Ranking Co-change Candidates of Micro-Clones

Manishankar Mondal Banani Roy Chanchal K. Roy Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
{mshankar.mondal, banani.roy, chanchal.roy, kevin.schneider}@usask.ca

ABSTRACT

Identical or nearly similar code fragments in a software system's code-base are known as code clones. Code clones from the same clone class have a tendency of co-changing (changing together) consistently during evolution. Focusing on this co-change tendency, existing studies have investigated prediction and ranking co-change candidates of regular clones. However, a recent study shows that micro-clones which are smaller than the minimum size threshold of regular clones might also need to be co-changed consistently during evolution. Thus, identifying and ranking co-change candidates of micro-clones is also important. In this paper, we investigate factors that influence the co-change tendency of the co-change candidates of a target micro-clone fragment.

We mine file level evolutionary coupling from thousands of revisions of our subject systems through mining association rules and analyze this coupling for the purpose of ranking. According to our findings on six open-source subject systems written in Java and C, consistent co-change tendency of micro-clones is influenced by file proximity of the micro-clone fragments as well as evolutionary coupling of the files containing those micro-clone fragments. On the basis of our findings we propose a composite ranking mechanism by incorporating both file proximity and file coupling for ranking co-change candidates for micro-clones and find that our proposed mechanism performs significantly better than File Proximity Ranking mechanism. We believe that our proposed ranking mechanism has the potential to help programmers in updating micro-clones consistently with less effort.

ACM Reference format:
Manishankar Mondal Banani Roy Chanchal K. Roy Kevin A. Schneider. 2019. Ranking Co-change Candidates of Micro-Clones . In *Proceedings of Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, Toronto, ON, Canada, Nov. 4–6, 2019 (CASCON '19),* 10 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Code clone has emerged as a controversial term [2, 5, 15] in the realm of software maintenance research and practice. Programmers often perform code cloning during programming for repeating common functionalities. Such an activity causes the existence of exactly or nearly similar code fragments, known as code clones [3, 7], in a software system's code-base. A group of similar code fragments forms a clone group or a clone class. Two code fragments that are similar (exactly or nearly) to each other form a clone-pair.

Code clones have a significant importance from the perspective of software maintenance and evolution. A great many studies [2, 5, 8, 11, 12, 15–18, 23–25, 27–30, 32–37, 41, 43] have been conducted on the detection [5, 9], analysis [23, 24, 36], and management [11, 33, 37] of code clones. Existing studies have revealed both positive [22, 23, 35] and negative [2, 12, 32] impacts of code clones on software evolution. Focusing on the issues related to code clones, researchers suggest to manage them through refactoring [37] and tracking [30]. Existing studies have investigated refactoring and tracking of regular clones only. However, a number of recent studies [26, 29, 42] have investigated the importance of micro-clones in software evolution. Mondal et al. [29] showed that micro-clones from the same clone class have a tendency of co-changing (changing together) consistently during evolution like the regular clones. Thus, tracking of micro-clones should be considered important as of the regular clones. By considering micro-clones for tracking, we can avoid inconsistencies in the code-base and can help programmers in updating micro-clones with less effort. The primary purpose of a clone-tracker is to suggest co-change candidates of a target clone fragment. In our research, we investigate ranking of co-change candidates for micro-clones from the tracking perspective.

When a programmer attempts to make changes to a target micro-clone fragment from a micro-clone class, the other fragments in the class might also need to be changed together (co-changed) consistently with the target fragment in order to ensure consistency of the software system. Thus, these other fragments are called the co-change candidates of the target fragment. The task of a clone tracker is to identify these co-change candidates when a programmer attempts to change a target clone fragment. However, all these co-change candidates might not need to be actually co-changed with the target fragment because clone fragments can evolve independently. In such a situation, it is important to identify which of the co-change candidates are highly likely to be co-changed consistently. Our goal in this research is to rank the co-change candidates on the basis of their likeliness of being co-changed with the target fragment so that the programmers can easily identify the highly likely ones. Fig. 1 shows a target micro-clone fragment CF2 in a micro-clone class having seven clone fragments. We can also see the six co-change candidates of the target fragment and a possible ranking of the co-change candidates. According to the ranking in Fig. 1, CF7 has the highest likeliness of being co-changed with CF2 and the likeliness of CF5 is the lowest. To the best of our knowledge, our study is the first one to investigate ranking co-change candidates of micro-clones.

Mondal et al. [31] previously conducted a genealogy-based study on ranking co-change candidates for regular code clones. They first extracted clone genealogies, and then analyzed evolutionary

CASCON '19, Nov. 4–6, 2019, Toronto, ON, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

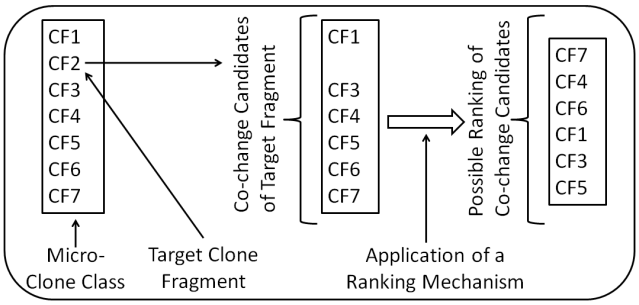

Figure 1: Ranking co-change candidates of a target micro-clone fragment

coupling among the genealogies. However, the number of micro-clones is much higher in a software system than regular clones. As a result, a genealogy-based ranking is not suitable for micro-clones. Extracting micro-clone genealogies and analyzing evolutionary coupling among those will be very time consuming and will not be suitable for real-time coding environment. Focusing on this, we investigate ranking co-change candidates of micro-clones in the following way.

We investigate two ranking mechanisms, *file proximity ranking* and *file coupling ranking*, in our research for ranking the co-change candidates of a target micro-clone fragment. While file proximity ranking ranks the co-change candidates on the basis of their distances from the target fragment in the file system tree, file coupling ranking ranks the candidates considering the evolutionary coupling among their container files. We realize evolutionary coupling among files through mining association rules [47]. We answer the research questions listed in Table 1 by investigating thousands of revisions of six open-source subject systems written in Java and C. We have the following findings:

(1) While a micro-clone class can contain a large number of clone fragments (for example, 67 which is the average number of fragments per affected micro-clone class of our subject system jEdit), only a little proportion (for example, 3% for jEdit) of these fragments might actually need to be co-changed consistently. Such a finding implies that ranking of co-change candidates for micro-clones is necessary. An efficient ranking mechanism can be beneficial to the programmers in selecting the highly likely co-change candidates from a long list of possible candidates.

(2) According to our statistical significance tests, micro-clones belonging to the same source code file have a significantly higher tendency of co-changing consistently compared to micro-clones belonging to different source code files.

(3) Evolutionary coupling among the container files of the micro-clone fragments have a positive impact on ranking co-change candidates of micro-clones.

(4) On the basis of our second and third findings we propose a composite mechanism for ranking co-change candidates of micro-clones. Our composite ranking mechanism performs significantly better than file proximity ranking mechanism.

We finally suggest that ranking co-change candidates of micro-clones should be taken into proper consideration when making clone management decisions. Our proposed composite ranking mechanism has the potential to assist programmers in identifying likely co-change candidates of target micro-clone fragments during programming with considerably less effort and time. It can complement the existing clone tracking techniques so that they can consider micro-clones for tracking.

Table 1: Research Questions

SL	Research Question
RQ 1	What percentage of the micro-clone fragments in a micro-clone class get changed together consistently during evolution?
RQ 2	Do micro clones from the same source code file have a higher tendency of co-changing consistently compared to micro clones from different source code files?
RQ 3	Does consistent co-change tendency of micro clones belonging to different source code files depend on the presence of evolutionary coupling among the files?
RQ 4	Can we rank co-change candidates for micro clones using proximity as well as evolutionary coupling among their container files?

Paper Organization. The rest of our paper is organized as follows. Section 2 describes the terminology, Section 3 discusses the experimental steps, Section 4 presents our experiment results and analyzes those to answer our research questions, Section 5 discusses the related work, Section 6 mentions the threats to validity, and Section 7 concludes the paper by mentioning future work.

2 TERMINOLOGY

Regular clones. We detect and analyze three major types (Type 1, Type 2, and Type 3) of regular code clones in our research. We define these clone-types in the following way according to the literature [3, 7]. The identical code fragments residing in a software system's code-base disregarding the comments and indentations are called **Type 1** clones. Syntactically similar code fragments are known as **Type 2** clones. These clones are generally created from Type 1 clones because of renaming identifiers and/or changing data types. Finally, **Type 3** clones, also known as gapped clones, are generally created from Type 1 or Type 2 clones because of additions, deletions, or modifications of lines in these clones.

Micro-Clones. According to the literature [26, 29, 42], *Code clones having a size that is smaller than the minimum size of regular clones are called micro-clone. Micro-clones are not parts of regular code clones.* The minimum size of a micro-clone fragment is 1 LOC. In our experiment we use the NiCad clone detector [19] for detecting regular clones of at least 5 lines which is the best threshold value for NiCad for detecting regular clones from Java and C source code as was reported by Wang et al. [46]. Thus, in our experiment, a micro-clone fragment can have at most 4 LOC.

Consistent changes in code clones. Code clones from the same clone class have a tendency of getting changed consistently during evolution. We define consistent changes in code clones in the following way.

If both clone fragments in a clone-pair residing in a particular revision of a software system experienced the same changes in the commit operation that was applied on that revision, we say that the clone fragments in the pair were changed together (co-changed) consistently in the commit operation.

Identifying consistent changes in code clones. Let us assume that the two clone fragments belonging to a clone-pair were changed in a particular commit operation. We identify these changes using UNIX diff. Diff outputs three types of changes: addition, modification, and deletion. We consider that the clone fragments were changed consistently if the following conditions hold:

Table 2: Subject Systems

Systems	Lang.	Domains	LLR	Revs
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Management	45,515	1545
Ctags	C	Code Def. Generator	33,270	774
Camellia	C	Image Processing Library	89,063	170

LLR = LOC in the Last Revision Revs = No. of Revisions

Table 3: Number of distinct regular and micro clone fragments in each of our subject systems

	jEdit	Freecol	Carol	Jabref	Ctags	Camellia
Regular Clones	3823	305	226	171	139	93
Micro-Clones	6146	4449	2244	2306	284	203

Regular Clones = Number of Regular Clones in the last revision.
Micro-Clones = Number of Micro Clones in the last revision.

- In the case of addition, the same line(s) were added after the same line in each clone fragment.
- In the case of modification, the same line(s) in each clone fragment were modified in the same way. Thus, the lines that were modified and the lines that we obtain after modification should be the same in each clone fragment.
- In the case of deletion, the same line(s) were deleted from each clone fragment in the pair.

We can check these conditions by comparing the diff outputs corresponding to the changes in the two clone fragments.

Evolutionary coupling. Evolutionary coupling [13, 47] is a well investigated phenomenon during software maintenance and evolution. If a group of program entities (such as files, classes, or methods) change together (i.e., co-change) frequently during software evolution, it is expected that the entities in the group are coupled. In future, a change in one entity might require corresponding changes to the other entities in the group. In such a situation we say that the entities in the group have evolutionary coupling. In our research we investigate file level evolutionary coupling through mining association rules [47] for ranking co-change candidates of micro-clones. The details how we detect evolutionary coupling will be discussed in Section 4.3.

3 EXPERIMENT STEPS

We conduct our experiment on six open-source subject systems (Table 2) written in Java and C. We download these systems from an on-line SVN repository [38]. We select these systems for our investigation, because these are of diverse variety in terms of their application domains, sizes, and revision history lengths. Moreover, the systems are written in two different programming languages. We intentionally select our subject systems emphasizing their diversity in order to generalize our findings. We perform the following experiment steps before answering our research questions.

- Downloading each of the revisions (as mentioned in Table 2) of the subject system from the SVN repository.
- Preprocessing the source code files in each revision by removing comments and blank lines.
- Detecting changes between the corresponding source code files of every two consecutive revisions by applying UNIX diff operation.

Table 4: NiCad settings for regular and micro-clones

	Clone Granularity	Min Line	Max Line	Identifier Renaming	Dissimilarity Threshold
Regular Clones	block	5	20000	blind renaming	20%
Micro Clones	block	1	4	blind renaming	20%

- Detecting regular and micro-clones from each of the revisions by applying a clone detector called NiCad [19].
- Mapping the changes that occurred to each revision to the already detected micro-clones in that revision by using line numbers of the changes and micro-clones.
- Identifying consistent changes in micro-clones following the procedure described in Section 2.

We detect code clones using the well known clone detector NiCad [19] that can detect all three types of clones (Type 1, Type 2, and Type 3) with high precision and recall [4, 6]. A recent study [20] shows that NiCad is a good choice among the modern clone detectors in terms of detection accuracy. As suggested in Wang et al.'s [46] study, we detect regular code clones of at least 5 LOC using the clone detection tool, NiCad [19]. We also use NiCad for detecting micro-clones of at most 4 LOC. The settings that we have used for detecting regular and micro-clones are shown in Table 4.

4 EXPERIMENT RESULTS AND ANALYSIS

In this section, we present our experiments, report our experiment results, and analyze our results to answer the research questions in Table 1. Table 3 shows the total number of regular and micro-clones from the last revisions of each of our subject systems. We see that the number of micro-clones is always higher than the number of regular clones. Table 5 shows the following three measures for each of our investigated subject systems listed in Table 2: **(1)** Total number of changes during the entire period of evolution of the subject system, **(2)** Total number of consistent changes during evolution, and **(3)** Total number of consistent changes that occurred in micro-clones during evolution.

By considering all the consistent changes that occurred in all subject systems, we calculate the overall percentage of consistent changes that occurred in micro-clones using Eq. 1.

$$OPCCMC = 100 \times \frac{\sum_{s \in systems} NCCMC_s}{\sum_{s \in systems} NCC_s} \quad (1)$$

In the above equation, *OPCCMC* is the overall percentage of consistent changes in micro-clones with respect to all consistent changes, $NCCMC_s$ is the number of consistent changes that occurred in micro-clones of a subject system s, and NCC_s is the number of all consistent changes in s. According to our calculation, around 79.98% of all consistent changes occurred in micro-clones. In our calculation, we disregarded those consistent changes that consisted of addition, deletion, or modification of lines containing only a single character such as "{" or "}". We analyze the consistent changes in micro-clones and answer our research questions.

4.1 Answering the first research question

RQ 1: *What percentage of the micro-clone fragments in a micro-clone class get changed consistently during evolution?*

Rationale. Answering RQ 1 is important for ranking co-change candidates of micro-clones. If we observe that almost all clone

Table 5: Consistent changes in micro-clones

Systems	jEdit	Freecol	Carol	Jabref	Ctags	Camellia
NC	6261	16320	8253	14917	2114	3118
NCC	669	3565	1714	1777	109	749
NCCMC	554	3063	1551	904	74	719

NC = Number of changes during the whole period of evolution
NCC = Number of consistent changes during evolution
NCCMC = Number of consistent changes in Micro-Clones

fragments in a micro-clone class generally get updated consistently, then ranking co-change candidates of micro clones might not be particularly important. We just need to detect the other micro-clone fragments in the micro-clone class that includes the particular micro-clone fragment which is going to be changed. However, if it is found that only a subset of all the members in a micro-clone class is likely to be updated consistently, then further investigation should be done towards ranking co-change candidates of micro clones. Whenever a programmer attempts to make changes to a target micro clone fragment from such a subset of a micro clone class, the other members in the subset should be given higher priorities for co-changing with the target fragment than the class members that are not included in this subset. We perform our investigation for answering RQ 1 in the following way.

Investigation Procedure. Our investigation involves examining the entire evolution history of our candidate systems. For a particular software system we automatically examine each of its commit operations from the beginning as mentioned in Table 2. While examining a particular commit operation we determine all the changes that occurred in that commit operation using UNIX diff. We analyze these changes in the following way to identify the affected micro-clone classes.

UNIX diff reports three types of changes: additions, deletions, and modifications. Let us consider a particular change (an addition, a deletion, or a modification) in a particular commit operation. We first determine whether this change occurred in a regular clone fragment or not. For this purpose we detect the regular code clones in the revision where the commit operation was applied and check whether the change occurred in any of these regular code clones. We use the NiCad clone detector [19] for detecting regular clones. If the change occurred in a regular clone fragment, we discard it from our consideration. However, if it did not occur in a regular clone fragment, then it might occur in a micro clone fragment. We detect all the micro clone fragments in the revision using the NiCad clone detector [19]. By applying NiCad we detect code clones with a minimum size of 1 LOC and a maximum size of 4 LOC. We then exclude those clones from considerations that reside in our already detected regular clones. The remaining ones are micro clones according to the definition in Section 2. We then determine whether the change occurred in any of these micro clone fragments. Let us assume that the change occurred in a particular micro clone fragment. In this situation we determine the other micro clone fragments in the same micro clone class. We determine which of these other fragments also experienced the same change. Finally, we determine how many micro clone fragments in the micro clone class experienced the same change. In other words, we determine how many micro clone fragments in the micro clone class were changed together consistently.

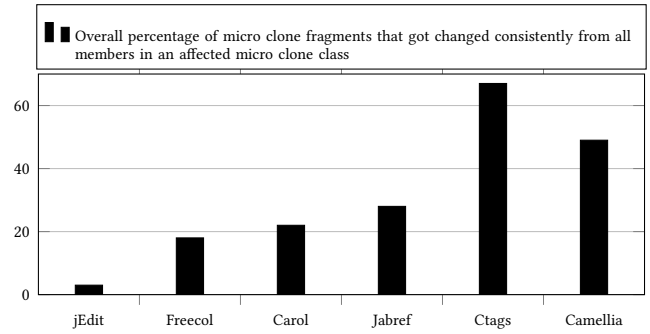

Figure 2: Percentage of micro clone fragments that got changed consistently from all micro clone fragments of an affected micro clone class

In the way described above, we examine all the affected micro clone classes in each commit operation. For each affected micro clone class we determine the total number of micro clone fragments in the class, and the total number of micro clone fragments that were updated consistently from that class. Table 6 shows statistics regarding the affected micro clone classes from the entire evolution history of our subject systems. From this table it is clear that micro clone classes often get updated partially. According to our subject systems, the proportion of partially updated micro clone classes can be up to 60% (for Freecol) of all affected micro clone classes. Table 6 also shows that the average number of micro clone fragments per affected micro clone class can be up to 67 (for jEdit). Moreover, the overall proportion of the class members that got modified per affected micro clone class is very low for some subject systems such as jEdit, Freecol, Carol, and Jabref. We show this proportion for each of our subject systems in Fig. 2. According to Fig. 2, Ctags shows the highest proportion.

> **Answer to RQ 1:** The percentage of class members (i.e., micro clone fragments) that get changed together consistently from a micro clone class can be up to 67% as indicated by our subject system Ctags in Table 6. However, this percentage can be very low, for example 3% for our subject system jEdit.

According to our findings from RQ 1 we realize that further investigation is necessary towards identifying which members in a micro clone class generally have a high tendency of getting changed together consistently. We perform such investigations in the following subsections.

4.2 Answering the second research question

RQ 2: *Do micro clones from the same source code file have a higher tendency of co-changing consistently compared to micro clones from different source code files?*

Rationale. From our answer to RQ 1 we realize that only a small subset of all the micro clone fragments in a micro clone class might get updated consistently. Thus, it is important to investigate which of the clone fragments in a micro clone class have a tendency of getting updated consistently. Such an investigation can help us rank the co-change candidates for a micro clone fragment. If we can identify that a particular subset of the clone fragments in a micro clones class have a tendency of getting modified consistently, then,

Table 6: The measures regarding answering RQ 1

Measure	jEdit	Freecol	Carol	Jabref	Ctags	Camellia
Number of affected micro clone classes during the entire period of evolution	147	653	231	351	23	168
Number of micro clone classes where all the members were changed	65	256	176	229	13	126
Number of micro clone classes where only a subset of all the members was changed	82	397	55	122	10	42
Percentage of partially affected micro clone class w.r.t all affected classes	55.78%	60.79%	23.8%	34.75%	43.47%	25%
Average number of members (micro clone fragments) per affected micro clone class	67.3	22.49	10.4	8.1	3.52	7.5
Overall percentage of members that were modified consistently from an affected micro clone class	3%	18%	22%	28%	67%	49%

Table 7: Summations for measures (M1, M2, M3, and M4) regarding answering RQ 2

	jEdit	Freecol	Carol	Jabref	Ctags	Camellia
Summation for M1	6716	897	941	1181	82	3882
Summation for M2	508	578	798	503	44	2456
Summation for M3	19768	96666	4166	4318	1	2318
Summation for M4	65	40366	289	221	0	876

when a programmer will attempt to make changes to a particular fragment from the subset, the other fragments in the subset can be suggested as the co-change candidates of the particular fragment with high priority. In other words, the other fragments in the subset can be given a higher priority compared to those fragments in the micro clone class that are not included in the subset. In RQ 2, we particularly investigate whether file proximity of the micro clone fragments have any impact on their possibility of getting co-changed consistently. We investigate in the following way.

Investigation procedure. For answering RQ 2, we investigate whether micro clones belonging to the same source code file have a higher tendency of co-changing consistently compared to the micro clones belonging to different files. We first identify all the affected micro clone classes by automatically analyzing the entire evolution history of a subject system as we did in RQ 1. For each of the affected micro clone classes we identify which of the members in the class were changed together consistently. Let us assume that a particular affected micro clone class has n micro clone fragments in total. The number of fragments that were changed consistently is n_c. We select one fragment from the set of consistently changed micro clone fragments. Let us denote this fragment by F. We make all possible clone pairs considering all the members (n members) in the clone class. From these clone pairs we select each of those pairs that include F. Thus, for F, we obtain $n-1$ pairs in total and n_c-1 of these pairs were made with the remaining n_c-1 consistently changed fragments. By considering all the pairs containing F, we determine the following measures:

- **M1:** Number of pairs where F and the other fragment in the pair belong to the same source code file.
- **M2:** Number of pairs where F and the other fragment in the pair belong to the same source code file and both fragments (F and the other fragment) in the pair were changed together consistently.
- **M3:** Number of pairs where F and the other fragment in the pair belong to different source code files.
- **M4:** Number of pairs where F and the other fragment in the pair belong to different source code files, however, both fragments (F and the other fragment) in the pair were changed together consistently.

We determine these four measures for each of the consistently updated micro clone fragments of each of the affected micro clone classes from the entire period of evolution of a candidate system. We determine the respective sum for each of these four measures (the summations are presented in Table 7) and then determine two probabilities. Before elaborating these two probabilities, we define a target micro clone fragment and its co-change candidates in the following way. These definitions will help us easily understand the probabilities.

Target micro clone fragment: Let us assume that a programmer is going to make some changes to a particular micro clone fragment in a micro clone class. We call this fragment the target micro clone fragment.

Co-change candidates of the target micro clone fragment: When a programmer attempts to make changes to a target micro clone fragment from a micro clone class, the other fragments in the same class might also need to be updated together consistently with the target fragment. These other fragments are called the co-change candidates of the target fragment as shown in Fig. 1.

On the basis of the above definitions, we now define two probabilities **CPSFC** and **CPDFC** in the following way using the four measures: **M1**, **M2**, **M3**, and **M4**.

- **CPSFC (Co-change Probability of Same File Candidates):** Let us assume that a programmer is going to make changes to a target micro clone fragment in a micro clone class. **CPSFC** is the probability that a co-change candidate (of the target micro clone fragment) residing in the same source code file as of the target fragment also needs to be co-changed consistently with the target fragment. We determine this probability in percentage using the following equation.

$$CPSFC = 100 \times \frac{\sum_{all\ affected\ classes} M2}{\sum_{all\ affected\ classes} M1} \quad (2)$$

- **CPDFC (Co-change Probability of Different File Candidates):** Let us assume that a programmer is going to make changes to a target micro clone fragment in a micro clone class. **CPDFC** is the probability that a co-change candidate (of the target micro clone fragment) which does not reside in the source code file where the target fragment resides also need to be co-changed consistently with the target fragment. We determine this probability in percentage using the following equation.

$$CPDFC = 100 \times \frac{\sum_{all\ affected\ classes} M4}{\sum_{all\ affected\ classes} M3} \quad (3)$$

Fig. 3 shows the two probabilities (**CPSFC** and **CPDFC**) for each of our candidate systems. From the graph it is clear that the probability **CPSFC** is always higher than **CPDFC**. In other words, the probability that a co-change candidate from the same source code file will co-change consistently with the target micro clone fragment is always higher than the probability that a co-change candidate

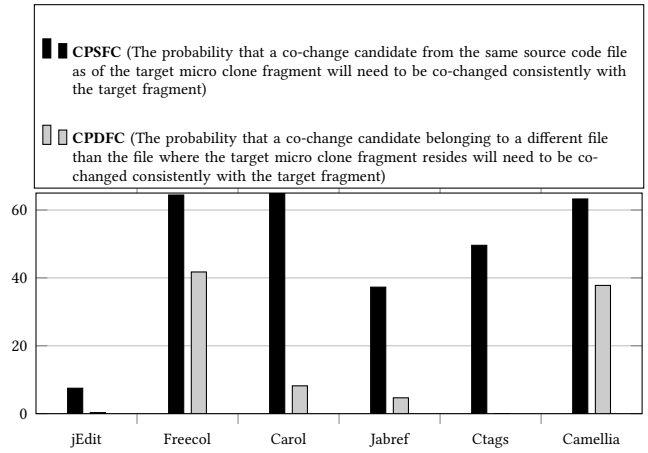

Figure 3: Probabilities that a co-change candidate belonging to the same or a different source code file will need to be co-changed with a target fragment

from a different source code file will co-change consistently with the target micro clone fragment. Such a finding helps us decide that *when ranking the co-change candidates for a target micro clone fragment from a micro clone class, co-change candidates belonging to the same source code file as of the target clone fragment should be given higher ranks compared to the co-change candidates belonging to different source code files.*

Significance test regarding the two probabilities: CPSFC and CPDFC. As we see that the value of CPSFC is always greater than CPDFC, we wanted to determine whether CPSFC is significantly greater than CPDFC. For this purpose we perform Wilcoxon Signed Rank Test[48, 49] to determine whether the values of CPSFC are significantly different than the values of CPDFC. We should note that this test is non-parametric [48], and thus, the samples do not need to be normally distributed for applying this test. Moreover, this test can be applied to both small and large data sets [48]. We perform our test considering a significance level of 5%. We used the statistical software package of SPSS [45] for conducting our test. From our two-tailed test result we realize that the values of **CPSFC** are significantly different than the values of **CPDFC** with a *p-value* of 0.028 which is less than 0.05. As CPSFC is always greater than CPDFC, we realize that CPSFC is significantly greater than CPDFC.

> **Answer to RQ 2:** Micro clone fragments belonging to the same source code file have a significantly higher tendency of co-changing consistently compared to micro clone fragments belonging to different source code files.

Our answer to RQ 2 implies that when a programmer attempts to make changes to a target micro clone fragment in a micro clone class, the other micro clone fragments in the class (i.e., the co-change candidates) that belong to the same source code file as of the target fragment should be given higher ranks compared to the fragments that do not belong to the source code file where the target clone fragment resides. Although ranking co-change candidates on the basis of file proximity might be a good idea, Table 7 makes us realize that consistent co-change of micro clone fragments from different source code files is also a common phenomenon. Thus, ranking co-change candidates belonging to different source code files than the target micro clone fragment should also be investigated. We perform such an investigation for answering our next research question (RQ 3).

Table 8: Summations for the four measures from RQ 3

	jEdit	Freecol	Carol	Jabref	Ctags	Camellia
Summation for Measure 1	11	78124	496	918	0	1956
Summation for Measure 2	11	24706	138	81	0	835
Summation for Measure 3	19757	18542	3670	3400	1	362
Summation for Measure 4	54	15660	151	140	0	41

4.3 Answering the third research question

RQ 3: *Does consistent co-change tendency of micro clones belonging to different source code files depend on the presence of evolutionary coupling among the files?*

Rationale. In RQ 3, we investigate which of the co-change candidates that do not belong to the source code file of the target micro clone fragment exhibit high tendency of getting co-changed with the target fragment. We particularly analyze whether evolutionary coupling among source code files containing micro clone fragments influence their tendency of getting co-changed consistently. We investigate in the following way.

Detecting evolutionary coupling through mining association rules. Let us consider a micro clone class in a particular revision r of a subject system. Two micro clone fragments in the clone class belong to two different source code files: *file1* and *file2*. We mine file level association rules [47] from the past evolutionary history of the software system consisting of the commits that were applied on the revisions 1 to $r-1$. In our research, an association rule takes the form, 'x => y', where x and y are source code files. Here, 'x' is the antecedent and 'y' is the consequent. Such a rule implies that if 'x' changes in a commit operation, there is a possibility that 'y' will also change in that commit operation. We mine file level association rules along with their support and confidence values following the procedure of Zimmerman et al. [47] from the past evolutionary history (i.e., from the commits applied on the revisions 1 to r-1). We consider rules with support >= 1. If any of these rules contains both of the files, *file1* and *file2*, we consider that the files exhibited evolutionary coupling.

Analyzing the impact of file evolutionary coupling on co-change tendency of micro clones. We first identify the affected micro clone classes by examining the entire evolution history of our subject systems as we did when answering RQ 1 and RQ 2. Let us consider a particular micro clone class containing n micro clone fragments in total. We automatically identify the micro clone fragments that were consistently updated from this class. Let us assume that F is a consistently updated micro clone fragment. We make all possible clone pairs considering the n micro clone fragments from the affected class, and then identify those pairs where the two clone fragments in a pair belong to two different source code files. From these identified pairs, we select those pairs that contain F. Thus, each of the clone pairs that we finally select contains F, and the two fragments (F and the other fragment) in the pair belong to two different files. From these selected clone pairs we determine the following four measures as we did when answering RQ 2.

- **Measure 1:** The number of clone pairs where the two source code files containing the two micro clone fragments in each pair exhibited evolutionary coupling during the past evolution. If the clone class containing F resides in revision r of a subject system, then past evolution consists of the commit operations that were applied on the revisions 1 to $r-1$.
- **Measure 2:** The number of clone pairs where the two fragments in each pair co-changed consistently and the two source code files containing the fragments in the pair exhibited evolutionary coupling during the past evolution.
- **Measure 3:** The number of clone pairs where the two source code files containing the two micro clone fragments in each pair did not exhibit evolutionary coupling during the past evolution.
- **Measure 4:** The number of clone pairs where the two fragments in each pair co-changed consistently, however, the two source code files containing the fragments in the pair did not exhibit evolutionary coupling during the past evolution.

We determine these four measures for each of the consistently updated micro clone fragments from each of the affected micro clone classes obtained from the whole period of evolution of a subject system. We calculate the respective sum of each of these measures (the summations are presented in Table 8) and then determine the following two probabilities:

CPCC (Co-change Probability of Coupled Candidates): Let us assume that a programmer is going to make some changes to a target micro clone fragment in a micro clone class. Let us also assume that we have the set of its co-change candidates that do not belong to the same source code file as of the target fragment and the files containing these co-change candidates exhibited evolutionary coupling with the file having the target fragment during the past evolution. **CPCC** is the probability that such a co-change candidate will co-change (change together) consistently with the target micro clone fragment. We determine **CPCC** in percentage using the following equation.

$$CPCC = 100 \times \frac{\sum_{all\ affected\ classes} Measure\ 2}{\sum_{all\ affected\ classes} Measure\ 1} \quad (4)$$

CPNC (Co-change Probability of Non-coupled Candidates): Let us consider that a programmer is going to make some changes to a target micro clone fragment in a micro clone class. We determine the set of its co-change candidates that do not belong to the same source code file as of the target fragment and the files containing these co-change candidates did not exhibit evolutionary coupling with the file having the target fragment during the past evolution. **CPNC** is the probability that such a co-change candidate will co-change (change together) consistently with the target micro clone fragment. We determine **CPNC** in percentage using the following equation.

$$CPNC = 100 \times \frac{\sum_{all\ affected\ classes} Measure\ 4}{\sum_{all\ affected\ classes} Measure\ 3} \quad (5)$$

We determine these two probabilities for each of our subject systems and plot those in the graph of Fig. 4. From Fig. 4 it is clear that **CPCC** is greater than **CPNC** for most of the subject systems (4 out of 6) except Ctags and Freecol. For Ctags, we did not get any pair such that the two fragments in the pair co-changed consistently and the fragments reside in different source code files. Finally, Freecol

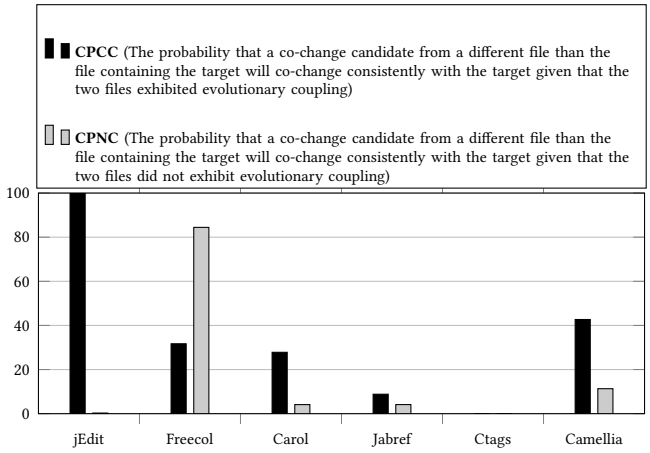

Figure 4: Probabilities that a co-change candidate belonging to a different source code file will need to be co-changed with a target micro clone fragment

is the only exception where **CPNC** is greater than **CPCC**. Considering the agreement among the majority of our subject systems we realize that evolutionary coupling among source code files have a positive impact on the co-change possibility of micro clone fragments belonging to different files. We performed Wilcoxon Signed Rank Test [48, 49] considering a significance level of 5% to determine whether the **CPCC** values in Fig. 4 are significantly different than the **CPNC** values in that figure. However, we found that the values are not significantly different.

> **Answer to RQ 3:** *Consistent co-change tendency of micro clone fragments belonging to different source code files can be impacted by the presence of evolutionary coupling among the source code files.* Generally, consistent co-change tendency of micro clone fragments belonging to different source code files having evolutionary coupling among them is higher than the consistent co-change tendency of micro clone fragments belonging to different files having no evolutionary coupling.

Our answer to RQ 3 implies that evolutionary coupling among files containing micro clone fragments should be considered when ranking co-change candidates for a target micro clone fragment. In RQ 4, we investigate ranking of co-change candidates for micro clones using proximity of files containing micro clones as well as evolutionary coupling of the container files.

4.4 Answering the fourth research question

RQ 4: *Can we rank co-change candidates for micro clones using proximity as well as evolutionary coupling among their container files?*

Rationale. From our answers to RQ 2 and RQ 3, we realize that when ranking co-change candidates for a target micro clone fragment we should consider both proximity and evolutionary coupling among the files that contain the target micro clone fragment and it's co-change candidates. For answering RQ 4, we investigate two ranking mechanisms: (1) file proximity ranking, and (2) a composite ranking mechanism that combines file proximity ranking and ranking on the basis of evolutionary coupling among the source

code files of the co-change candidates and the target fragment. We perform our investigation in the following way.

Investigation procedure. Let us assume that a micro clone class has n micro clone fragments. A programmer is going to make changes to a particular micro clone fragment which we call the target micro clone fragment. The remaining $n-1$ members (i.e., excluding the target fragment) in the class are the co-change candidates of the target fragment. We separate these co-change candidates into following two disjoint subsets:

- **SFC (Same File Candidates):** These are the co-change candidates that belong to the same file as of the target micro clone fragment.
- **DFC (Different File Candidates):** These are the co-change candidates that do not belong to the source code file where the target fragment resides.

We can easily realize that $|SFC| + |DFC| = n - 1$. We now define the following ranking mechanisms for ranking the co-change candidates of a target micro clone fragment in a micro clone class:

- **FPR (File Proximity Ranking):** Ranking co-change candidates essentially means listing the candidates in such a way that candidates with high possibility of getting co-changed with the target micro clone fragment take place to the top of the list. In *File Proximity Ranking*, we ensure that the co-change candidates in the subset called SFC (Same File Candidates) take the top $|SFC|$ positions in the list so that the co-change candidates residing in the same file as of the target micro clone fragment get a higher co-change priority compared to the candidates in the subset called DFC.
- **FCR (File Coupling Ranking):** *FCR* focuses on the co-change candidates in the subset called DFC (Different File Candidates). For each of the co-change candidates in DFC we determine whether the source code file containing the candidate exhibited evolutionary coupling with the file where the target micro clone fragment resides. Such co-change candidates from DFC are given higher priorities than the remaining candidates in DFC in *File Coupling Ranking* (FCR).
- **CR (Composite Ranking):** In *Composite Ranking* we rank the whole set of co-change candidates ($n-1$ candidates in total) by applying both *File Proximity Ranking* (FPR) and *File Coupling Ranking* (FCR). We first place the co-change candidates in SFC in the top $|SFC|$ positions in the list. Then, the remaining co-change candidates (i.e., the candidates in DFC) are listed according to FCR.

We analyze whether ordering co-change candidates according to *Composite Ranking* provides better ranks for the actually co-changed candidates compared to *File Proximity Ranking*. Before showing comparison results, we discuss how we compare two ranking mechanisms by analyzing evolutionary history of our subject systems.

Comparing two ranking mechanisms. Let us assume that we have two ranking mechanisms: **RM1** and **RM2**. Our goal is to compare these two ranking mechanisms in ranking the co-change candidates for a target micro clone fragment. We compare by automatically analyzing evolution history of each of our subject systems. We first identify all the affected micro clone classes from the evolution history of a particular subject system. Let us consider a particular micro clone class, c, residing in revision r of the subject system. The commit operation which was applied on revision r consistently changed some of the micro clone fragments in the class c. We identify which of the micro clone fragments from class c were updated consistently in the commit operation. Let f be such a micro clone fragment. We consider f as the target micro clone fragment. Thus, all the other members in c (i.e., excluding f) are the co-change candidates of f. However, as we examine the commit operation, we know which of these co-change candidates really co-changed consistently with f. We now apply the two ranking mechanisms: *RM1* and *RM2* on the co-change candidates of f and make two lists of these candidates from the two ranking mechanisms respectively. Each list contains all the co-change candidates of f. We consider one co-change candidate that really co-changed with f in the commit operation. From the co-change candidate list obtained by applying RM1, we determine the rank (i.e., the position number) of that particular co-changed candidate. In the same way, we again obtain the rank of that co-changed candidate from the candidate list obtained by applying RM2. Now we easily understand that the ranking mechanism which will provide a better rank (i.e., a smaller position number) to the co-changed candidate should be considered the better ranking mechanism. By considering each of the micro clone fragments changed from each of the affected micro clone classes, we determine two ranks from two ranking mechanisms (RM1 and RM2) for each of the co-change candidates that really co-changed with that particular micro clone fragment. We determine the average rank of co-changed candidates considering each ranking mechanism and then compare these average ranks to determine which ranking mechanism performs better.

Comparison between *Composite Ranking* and *File Proximity Ranking*. Fig. 5 compares the average ranks provided by *Composite Ranking* and *File Proximity Ranking* mechanisms for each of our subject systems. We should note that a lower average is considered better because, ranks are the serial numbers of the co-changed candidates in the lists obtained by the two ranking mechanisms. A lower serial number is considered a better position. From Fig. 5 we realize that for five subject systems (except Ctags), *Composite Ranking* mechanism performs better (i.e., Composite Ranking mechanism provides better ranks) than *File Proximity Ranking*. For Camellia, the average ranks, 4.96 and 4.87, provided by respectively *file proximity ranking* and *composite ranking* are very near to each other with *composite ranking* being a bit better. For Ctags, we did not find any commit operation where micro-clone fragments belonging to different files co-changed. Thus, we could not apply Composite Ranking for Ctags. As a result, Fig. 5 does include a comparison for Ctags. For making comparison, we consider those cases where the two ranking mechanisms provide different ranks.

Statistical significance tests. We also performed Wilcoxon Signed Rank test [48, 49] (as we did for answering RQ 2) using SPSS [45] to determine whether the average ranks provided by *Composite Ranking* are significantly different than those provided by *File Proximity Ranking*. According to our 2-tailed test considering a significance level of 5%, the average ranks from the composite ranking mechanism are significantly different those from file proximity ranking mechanism with a p-value of 0.043 which is smaller than 0.05. As the average rank from the composite mechanism is always better (i.e., smaller) than that of file proximity ranking mechanism,

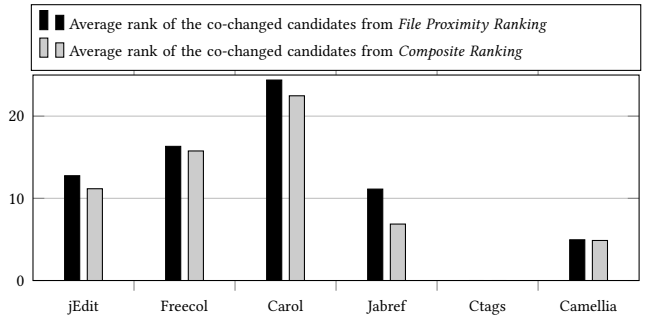

Figure 5: Comparing *File Proximity Ranking* and *Composite Ranking*

we can state that composite mechanism performs significantly better than file proximity ranking mechanism.

> **Answer to RQ 4:** *Composite Ranking* mechanism performs significantly better than *File Proximity Ranking* mechanism in ranking co-change candidates of micro-clones.

Our proposed *Composite Ranking* mechanism that combines file proximity ranking and ranking on the basis of evolutionary coupling can be useful for ranking co-change candidates for micro clones when only a subset of the co-change candidates need to be updated consistently.

5 RELATED WORK

A large number of studies [2, 8, 12, 15, 16, 18, 22–25, 28, 32, 35, 36, 41, 43, 44, 46] have been done on detecting, analyzing, and managing regular code clones. However, our research involves investigating micro-clones which are smaller than the minimum size of regular clones. Recent studies [26, 29, 42] have shown the importance of micro-clones during software evolution and maintenance.

Beller et al. [26] discover that micro-clones can often be faulty during software evolution. Tonder and Goues [42] detected micro-clones from a large number of Java projects and found that such clones can often indicate flaws in software systems. They suggested that removal of micro-clones can be important from development perspectives. Mondal et al. [29] showed that micro-clones have a tendency of co-changing consistently during evolution. Islam et al. [21] found that micro-clones can exhibit a higher bug-proneness compared to regular clones. Our research in this paper is different from all these existing studies on micro-clones, because we investigate ranking co-change candidates of micro-clones. Our research findings as well as the proposed ranking mechanism can be beneficial to the programmers in consistently updating micro-clones during evolution to ensure software consistency.

A number of studies [10, 11, 33, 39, 40] have investigated tracking of regular clones. The main purpose of clone tracking is to ensure consistent updates to the code clones. Our study is different than these existing studies because we focus on consistent updates in micro-clones. In particular, we investigate ranking co-change candidates of micro-clones so that programmers can easily update them consistently during programming to ensure consistency of software systems.

Mondal et al. [31] previously investigated ranking co-change candidates for regular code clones. Their investigation was based on clone genealogies and they analyzed evolutionary coupling among code clones by analyzing the genealogies. However, genealogy-based investigation is infeasible for micro-clones because micro-clones are around three times as much as the regular clones in a software system [29]. Detecting genealogies of this huge number of micro-clones will be very time consuming and it will not be suitable for real-time programming environment. Our investigation on ranking co-change candidates of micro-clones does not include clone genealogy detection. We investigate evolutionary coupling among the container files of micro-clones for the purpose of ranking. We combine file proximity with evolutionary coupling to device a composite ranking mechanism for micro-clones.

A number of studies [1, 14, 47] have been conducted on detecting and analyzing the evolutionary coupling of software entities such as files, classes, and methods. The goal of these studies is to identify which entities have a tendency of getting changed together. However, none of these studies investigated co-change tendencies of micro-clone fragments. By analyzing co-change tendencies of micro-clones, we can help programmers update micro-clones consistently with less effort and time during evolution, and thus, minimize inconsistencies and bugs in software systems. We leverage evolutionary coupling among the container files of micro-clones for ranking co-change candidates of micro-clones.

From our previous discussion it is clear that ranking co-change candidates of micro-clones was never investigated by the existing studies. As micro-clones exhibit consistent co-change tendencies like the regular clones, we believe that ranking co-change candidates of micro-clones should also be considered important. We answer four research questions in our study and propose a composite ranking mechanism on the basis of our findings from the questions. According to our experiment, our proposed mechanism can be beneficial to the programmers in selecting the likely co-change candidates of a target micro-clone fragment during programming.

6 THREATS TO VALIDITY

Our investigation involves detecting regular code clones using the NiCad clone detector [19]. For different settings of the clone detector, the experimental results and findings can be different. Wang et al. [46] mentioned this problem as the *confounding configuration choice problem* and performed an in-depth investigation in order to find the most suitable configurations for different clone detectors. The setting that we have used for NiCad for detecting regular code clones was suggested to be the most suitable one by Wang et al. [46]. Thus, we believe that our findings are important.

Micro-clones are of at most 4LOC in our study. With this upper threshold, we found that the number of micro-clones is higher than that of regular clones. While a higher value for the upper threshold could give us more micro-clones, we restrict our study to 4LOC because the best minimum size threshold of regular clones is 5LOC for Java and C systems as was reported by Wang et al. [46]. Thus, our consideration of the maximum size threshold of 4LOC for micro-clones is reasonable.

We did not study enough subject systems to be able to generalize our findings regarding ranking co-change candidates of micro clones. However, our candidate systems were of diverse variety in terms of application domains, sizes, revisions, and implementation

languages. Thus, our findings cannot be attributed to a chance, and these are important from the perspective of managing micro-clones.

7 CONCLUSION

Existing studies show that both regular and micro-clones have tendencies of co-changing consistently during evolution. Thus, tracking of such clones can help us minimize inconsistencies in the code-base and reduce programmer effort for consistently updating those. In our research, we investigate automatic ranking of co-change candidates of micro-clones from the tracking perspective. While ranking co-change candidates for regular clones was previously investigated, such an investigation was not done before considering micro-clones. According to our investigation on thousands of revisions of six open-source subject systems written in Java and C we find that micro-clones residing in the same file have a significantly higher tendency of co-changing consistently compared to micro-clones residing in different files. Moreover, evolutionary coupling among the container files of the micro-clones that reside in different files have a positive impact on the co-change tendencies of such micro-clones. We finally device a composite ranking mechanism by combining proximity and evolutionary coupling among the container files of the micro-clone fragments and find that the composite mechanism performs significantly better than file proximity ranking mechanism. Our findings as well as the proposed ranking mechanism are important for consistently updating micro-clones with less effort during software evolution.

ACKNOWLEDGEMENT

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by two Canada First Research Excellence Fund (CFREF) grants coordinated by the Global Institute for Food Security (GIFS) and the Global Institute for Water Security (GIWS).

REFERENCES

[1] A. Alali, B. Bartman, C. D. Newman, J. I. Maletic, "A Preliminary Investigation of Using Age and Distance Measures in the Detection of Evolutionary Couplings", Proc. *MSR*, 2013, pp. 169 – 172.
[2] A. Lozano, M. Wermelinger, "Assessing the effect of clones on changeability", Proc. *ICSM*, 2008, pp. 227 – 236.
[3] C. K. Roy, "Detection and analysis of near-miss software clones", Proc. *ICSM*, 2009, pp. 447 – 450.
[4] C. K. Roy, J. R. Cordy, "A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools", Proc. *Mutation*, 2009, pp. 157 – 166.
[5] C. K. Roy, J. R. Cordy, "A Survey on Software Clone Detection Research", *Technical Report No. 2007-541*, 2007, School of Computing Queen's University, pp. 1 - 115.
[6] C. K. Roy, J. R. Cordy, R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", *Science of Computer Programming*, 2009, 74 (2009): 470 – 495.
[7] C. K. Roy, M. F. Zibran, R. Koschke, "The Vision of Software Clone Management: Past, Present, and Future (Keynote paper)", Proc. *CSMR-WCRE*, 2014, pp. 18 – 33.
[8] C. Kapser, M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software", *Empirical Software Engineering*, 2008, 13(6): 645 – 692.
[9] D. Rattan, R. Bhatia, M. Singh, "Software Clone Detection: A Systematic Review", *Information and Software Technology*, 2013, 55(7): 1165 – 1199.
[10] E. Duala-Ekoko, M. P. Robillard, "CloneTracker: Tool Support for Code Clone Management", Proc. *ICSE*, 2008, pp. 843 – 846.
[11] E. Duala-Ekoko, M. P. Robillard, "Tracking Code Clones in Evolving Software", Proc. *ICSE*, 2007, pp. 158 – 167.
[12] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, "Do Code Clones Matter?", Proc. *ICSE*, 2009, pp. 485 – 495.
[13] H. Gall, M. Jazayeri, J. Krajewski,"CVS Release History Data for Detecting Logical Couplings", Proc. *IWPSE*, 2003, pp. 13 – 23.
[14] H. Kagdi, M. Gethers, D. Poshyvanyk, M. L. Collard,"Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code", Proc. *WCRE*, 2010, pp. 119 – 128.
[15] J. Krinke, "A study of consistent and inconsistent changes to code clones", Proc. *WCRE*, 2007, pp. 170 – 178.
[16] J. Krinke, "Is cloned code more stable than non-cloned code?", Proc. *SCAM*, 2008, pp. 57 – 66.
[17] J. Krinke, "Is Cloned Code older than Non-Cloned Code?", Proc. *IWSC*, 2011, pp. 28 – 33.
[18] J. Li, M. D. Ernst, "CBCD: Cloned Buggy Code Detector", Proc. *ICSE*, 2012, pp. 310 – 320.
[19] J. R. Cordy, C. K. Roy, "The NiCad Clone Detector", Proc. *ICPC Tool Demo*, 2011, pp. 219 – 220.
[20] J. Svajlenko, C. K. Roy, "Evaluating Modern Clone Detection Tools", Proc. *ICSME*, 2014, pp. 321 – 330.
[21] Judith F. Islam, Manishankar Mondal, Chanchal K. Roy, "A Comparative Study of Software Bugs in Micro-clones and Regular Code Clones", Proc. *SANER*, 2019, 11 pp.
[22] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto, "Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software", Proc. *EVOL/IWPSE*, 2010, pp. 73 – 82.
[23] L. Aversano, L. Cerulo, and M. D. Penta, "How clones are maintained: An empirical study", Proc. *CSMR*, 2007, pp. 81 – 90.
[24] L. Barbour, F. Khomh, Y. Zou, "Late Propagation in Software Clones", Proc. *ICSM*, 2011, pp. 273 – 282.
[25] L. Jiang, Z. Su, E. Chiu, "Context-Based Detection of Clone-Related Bugs", Proc. *ESEC-FSE*, 2007, pp. 55 – 64.
[26] M. Beller, A. Zaidman, A. Karpov, "The Last Line Effect", Proc. *ICPC*, 2015, pp. 240 – 243.
[27] M. F. Zibran, C. K. Roy, "Conflict-aware Optimal Scheduling of Code Clone Refactoring", *IET Software*, 2013, 7(3): 167 – 186.
[28] M. Kim, V. Sazawal, D. Notkin, G. C. Murphy, "An empirical study of code clone genealogies", Proc. *ESEC-FSE*, 2005, pp. 187 – 196.
[29] M. Mondal, C. K. Roy and K. A. Schneider, "Micro-clones in evolving software," Proc. *SANER*, 2018, pp. 50 – 60.
[30] M. Mondal, C. K. Roy, K. A. Schneider, "Automatic Identification of Important Clones for Refactoring and Tracking", Proc. *SCAM*, 2014, pp. 11 – 20.
[31] M. Mondal, C. K. Roy, K. A. Schneider, "Prediction and Ranking of Co-change Candidates for Clones", Proc. *MSR*, 2014, pp. 32 – 41.
[32] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, K. A. Schneider, "Comparative Stability of Cloned and Non-cloned Code: An Empirical Study", Proc. *SAC*, 2012, pp. 1227 – 1234.
[33] M. Toomim, A. Begel, S. L. Graham. "Managing duplicated code with linked editing", Proc. *IEEE Symposium on Visual Languages and Human Centric Computing*, 2004, pp. 173 – 180.
[34] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, A. E.Hassan, "An empirical study on inconsistent changes to code clones at the release level", *Science of Computer Programming Journal*, 2012, 77(6): 760 – 776.
[35] N. Göde, J. Harder, "Clone Stability", Proc. *CSMR*, 2011, pp. 65 – 74.
[36] N. Göde, Rainer Koschke, "Frequency and risks of changes to clones", Proc. *ICSE*, 2011, pp. 311 – 320.
[37] N. Tsantalis, D. Mazinanian, G. P. Krishnan, "Assessing the Refactorability of Software Clones", *IEEE Transactions on Software Engineering*, 41(11):1055 – 1090.
[38] Online SVN repository: http://sourceforge.net/.
[39] P. Jablonski, D. Hou, "CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE", Proc. *Eclipse Technology Exchange at OOPSLA*, 2007.
[40] R. C. Miller, B. A. Myers. "Interactive simultaneous editing of multiple text regions.", Proc. *USENIX 2001 Annual Technical Conference*, 2001, pp. 161 – 174.
[41] R. K. Saha, C. K. Roy and K. A. Schneider, "An automatic framework for extracting and classifying near-miss clone genealogies," Proc. *ICSM*, 2011, pp. 293 – 302.
[42] R. van Tonder and C. Le Goues, "Defending against the attack of the micro-clones", Proc. *ICPC*, 2016, pp. 1 – 4.
[43] S. Thummalapenta, L. Cerulo, L. Aversano, M. D. Penta, "An empirical study on the maintenance of source code clones", *Empirical Software Engineering*, 2009, 15(1): 1 – 34.
[44] S. Xie, F. Khomh, Y. Zou, "An Empirical Study of the Fault-Proneness of Clone Mutation and Clone Migration", Proc. *MSR*, 2013, pp. 149 – 158.
[45] SPSS: https://en.wikipedia.org/wiki/SPSS .
[46] T. Wang, M. Harman, Y. Jia, J. Krinke, "Searching for Better Configurations: A Rigorous Approach to Clone Evaluation", Proc. *ESEC/SIGSOFT FSE*, 2013, pp. 455 – 465.
[47] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, "Mining version histories to guide software changes", Proc. *ICSE*, 2004, pp. 563–572.
[48] Wilcoxon Signed Rank Test. https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test.
[49] Wilcoxon Signed Rank Test. http://www.socscistatistics.com/tests/signedranks/Default2.aspx.