Investigating Near-Miss Micro-Clones in Evolving Software

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

Code clones are the same or nearly similar code fragments in a software system's code-base. While the existing studies have extensively studied regular code clones in software systems, micro-clones have been mostly ignored. Although an existing study investigated consistent changes in exact micro-clones, near-miss micro-clones have never been investigated. In our study, we investigate the importance of near-miss micro-clones in software evolution and maintenance by automatically detecting and analyzing the consistent updates that they experienced during the whole period of evolution of our subject systems. We compare the consistent co-change tendency of near-miss micro-clones with that of exact micro-clones and regular code clones. According to our investigation on thousands of revisions of six open-source subject systems written in two different programming languages, near-miss micro-clones have a significantly higher tendency of experiencing consistent updates compared to exact micro-clones and regular (both exact and near-miss) code clones. Consistent updates in near-miss micro-clones have a high tendency of being related with bug-fixes. Moreover, the percentage of commit operations where near-miss micro-clones experience consistent updates is considerably higher than that of regular clones and exact micro-clones. We finally observe that near-miss micro-clones staying in close proximity to each other have a high tendency of experiencing consistent updates. Our research implies that near-miss micro-clones should be considered equally important as of regular clones and exact micro-clones when making clone management decisions.

KEYWORDS

Code Clones, Micro-Clones, Near-Miss Micro-Clones

ACM Reference Format:
Manishankar Mondal Banani Roy Chanchal K. Roy Kevin A. Schneider. 2020. Investigating Near-Miss Micro-Clones in Evolving Software. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389262

1 INTRODUCTION

Code cloning, copy/pasting code fragments for reusing existing functionalities, is a common but controversial activity of the programmers during software development and maintenance. Such an activity causes the existence of the same or nearly similar code fragments, known as code clones [44, 45], in the code-base of a software system. Two code fragments that are exactly or nearly similar to each other form a clone-pair. A group of similar code fragments forms a clone class. Beside copy/pasting, code clones might be created in a number of other ways [43]. Whatever maybe the reasons behind the existence of code clones, clones are of significant importance from the perspectives of software maintenance and evolution.

A great many studies [1, 2, 11, 12, 14, 17, 19, 21, 23–28, 37, 38, 53, 57] have already been conducted on the detection, analysis, and management of code clones. While a number of clone analysis studies [11, 12, 14] identify some positive impacts of code clones, there is strong empirical evidence [19, 27, 28, 37, 38] of some negative impacts (such as hidden bug-propagation [40], late-propagation [2], high instability [38]) as well. Emphasizing the negative impacts of code clones, researchers suggest to manage them through refactoring [35] or tracking [36]. While clone refactoring is a technique for merging two or more clone fragments from a clone class, clone tracking helps us ensure consistent updates in code clones.

Existing studies [1, 43] report that it is often important to update code clones together consistently during software evolution and maintenance. A number of techniques and tools [9, 20, 31, 58] have been proposed and developed in order to ensure consistent updates in code clones. While the existing techniques and tools support consistent updates in regular code clones only, a recent study conducted by Mondal et al. [39] reports that micro-clones should also be considered equally important for updating consistently. According to Mondal et al. [39], micro-clones are clone fragments that are smaller than the minimum size of the regular clones. They showed that micro-clones have a tendency of getting updated consistently. Their study was restricted to consistent updates in exact micro-clones. However, near-miss micro-clones can also experience consistent updates during evolution. The example in Fig. 1 will explain this.

Fig. 1 shows that two nearly similar source code lines residing in revision 468 of our subject system Ctags were updated consistently in the commit operation which was applied on revision 468. We have highlighted these two lines in revision 468. While these two lines are very similar to each other, they are syntactically different. We also see that the surrounding code of one line is different than the surrounding code of the other line. We consider these two highlighted lines in revision 468 as near-miss micro-clones of each other. We also see the snap-shots of these two near-miss micro-clones in revision 469. We can easily identify the changes between the corresponding snap-shots. We see that the two near-miss micro-clone fragments in revision 468 were updated consistently (i.e., were updated in the same way) in the commit operation. We analyzed the commit log regarding the changes in Fig. 1 and find that the changes occurred for fixing a bug in Ctags. According to this example, near-miss micro-clone fragments can also experience consistent updates

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389262

Code Fragment 1, Revision 468		**Code Fragment 1, Revision 469**

if (longStringLiteral) {
cp = (const unsigned char*) strstr ((const char*) cp, "\"\"\"");
if (cp == NULL) continue;
	Change	if (longStringLiteral) {
		cp = strstr (cp, "\"\"\"");
		if (cp == NULL) continue;

Code Fragment 2, Revision 468		**Code Fragment 2, Revision 469**

skip_indent = indent; } } } } }
if ((cp = (const unsigned char*) strstr ((const char*)cp, "\"\"\"")) != NULL) {
cp += 3;
	Change	skip_indent = indent; } } } } }
		if ((cp = strstr (cp, "\"\"\"")) != NULL) {
		cp += 3;

Figure 1: The figure shows two code fragments, Code Fragment 1 and Code Fragment 2, from two revisions, 468 and 469, of our subject system Ctags. The lines that got changed in these two fragments have also been highlighted. The highlighted line in Code Fragment 1 of revision 468 has been detected as a single line near-miss micro-clone of the highlighted line in Code Fragment 2 of the same revision by our implementation. The Dice Sorensen Coefficient between these two highlighted lines in revision 468 is 88.88%. From the figure we realize that these two single line micro-clones were updated consistently (in the same way) while being propagated to revision 469. Now, if we look at the two code fragments in revision 468, we see that the corresponding lines of the fragments are syntactically dissimilar. These two fragments will not be detected as clones by any of the existing clone detectors even after tuning their detection parameters because of the syntactic dissimilarity. The same is true for the fragments in revision 469 as well. However, from the highlighted lines in the two fragments it is evident that the lines were changed together consistently. Thus, near-miss micro-clones can change together consistently during evolution.

during software evolution and maintenance. However, none of the existing studies investigated consistent updates in near-miss micro-clones. Focusing on this drawback of the existing studies, we investigate consistent co-change tendencies of near-miss micro-clones in our study.

We automatically extract and examine the source code change history of each of our subject systems by using UNIX *diff* tool, identify the consistent updates using Dice Sörensen similarity detection technique [51], and determine how many of these consistent updates occurred in regular as well as exact and near-miss micro-clones. For detecting regular code clones, we used the well-known clone detector NiCad [6]. We also determine how many of the consistent updates occurring in different categories of code clones are related with bugs. From our investigation on the consistent updates from thousands of commit operations of six open-source subject systems written in two different programming languages, we answer the four research questions listed in Table 1. We have the following findings from our investigation.

- The percentage of consistent updates occurring in near-miss micro-clones is significantly higher (according to our statistical significance tests) compared to the percentages of consistent updates occurring in exact micro-clones and regular (exact + near-miss) code clones.
- The percentage of commit operations where near-miss micro-clones experienced consistent updates is significantly higher compared to the corresponding percentages regarding exact micro-clones and regular code clones.
- Consistent updates in near-miss micro-clones have a high tendency of being related with bug-fixes.
- Near-miss micro-clones residing in the same file generally have a higher tendency of experiencing consistent updates compared to near-miss micro-clones in different files.

According to our findings, near-miss micro-clones should not be ignored when making clone management decisions because they exhibit a high tendency of experiencing consistent updates during evolution. We need further investigations towards updating the existing clone management techniques and tools so that these can incorporate exact and near-miss micro-clones as well.

Table 1: Research Questions

SL	Research Question
RQ 1	What percentage of the consistent updates can be experienced by near-miss micro-clones?
RQ 2	What percentage of the consistent updates in different categories of code clones are related to fixing bugs?
RQ 3	How often do near-miss micro-clones experience consistent updates?
RQ 4	Do near-miss micro-clones staying in the same file have a higher tendency of consistent updates compared to those staying in different files?

The rest of the paper is organized as follows. Section 2 describes the terminology, Section 3 discusses the experiment setup and steps, Sections 4 to 7 answer the four research questions listed in Table 1, Section 8 mentions the possible threats to the validity of our experiment results, Section 9 discusses the existing works that are related to our research, and finally, Section 10 makes the concluding remarks by mentioning some possible future work.

2 TERMINOLOGY

This section explains the terms that we have used in our paper.

2.1 Different types of code clones

Our research involves detection and analysis of regular code clones of all three major clone-types: Type 1, Type 2, and Type 3. According to the literature [44, 45], the identical code fragments residing in a software system's code-base are called **Type 1** clones. More elaborately, if two or more code fragments in a code-base are exactly the same disregarding their comments and indentations, then we call these code fragments identical clones or Type 1 clones of one another. Syntactically similar code fragments residing in a software system's code-base are known as **Type 2** clones. Type 2 clones generally get created from Type 1 clones because of renaming identifiers and/or changing data types. Finally, **Type 3** clones (also known as gapped clones) generally get created from both Type 1 or Type 2 clones because of additions, deletions, or modifications of lines in these clones.

2.2 Micro-Clones

According to the literature [39], *Code clones having a size that is smaller than the minimum size of regular code clones are called micro-clones. Micro-clones are not parts of regular code clones.* The minimum

Table 2: Subject Systems

Systems	Lang.	Domains	LLR	SRev	LRev
Ctags	C	Code Definition Generator	33,270	1	774
Camellia	C	Multimedia	85,015	1	140
Carol	Java	Game	25,091	1	1700
Freecol	Java	Game	91,626	1000	1950
JabRef	Java	Reference Management	45,515	1	1545
jEdit	Java	Text Editor	191,804	3791	4000

LLR = LOC in the Last Revision
SRev = Starting Revision LRev = Last Revision

size of a micro-clone fragment is 1 LOC. Our study involves investigating regular clones detected by NiCad [6] clone detector. NiCad detects regular code clones of minimum 5LOC. Thus, the maximum size of a micro clone fragment can be 4LOC in our study.

2.3 Near-miss micro-clones

Micro-clones with textual or syntactic dissimilarities have been termed as near-miss micro-clones in our study. In a previous study, Mondal et al. [39] investigated consistent updates in exact micro-clones. However, near-miss micro-clones were never analyzed. Our study is the first one to investigate consistent updates in near-miss micro-clones. We use Dice Sørensen similarity detection technique for our investigation. Fig. 1 contains an example of single line near-miss micro-clone pair.

2.4 Dice Sørensen Coefficient

Dice–Sørensen coefficient [8, 51, 52] helps us determine the lexical similarity of source code lines. For any two lines, we calculate their similarity from their bigrams (the set of every sequence of two adjacent characters) as follows:

$$Similarity = \frac{2 \times |bigrams(line_1) \cap bigrams(line_2)|}{|bigrams(line_1)| + |bigrams(line_2)|} \quad (1)$$

Here, *Similarity* is the Dice Sörensen similarity between the two lines: $line_1$ and $line_2$. The Dice–Sørensen coefficient reports similarity values in the range 0 and 1, which we express as percentages. A similarity value of 100% indicates that the strings are the same, whereas 0% indicates that the strings are dissimilar. Dice–Sørensen coefficient rewards both common substrings and a common ordering of those substrings. It is robust to changes in word order. It outperforms the existing algorithms, such as Soudex Algorithm, Edit Distance, and Longest Common Subsequence in determining lexical similarity between strings [54].

3 EXPERIMENT SETUP AND STEPS

We conduct our experiment by downloading six subject systems written in Java and C from an on-line SVN repository called SourceForge [42]. Our subject systems have been listed in Table 2. For each system, the table shows the range of revisions that we investigated. The beginning and ending revision numbers of the range are recorded in the table. While for most of the systems we could investigate starting from revision 1, this was not possible for jEdit and Freecol. The starting revisions of jEdit and Freecol are respectively 3791 and 1000 in SourceForge. The former revisions are missing possibly because these two systems were taken under SourceForge from 1000-th and 3791-th revisions respectively. We select the systems listed in Table 2 for our study, because these are of diverse variety in terms of application domains, and size. The revision histories of these systems are also of different lengths. Moreover, the systems are written in two different programming languages. We intentionally select our subject systems emphasizing their diversity so that we can generalize our findings. We perform a number of experiment steps as listed below for each of the systems.

- Downloading each of the revisions (as mentioned in Table 2) of the subject system from the SVN repository.
- Detecting changes between the corresponding source code files of every two consecutive revisions by applying UNIX *diff* operation.
- Detecting regular code clones from each of the revisions by applying the NiCad clone detector [6].
- Identifying consistent updates by following the procedure described in Section 4.
- Analyzing the consistent updates to analyze which updates occurred in regular code clones, exact micro-clones and near-miss micro-clones.
- Identifying the bug-fix commit operations by automatically analyzing the commit messages using the procedure proposed by Mockus and Votta [32].
- Identifying which of the consistent updates occurring in micro-clones were made for fixing bugs.

We will describe the last step in Section 5. We detect regular code clones using the well known clone detector NiCad [6] that can detect all three types of clones (Type 1, Type 2, and Type 3) with high precision and recall [47, 48]. A recent study [55] shows that NiCad is a good choice among the modern clone detectors in term of detection accuracy. As suggested in Wang et al.'s [59] study, we detect regular code clones of at least 5 LOC using NiCad.

In our experiment, we disregarded the changes that occurred to the comments and indentations so that such changes cannot affect our experiment results. One possibility would be to first preprocess the source code by removing comments and blank lines and then doing our experiment. However, such an experiment cannot indicate the real-world scenario of consistent updates in reglar and micro-clones. Thus, we decided not to preprocess the source code. Beside disregarding changes in comments and indentations, we also disregarded changes to source code lines of a single character such as '{' or '}'. In the following sections, we describe our experiments towards answering the research questions.

Identifying the bug-fix commits. Let us assume that we have a subject system. We first retrieve its commit messages by applying the 'SVN log' command. A commit message describes the purpose of the corresponding commit operation. We automatically examine the commit messages using the heuristic proposed by Mockus and Votta [32] to identify those commits that occurred for the purpose of fixing bugs. The way we detect the bug-fix commits was also followed by Barbour et al. [2]. They investigated whether late propagation in code clones [44] are related to bugs. Our study is different. We investigate whether consistent changes in near-miss micro-clones can be related with bug-fixes.

4 INVESTIGATING CONSISTENT UPDATES IN NEAR-MISS MICRO-CLONES

In this section, we answer our first research question (RQ 1) through our investigations.

RQ1: *What percentage of the consistent updates can be experienced by near-miss micro-clones?*

Answering this question is the primary goal of our research. Existing studies show that it is important to update regular code clones consistently during evolution in order to ensure consistency of a software system. On the basis of this finding, a number of techniques [10] and tools [9, 50] have been developed for assisting developers in consistent modification of exact and near-miss regular clones. In a recent study, Mondal et al. [39] showed that micro-clones also require consistent updates during software evolution and maintenance. However, Mondal et al.'s [39] investigation was restricted to the consistent updates in exact micro-clones only. Consistent updates in near-miss micro clones were ignored in their study. In RQ 1, we investigate whether near-miss micro-clones also require consistent updates during evolution. Our finding from RQ 1 can have a considerable impact on the existing clone management technologies. If near-miss micro-clones appear to exhibit a tendency of experiencing consistent updates, then we need further investigations focusing on improving the existing clone management techniques so that they can incorporate exact and near-miss micro-clones as well. We perform our investigation for answering RQ 1 in the following way.

4.1 Detecting pairs of consistent updates

We examine each of the commit operations of a subject system from the very beginning one and determine the changes that occurred in each commit using UNIX *diff*. Let us consider the commit operation c which was applied on revision R of a subject system. One or more source code files in revision R were changed because of the commit operation and the immediate next revision $R + 1$ was created. We obtain the source code changes that occurred in commit c using UNIX *diff* operation. We apply *diff* between the corresponding source code files of the two revisions (R and $R+1$) and determine the changes. *Diff* outputs three types of changes: additions, deletions, and modifications. Two changes (i.e., two updates) are considered consistent if the following conditions hold.

- **Condition 1.** The two changes will be of the same type. For example, if one change is a modification, the other one should also be a modification.
- **Condition 2:** Let us consider that the two changes are two additions: a_1 and a_2. From each addition reported by *diff* we determine two sets: s_1 and s_2. The set s_1 contains the source code line after which the addition was made. The other set, s_2, contains the consecutive source code lines that were added. Now, the two additions a_1 and a_2 are considered consistent if the two sets obtained from a_1 are similar to the corresponding sets obtained from a_2. The technique that we apply for detecting similarity between two sets of source code lines will be described later in this section.
- **Condition 3:** Let us consider that the two changes are two deletions. From each deletion reported by *diff*, we determine the set of consecutive lines that were deleted. If the two sets corresponding to the two deletions are similar, the deletions are considered similar.
- **Condition 4:** Let us consider the two changes are two modifications: m_1 and m_2. From each modification reported by *diff*, we determine two sets: s_1 and s_2. While s_1 contains the consecutive source code lines that were modified, s_2 contains the lines that we obtained after the modification. The two modifications, m_1 and m_2, are considered similar if the two sets obtained from m_1 are similar to the corresponding sets obtained from m_2.

We determine all the changes that occurred in all the commit operations during the whole period of evolution of a candidate software system. By considering the changes in each of the commit operations, we determine pairs of consistent updates.

4.2 Detecting similarity between two sets of source code lines

Let us consider two sets, s_1 and s_2, where each set contains one or more consecutive source code lines. From each of these sets, we determine a single string by sequentially adding source code lines in the set one after another. We then determine the Dice Sörensen similarity between the two strings obtained from the two sets: s_1 and s_2. The details of calculating Dice Sörensen similarity between two strings have been described in Section 2. We obtain the similarity value as a percentage. If the two strings from two sets of source code lines exhibit at least 70% similarity, we consider that the two sets are similar. We select this similarity threshold, because in an existing work, Lozano and Wermelinger [28] considered the same threshold value for detecting similar methods using Dice Sörensen similarity detection technique.

4.3 Categorizing the pairs of consistent updates

Let us consider the pairs of consistent updates that occurred in revision R of a subject system. We categorize these consistent update pairs into the following three categories.

Category 1 (Consistent updates in regular code clones). If both of the updates in a consistent update pair occurred in regular code clones (exact or near-miss), then we consider this pair in **Category 1**. For determining whether a consistent update pair belongs to **Category 1**, we first detect regular code clones (both exact and near-miss clones) from revision R using the NiCad [6] clone detector. After detecting regular clones, we determine whether the two updates in a consistent update pair occurred in two regular clone fragments belonging to the same clone class.

Category 2 (Consistent updates in exact micro-clones). Let us assume that a consistent update pair does not belong to **Category 1**. We consider this pair in **Category 2**, if the two updates (i.e., two changes) in the consistent update pair occurred in two identical micro-clone fragments. We satisfy this condition in the following way.

- If the two consistent updates are additions, then the two additions should be done after two identical source code lines. If two source code lines are identical and they are not included in regular code clones, then they should be considered as exact micro-clones of each other.

Table 3: Number of consistent changes in different categories of code clones

SL	Measures	Ctags	Camellia	Carol	Freecol	Jabref	jEdit
1	Total number of changes during the entire period of evolution of the subject system	3284	2243	6659	13255	13092	5566
2	Total number of pairs of consistent changes that occurred in regular clones during system evolution	166	130	3235	818	1330	70
3	Total number of pairs of consistent changes that occurred in exact micro clones during evolution	296	2196	1262	35890	737	704
4	Total number of pairs of consistent changes that occurred in near-miss micro-clones during evolution	636	3619	2624	60049	2304	9543

- If the two consistent updates are deletions, then the two sets of lines that were deleted in the two updates should be identical. Moreover, the number of deleted lines in each set should be at most four. As these two sets of lines are not included in regular code clones, they are exact micro-clones of each other.
- Finally, if the two consistent updates are modifications, then the two sets of lines that got modified in the two updates should be identical and the number of modified lines in each set should be at most four. As these two sets of source code lines do not belong to regular code clones, they are exact micro-clones.

Category 3 (Consistent updates in near-miss micro-clones). Let us assume that a consistent update pair does not belong to any of the two categories described above. If each of the two sets of lines that experienced the two updates contains at most four lines of code, then these two sets of lines are near-miss micro-clones of each other. Near-miss micro-clones can even be of single lines. Fig. 1 shows consistent updates in two near-miss single line micro-clones. From the figure we see that the single line micro-clones in revision 468 (i.e., the two highlighted lines in the two code fragments in revision 468) are syntactically different. Moreover, the surrounding code of these two highlighted lines are also different. None of the existing clone detectors can detect these two lines as clones of each other even by varying the detection parameters. However, the Dice Sörensen similarity value for these two highlighted lines is 88.88% and this value is greater than the similarity threshold of 70%. Thus, these two lines are near-miss micro-clones of each other. From the figure, we also see that the modified versions of these two lines in revision 469 are near-miss micro-clones of each other as well. Our implemented prototype tool automatically mined this example of consistent updates in near-miss micro-clone fragments by analyzing the evolution history of our subject system Ctags.

We applied our tool on each of our subject systems. It automatically detects the pairs of consistent updates from the entire evolutionary history of the system and categorizes the consistent update pairs into the above three categories.

4.4 Investigation result

Table 3 shows the values of four different measures from our investigation for each of our subject systems. The first measure (the top most one) is the total number of changes that occurred during the whole period of evolution. The second, third, and fourth measures respectively indicate the number of consistent update pairs in regular clones, exact micro-clones, and near-miss micro-clones. Here we should note that Mondal et al. [39] also reported the total number of changes during the entire period of evolution of the subject systems that we have used. The change counts that we have reported in the first row of Table 3 are different than those reported by Mondal et al. [39]. The reason behind this is that they performed preprocessing of the source code before detecting the changes. We previously explained (Section 3) that we do not perform any preprocessing of the source code.

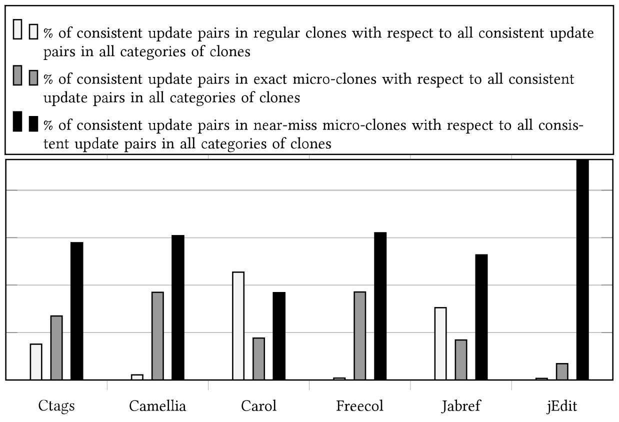

Figure 2: Comparing the percentages of consistent updates in different categories of code clones

From the second, third, and fourth measures in Table 3, we have the following two observations.

- The number of pairs of consistent updates that occurred in near-miss micro-clones is mostly (i.e., for all subject systems except Carol) higher than the number of consistent update pairs in regular code clones and in exact micro-clones.
- The number of consistent update pairs in exact micro-clones is mostly higher than that of regular code clones except for the subject systems: Carol and Jabref. Such an observation agrees with the findings from Mondal et al. [39].

The above observations are also evident from Fig. 2 that shows the percentages of consistent update pairs occurring in regular clones, exact micro-clones, and near-miss micro-clones with respect to all consistent updates in all categories of clones.

Statistical Significance Tests. We wanted to see whether the percentage of similar update pairs occurring in near-miss micro-clones is significantly higher than that of regular code clones. For this purpose we conducted Wilcoxon Signed Rank (WSR) test [29, 30] considering the percentages regarding near-miss micro-clones and regular clones plotted in Fig. 2. We should note that WSR test is non-parametric, and thus, the samples in the test do not need to be normally distributed [29]. This test can be applied to both large and small data sets [29]. We conducted this test considering a significance level of 5%. From our test we see that the percentage regarding near-miss micro-clones is significantly different than the percentage regarding regular code clones with a p-value of 0.046 which is smaller than 0.05. As the percentage regarding near-miss micro-clones is mostly higher, we can say that this percentage is significantly higher than the percentage regarding regular clones.

We also conducted WSR tests [29, 30] in a similar way in order to determine whether the percentage of similar update pairs in near-miss micro-clones is significantly higher than the corresponding percentage for exact micro-clones. We again conducted the test with a significance level of 5% and found that the percentages regarding near-miss micro-clones are significantly different than the percentages regarding the exact micro-clones with a p-value of 0.028 (< 0.05). As the percentages for near-miss micro-clones are always higher than those of exact micro-clones, we can say that the percentages of consistent updates in near-miss micro-clones are significantly higher than those of exact micro-clones.

Answer to RQ 1: According to our experimental results and investigations we decide that the percentage of consistent updates experienced by near-miss micro-clones is significantly higher than the percentages of consistent updates experienced by regular clones and exact micro-clones. In other words, the possibility of occurring consistent updates in near-miss micro-clones is considerably higher than that of regular clones and exact micro-clones.

As we see that near-miss micro-clones have a high tendency of experiencing of consistent updates, we should consider such clones for management. Our next research questions focus on the bug-proneness and management of near-miss micro-clones.

5 INVESTIGATING CONSISTENT UPDATES THAT ARE RELATED WITH BUG-FIXES

From our answer to RQ 1 we realize that near-miss micro-clones have a significantly higher tendency of experiencing consistent updates during evolution compared to regular clones and exact micro-clones. Such an observation inspires us to investigate how many of the consistent updates in different categories are related with fixing bugs. If we observe that a high percentage of the consistent updates occurring in near-miss micro-clones are made for fixing bugs, it can establish the importance of near-miss micro-clones in software systems. Through our investigation in this section, we answer the second research question (RQ 2).

RQ 2: *What percentage of the consistent updates in different categories of code clones are related to fixing bugs?*

5.1 Investigation Procedure

As we did in RQ 1, we identify the pairs of consistent updates in three categories of code clones: regular clones, exact micro-clones, and near-miss micro-clones. We also determine the bug-fix commits following the procedure described in Section 3. We now determine which of the consistent updates were related with fixing bugs. If a pair of consistent updates occur in a bug-fix commit operation, we realize that the consistent updates were necessary for fixing a bug. We consider such a pair as a pair of consistent updates related with bug-fix. For different categories of code clones, we determine the number of consistent update pairs that are related with bug-fixes. Table 4 shows this number for each of the three clone categories of each of our subject systems. We finally determine the percentage of consistent update pairs that are related with bug-fix with respect to all pairs of consistent updates for each clone category of each subject system. Fig. 3 shows these percentages for our subject systems.

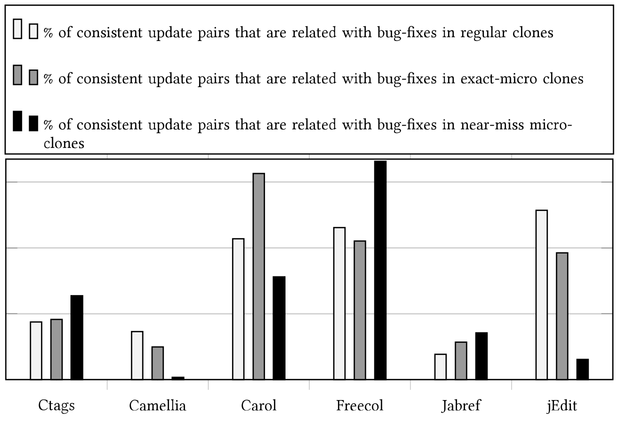

Figure 3: Comparing the percentages of consistent updates that are related with bug-fixes in different categories of code clones

5.2 Investigation Result

From Table 4 we realize that the number of consistent update pairs that are related with fixing bugs in near-miss micro-clones is mostly the highest (except for two subject systems: Camellia and Carol) compared to regular clones and exact micro-clones. Fig. 3 makes us understand that the percentage of consistent update pairs that are related with bug-fix in near-miss micro-clones is the highest for three subject systems: Ctags, Freecol, and Jabref. For two systems, Camellia and jEdit, the percentage regarding regular clones is the highest. For the remaining system, Carol, the percentage regarding exact micro-clones is the highest.

We performed Wilcoxon Signed Rank tests [29] to determine whether the percentages regarding near-miss micro-clones are significantly different than the percentages regarding the other two categories of clones. However, we found that the percentages regarding near-miss micro-clones are not significantly different than the percentages regarding the other two clone categories.

Answer to RQ 2: According to our investigation and analysis, the percentage of consistent update pairs that are related with fixing bugs in near-miss micro-clones can sometimes be very high (for example 66% for our subject system Freecol). Thus, near-miss micro-clones should not be ignored. By considering such clones for management, we can likely minimize bugs in the source code.

6 INVESTIGATING THE OFTENNESS OF CONSISTENT UPDATES IN NEAR-MISS MICRO-CLONES

In this section we answer our third research question (RQ 3) regarding how frequently consistent updates occur in near-miss micro-clones during evolution.

Table 4: Number of consistent update pairs that are related with bug-fixes in different categories of code clones

SL	Measures	Ctags	Camellia	Carol	Freecol	Jabref	jEdit
1	Total number of bug-fixes during the entire period of evolution of a subject system	300	36	137	336	233	58
2	Total number of consistent update pairs that are related with bug-fix in regular clones (exact + near-miss)	29	19	1384	378	102	36
3	Total number of consistent update pairs that are related with bug-fix in exact micro-clones	54	218	790	15118	84	271
4	Total number of consistent update pairs that are related with bug-fix in near-miss micro-clones	162	26	819	39813	327	581

Table 5: The number of commits where different categories of code clones experienced consistent updates

SL	Measures	Ctags	Camellia	Carol	Freecol	Jabref	jEdit
1	Total number of commits where there were some changes to the source code	447	126	454	836	860	145
2	No. of commits where regular code clones (both exact and near-miss) experienced consistent updates	41	38	129	121	173	12
3	No. of commits where exact micro-clones experienced consistent updates	47	36	82	258	138	80
4	No of commits where near-miss micro-clones experienced consistent updates	108	56	141	441	285	90

RQ 3: *How often do the consistent updates occur in near-miss micro-clones?*

From our answer to RQ 1 we realize that near-miss micro-clones generally experience a higher percentage of consistent updates compared to exact micro-clones, and regular code clones. In RQ 3, we investigate whether near-miss micro-clones experience consistent updates in many of the commit operations or not. If we find that near-miss micro-clones undergo consistent updates in many commits during system evolution, then it is important for the modern clone tracking tools to consider near-miss micro-clones for tracking so that consistent updates in such code clones can be done with reduced effort.

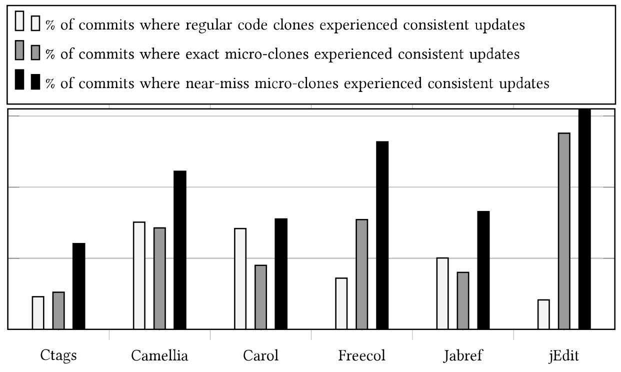

Figure 4: Percentages of commits where code cones of different categories experienced consistent updates

6.1 Investigation Procedure

As we did in RQ 1, we identify consistent updates in three categories of code clones: (1) regular code clones, (2) exact micro-clones, and (3) near-miss micro-clones during the whole period of evolution of a subject system. For answering RQ 3, we also identify which of the commit operations made changes to the source code. Then, for each of the three categories of code clones, we determine the percentage of commits where the code clones of that category experienced consistent updates.

Let us assume that the number of commit operations where there were some changes to the source code is C during the whole period of evolution of a subject system. The number of commits where a particular category of code clones experienced consistent updates is $C_{consistent}$. Then, the percentage of commits where that particular category of code clones experienced consistent updates is calculated using the following equation (Eq. 2).

$$P = \frac{C_{consistent} \times 100}{C} \tag{2}$$

Here, P is the percentage of commits with consistent updates.

6.2 Investigation result

For each of our subject systems, Table 5 shows four measures that we obtained from our investigation. These measures are: (1) total number of commit operations where there were some changes to the source code, (2) number of commit operations where regular code clones experienced consistent updates, (3) number of commits where exact micro-clones experienced consistent updates, and (4) number of commits where near-miss micro-clones experienced consistent updates. In a particular commit operation, more than one category of code clones can experience consistent updates. Thus, the commit operations in the second, third, and fourth measures are not disjoint. We determine the percentages of these last three measures with respect to the first measure (total number of commits with changes to the source code) as indicated in Eq. 2 and plot these percentages in the bar-graph of Fig. 4.

Fig. 4 shows that the percentage of commits where near-miss micro-clones experienced consistent updates is always higher compared to the percentages of commits where regular code clones (both exact and near-miss) and exact micro-clones experienced consistent updates. In other words, near-miss micro-clones experience consistent updates for the longest duration among the three categories of code clones. We also wanted to investigate whether the percentages regarding near-miss micro-clones in Fig. 4 are significantly higher compared to the percentages regarding regular clones and exact micro-clones. For this purpose, we again conduct the Wilcoxon Signed Rank tests [29, 30] in a similar way as we did Section 4. We consider a significance level of 5% for conducting the tests. According to our test results, the percentages regarding near-miss micro-clones are significantly higher than the percentages regarding regular code clones with a p-value of 0.03 (< 0.05) for two-tailed test case. We also obtain the same result regarding the tests between near-miss micro-clones and exact micro-clones. The percentages regarding near-miss micro-clones are significantly higher compared to the percentages regarding exact micro-clones with a p-value of 0.028 which is less than 0.05.

Answer to RQ 2. According to our investigation results and analysis, near-miss micro-clones experience consistent updates for a significantly higher percentage of commit operations during evolution compared to regular clones and exact micro-clones.

As near-miss micro-clones experience consistent updates for the longest duration, it is important to consider such clones for management such as tracking. However, refactoring near-miss micro-clones might not be a good idea because they are small in size. Existing research [62] indicates that larger code clones are more promising for refactoring to the programmers. Further research should be conducted on the clone tracking techniques so that these techniques can consider near-miss micro-clones for tracking as well.

Mondal et al. [39] showed that micro-clones can be three times as much as the regular clones in a software system. We also experienced a similar scenario in our research by detecting code clones using the same clone detector and detection parameters that they used. We wanted to determine the total number of near-miss micro-clones in our subject systems. However, the existing clone detectors cannot detect near-miss micro-clones with syntactic differences well. Fig. 1 shows an example of consistent updates in single line near-miss micro-clones with syntactic dissimilarity. None of the existing clone detectors can detect it. Possibly, we need further investigations on customizing the existing clone detectors to incorporate Dice Sörensen similarity detection technique for detecting near-miss micro-clones with syntactic differences. We should note that for answering research questions (RQ 1, RQ 2, and RQ 3), we detected consistent updates in exact and near-miss micro-clones using Dice Sörensen similarity detection technique.

7 IDENTIFYING NEAR-MISS MICRO-CLONES THAT ARE IMPORTANT FOR MANAGEMENT

In this section we answer our fourth research question (RQ 4) on identifying near-miss micro-clones that can be important for management such as tracking.

RQ 4. *Do near-miss micro-clones that experience consistent updates generally reside in the same file?*

From our answer to the previous research questions we realize that it is important to consider near-miss micro-clones for management such as tracking. When deciding about clone management, we should identify which of the near-miss micro-clones should be considered important for management. Intuitively, near-miss micro-clones that exhibit a high tendency of experiencing consistent updates should be considered important for management. In RQ 4, we particularly investigate whether near-miss micro-clones residing in the same file have a higher tendency of experiencing consistent updates compared to those residing in different files. We perform our investigation for answering RQ 4 in the following way.

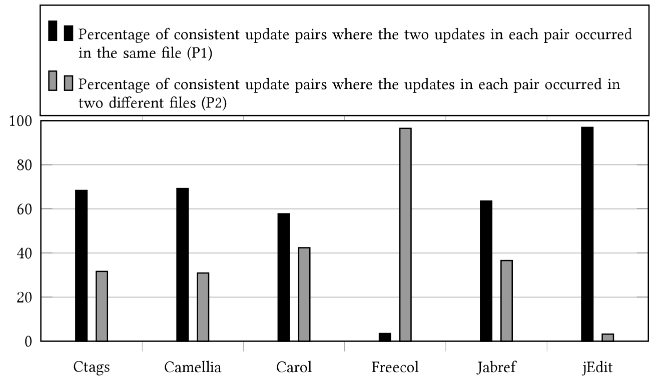

Figure 5: Percentage of consistent updates that occurred in the same or different files

7.1 Investigation procedure

We analyze each of the commit operations from the beginning one and identify pairs of consistent updates in near-miss micro-clones as we did before. For each such pair, we determine whether the two updates in the pair took place in the same source code file or in different files. By analyzing the all the commit operations from the whole period of evolution of a subject system we determine the following two measures:

- **Measure 1.** The total number of the pairs of consistent updates where the two updates in each pair took place in the same file.
- **Measure 2.** The total number of the pairs of consistent updates where the two updates in each pair occurred in two different source code files.

We show these two measures in Table 6. This table also shows the number of consistent update pairs that occurred in the near-miss micro-clones of different subject systems during their evolution. From the data recorded in Table 6 we determine the following two percentages:

- **P1.** This is the percentage of consistent update pairs each having updates occurring in the same file with respect to all consistent update pairs in near-miss micro-clones. We calculate this percentage in the following way.

$$P1 = (Measure\ 1 \times 100)/NPCU \quad (3)$$

Here, NPCU is the number of pairs of consistent updates that occurred in near-miss micro-clones during the whole period of evolution of a subject system. The first row in Table 6 represents this data (NPCU).

- **P2.** This is the percentage of consistent update pairs where the updates in each pair occurred in two different files. We calculate this percentage in the following way.

$$P2 = (Measure\ 2 \times 100)/NPCU \quad (4)$$

We show these percentages in Fig. 5 in order to visually realize the comparative scenario of these two percentages. We see that the percentage P1 (the percentage of consistent update pairs each having updates occurring in the same file) is always higher than

Table 6: Statistics regarding the consistent updates occurring in the same or different files

SL	Measures	Ctags	Camellia	Carol	Freecol	Jabref	jEdit
1	No. of pairs of consistent changes that occurred in near-miss micro-clones	636	3619	2624	60049	2304	9543
2	Total number of the pairs of consistent updates where the two updates in each pair took place in the same file	435	2502	1514	2085	1463	9242
3	Total number of the pairs of consistent updates where the two updates in each pair took place in different files	201	1117	1110	57964	841	301

P2 (the percentage of consistent update pairs where the updates in each pair occurred in two different files) except for our subject system Freecol. In other words, near-miss micro-clones from the same source code file mostly have a higher tendency of experiencing consistent updates compared to near-miss micro-clones residing in different files. We also wanted to see whether the percentage, P1, is significantly higher than the other percentage. We performed Wilcoxon signed Rank tests [29, 30] for this purpose as we did for answering our previous research questions. We conducted the tests considering a significance level of 5%. However, according to our tests, the two percentages P1 and P2 are not significantly different than each other.

Answer to RQ 4. According to our investigation result and analysis, near-miss micro-clones residing in the same file generally have a higher tendency (according to our statistical significance test) of experiencing consistent updates compared to those that reside in different files.

Such a finding is important for making decisions regarding tracking code clones. Clone tracking is the technique for ensuring consistent updates to code clones. If a programmer attempts to make changes to a code fragment, the clone tracker automatically finds the clones of the fragment so that the programmer can consistently change these clones as well. Our finding from RQ 4 can be useful to rank the near-miss micro-clones of the target code fragment. The near-miss micro-clones residing in the same file as of the target code fragment can be given a higher priority compared to the near-miss micro-clones that reside in different files.

8 THREATS TO VALIDITY

In our research, we detect regular code clones from our subject systems using the NiCad clone detector [6]. For different settings of NiCad, the clone detection results for regular code clones can be different, and thus, the statistics reported in our research can also be different. However, the settings that we have used for NiCad are considered standard [46]. NiCad has been shown to exhibit high precision and recall with these settings [47, 48, 55]. Thus, our findings are important and can have a considerable impact on the evolution and maintenance of software systems.

In our experiment we did not study enough subject systems to be able to generalize our findings. However, we selected our candidate systems emphasizing their diversity in sizes and revision history lengths. Thus, we believe that our findings cannot be attributed to a chance. Our experiment results regarding near-miss micro-clones should be considered important.

We detect bug-fix commits in our research using the technique proposed by Mockus and Votta [32]. Such a technique which was also used by Barbour et al. [2] can sometimes detect a non bug-fix commit as a bug-fix commit. However, Barbour et al. [2] showed that the technique has an accuracy of 87% in detecting bug-fix commits. Thus, we believe that our findings regarding the occurrence of consistent changes in different categories of code clones in bug-fix commits is reasonable.

9 RELATED WORK

Code clones have been a matter of great importance from the perspectives of software maintenance and evolution. A great many studies have investigated detection [6, 7, 13, 18, 43, 45, 46, 48, 55], analysis [1–3, 5, 11, 12, 14–17, 19, 21, 22], and management [9, 10, 20, 33, 35] of code clones in different ways. Our research in this paper focuses on near-miss micro-clones.

The existing study which is most relevant to our study was conducted by Mondal et al. [39]. They investigated the importance of micro-clones during the evolution of a software system. They showed that micro-clones can also experience consistent updates during software maintenance and evolution like the regular code clones. However, they only investigated consistent updates to exact micro-clones in their study. In our study, we investigate consistent updates in near-miss micro-clones by using Dice Sörensen similarity detection technique [52] and make a comparison among the consistent updates experienced by near-miss micro-clones, exact micro-clones, and regular code clones. We found that near-miss micro-clones have a significantly higher tendency of experiencing consistent updates during evolution compared to the other two categories of code clones. Thus, such code clones should also be considered for management.

Betterburg et al. [4] conducted a study on the consistent and inconsistent changes to regular code clones by considering different releases of a number of subject systems. They detected regular code clones in their study using the SimScan clone detector. They found that a very little proportion of the regular code clones create faults in the software systems because of being changed inconsistently. In other words, most of the regular code clones undergo consistent changes during evolution. While they investigate consistent and inconsistent changes to regular code clones, we study consistent changes to near-miss micro-clones in our study. We found that near-miss micro-clones have a higher tendency of experiencing consistent updates than the regular code clones.

Jurgens et al. [19] investigated whether inconsistent changes to regular code clones introduce faults in a software system. They found that every second to third unintentional inconsistent change to a clone fragment leads to a fault. They conducted their study using the clone detection tool called ConQAT and investigated regular clones of at least 10 lines of code. In our study, we investigate the consistent co-change tendencies of near-miss micro-clones. We find that near-miss micro-clones have a higher tendency of experiencing consistent updates than regular code clones, thus, near-miss micro-clones should also be considered for management.

Saha et al. [49] studied the genealogies of regular code clones at release level and found that most of the clone genealogies were updated consistently during evolution. They detected regular code clones using CCFinderX clone detector and reported that we should focus on managing code clones through tracking so that we can update them consistently with reduced effort. In our study we investigate the intensities of consistent updates in near-miss micro-clones and found that such code clones have a higher tendency of experiencing consistent updates compared to regular code clones.

Krinke [23] measured how consistently the code clones are changed during maintenance using Simian [56] (clone detector) on Java, C and C++ code bases considering exact regular clones only. He found that clone groups are mostly updated consistently during evolution. In two other studies [24, 25] he showed that cloned code is more stable than non-cloned code [21]. While Krinke's studies have only considered regular code clones, we investigate near-miss micro-clones in our study and find that near-miss micro-clones have a strong tendency of experiencing consistent updates during evolution.

A number of studies have also investigated regular code clones from management perspectives such as clone refactoring and tracking. Kim et al. [22] investigated genealogies of regular code clones and reported that aggressive refactoring of regular code clones is not necessary during evolution. Mondal et al. [35, 36] investigated which of the regular code clones can be important for management such as refactoring or tracking. They applied the technique of mining association rules for investigating co-evolution of regular code clones and found that only a very small proportion of the regular code clones can be important for management. We see that none of these studies investigated near-miss micro-clones from the perspectives of clone management.

A number of tools for tracking regular code clones also exist. Miller and Myer [31] implemented 'Simultaneous Editing' tool to automatically propagate changes from one clone fragment to it's peer clone fragments. Their tool can consider exact regular code clones only. Toomim et al. [58] introduced the tool called 'Linked Editing' for consistently updating exact regular code clones. Duala-Ekoko and Robillard [10] introduced CRD (Clone Region Descriptor) based clone tracking technique and implemented 'CloneTracker' [9] for tracking the evolution of regular code clones only. Jablonski and Hou [20] introduced CReN that can help us in consistent renaming of regular code clones. We see that none of these existing clone tracking tools consider near-miss micro-clones for tracking. However, our research indicates that near-miss micro-clones have a significantly higher tendency of experiencing consistent updates during evolution. Thus, the existing clone tracking tools should be investigated so that they can consider near-miss micro-clones for tracking.

A number of clone refactoring techniques and tools [41, 61] also exist. However, these techniques and tools can only incorporate regular code clones. A number of studies [16, 17, 26, 34] investigated the bug-proneness of regular code clones during software evolution and maintenance. These studies have ignored exact as well as near-miss micro-clones. However, research shows that near-miss micro-clones should also be considered important for management and they should be investigated from different management perspectives.

While the existing studies, techniques, and tools have only considered different types of regular code clones by ignoring the micro-clones because of their small size, our findings from our research imply that micro-clones should not be ignored. In our research, we make a comparison among the consistent co-change tendencies of regular clones, exact micro-clones, and near-miss micro-clones. We find that near-miss micro-clones have the highest tendency of experiencing consistent updates. We also observe that near-miss micro-clones experience consistent updates for the longest duration (i.e., for the highest number of commit operations) during software evolution. Thus, such code clones should not be ignored, rather, they should be considered for management. Our findings can have a significant impact on the current clone detection and management technologies. Further research will be required for improving the existing technologies so that they can incorporate exact as well as near-miss micro-clones.

10 CONCLUSION

In this paper, we investigate the importance of near-miss micro-clones in software evolution and maintenance. We automatically detect consistent changes from the entire period of evolution of our subject systems using UNIX *diff* operation and Dice Sörensen similarity detection technique. We identify which consistent changes occurred in regular code clones, and which ones occurred in exact micro-clones and near-miss micro-clones. According to our investigation on thousands of revisions of six open-source subject systems written in two different programming languages:

- Near-miss micro-clones have a significantly higher tendency of experiencing consistent updates compared to exact micro-clones and regular code clones.
- Near-miss micro-clones experience consistent updates for the longest duration (i.e., for the highest number of commit operations) among the three categories of code clones (regular clones, exact micro-clones, near-miss micro-clones).
- Consistent updates in near-miss micro-clones have a high tendency of being related with fixing bugs.

From such findings we realize that near-miss micro-clones should also be considered important for management like regular clones and exact micro-clones. When making decisions regarding managing near-miss micro-clones, we should identify which ones should be considered important for management. According to our investigation for answering the fourth research question, near-miss micro-clones residing in the same file generally have a higher tendency of experiencing consistent updates compared to those that reside in different files. Thus, near-miss micro-clones residing in the same file should be given a high priority for managing. The existing clone trackers do not consider near-miss micro-clones for tracking. Future investigations on updating the existing clone trackers to track near-miss micro-clones can add value to the existing clone management research.

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] L. Aversano, L. Cerulo, and M. D. Penta, "How clones are maintained: An empirical study", Proc. *CSMR*, 2007, pp. 81 – 90.

[2] L. Barbour, F. Khomh, Y. Zou, "Late Propagation in Software Clones", Proc. *ICSM*, 2011, pp. 273 – 282.

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

[4] 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.

[5] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, N. A. Kraft, "Measuring the Efficacy of Code Clone Information in a Bug Localization Task: An Empirical Study", Proc. *ESEM*, 2011, pp. 20 – 29.

[6] J. R. Cordy, C. K. Roy, "The NiCad Clone Detector", Proc. *ICPC Tool Demo*, 2011, pp. 219 – 220.

[7] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code", *IEEE Trans. Software Engineering*, 2002, 28(7):654 – 670.

[8] L. R. Dice, "Measures of the Amount of Ecologic Association Between Species", *Ecology*, 1945, 26(3): 297 – 302.

[9] E. Duala-Ekoko, M. P. Robillard, "CloneTracker: Tool Support for Code Clone Management", Proc. *ICSE*, 2008, pp. 843 – 846.

[10] E. Duala-Ekoko, M. P. Robillard, "Tracking Code Clones in Evolving Software", Proc. *ICSE*, 2007, pp. 158 – 167.

[11] N. Göde, Rainer Koschke, "Frequency and risks of changes to clones", Proc. *ICSE*, 2011, pp. 311 – 320.

[12] N. Göde, J. Harder, "Clone Stability", Proc. *CSMR*, 2011, pp. 65 – 74.

[13] N. GÃűde and R. Koschke, "Incremental Clone Detection," 2009 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, 2009, pp. 219-228.

[14] 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.

[15] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, E. Lee, "Experience of Finding Inconsistently-Changed Bugs in Code Clones of Mobile Software", Proc. *IWSC*, 2012, pp. 94 – 95.

[16] J. F. Islam, M. Mondal, C. K. Roy, "Bug Replication in Code Clones: An Empirical Study", Proc. *SANER*, 2016, pp. 68 - 78.

[17] L. Jiang, Z. Su, E. Chiu, "Context-Based Detection of Clone-Related Bugs", Proc. *ESEC-FSE*, 2007, pp. 55 – 64.

[18] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones", Proc. *ICSE*, 2007, pp. 96 - 105.

[19] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, "Do Code Clones Matter?", Proc. *ICSE*, 2009, pp. 485 – 495.

[20] 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.

[21] C. Kapser, M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software", *Empirical Software Engineering*, 2008, 13(6): 645 – 692.

[22] M. Kim, V. Sazawal, D. Notkin, G. C. Murphy, "An empirical study of code clone genealogies", Proc. *ESEC-FSE*, 2005, pp. 187 – 196.

[23] J. Krinke, "A study of consistent and inconsistent changes to code clones", Proc. *WCRE*, 2007, pp. 170 – 178.

[24] J. Krinke, "Is cloned code more stable than non-cloned code?", Proc. *SCAM*, 2008, pp. 57 – 66.

[25] J. Krinke, "Is Cloned Code older than Non-Cloned Code?", Proc. *IWSC*, 2011, pp. 28 – 33 .

[26] J. Li, M. D. Ernst, "CBCD: Cloned Buggy Code Detector", Proc. *ICSE*, 2012, pp. 310 – 320.

[27] A. Lozano, M. Wermelinger, "Tracking clones' imprint", Proc. *IWSC*, 2010, pp. 65 – 72.

[28] A. Lozano, M. Wermelinger, "Assessing the effect of clones on changeability", Proc. *ICSM*, 2008, pp. 227 – 236.

[29] Wilcoxon Signed Rank Test. https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

[30] Wilcoxon Signed Rank Test online calculator. http://www.socscistatistics.com/tests/signedranks/Default2.aspx

[31] R. C. Miller, B. A. Myers. "Interactive simultaneous editing of multiple text regions.", Proc. *USENIX 2001 Annual Technical Conference*, 2001, pp. 161 – 174.

[32] A. Mockus, L. G. Votta, "Identifying Reasons for Software Changes using Historic Databases", Proc. *ICSM*, 2000, pp. 120 – 130.

[33] M. Mondal, C. K. Roy, K. A. Schneider, "SPCP-Miner: A Tool for Mining Code Clones that are Important for Refactoring or Tracking", Proc. *SANER*, 2015, 5pp. (to appear).

[34] M. Mondal, C. K. Roy, K. A. Schneider, "A Comparative Study on the Bug-proneness of Different Types of Code Clones", Proc. *ICSME*, 2015, pp. 91 - 100.

[35] M. Mondal, C. K. Roy, K. A. Schneider, "Automatic Ranking of Clones for Refactoring through Mining Association Rules", Proc. *CSMR-WCRE*, 2014, pp. 114 – 123.

[36] M. Mondal, C. K. Roy, K. A. Schneider, "Automatic Identification of Important Clones for Refactoring and Tracking", Proc. *SCAM*, 2014, pp. 11 – 20.

[37] 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.

[38] M. Mondal, C. K. Roy, K. A. Schneider, "An Empirical Study on Clone Stability", *ACM SIGAPP Applied Computing Review*, 2012, 12(3): 20 – 36.

[39] M. Mondal, C. K. Roy and K. A. Schneider, "Micro-clones in evolving software," 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, 2018, pp. 50-60.

[40] M. Mondal, C. K. Roy, K. A. Schneider, "Bug Propagation through Code Cloning: An Empirical Study", Proc. *ICSME*, 2017, pp. 227 – 237.

[41] D. Mazinanian, N. Tsantalis, R. Stein, Z. Valenta, "JDeodorant: Clone Refactoring", Proc. *ICSE*, 2016, pp.14 – 22.

[42] Online SVN repository: http://sourceforge.net/

[43] C. K. Roy, J. R. Cordy, "A Survey on Software Clone Detection Research", *Technical Report No. 2007-541*, 2007, School of Computing QueenâĂŹs University, pp. 1 - 115.

[44] 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.

[45] C. K. Roy, "Detection and analysis of near-miss software clones", Proc. *ICSM*, 2009, pp. 447 – 450.

[46] C. K. Roy, J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization", Proc. *ICPC*, 2008, pp. 172 – 181.

[47] 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.

[48] C. K. Roy, J. R. Cordy, "A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools", Proc. *Mutation*, 2009, pp. 157 – 166.

[49] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, K. A. Schneider, "Evaluating Code Clone Genealogies at Release Level: An Empirical Study", Proc. *SCAM*, 2010, pp. 87 – 96.

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

[51] T. Sørensen, "A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons", *Kongelige Danske Videnskabernes Selskab*, 1948, 5(4): 1 – 34.

[52] Sørenson–Dice coefficient: https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient

[53] D. Steidl, N. Göde, "Feature-Based Detection of Bugs in Clones", Proc. *IWSC*, 2013, pp. 76 – 82.

[54] Strike A Match: http://www.catalysoft.com/articles/strikeamatch.html

[55] J. Svajlenko, C. K. Roy, "Evaluating Modern Clone Detection Tools", Proc. *ICSME*, 2014, pp. 321 – 330.

[56] Simian-similarity analyser. http://www.redhillconsulting.com.au/products/simian/

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] M. F. Zibran, C. K. Roy, "Conflict-aware Optimal Scheduling of Code Clone Refactoring", *IET Software*, 2013, 7(3): 167 – 186.

[62] M. F. Zibran, R. K. Saha, C. K. Roy, K. A. Schneider,"Evaluating the Conventional Wisdom in Clone Removal: A Genealogy-based Empirical Study", *SAC*, 2013, pp. 1123 – 1130.