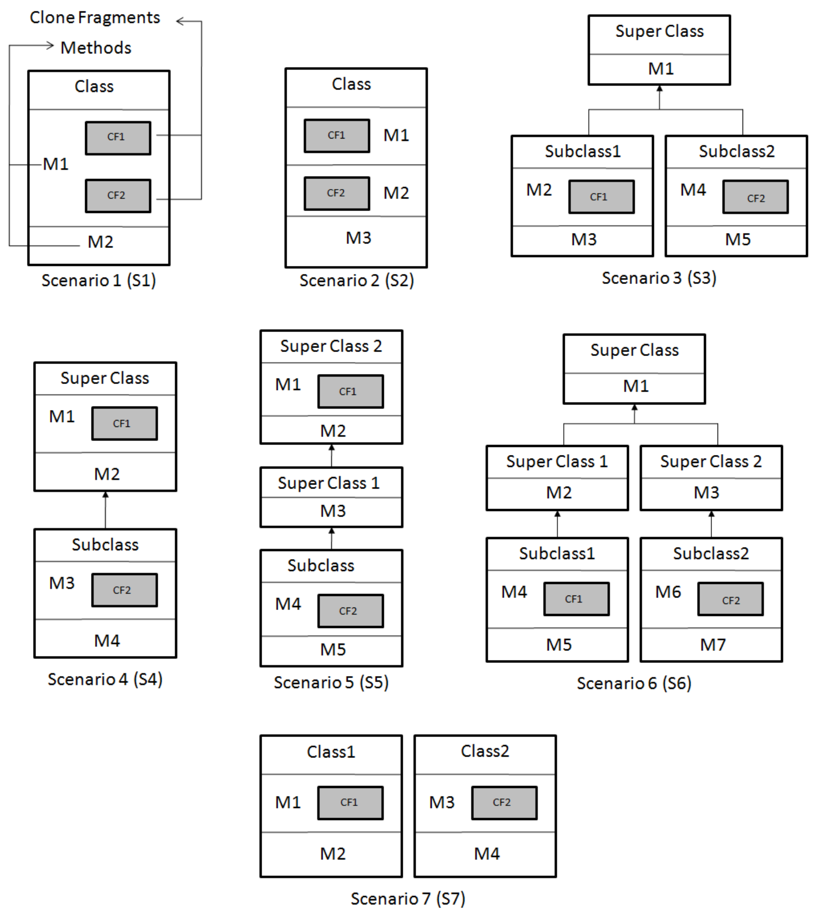

Fig. 2. Clone refactoring scenarios have been shown in this figure. In each scenario we can see two clone fragments, CF1 and CF2, that are candidates for refactoring.

Table XI. Tool capabilities in different refactoring scenarios

Clone Refactoring Tool	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**
CLoRT [5]	N	Y	Y	Y	Y	Y	Y
SUPREMO [70]	Y	Y	Y	Y	Y	Y	Y
CCShaper [49]	N	N	N	N	N	N	N
ARIES [53]	Y	Y	Y	Y	Y	Y	Y
CeDAR [133]	Y	Y	Y	N	N	N	Y
DCRA [38]	Y	Y	Y	Y	Y	Y	N
RASE [90]	Y	Y	Y	N	N	N	Y
JDeodorant [88]	Y	Y	Y	Y	Y	Y	Y

Y = The tool supports refactoring in the scenario N = The tool cannot refactor in the scenario tored. For each scenario it suggests particular refactoring patterns. The programmers can apply these patterns to perform the refactoring. However, this tool cannot automatically assess refactorability of two clone fragments. Programmers are responsible for assessing the refactorability. CCShaper [49] does not particularly facilitate refactoring in any of the seven scenarios. Given two clone fragments, it only shows the structural blocks in the fragments. The programmers are to make refactoring decisions seeing these structural blocks. The tool ARIES [53] is a significant improvement over CCShaper [49], and it facilitates refactoring in these seven scenarios by providing necessary metrics which are indicators of particular refactoring patterns applicable in particular scenarios. However, it cannot automatically assess refactorability of clone fragments. The tool CeDAR [133], implemented as a plug-in of Eclipse IDE, supports refactoring in four scenarios as mentioned in Table XI. It uses Eclipse refactoring engine for automatically determining the refactorability of two clone fragments. Also, CeDAR provides support for applying three refactoring patterns: *extract method*, *pull up method*, and *introduce utility method*. DCRA [38] supports refactoring in six scenarios (excluding the seventh scenario). It automatically detects the scenarios by analyzing clone locations and suggests a set of refactoring patterns for each scenario. However, it cannot automatically assess refactorability of clone fragments. RASE [90] can perform automatic refactoring of clone fragments in four scenarios: S1, S2, S3, and S7 as mentioned in Table XI. JDeodorant [88], however, supports all the seven refactoring scenarios listed above. This tool performs both automatic and semi-automatic refactoring. It can automatically assess refactorability of any clone-pair by analyzing a number of preconditions. It also provides support for applying many refactoring patterns including *extract method*, *pull up method*, *create template method*, *introduce utility method*, *parameterize method*, and *introduce Lambda Expressions*.

6. CLONE TRACKING

Clone tracking means remembering all clone fragments in a particular clone class during evolution so that when a programmer wants to make some changes to a particular fragment in the class, he/she gets notified about the existence of the other clone fragments in the same class. The programmer can then decide whether to implement the same changes to these other clone fragments to ensure consistency of the code-base. The main purpose of clone tracking is to ensure consistent updates of the code clones that are not suitable for refactoring. Updating code clones by ensuring their consistency is also known as clone synchronization in the literature. A number of studies [92; 137; 30; 31; 32; 58; 48; 125] have been done on clone tracking as well as clone synchronization resulting a number of techniques and tools. Fig. 3 shows a bar graph showing the number of publications on clone tracking in different years. We see that clone tracking research started from the year of 2001. By comparing the graphs in Fig. 1 and Fig. 3 we see that there are more publications on clone refactoring compared to clone tracking. We discuss the studies on clone tracking and synchronization in the following subsections by separating them into the following four categories.

— Clone synchronization without clone tracking
— Clone synchronization with clone tracking
— Clone tracking without facilities for synchronization
— Ranking code clones for tracking

6.1. Clone synchronization without clone tracking

A number of studies were conducted with an aim to synchronize code clones without tracking them through evolution. Such studies mainly consider Type 1 clones for synchronization. We discuss these in the following paragraphs.

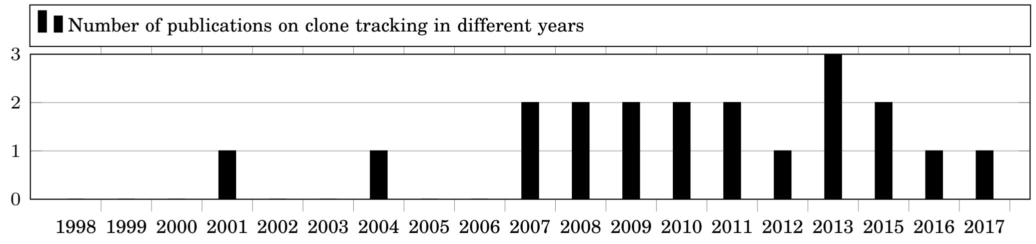

Fig. 3. Number of publications on clone tracking in different years

The first ever study on clone synchronization was done by Miller and Myer [92]. They implemented a tool that supports interactive simultaneous editing of multiple text regions that are similar to one another. The programmer first defines a set of text regions by manually selecting those. These regions were called records by Miller and Myer. After defining the record set if a programmer edits one record, the other records in the same set also experience the same edit. Simultaneous editing was integrated with an editor called LAPIS [91].

Toomim et al. [137] implemented a prototype tool called Linked Editing as an extension of XEmacs. The tool supports simultaneous editing of multiple clone fragments in a clone group. The user first selects the clone fragments that need to be linked together and then applies the tool to link those. After the linking, any further edits to any of the linked clone fragment will be automatically reflected to the other clone fragments in the group. Toomim et al did not apply any particular clone detector. Selecting groups of duplicated code fragments and linking them need to be done by the programmers. Also, such an approach can only be applied to Type 1 clones. Consistent updating of Type 2 and Type 3 clones were not considered by Toomim et al. [137].

Nguyen et al. [105] developed a clone-aware software configuration management system, Clever, that represents code clones as subtrees in ASTs and provides supports for clone detection and synchronization. Clever provides suggestions for clone synchronization. It cannot track clones through evolution. Wit et al. [25] proposed a mechanism called CloneBorard for instantly identifying if a programmer is making changes to code clones. They also proposed strategies for resolving inconsistencies in clones.

Lin et al. [82] implemented a tool called CCDemon which is capable of synchronizing only the copy-paste induced code clones. CCDemon mines the software evolution history to identify synchronizing modifications in the copy-paste clones, and uses these modifications for suggesting changes to pasted clones in future. The authors applied the tool on five open source software systems and found that it can identify 96.9% of the to be modified positions in the pasted code and can suggest 75% of the required modifications.

6.2. Clone synchronization with clone tracking

The studies that we will discuss in this subsection investigated supporting clone synchronization through clone tracking. Clone tracking through evolution helps us identify how clones co-changed in the past. This co-change information can help us suggest synchronizing changes in future.

Duala-Ekoko and Robillard [32; 30; 31] proposed a clone tracking technique by introducing the concept of clone region descriptor (CRD). CRD is used to uniquely describe a clone fragment residing in a method. It consists of file name, class name, method name, and relative location of a clone fragment in the method. CRD is independent of the text in the clone fragment. However, the concept of CRD works only for Type 1 and Type 2 clones. Duala-Ekoko and Robillard implemented a clone tracking tool called ‘CloneTracker’ that works on the basis of CRD. CloneTracker relies on the clone detector SimScan. In order to track code clones using CloneTracker, the programmers are to manually select which clones they want to track. While code clones are selected for tracking, CloneTracker builds a clone model on the basis of these selected clone fragments and tracks this model as well as the code clones during evolution. When any change occurs to any of these tracked clone fragments, CloneTracker notifies the programmers about the other fragments in the same clone group, supports consistent updating of the clone fragments, and also updates the clone model. The clone model created by CloneTracker can be used for collaboration among the team members of the project. The drawback of CloneTracker is that it cannot track clones if they are moved to a different place in the code-base.

Jablonski and Hou [58] implemented a clone tracking tool called CReN as an Eclipse plug-in which is capable of tracking copy/paste clones and consistent renaming identifiers. When a programmer copies a code fragment and pastes it any where in the code-base, CReN can identify this activity and tracks both the original (from which the copy was made) and pasted code fragments. CReN also infers a set of rules on the basis of the relationships between the identifiers of these fragments. CReN only identifies and tracks copy/paste induced clones, and it does not use any external clone detector to detect the clones. However, code clones are not only created by copy/paste activities. Many code clones can be created accidentally. Dissimilar code fragments might become similar because of changes during software evolution. These code clones are ignored by CReN. Also, copy/pasted fragments might not always appear as clones by definition after the programmer has made changes to the pasted fragment. There is no implication on how CReN can treat these fragments.

Nguyen et al. [104] developed a tool called JSync which is capable of incremental detection and tracking of code clones, detecting changes to code clones, notifying developers about changes to clones, and consistently updating (i.e., synchronizing) clone groups. JSync works on the ASTs of a code-base and represents changes at tree editing scripts. The authors applied JSync on Bellons benchmark database and found that its synchronization accuracy varies between 70% to 90%. JSync is capable of handling mainly two types of clones (Type 1 and Type 2) in Java systems.

Cheng et al. [19] developed a clone synchronization tool called CCSync which is capable of synchronizing structurally dissimilar clones through a complex matching of ASTs of the clone fragments. The working procedure of CCSync involves detecting synchronization rules among clone fragments. CCSync can identify code clones that are suitable for synchronization with a precision of 92% and a recall of 84%. It works on Type 1, Type 2, and Type 3 code clones of Java systems only.

6.3. Clone tracking without facilities for synchronization

A number of studies only investigated clone tracking. The aim of these studies is to track the clone genealogies through evolution. The tools implemented in these studies were not associated with IDEs, and thus, these tools cannot facilitate clone synchronization. We discuss the studies in the following paragraphs.

Harder and Göde [45] introduced the tool CYCLONE which is capable of detecting and showing the clone genealogies from the software evolution history. CYCLONE works on the clone detection results of iClones [43]. Although CYCLONE detects and shows clone genealogies, it cannot facilitate simultaneous editing of clone fragments in the IDE. Also, as it is not integrated with the IDE, it cannot notify programmers when they attempt to modify clone fragments.

Saha et al. [118] developed a clone genealogy extractor called gCad. It can form clone genealogies considering the clone results of the NiCad clone detector [23]. Saha et al. [117] also compared the genealogy detection capability of gCad with that of CYCLONE

[45] and CloneTracker [31], and show that gCad is superior to the other two tools in extracting clone genealogies. However, gCad was not integrated with any IDE, and thus, it cannot support programmer notification when clone fragments get changed. Also, it cannot facilitate simultaneous editing of code clones.

Higo et al. [48] enhanced the CRD (Clone Region Descriptor) based clone tracking technique which was proposed by Duala-Ekoko and Robillard [32]. The drawback of the CRD based technique is that it cannot track code clones if they are moved to a different place in the code-base. Higo et al. enhanced this technique to make it capable of realizing movements of code clones. They experimented on two open source subject systems and found 44 clone classes which could not be tracked by the original CRD based technique. However, the enhanced technique could track these clone classes with a precision of 91%.

Ci et al. [22] proposed an algorithm for mapping clone groups across multiple revisions of software systems. Their algorithm is based on CRD [30] and they applied it on the clone detection results of NiCad [23] clone detector. Their experiment on three software systems indicates that their proposed algorithm can efficiently track clone groups across software revisions.

Bakota [2] investigated the evolution of code clones across software revisions in order to identify faults due to inconsistent changes in code clones. The evolution of a software system called jEdit was investigated in this research, and it was found that around half of the smells reported during evolution were caused by inconsistent changes to code clones. Lozano and Wermelinger [85; 84] tracked clone evolution for comparing the change-proneness of clone code with that of non-clone code and found that clone code exhibits a higher change-proneness.

6.4. Ranking code clones for tracking

A software system may contain a huge number of code clones. A large portion of these code clones might not be suitable for tracking. Identifying the ones that can be important for tracking is a challenge. Only one study investigated this issue. We discuss this in the following paragraph.

Mondal et al. [97] investigated ranking of the co-change candidates of code clones from the perspective of clone tracking. When a programmer attempts to make changes to a particular clone fragment, they can find which other clone fragments in the same clone class have high possibilities of getting co-changed with that particular fragment. Mondal et al. also ranked these other clone fragments (i.e., the co-change candidates) on the basis of their evolutionary coupling. They investigated two types of ranking: frequency ranking and recency ranking of the co-change candidates. According to their analysis, recency ranking performs better than frequency ranking mechanism when ranking co-change candidates for code clones.

7. QUALITATIVE ANALYSIS OF THE CLONE TRACKING TOOLS

We perform a qualitative analysis of the clone tracking tools (i.e., Table XII) on the basis of the following six features.

— Tool's capability in synchronizing the clone fragments in a clone class
— Tool's capability in automatic tracking of clone fragments through evolution
— Tools capability in automatically notifying programmers about changes in the tracked clone fragments
— Tool's capability in handling different types of clones
— Language coverage of the tool
— Tool's dependency on clone detector

Table XII. Clone Tracking Tools

Tool	**Authors**	**Description**
Simultaneous Editing	Miller and Myer [92]	This tool facilitates clone synchronization through simultaneous editing of multiple clone fragments selected by a programmer.
Linked Editing	Toomim et al. [137]	This tool facilitates the programmers to define groups of identical code clones and editing of these code clones simultaneously.
CloneTracker	Duala-Ekoko and Robillard [31]	CloneTracker facilitates tracking of user selected clone classes throughout the evolution using CRD (Clone Region Descriptor) based clone tracking technique. It can notify programmers if changes were made to clone regions tracked by it. CloneTracker is implemented as a plug-in for Eclipse IDE.
CReN	Jablonski et al. [58]	CReN is capable of tracking copy/paste induced clones and enforcing consistent renaming of identifiers in such clones. CReN was implemented as an Eclipse plug-in.
CYCLONE	Harder and Göde [45]	CYCLONE is capable is detecting and showing genealogies of code clones detected by iClones [43] clone detector. It was implemented as an standalone tool and it does not support clone synchronization.
gCad	Saha et al. [118]	gCad helps us in detecting clone genealogies and analyzing clone evolution. It works on the clone detection results of the NiCad clone detector [23]. gCad was also implemented as a standalone tool. It does not support clone synchronization.
JSync	Nguyen et al. [104]	JSync detects and tracks code clones incrementally, senses changes to code clones, notifies programmers about such changes, and helps programmers in updating code clones consistently.
CCSync	Cheng et al. [19]	CCSync can synchronize structurally dissimilar code clones through a complex matching of their ASTs. For suggesting synchronizing changes, CCSync detects and analyzes synchronization rules among code clones.
CCDemon	Lin et al. [82]	CCDemon was developed for synchronizing copy/paste induced code clones. It mines the software evolution history to identify synchronizing modifications in copy/paste clones and uses these modifications to suggest synchronizing changes in future.

Table XIII. Capabilities of the tools in synchronizing code clones

Clone Tracking Tool	**Simultaneous editing capability**
Simultaneous Editing [92]	It **supports clone synchronization** through simultaneous editing of the programmer selected identical code clones.
Linked Editing [137]	It **facilitates synchronization** of the programmer selected code clones.
CloneTracker [31]	It **supports synchronization** of the clone fragments detected by the clone detector SimScan.
CReN [58]	It **supports clone synchronization** in a restricted way. It only provides consistent renaming facility of the identifies in the copy/paste induced clones.
CYCLONE [45]	It **does not support clone synchronization**, because it was not integrated with an IDE.
gCad [118]	gCad **does not support synchronization** of code clones. It was not integrated with an IDE.
JSync [104]	JSync **supports clone synchronization** through matching the ASTs of the clone fragments.
CCSync [19]	CCSync **supports synchronizing code clones** through a complex matching of their ASTs and identifying synchronization rules.
CCDemon [82]	CCDemon **supports synchronization** of the copy/paste induced code clones through learning from clone synchronization history.

In the following paragraphs we perform a comparative analysis of clone tracking tools on the basis of the features mentioned above.

7.1. Analysis of the tools regarding their capabilities in clone synchronization

From Table XIII we see that seven tools (i.e., excluding CYCLONE and gCad) facilitate clone synchronization; however, the tools, CReN and CCDemon, facilitate it in a

Table XIV. Capabilities of the tools in tracking code clones through evolution

Clone Tracking Tool	**Clone tracking capability**
Simultaneous Editing [92]	It **does not support tracking** of clone fragments through evolution.
Linked Editing [137]	It **does not support clone tracking** through evolution.
CloneTracker [31]	It **supports clone tracking** through evolution using the technique called CRD (Clone Region Descriptor).
CReN [58]	It **supports tracking** of copy/paste induced clone fragments. It cannot consider clone fragments that get created in ways other than copy/pasting.
CYCLONE [45]	It **supports clone tracking** through detection of clone genealogies.
gCad [118]	It **supports clone tracking** by detecting clone genealogies from software evolution history.
JSync [104]	It **supports clone tracking** through incremental detection of code clones.
CCSync [19]	It **supports clone tracking** through mining synchronization rules among clones.
CCDemon [82]	It **does not support clone tracking** during software evolution.

Table XV. Capabilities of the tools in notifying programmers automatically

Clone Tracking Tool	**Automatic programmer notification**
Simultaneous Editing [92]	It **does not provide notifications** to the programmers when clone fragments get changed, because it cannot track clone evolution.
Linked Editing [137]	It **does not provide notifications** to the programmers when clone fragments get changed, because it cannot track clone evolution.
CloneTracker [31]	It **notifies the programmer** when a clone fragment tracked by it gets modified.
CReN [58]	It **supports programmer notification** in a restricted way. When a variable name in a copy/paste clone fragment gets changed, it notifies programmers to help them make similar changes to the corresponding variables in the other clone fragments in the same clone group.
CYCLONE [45]	It was not implemented as an IDE Plug-in, and thus, it **cannot support programmer notification**.
gCad [118]	As gCad was not integrated with any IDE, it **does not support programmer notification** during software development.
JSync [104]	It **supports programmer notification** through sensing changes to code clones.
CCSync [19]	It **notifies programmers** about changes in code clones, and also, suggests synchronizing changes to those.
CCDemon [82]	It **supports notification** for copy/paste induced clones.

restricted way. CReN only supports consistent renaming of the identifiers involved in the clone fragments. Other types of editing are not supported by CReN. CCDemon can only support synchronization of copy/paste induced clones.

7.2. Analysis regarding the capability of the tools in tracking code clones through evolution

Table XIV demonstrates the clone tracking capabilities of the tools during system evolution. We see that three tools (Simultaneous Editing [92], Linked Editing [137], and CCDemon [82]) do not support tracking of code clones during system evolution. Thus, these tools are not really clone tracking tools. Clone tracking facility is available in CloneTracker [31], CYCLONE [45], gCad [118], JSync [104], and CCSync [19]. CReN also support clone tracking; however, in a restricted way. It only supports tracking of copy/paste induced clones. Clone fragments created in ways other than copy/pasting cannot be tracked by CReN. Saha et al. [117] shows that gCad is more efficient in detecting clone genealogies compared to CloneTracker and CYCLONE.

Table XVI. Capabilities of the tools in handling different clone-types

Clone Tracking Tool	**Consideration of clone-types**
Simultaneous Editing [92]	It only supports synchronization of **Type 1 (identical) clones**.
Linked Editing [137]	It only supports synchronization of **Type 1 (identical) clones**.
CloneTracker [31]	It supports synchronization and tracking of **Type 1 and Type 2 clones**.
CReN [58]	It supports consistent renaming and tracking of **Type 1 and Type 2 clones**.
CYCLONE [45]	It supports genealogy detection of **Type 1, Type 2, and Type 3** clones.
gCad [118]	It facilitates genealogy detection of **Type 1, Type 2, and Type 3** clones.
JSync [104]	It supports detection and synchronization of **Type 1 and Type 2** clones.
CCSync [19]	It supports synchronization of **Type 1, Type 2, and Type 3** clones.
CCDemon [82]	It supports synchronization of **copy/paste induced Type 1** clones.

Table XVII. Capabilities of the tools in handling programming languages

Clone Tracking Tool	**Programming language capability**
Simultaneous Editing [92]	It supports simultaneous editing for **Java** and **HTML** code.
Linked Editing [137]	It supports simultaneous editing of code clones in **Java** systems only.
CloneTracker [31]	It supports simultaneous editing and tracking of code clones in **Java** systems only.
CReN [58]	It supports consistent renaming and tracking of copy/paste induced clones in **Java** systems only.
CYCLONE [45]	It supports clone genealogy detection and analysis in **Java** systems only.
gCad [118]	gCad helps us detect and analyze clone genealogies considering multiple programming languages including **Java, C, and C#**.
JSync [104]	It works only on software systems written in **Java**.
CCSync [19]	It supports clone tracking and synchronization in **Java** systems only.
CCDemon [82]	It supports clone synchronization in **Java** systems only.

7.3. Analysis regarding the capability of the tools in notifying programmers

We show the programmer notification capabilities of the tools in Table XV. We see that two tools: Simultaneous Editing [92] and Linked Editing [137] do not provide supports for notifying programmers when clone fragments get changed. The reason is that these two tools cannot track clone evolution. Although the tools CYCLONE [45] and gCad [118] facilitates genealogy detection, they do not support programmer notification during development because they are not integrated with any IDEs. We see that CloneTracker, JSync, CCSync, and CCDemon support programmer notification. CReN also provides limited support for programmer notification. When a variable name in a copy/paste clone fragment gets changed, the other fragments in the same group are notified to the programmer to ensure similar changes to the corresponding variables in these other fragments. CCDemon also supports programmer notification only for copy/paste clones.

7.4. Clone-type centric analysis of the tools

Table XVI shows a clone-type centric comparison of the clone tracking tools. We see that each of the two tools: CloneTracker, and CReN supports tracking of Type 1 and Type 2 clones. Both CYCLONE and gCad support genealogy detection of all three clone-types: Type 1, Type 2, and Type 3. CCSync supports tracking of all three clone types (Type 1, Type 2, and Type 3). JSync only supports synchronization of minor modifications (such as a single insert or delete) in Type 3 clones. CCDemon supports synchronization of copy/paste induced Type 1 clones. The remaining two tools only support simultaneous editing of Type 1 clones.

7.5. Programming language centric analysis of the tools

We show the programming language capabilities of the clone tracking tools in Table XVII. We see that the tool called Simultaneous Editing [92] supports two programming

Table XVIII. Capabilities of the tools in handling code clones from different clone detectors

Clone Tracking Tool	**Capability in handling clone detector output**
Simultaneous Editing [92]	It does not apply any clone detector for detecting code clones. It completely relies on the manual detection of the programmers. A programmer first manually selects the code clones she intends to work on. The tool can then support editing those code clones simultaneously.
Linked Editing [137]	Like the Simultaneous Editing tool [92], this tool also relies on the manual selection of the programmers to realize code clones to edit simultaneously.
CloneTracker [31]	This tool is capable of tracking code clones detected by the clone detector SimScan.
CReN [58]	This tool cannot work on the clone detection results of any clone detector. It can detect copy/paste induced code clones by itself. A copy/paste activity performed by a programmer triggers the tool. The tool then stores the code clones and continues to track them for ensuring consistent renaming in future.
CYCLONE [45]	It can work on clone detection results from iClones [43].
gCad [118]	It works on clone detection results from NiCad [23].
JSync [104]	It incrementally detects code clones by itself.
CCSync [19]	It works on top of ConQAT [63].
CCDemon [82]	It detects copy/paste induced clones by using JCCD [14]

languages: Java and HTML. Each of the other tools except gCad [118] only supports Java. gCad supports genealogy detection of code clones considering three programming languages: Java, C, and C#. Future research on how to support more languages by the tracking tools is important.

7.6. Analysis for the capability of the tools in handling clone results from clone detectors

We now discuss the capabilities of the tracking tools in handling clone results from clone detectors. From Table XVIII we see that each of the two tools: Simultaneous Editing [92] and Linked Editing [137] depends on manual identification of the programmers to realize code clones for simultaneous editing. CReN also does not apply any clone detector for detecting clones for tracking. It can detect code clones by itself. However, it can only realize code clones from the copy/paste activities of the programmers. Code clones can be created in many other ways such as forking, merging of similar software systems, design reuse etc. Such code clones are also important for tracking. CReN cannot consider such code clones for tracking. We think it is better to use a clone detector for detecting code clones so that we do not disregard particular clones from consideration. From Table XVIII we see that the tool called CloneTracker applies the clone detector SimScan for clone detection. The tools CYCLONE [45] and gCad [118] can work on clone results from iClones [43] and NiCad [23] respectively. While JSync detects clones by itself, CCSync works on top of ConQAT. CCDemon uses JCCD [14] for clone detection. Future research on customizing the clone tracking tools to make them capable of working with other state-of-the-art clone detectors can make a significant contribution towards clone management.

7.7. Overall analysis of the clone tracking tools

We draw Table XIX accumulating all the features, and demonstrating tool capabilities with respect to these features. We see that CCSync can detect and track all three types of clones (Type 1, Type 2, and Type 3). The tools CYCLONE [45] and gCad [118] are also promising for clone genealogy detection. Specially, gCad is the most promising clone genealogy detector. However, these two tools (i.e., CYCLONE and gCad) were not integrated with IDE, and thus, do not support programmer notification and synchronization of clone fragments. Future research involving integration of these two tools with an IDE to facilitate synchronization and programmer notification can make an important contribution in clone management.

Table XIX. Clone Tracking Features

	Clone synchronization	Tracking clones through evolution	Programmer notification	Capability in handling clone-types	Language support	Dependency on a clone detector
Simultaneous Editing [92]	Yes	No	No	T1	Java, HTML	No
Linked Editing [137]	Yes	No	No	T1	Java	No
CloneTracker [31]	Yes	Yes	Yes	T1 T2	Java	SimScan
CReN [58]	Yes	Yes	Yes	T1 T2	Java	No
CYCLONE [45]	No	Yes	No	T1 T2 T3	Java	iClones
gCad [118]	No	Yes	No	T1 T2 T3	Java C C#	NiCad
JSync [104]	Yes	Yes	Yes	T1 T2	Java	No
CCSync [19]	Yes	Yes	Yes	T1 T2 T3	Java	ConQAT
CCDemon [82]	Yes	No	Yes	T1	Java	JCCD

Table XX. Capabilities of the tools in different tracking scenarios

Clone Tracking Tool	**Scenario 1**	**Scenario 2**	**Scenario 3**	**Scenario 4**
CloneTracker [31]	Y	N	Y	N
CReN [58]	Y	Y	Y	N
CYCLONE [45]	Y	Y	Y	N
gCad [118]	Y	Y	Y	N
JSync [118]	Y	Y	Y	N
CCSync [118]	Y	Y	Y	N
Y = The tool can track in the scenario. N = The tool cannot track in the scenario.				

7.8. Comparative analysis of the clone tracking tools on the basis of tracking scenarios

We now analyze the clone tracking tools on the basis of a number of tracking scenarios. We define the scenarios emphasizing tool capabilities in detecting clone genealogies. From Table XIX we see that the tools: Simultaneous Editing [92], Linked Editing [137], and CCDemon [82] cannot track clone fragments through evolution. We limit our scenario-based analysis on the remaining six tools. The scenarios have been defined below.

— **Scenario 1:** Tracking a clone fragment when it neither experiences changes nor is moved to a different location in the code-base.
— **Scenario 2:** Tracking a clone fragment when it is moved to a different place.
— **Scenario 3:** Tracking a clone fragment when it experiences changes.
— **Scenario 4:** Tracking a clone fragment when it becomes a non-clone fragment.

Table XX shows the capabilities of the four clone trackers in different tracking scenarios. We see that most of the clone trackers are capable of clone tracking in the first three scenarios. They cannot track the evolution of a clone fragment in the last scenario. Tracking a clone fragment in the last scenario (Scenario 4) where a clone fragment becomes a non-clone fragment is also important. It might be the case that a particular clone fragment has become a non-clone fragment because the other fragments in its group have been deleted. In such a case, it is important to track this non-clone fragment (i.e., which was a clone fragment previously) because it can again make a clone group with one or more other code fragments. Also, the deleted code fragments might reappear. If we cannot track such a non-clone fragment, we will miss its evolution during its non-cloned period. Tracking a clone fragment during its non-cloned period can help us analyze late propagations [6] in code clones. From Table XX we also see that CloneTracker [31] cannot track a clone fragment if it is moved to a different place. Higo et al. [48] performed an investigation emphasizing this problem of CloneTracker and proposed an improved tracking mechanism called Enhanced CRD which was capable of tracking clone fragments if they were moved to different places.

Using Enhanced CRD in CloneTracker can improve its clone tracking capability. Table XX indicates that CYCLONE and gCad are promising clone trackers on the basis of different scenarios. However, as we previously discussed, these two tools are not integrated with IDEs, and thus, cannot provide programmer notification and simultaneous editing facilities. Also, CReN is capable of tracking only the copy/paste clones.

8. FUTURE RESEARCH POSSIBILITIES ON CLONE REFACTORING AND TRACKING

From our analysis on the existing clone refactoring and tracking research we feel the necessity of further research in the following directions:

8.1. Post-refactoring analysis on the effects of clone refactoring on system performance

Analyzing the effect of clone refactoring on system performance is important. Rajapakse and Jarzabek [109] showed that clone refactoring negatively affects the performance of web applications written in PHP and significantly increases the testing effort. Such studies should also be performed considering software systems developed in other programming languages such as: Java, C, and C#. It is also important to investigate whether clone refactoring affects energy consumptions of software systems. A recent study [81] shows that small changes in the code-base can cause significant difference in the energy consumption of a software system. Mahmoud and Niu [86] discovered that removal of code clones through refactoring can negatively affect requirements to code traceability. We realize that clone refactoring should be investigated with a focus on software requirements engineering. The particular types of refactoring that are likely to be harmful for code traceability need to be identified so that software engineers can avoid such types of refactoring.

8.2. Increasing language support of the clone refactoring tools

Most of the existing clone refactoring tools support refactoring code clones from only one programming language. However, projects involving multiple programming languages exist. Refactoring code clones across multiple programming languages is still a challenge. The existing studies did not investigate this important issue. The existing clone refactoring tools apply clone detectors that can detect clones from multiple languages. For example, we consider the clone refactoring tool DCRA [38] that applies NiCad [23] clone detector for detecting clones. While NiCad [23] supports Java, C, C#, and Python programming languages, DCRA only supports clone refactoring in Java systems. We understand that different programming languages have different constructs, designs, and coding styles. Thus, refactoring patterns should be different for different programming languages. However, the same refactoring pattern might be applicable to multiple languages that support similar coding style. Future research on which refactoring patterns can be commonly applicable to which programming languages, and which are the language specific patterns can be much important.

8.3. Refactoring Type 4 clones

By the definition [110; 115], Type 4 clones (i.e., semantically similar code fragments) do not have syntactic similarity. Thus, the traditional refactoring tools cannot be used for refactoring semantic clones. However, if two code fragments in two places of a code-base are detected as semantic clones, then their refactoring might involve discarding one clone fragment and using the other one in both places possibly through method calls. Deciding which clone fragment to remove and which one to use should depend on the run-time complexity, coding standards, and code comprehensibility of the candidate Type 4 clone fragments. Intuitively, the clone fragment with lower run-time complexity should be more promising compared to the more complex one. Two studies [127; 79] have investigated refactoring functionally similar clones. However, these studies do not consider run-time complexity of the candidate clones. Future investigations on automatically comparing the run-time complexity as well as the comprehensibility of two semantically similar code fragments can add much to clone refactoring research.

8.4. Inter-project clone refactoring

The existing clone refactoring studies and techniques only deal with intra-project clone refactoring (i.e., refactoring of code clones in the same software system). However, different software systems that are written in the same programming language may have common code fragments. These code fragments are known as inter-project clones. It is important to detect and refactor inter-project code clones. A code fragment (for example, a method) that has been used for implementing more than one software systems should be given importance, because this code code fragment may again be used for implementing another project in future. Thus, such code fragments, that is the inter-project clones, should be managed with equal importance. Refactoring of inter-project clones can be done by developing a global library that will contain these clones and calling appropriate methods in this library in place of the corresponding code clones. A number of studies [57; 128] have been done on inter-project clone detection. Ishihara et al. [57] investigated detecting inter-project functional clones for building libraries. We think that similar studies should be done targeting block clones too.

8.5. Big-data clone refactoring

Inter-project clone detection and refactoring should be facilitated in a big-data environment empowered by Hadoop-MapReduce framework. Let us consider a particular software company where programmers are working on a number of projects. Also, a number of projects have already been developed in the company. Programmers can get coding help for their on-going projects from these already developed projects through inter-project clone detection and refactoring. Inter-connecting all the already developed as well as on-going projects, parallel detection and refactoring of inter-project code clones from these projects can only be facilitated in a big-data environment. A number of studies [119; 129; 120; 128] have been done on big-data clone detection. Future research on big-data clone refactoring has much potential to advance the state-of-the-art of clone detection and refactoring.

8.6. Increasing language support of the clone tracking tools

Most of the clone tracking tools only support tracking of code clones in Java systems. The tool Simultaneous Editing [92] also supports HTML. However, this tool cannot track code clones through evolution. The tool gCad [118] supports Java, C, and C#. However, it cannot support simultaneous editing, and programmer notification. Future research on enhancing clone trackers so that they can deal with code clones from different programming languages such as: C, C++, C#, and Python can make an important contribution towards clone management.

8.7. Clone tracking in a big-data environment

An alternative of automatic tracking of all important code clones in a code-base is instant detection of code clones in a time efficient manner. Let us assume a programmer is working on a piece of code. If we can detect all the duplicate copies of this piece of code instantly, then it might obviate the necessity of clone tracking. Clone tracking requires maintaining a clone database. Also, the evolution of each of the clone fragments need to be tracked through different revisions. For this purpose we need a clone genealogy analyzer. However, instant detection of code clones can possibly eliminate these necessities. In order to facilitate instant clone detection, we possibly need a big-data environment empowered by Hadoop-MapReduce framework. In the parallel computing environment of Hadoop we might be able to detect code clones instantly. Investigations in this direction can be much important.

8.8. Comparing the benefits of clone tracking and refactoring

The clone fragments in a particular clone class can be refactored or tracked. Refactoring removes all instances of the clone fragments by a single instance, whereas tracking does not remove any instance of clone fragments but ensures consistent updates of the fragments. Refactoring is beneficial because it obviates the necessity of implementing the same change to multiple clone fragments. Refactoring also reduces the size of the code-base. Moreover, refactoring of a clone class might not always be possible, however, tracking of the class is always possible. In such a situation it is important to perform a trade-off analysis of the benefits gained from refactoring and tracking. We should perform this trade-off analysis in a way that is similar to the investigation of Rajapakse et al. [109]. We should have two copies of the same software system. We should refactor a number of clone classes in one copy, and those clone fragments in the other copy should tracked for a certain period of evolution. Then we should analyze the evolution history considering the following points: (1) time and effort required for refactoring, (2) time and effort for updating code fragments after refactoring, (3) time and effort for updating clone fragments under tracking, (4) system performance after refactoring, (5) system performance while tracking. Future research on comparing refactoring and tracking benefits through evolution analysis can be much important for efficient software maintenance.

9. ANSWERING THE RESEARCH QUESTIONS

We conduct our survey with an aim to answer the three research questions listed in Table I. In answer to the first research question (RQ 1) we can say that we can categorize the existing studies on clone refactoring and tracking. Sections 4 and 6 mention these categories and discuss the clone refactoring and tracking studies on the basis of these categories. Our discussions demonstrate the extent to which each of these categories has been explored. We have also provided possible suggestions for further exploration. In answer to our second research question (RQ 2) we can state that we have identified six features for qualitative analysis of the clone refactoring and tracking tools. Sections 5 and 7 mention these features and make a comparison of the refactoring and tracking tools on the basis of these features. We have also answered our third research question (RQ 3) by discussing a number of future research possibilities in clone refactoring and tracking. Section 8 contains this discussion. Our study findings can be helpful for researchers aiming to explore the area of clone refactoring and tracking.

10. CONCLUSION

In this paper we present our survey on the existing research, tools, and techniques on clone refactoring and tracking. We categorize the studies on the basis of their research directions and discuss the extent to which each category has been explored. We also identify the existing clone refactoring and tracking tools and make a comparison among these tools on the basis of their features. From our survey on clone refactoring we realize that automatic refactoring cannot eradicate the necessity of manual effort regarding finding refactoring opportunities, and testing system behaviour after the application of refactoring. Research shows that post refactoring testing can require a significant amount of time and effort from the quality assurance engineers. There is a significant lack of research on how clone refactoing can affect system performance. Future investigations in this direction will add much value to clone refactoring research. We also feel the necessity of future research towards real-time detection, and tracking of code clones in a big-data environment.

REFERENCES

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

T. Bakota. Tracking the evolution of code clones. In *SOFSEM*, pages 86 – 98, 2011.

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

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

M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *WCRE*, pages 326 – 336, 1999.

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

L. Barbour, F. Khomh, and Y. Zou. An empirical study of faults in late propagation clone genealogies. *Journal of Software: Evolution and Process*, 25(11):1139 – 1165, 2013.

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

H. Basit and S. Jarzabek. Towards structural clones: Analysis and semi-automated detection of design-level similarities in software. In *VDM Verlag Dr. Müller*, 2010.

H. A. Basit, H. S. Khan, F. Hamid, and I. Suhail. Tool support for managing method clones. In *IWSC*, pages 40–46, 2015.

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

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

Y. Bian, X. Su, and P. Ma. Identifying accurate refactoring opportunities using metrics. In *ICSCTEA*, pages 141 – 146, 2014.

B. Biegel and S. Diehl. Jccd: a flexible and extensible api for implementing custom code clone detectors. In *ASE*, 2010.

S. Bouktif, G. Antoniol, M. Neteler, and E. Merlo. A novel approach to optimize clone refactoring activity. In *GECCO*, pages 1885 – 1892, 2006.

C. Brown and S. Thompson. Clone detection and elimination for haskell. In *PEPM*, pages 111 – 120, 2010.

D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, and N. A. Kraft. Measuring the efficacy of code clone information in a bug localization task: An empirical study. In *ESEM*, pages 20 – 29, 2011.

Z. Chen, M. Mohanavilasam, Y. W. Kwon, and M. Song. Tool support for managing clone refactorings to facilitate code review in evolving software. In *COMPSAC*, pages 288 – 297, 2017.

X. Cheng, H. Zhong, Y. Chen, Z. Hu, and J. Zhao. Rule-directed code clone synchronization. In *ICPC*, pages 1 – 10, 2016.

E. Choi, N. Yoshida, and K. Inoue. What kind of and how clones are refactored? : A case study of three oss projects. In *WRT*, pages 1 – 7, 2012.

E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano. Extracting code clones for refactoring using combinations of clone metrics. In *IWSC*, pages 7 – 13, 2011.

M. Ci, X. h. Su, T. t. Wang, and P. j. Ma. A new clone group mapping algorithm for extracting clone genealogy on multi-version software. In *Third International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pages 848–853, 2013.

J. R. Cordy and C. K. Roy. The nicad clone detector. In *ICPC Tool Demo*, pages 219 – 220, 2011.

N. Göde D. Steidl. Feature-based detection of bugs in clones. In *IWSC*, pages 76 – 82, 2013.

M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *ICSM*, pages 169 – 178, 2009.

M. Deepika and S. Sarala. Implication of clone detection and refactoring techniques using delayed duplicate detection refactoring. *International Journal of Computer Applications*, 93(6): 5 – 10, 2014.

F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model clone detection in practice. In *IWSC*, pages 57 – 64, 2010.

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

S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*, pages 166 – 177, 2000.

E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE*, pages 158 – 167, 2007.

E. Duala-Ekoko and M. P. Robillard. Clonetracker: Tool support for code clone management. In *ICSE*, pages 843 – 846, 2008.

E. Duala-Ekoko and M. P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology*, 20(1):1 – 31, 2010.

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

R. Ettinger and S. Tyszberowicz. Duplication for the removal of duplication. In *SANER*, pages 53 – 59, 2016.

R. Ettinger, S. Tyszberowicz, and S. Menaia. Efficient method extraction for automatic elimination of type-3 clones. In *SANER*, pages 1 – 11, 2017.

M. Fanqi. Using self organized mapping to seek refactorable code clone. In *CSNT*, pages 851 – 855, 2014.

W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake. Variant-preserving refactorings for migrating cloned products to a product line. In *SANER*, pages 316 – 326, 2017.

F. A. Fontana, M. Zanoni, and F. Zanoni. A duplicated code refactoring advisor. *Agile Processes, in Software Engineering, and Extreme Programming*, pages LNBIP(212): 3 – 14, 2015.

M. Fowler, K. Beck, J.Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-wesley, 1997.

N. Göde. Clone removal: Fact or fiction? In *IWSC*, pages 33 – 40, 2010.

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

N. Göde and R. Koschke. Incremental clone detection. In *CSMR*, pages 219 – 228, 2009.

N. Göde and Rainer Koschke. Frequency and risks of changes to clones. In *ICSE*, pages 311 – 320, 2011.

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

T. Hatano and A. Matsuo. Removing code clones from industrial systems using compiler directives. In *ICPC*, pages 336 – 345, 2017.

B. Hauptmann, E. Juergens, and V. Woinke. Generating refactoring proposals to remove clones from automated system tests. In *ICPC*, pages 115 – 124, 2015.

Y. Higo, K. Hotta, and S. Kusumoto. Enhancement of crd-based clone tracking. In *IWPSE*, pages 28 – 37, 2013.

Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. *Product Focused Software Process Improvement*, (LNCS 3009):220 – 233, 2004.

Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support tool for code clone. In *3-WoSQ*, pages 1 – 4, 2005.

Y. Higo and S. Kusumoto. Code clone detection on specialized pdgs with heuristics. In *CSMR*, pages 75 – 84, 2011.

Y. Higo and S. Kusumoto. Identifying clone removal opportunities based on co-evolution analysis. In *IWPSE*, pages 63 – 67, 2013.

Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE*, (20): 435 – 461, 2008.

K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *CSMR*, pages 53 – 62, 2012.

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

K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee. Experience of finding inconsistently-changed bugs in code clones of mobile software. In *IWSC*, pages 94 – 95, 2012.

T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects. In *WCRE*, pages 387 – 391, 2012.

P. Jablonski and D. Hou. Cren: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *OOPSLA*, pages 16 – 20, 2007.

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

L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE*, pages 55 – 64, 2007.

E. Juergens. Research in cloning beyond code: a first roadmap. In *IWSC*, pages 67 – 68, 2011.

E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can clone detection support quality assessments of requirements specifications? In *ICSE*, pages 79 – 88, 2010.

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

N. Juillerat and B. Hirsbrunner. An algorithm for detecting and removing clones in java code. *ELECTRONIC COMMUNICATIONS OF EASST*, (3):1 – 12, 2006.

T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. In *IEEE Transactions on Software Engineering*, volume 28(7):654 – 670, 2002.

J. Kanwal, K. Inoue, and O. Maqbool. Refactoring patterns study in code clones during software evolution. In *IWSC*, pages 1 – 2, 2017.

C. Kapser and M. W. Godfrey. “cloning considered harmful” considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6): 645 – 692, 2008.

M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *SIGSOFT/FSE*, pages 371 – 372, 2010.

M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study on code clone genealogies. In *FSE*, pages 187 – 196, 2005.

G. Koni-N’Sapu. A scenario based approach for refactoring duplicated code in object oriented systems. In *Diploma thesis, University of Bern*, 2001.

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

J. Krinke. Is cloned code more stable than non-cloned code? In *SCAM*, pages 57 – 66, 2008.

J. Krinke. Is cloned code older than non-cloned code? In *IWSC*, pages 28 – 33, 2011.

G. P. Krishnan and N. Tsantalis. Unification and refactoring of clones. In *CSMR-WCRE*, pages 104 – 113, 2014.

S. Lee, G. Bae, H. S. Chae, D. Bae, and Y. R. Kwon. Automated scheduling for clone-based refactoring using a competent ga. *SOFTWARE PRACTICE AND EXPERIENCE*, 41:521 – 550, 2011.

H. Li and S. Thompson. Clone detection and removal for erlang/otp within a refactoring environment. In *PEPM*, pages 169 – 177, 2009.

H. Li and S. Thompson. Incremental clone detection and elimination for erlang programs. In *FASE*, pages 356 – 370, 2011.

J. Li and M. D. Ernst. Cbcd: Cloned buggy code detector. In *ICSE*, pages 310 – 320, 2012.

X. Li, X. Su, P. Ma, and T. Wang. Refactoring structure semantics similar clones combining standardization with metrics. In *International Conference on Soft Computing Techniques and Engineering Application*, pages 361 – 367, 2014.

Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 20 – 33, 2004.

L. G. Lima, G. Melfe, F. Soares-Neto, P. Lieuthier, J. P. Fernandes, and F. Castor. Haskell in green land: Analyzing the energy behavior of a purely functional language. In *SANER*, page 12pp, 2016.

Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao. Clone-based and interactive recommendation for modifying pasted code. In *ESEC/FSE*, pages 520 – 531, 2015.

H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *APSEC*, pages 269 – 276, 2006.

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

A. Lozano and M. Wermelinger. Tracking clones’ imprint. In *IWSC*, pages 65 – 72, 2010.

A. Mahmoud and N. Niu. Supporting requirements to code traceability through refactoring. *Requirements Eng.*, (2014) 19:309329, 2013.

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

D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta. Jdeodorant: Clone refactoring. In *ICSE*, pages 613 – 616, 2016.

T. Mende, R. Koschke, and F. Beckwermert. An evaludation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):143 – 169, 2009.

N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does automated refactoring obviate systematic editing? In *ICSE*, pages 392 – 402, 2015.

R. C. Miller and B. A. Myers. Lightweight structured text processing. In *USENIX Annual Technical Conference*, pages 131 – 144, 1999.

R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference*, pages 161 – 174, 2001.

M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *SAC*, pages 1227 – 1234, 2012.

M. Mondal, C. K. Roy, and K. A. Schneider. An empirical study on clone stability. *ACM SIGAPP Applied Computing Review*, 12(3): 20 – 36, 2012.

M. Mondal, C. K. Roy, and K. A. Schneider. Automatic identification of important clones for refactoring and tracking. In *SCAM*, pages 11 – 20, 2014.

M. Mondal, C. K. Roy, and K. A. Schneider. Automatic ranking of clones for refactoring through mining association rules. In *CSMR-WCRE*, pages 114 – 123, 2014.

M. Mondal, C. K. Roy, and K. A. Schneider. Prediction and ranking of co-change candidates for clones. In *MSR*, pages 32 – 41, 2014.

M. Mondal, C. K. Roy, and K. A. Schneider. A comparative study on the bug-proneness of different types of code clones. In *ICSME*, pages 91 – 100, 2015.

M. Mondal, C. K. Roy, and K. A. Schneider. Spcp-miner: A tool for mining code clones that are important for refactoring or tracking. In *SANER*, pages 484 – 488, 2015.

B. Mourad, L. Badri, O. Hachemane, and A. Ouellet. Exploring the impact of clone refactoring on test code size in object-oriented software. In *ICMLA*, pages 586 – 592, 2017.

K. Narasimhan. Clone merge – an eclipse plugin to abstract near-clone c++ methods. In *ASE*, pages 819 – 823, 2015.

K. Narasimhan and C. Reichenbach. Copy and paste redeemed (t). In *ASE*, pages 630 – 640, 2015.

C. G. Nevill-Manning. Inferring sequential structure. In *Ph.D. dissertation, University of Waikato*, 1996.

H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008 – 1026, 2012.

T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-aware configuration management. In *ASE*, pages 123 – 134, 2009.

N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, and T. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE*, pages 276 – 286, 2009.

K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *ICSM*, pages 1 – 10, 2010.

F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *MSR*, pages 72 – 81, 2010.

D. C. Rajapakse and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *ICSE*, pages 116 – 126, 2007.

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

C. K. Roy and J. R. Cordy. A survey on software clone detection research. In *Tech Report TR 2007-541, School of Computing, Queens University, Canada*, pages 1 – 115, 2007.

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

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

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

C. K. Roy, M. F. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *CSMR-WCRE*, pages 18 – 33, 2014.

V. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *ASE*, pages 336 – 339, 2004.

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

R. K. Saha, C. K. Roy, and K. A. Schneider. gcad: A near-miss clone genealogy extractor to support clone evolution analysis. In *ICSM*, pages 488 – 491, 2013.

H. Sajnani, J. Ossher, and C. Lopes. Parallel code clone detection using mapreduce. In *ICPC*, pages 261 – 262, 2012.

H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big code. In *ICSE*, pages 1157 – 1168, 2016.

S. Sarala and M. Deepika. Unifying clone analysis and refactoring activity advancement towards c# applications. In *ICCCNT*, pages 1 – 5, 2013.

S. Schulze and M. Kuhlemann. Advanced analysis for code clone removal. In *WSR*, pages 1 – 2, 2009.

S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a refactoring guideline using code clone classification. In *WRT*, pages 1 – 4, 2008.

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

S. Shahzad, A. Hussain, and S. Nazir. A clone management framework to improve code quality of foss projects. In *C-CODE*, pages 253 – 258, 2017.

H. Störrle. Towards clone detection in uml domain models. In *ECSA*, pages 285 – 293, 2010.

X. Su, F. Zhang, X. Li, P. Ma, and T. Wang. Functionally equivalent c code clone refactoring by combining static analysis with dynamic testing. In *International Conference on Soft Computing Techniques and Engineering Application*, pages 247 – 256, 2014.

J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *ICSME*, pages 476 – 480, 2014.

J. Svajlenko, I. Keivanloo, and C. K. Roy. Big data clone detection using classical detectors: an exploratory study. *Journal of Software: Evolution and Process*, 27(6): 430 - 464, 2015.

R. Tairas. Clone detection and refactoring. In *OOPSLA*, pages 780 – 781, 2006.

R. Tairas. Clone maintenance through analysis and refactoring. In *FSE-16 (Doctoral Symp)*, pages 29 – 32, 2008.

R. Tairas and J. Gray. Sub-clone refactoring in open source software artifacts. In *SAC*, pages 2373 – 2374, 2010.

R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology*, 54 (2012) 1297 – 1307, 2012.

M. Takahashi, R. Nanba, Y. Anang, T. Uchiyama, and Y. Watanabe. A method of program refactoring based on code clone detection and impact analysis. In *SICE*, pages 673 – 678, 2016.

H. Thaller, R. Ramler, J. Pichler, and A. Egyed. Exploring code clones in programmable logic controller software. In *ETFA*, pages 1 – 8, 2017.

M. Tokunaga, N. Yoshida, K. Yoshioka, M. Matsushita, and K. Inoue. Towards a collection of refactoring patterns based on code clone categorization. In *AsianPLoP*, pages 1 – 6, 2011.

M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Symposium on Visual Languages and Human Centric Computing*, pages 173 – 180, 2004.

N. Tsantalis, D. Mazinanian, and G. P. Krishnan. Assessing the refactorability of software clones. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 41(11): 1055 – 1090, 2015.

N. Tsantalis, D. Mazinanian, and S. Rostami. Clone refactoring with lambda expressions. In *ICSE*, pages 20 – 28, 2017.

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

N. Volanschi. Safe clone-based refactoring through stereotype identification and iso-generation. In *IWSC*, pages 50 – 56, 2012.

W. Wang and M. W. Godfrey. Investigating intentional clone refactoring. *Electronic Communications of the EASST*, 63:1 – 7, 2014.

W. Wang and M. W. Godfrey. Recommending clones for refactoring using design, context, and history. In *ICSME*, pages 331 – 340, 2014.

S. Xie, F. Khomh, and Y. Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *MSR*, pages 149 – 158, 2013.

N. Yoshida, E. Choi, and K. Inoue. Active support for clone refactoring: A perspective. In *WRT*, pages 13 – 16, 2013.

N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. On refactoring support based on code clone dependency relation. In *METRICS*, pages 10 – 16, 2005.

L. Yu and S. Ramaswamy. Improving modularity by refactoring code clones: A feasibility study on linux. *ACM SIGSOFT Software Engineering Notes*, 33(2): 1 – 5, 2008.

M. F. Zibran. Analysis and visualization for clone refactoring. In *IWSC*, pages 47 – 48, 2017.

M. F. Zibran and C. K. Roy. Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach. In *ICPC*, pages 266 – 269, 2011.

M. F. Zibran and C. K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *SCAM*, pages 105 – 114, 2011.

M. F. Zibran and C. K. Roy. Towards flexible code clone detection, management, and refactoring in ide. In *IWSC*, pages 75 – 76, 2011.

M. F. Zibran and C. K. Roy. Conflict-aware optimal scheduling of prioritised code clone refactoring. *IET Software*, 7(3): 167 – 186, 2013.

M. F. Zibran, R. K. Saha, C. K. Roy, and K. A. Schneider. Genealogical insights into the facts and fictions of clone removal. *Applied Computing Review*, 13(4): 30 – 42, 2013.