A Comparative Study on the Bug-Proneness of Different Types of Code Clones

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

***Abstract*—Code clones are defined to be the exactly or nearly similar code fragments in a software system's code-base. The existing clone related studies reveal that code clones are likely to introduce bugs and inconsistencies in the code-base. However, although there are different types of clones, it is still unknown which types of clones have a higher likeliness of introducing bugs to the software systems and so, should be considered more important for managing with techniques such as refactoring or tracking. With this focus, we performed a study that compared the bug-proneness of the major clone-types: Type 1, Type 2, and Type 3. According to our experimental results on thousands of revisions of seven diverse subject systems, Type 3 clones exhibit the highest bug-proneness among the three clone-types. The bug-proneness of Type 1 clones is the lowest. Also, Type 3 clones have the highest likeliness of being co-changed consistently while experiencing bug-fixing changes. Moreover, the Type 3 clones that experience bug-fixes have a higher possibility of evolving following a *Similarity Preserving Change Pattern* (SPCP) compared to the bug-fix clones of the other two clone-types. From the experimental results it is clear that Type 3 clones should be given a higher priority than the other two clone-types when making clone management decisions. We believe that our study provides useful implications for ranking clones for refactoring and tracking.**

I. Introduction

Code cloning is a common yet controversial software engineering practice which is often employed by programmers during software development and maintenance for repeating common functionalities. Cloning refers to the task of copying a code fragment from one place of a code-base and pasting it to some other places with or without modifications [42]. The original code fragment (i.e., the code fragment from which the copies were made) and the pasted code fragments become clones of one another. Two exactly or nearly similar code fragments form a clone pair. A group of similar code fragments forms a clone class.

Code clones are of great importance from the perspectives of software maintenance and evolution. A great many studies [1], [2], [9]–[11], [13], [15], [17], [19]–[22], [24], [25], [35], [36], [48], [50] have already been conducted on the impacts of clones on the evolution and maintenance of software systems. While some of these studies [1], [10], [11], [17], [19]–[21] identify some positive impacts of code clones, a number of studies [2], [9], [13], [15], [22], [24], [25], [35], [36], [48] have shown empirical evidence of strong negative impacts of code clones such as hidden bug propagation [22], late propagation [2], unintentional inconsistencies [2], [9], and high instability [36]. Because of these negative impacts, code clones are considered to be the number one bad smell in a software system's code-base.

According to a number of studies [2], [4], [9], [12], [13], [22], [23], [48], [53], code clones are directly related to bugs and inconsistencies in a software system. However, although there are different types of code clones, none of the existing studies investigate the comparative bug-proneness of these different clone-types. Such an investigation is important because it can help us identify which type(s) of clones have the highest tendency of exhibiting bug-proneness and thus, should be considered to be the most important ones for management such as refactoring and tracking. Focusing on this issue in this research work we investigate the comparative bug-proneness of the major types of code clones: Type 1, Type 2, and Type 3 (defined in Section II). In particular, we answer four important research questions listed in Table I. According to our in-depth investigation on thousands of revisions of seven diverse subject systems written in two different programming languages (C and Java) we can state that:

(1) Type 3 clones have a higher bug-proneness compared to Type 1 and Type 2 clones. The bug-proneness of Type 1 clones is the lowest among the three clone-types. Our statistical significance tests show that Type 3 clones have a significantly higher bug-proneness than Type 1 clones.

(2) Type 3 clones have the highest likeliness of being co-changed (i.e., getting changed together) consistently among the three clone-types when changed to fix a bug.

(3) Type 3 bug-fix clones have the highest possibility of evolving following a *Similarity Preserving Change Pattern* called SPCP. According to our previous studies [33], [34], SPCP clones (i.e., clones that evolve following a Similarity Preserving Change Pattern) are the most important ones to consider for clone management.

Our experimental results imply that Type 3 clones should be given a higher priority than the other two clone-types when making clone management decisions (such as clone refactoring, or tracking) and our findings (points 2 and 3 above) can be used to rank code clones during clone management. In our previous studies [33], [34] we detected and ranked SPCP clones for refactoring and tracking on the basis of their co-change tendencies. However, we should also consider their bug-proneness. Our implemented prototype tool is capable of automatically detecting SPCP clones that exhibited bug-proneness during evolution. Thus, it can help us rank clones considering their bug-proneness too.

TABLE I
RESEARCH QUESTIONS

SL	Research Question
RQ 1	Which clone types have a higher possibility of experiencing bug fixing changes?
RQ 2	Do the clone fragments from the same clone class co-change (i.e., change together) consistently during a bug-fix?
RQ 3	What proportion of the clone fragments that experienced bug-fixing changes are SPCP clones?

The rest of the paper is organized is as follows: Section II describes the terminology, Section III discusses the experimental steps, Section IV answers the research questions by presenting and analyzing the experimental results, Section V mentions the possible threats to validity, Section VI discusses the related work, and finally, Section VII concludes the paper by mentioning possible future work.

II. TERMINOLOGY

Types of clones. We conduct our experiment considering both exact (Type 1) and near-miss clones (Type 2 and Type 3 clones). As is defined in the literature [42], [43], if two or more code fragments in a particular code-base are exactly the same disregarding the comments and indentations, these code fragments are called exact clones or Type 1 clones of one another. Type 2 clones are syntactically similar code fragments. In general, Type 2 clones are created from Type 1 clones because of renaming identifiers or changing data types. Type 3 clones are mainly created because of additions, deletions, or modifications of lines in Type 1 or Type 2 clones.

Similarity Preserving Change Pattern (SPCP). In our previous studies [33], [34] we showed that the code clones that evolve following a *Similarity Preserving Change Pattern* (SPCP) are the most important ones for refactoring or tracking. A *Similarity Preserving Change Pattern* consists of a *Similarity Preserving Change* and/or a *Re-synchronizing Change*.

Similarity Preserving Change. Let us consider two code fragments that are clones of each other in a particular revision of a subject system. A commit operation was applied to this revision, and any one or both of these code fragments (i.e., clone fragments) received some changes. However, in the next revision (created because of the commit operation) if these two code fragments are again considered clones of each other (i.e., the code fragments preserve their similarity), then we say that the code fragments received a *Similarity Preserving Change* in the commit operation.

Re-synchronizing Change. A re-synchronizing change consists of a *diverging change* followed by a *converging change*. Let us consider two code fragments that are clones of each other in a particular revision. A commit operation C_i was applied to this revision, and any one or both of the fragments received some changes in such a way that the code fragments were not considered clones of each other in the next revision. We say that the code fragments experienced a *diverging change*. However, in a later commit operation C_{i+n} ($n >= 1$) any one or both of the code fragments received some changes, and because of these changes the code fragments again became clones of each other. We say that the code fragments experienced a *converging change* in commit C_{i+n}. A *diverging change* followed by a *converging change* is termed a *re-synchronizing change*.

TABLE II
SUBJECT SYSTEMS

Systems	Lang.	Domains	LLR	Revisions
Ctags	C	Code Def. Generator	33,270	774
Camellia	C	Image Processing Library	89,063	170
BRL-Cad	C	3-D Modeling	39,309	735
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

LLR = LOC in the Last Revision

III. EXPERIMENTAL STEPS

We perform our investigation on seven subject systems (Table II) downloaded from an on-line SVN repository [39].

A. Preliminary Steps

We perform the following preliminary steps before analyzing bug-proneness: **(1)** Extraction of all revisions (as mentioned in Table II) of each of the subject systems from the online SVN repository; **(2)** Method detection and extraction from each of the revisions using CTAGS [6]; **(3)** Detection and extraction of code clones from each revision by applying the NiCad [5] clone detector; **(4)** Detection of changes between every two consecutive revisions using *diff*; **(5)** Locating these changes to the already detected methods as well as clones of the corresponding revisions; **(6)** Locating the code clones detected from each revision to the methods of that revision; **(7)** Detection of method genealogies considering all revisions using the technique proposed by Lozano and Wermelinger [25]; **(8)** Detection of clone genealogies by identifying the propagation of each clone fragment through a method genealogy; and **(9)** Detection of SPCP clone fragments by analyzing clone change patterns. For completing these steps we use the tool SPCP-Miner [30]. For the details of these steps we refer the interested readers to our earlier work [32].

We use NiCad [5] for detecting clones because it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [45], [46]. Using NiCad we detect block clones including both exact (Type 1) and near-miss (Type 2, Type 3) clones of a minimum size of 10 LOC with 20% dissimilarity threshold and blind renaming of identifiers. These settings are explained in detail in our earlier work [32]. For different settings of a clone detector the clone detection results can be different and thus, the findings regarding the bug-proneness of code clones can also be different. Thus, selection of reasonable settings (i.e., detection parameters) is important. We used the mentioned settings in our research, because in a recent study [49] Svajlenko and Roy show that these settings provide us with better clone detection results in terms of both precision and recall.

Clone Genealogies of Different Clone-Types. SPCP-Miner [30] detects clone genealogies considering each clone-type (Type 1, Type 2, and Type 3) separately. Considering a

particular clone-type it first detects all the clone fragments of that particular type from each of the revisions of the candidate system. Then, it performs origin analysis of these detected clone fragments and builds the genealogies. Thus, all the instances in a particular clone genealogy are of a particular clone-type. An instance is a snap-shot of a clone fragment in a particular revision. A detailed elaboration of the genealogy detection approach is presented in our previous study [33]. As we obtain three separate sets of clone genealogies for three different clone-types, we can easily determine and compare the bug-proneness of these clone-types.

Tackling Clone-Mutations. Xie et al. [53] found that mutations of the clone fragments (i.e., a particular clone fragment may change its type) might occur during evolution. If a particular clone fragment is considered of a different clone-type during different periods of evolution, SPCP-Miner extracts a separate clone-genealogy for this fragment for each of these periods. Thus, even with the occurrences of clone-mutations, we can clearly distinguish which bugs were experienced by which clone-types.

B. Bug-proneness Detection Technique

For a particular candidate system, we first retrieve the commit messages by applying the 'SVN log' command. A commit message describes the purpose of the corresponding commit operation. We automatically infer the commit messages using the heuristic proposed by Mockus and Votta [29] in order to identify those commits that occurred for the purpose of fixing bugs. Then we identify which of these bug-fix commits make changes to clone fragments. If one or more clone fragments are modified in a particular bug-fix commit, then it is an implication that the modification of those clone fragment(s) was necessary for fixing the corresponding bug. In other words, the clone fragment(s) are related to the bug. In this way we examine the commit operations of a candidate system, analyze the commit messages to retrieve the bug-fix commits, and identify those clone fragments that are related to the bug-fix. We also determine the number of changes that occurred to such a clone fragment in a bug-fix commit using the UNIX *diff* command. For the details of change detection we refer the interested readers to our earlier work [32].

The way we detect the bug-fix commits was also previously followed by Barbour et al. [2]. Barbour et al. [2] detected bug-fix commits in order to investigate whether late propagation in clones is related to bugs. They at first identified the occurrences of late propagations and then analyzed whether the clone fragments that experienced late propagations are related to bug-fix. In our study we detect bug-fix commits in the same way as they detected, however, our study is not limited to the late propagation clones only. We investigate the bug-proneness of all clone fragments in a software system. Also, Barbour et al. [2] did not investigate Type 3 clones in their study. We consider Type 3 clones in our bug-proneness analysis. Moreover, we compare the bug-proneness of different types of code clones from different perspectives. None of the existing studies do such comparisons.

IV. Experimental Results and Analysis

We present and analyze our experimental results in the following subsections in order to answer the research questions mentioned in Table I.

RQ 1: *Which clone types have a higher possibility of experiencing bug fixing changes?*

Rationale. It is important to know which types of clones have a higher probability of experiencing bug-fix changes compared to the others. The code clones exhibiting higher bug-proneness should be given higher priorities when making clone management decisions (such as refactoring and tracking). Refactoring or tracking of such clone fragments (i.e., highly bug-prone clones) could help us minimize the probability of the occurrences of bugs or inconsistencies in these fragments in the future. In a previous study [36] we found Type 1 and Type 2 clones to be more unstable (i.e., change-prone) than Type 3 clones. However, there is no empirical study on the correlation between change-proneness and bug-proneness of code clones. Thus, we should not infer the bug-proneness of clone types from their change-proneness. A comparative study on the bug-proneness of different types of code clones is important. We perform our investigations for answering *RQ 1* in the following two ways.

- **Investigation 1:** Investigation regarding the proportion of bug-fix changes experienced by the code clones.
- **Investigation 2:** Investigation regarding the proportion of code clones experiencing bug-fix changes.

Investigation 1. *Investigating what proportion of the changes that occurred to the clone fragments of different clone-types are related to a bug-fix.*

Considering the code clones of a particular clone-type of a particular subject system, we first determine how many changes occurred to the code clones during the period of evolution (consisting of the revisions mentioned in Table II). Then we identify which of these changes were related to a bug-fix. Finally, we calculate the percentage of changes related to a bug-fix considering each clone-type of each of the candidate systems using the following equation.

$$PCB = NBC * 100 \ / \ TNC \quad (1)$$

TNC is the total number of changes that occurred to the code clones of a particular clone type of a particular subject system, *NBC* is the number of bug-fix changes that occurred to those code clones, and lastly, *PCB* denotes the percentage of changes related to a bug-fix with respect to all the changes (*TNC*) that occurred to those code clones. Table III shows the *TNC* and *PCB* for each clone-type of each of the subject systems. We also plot the percentages (*PCB*) in the graph of Fig. 1 to get a visual understanding regarding their comparison.

From Fig. 1 we see that for six out of seven subject systems (i.e., except Camellia) the percentage of bug-fix changes is the lowest for the Type 1 case. For four systems (Ctags, Camellia, Freecol, and Carol) the percentage regarding the Type 3 case is the highest among the three cases (Type 1, Type 2, and Type 3). For the remaining three systems, the

TABLE III
PERCENTAGE OF CHANGES RELATED TO BUG-FIX

	Type 1		Type 2		Type 3	
Systems	TNC	PCB	TNC	PCB	TNC	PCB
Ctags	40	10%	84	11.90%	161	14.29%
Camellia	21	9.52%	20	0%	259	14.67%
BRL-Cad	322	0.93%	41	19.51%	215	7.9%
Freecol	134	20.89%	126	21.43%	766	30.42%
jEdit	1594	25.91%	145	47.59%	1265	43.08%
Carol	245	14.69%	279	21.86%	1123	23.15%
Jabref	304	4.27%	244	6.56%	1164	6.44%

TNC = Total Number of Changes that occurred to the Clones
PCB = Percentage of Changes related to a Bug-fix.

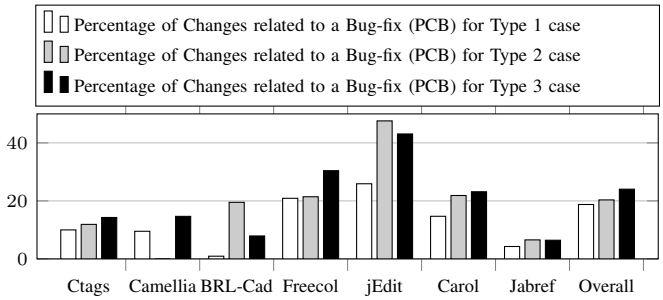

Fig. 1. Comparison regarding the percentage of bug-fix changes that occurred to the clone fragments.

TABLE IV
PERCENTAGE OF CLONES RELATED TO BUG-FIX

	Type 1		Type 2		Type 3	
Systems	TNCG	PCGB	TNCG	PCGB	TNCG	PCGB
Ctags	52	7.69%	88	4.55%	155	9.03%
Camellia	300	0.67%	48	0%	177	6.21%
BRL-Cad	136	2.2%	28	7.14%	127	7.87%
Freecol	239	5.86%	162	7.41%	752	14.23%
jEdit	7398	0.99%	399	5.01%	2688	6.85%
Carol	415	7.47%	211	15.17%	682	19.65%
Jabref	483	1.66%	228	6.14%	1363	2.27%

TNCG = Total Number of Clone Genealogies created during evolution.
PCGB = Percentage of Clone Genealogies related to a Bug-fix.

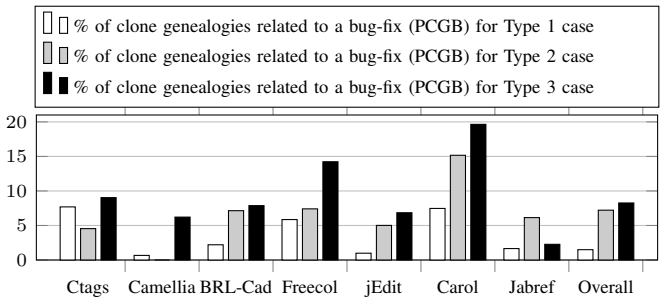

Fig. 2. Comparison regarding the percentage of clone fragments that experienced bug-fixing changes.

percentage regarding the Type 2 case is the highest. The figure also shows the overall percentages (i.e., measured over all the subject systems) for the three clone-types. We see that the percentage of bug-fix changes is the highest in Type 3 case. The overall percentages regarding the other two cases (Type 1, and Type 2) are almost the same. We calculate the overall percentages using the following equation.

$$OP_{Type\ i} = \frac{100 * \sum_{for\ all\ systems} NBC_{Type\ i}}{\sum_{for\ all\ systems} TNC_{Type\ i}} \quad (2)$$

$OP_{Type\ i}$ is the overall percentage of bug-fix changes occurred to the $Type\ i$ clones. $NBC_{Type\ i}$ is the number of bug-fix changes to the $Type\ i$ clones of a particular subject system. $TNC_{Type\ i}$ is the total number of changes that occurred to the $Type\ i$ clones of a subject system.

Investigation 2. *Investigating what proportion of the clone fragments in different clone-types are related to bug-fix changes?*

We mentioned (in Section III) that we determine the genealogies of the detected clone fragments. Considering each clone-type of each of the subject systems we determine how many clone genealogies were created during the evolution and how many of these experienced a bug-fix. From these two values we determine the percentage of clone genealogies that experienced bug-fixing changes using a similar equation to Eq. 1. Table IV shows the total number of clone genealogies (the column *TNCG*) as well as the percentage of bug-fix clone genealogies (the column *PCGB*) for each clone-type of each of the candidate systems. We also plot the percentages (*PCGB*) in the graph of Fig. 2 for easily understanding the comparison of bug-proneness among the three clone-types. The figure also shows the overall percentages of clone genealogies related bug-fix for each clone-type. Overall percentages were calculated using a similar equation to Eq. 2.

From Fig. 2 we see that for all of the subject systems except Jabref, the percentage of clones related to bug-fix is the highest in the Type 3 case. Also, the percentage of bug-fix clones is the lowest in the Type 1 case for most of the systems except Ctags, and Camellia. The overall percentages of the bug-fix clones in the three clone-types provide such implications. Finally, the graph in Fig. 2 implies that Type 3 clones generally have a much higher tendency of experiencing bug-fixing changes compared to the clone fragments of the other two clone-types.

Statistical Significance Tests. We were also interested to investigate whether Type 3 clones have a significantly higher tendency of experiencing bug-fixing changes compared to the clones of the other two types. We performed Mann-Whitney-Wilcoxon (MWW) tests [27] considering the percentages of the bug-fix clone genealogies of the three cases (Type 1, Type 2, and Type 3) as recorded in Table IV. We first determine whether the percentages regarding the Type 3 case are significantly higher than those of the Type 1 case. Our MWW test result implies that *the percentages regarding Type 3 case are significantly higher than the percentages regarding Type 1 case with a* p-value *of 0.026 (for two tailed test) which is less than 0.05*. However, we observe that the percentages for the Type 3 case are not significantly higher than those of the Type 2 case. The MWW test is non-parametric and does not require the samples to be normally distributed [26]. This test can be applied to both small and large sample sizes [38]. In our research, we perform this test considering a significance level of 5%. Finally, it appears that *the percentage of Type 3 clones that experience bug-fixing changes is significantly higher than the percentage of bug-fix clones in the Type 1 case*.

Answer to RQ 1. From our investigations we can state that *while Type 3 clones have a higher bug-proneness compared to the other two clone-types in general, the bug-proneness of Type 1 clones is the lowest for most of our subject systems*. Our statistical significance test results indicate that *Type 3 clones have a significantly higher bug-proneness compared to Type 1 clones*.

In general, the total number of Type 3 clones in a software system is higher compared to the other two clone-types as is evident in Table IV (except Camellia, jEdit, and BRL-Cad). Also, our investigation results indicate that Type 3 clones have the highest possibility of introducing bugs. Finally, our findings imply that possibly Type 3 clones should be managed (i.e., refactored or tracked) with the highest priority.

A possible reason behind why Type 3 clones exhibit the highest bug-proneness is that these are gapped clones (i.e., there are some non-clone lines in the Type 3 clone fragments). Thus, copy-pasting and consistently changing a Type 3 clone fragment is not as straight forward as in the cases of Type 1 and Type 2 clones. Also, because of the gaps in the Type 3 clones, refactoring of such clones might sometimes be difficult, and it causes an increased number of Type 3 clones in the software systems (i.e., as can be seen from our experimental results). Because of the existence of the gaps, possibly tracking is the best suitable management technique for Type 3 clones.

RQ 2: *Do the clone fragments from the same clone class co-change (i.e., change together) consistently during a bug-fix?*

Rationale. From our answer to *RQ 1* we understand that code clones of each clone-type have a tendency of experiencing bug-fixing changes, and Type 3 clones have the highest tendency. However, it is also important to know whether two or more clone fragments from the same clone class co-changed (i.e., changed together) consistently (i.e., the clone fragments were modified in the same way) during bug-fixes. Such clones are more important for clone management than those clones that did not experience consistent co-change during bug-fixes for the following reasons.

(1) If more than one clone fragments from the same clone class are changed together consistently during a bug-fix, then it is an implication that those clone fragments contained the same bug and fixing of that bug required those clone fragments to be modified together consistently. Unification of these clone fragments (i.e., that co-changed consistently during bug-fixes) into a single one through refactoring can possibly help us fix future bugs or inconsistencies with reduced effort, because in that case the bug-fixing changes will require to be implemented in a single code fragment rather than implementing/propagating the same changes to multiple similar code fragments.

(2) If only a single clone fragment from a particular clone class is modified for fixing a bug leaving the other fragments in that class as they are, then it is an implication that this particular clone fragment does not require to maintain consistency with the other clone fragments in its class, and it has a tendency of evolving independently. Such a fragment might not be regarded as a member of the class if it continues to evolve independently, and in that case it should not be considered for clone management.

For this research question we investigate whether clone fragments from the same clone class have a tendency of co-changing consistently during a bug-fix, and if so, how this tendency differs across the clone-types. The clone-type with a higher tendency should be given a higher priority when making clone management decisions.

Methodology. In a previous study [33] we showed that if two or more clone fragments from the same clone class experience a *similarity preserving co-change* (we define it in the next paragraph) in a particular commit operation, then it is an implication that they co-changed consistently (i.e., they were changed in the same way) in that commit. Considering this fact we answer this research question by automatically examining the bug-fix commits and determining whether two or more clone fragments from the same clone class experienced *similarity preserving co-change*s in these commits. If such clone fragments really exist, then these should be given higher priorities for management as we have just discussed.

Similarity Preserving Co-change. Let us consider that two code fragments *CF1* and *CF2* are clones of each other in revision R. A commit operation C was applied on this revision and both of these two code fragments were changed (i.e., the clone fragments were co-changed) in this commit. If in revision *R+1* (created because of the commit operation C) these two code fragments are again considered as clones of each other (i.e., if they preserve their similarity), then we say that *CF1* and *CF2* experienced a *similarity preserving co-change* in the commit operation C.

Considering each clone-type of each of the subject systems we determine which clone fragments experienced bug-fix commits and which of these clone fragments received similarity preserving co-changes in the bug-fix commits. Finally, we determine the percentage of clone fragments that received similarity preserving co-changes in the bug-fix commits with respect to all clone fragments related to bug-fix. Table V shows the total number of clones related to bug-fix and the percentage of bug-fix clones that experienced similarity preserving co-changes during bug-fix commits. We also show these percentages in Fig. 3 to do a visual comparison of the percentages regarding different clone-types.

From Fig. 3 we see that there are no vertical bars for Type 2 and Type 3 cases of Ctags, and also, for Type 2 case of Camellia. The reason is that the number of bug-fix clones that experienced similarity preserving co-changes is zero for each of these cases. This is also evident from Table V. From the overall percentages we see that bug-fix clones of Type 3 have the overall highest tendency of experiencing similarity preserving co-changes in the bug-fix commits. The tendency for Type 2 case is also very near to that of the Type 3 case. Bug-fix clones of Type 1 have the lowest tendency of experiencing similarity preserving co-changes during bug-fix.

We also manually analyzed the similarity preserving co-changes that occurred to the bug-fix clones of each clone-type of Freecol during the bug-fix commits to see whether the clone

TABLE V
PERCENTAGE OF BUG-FIX CLONES THAT EXPERIENCED SIMILARITY PRESERVING CO-CHANGE IN THE BUG-FIX COMMITS

	Type 1		Type 2		Type 3	
Systems	CGBF	BFCS	CGBF	BFCS	CGBF	BFCS
Ctags	4	50%	4	0%	14	0%
Camellia	2	100%	0	0%	11	45.45%
BRL-Cad	3	66.67%	2	100%	10	60%
Freecol	14	57.14%	12	50%	107	51.4%
jEdit	73	8.21%	20	30%	184	24.45%
Carol	31	38.7%	32	50%	134	50.74%
Jabref	8	25%	14	28.57%	31	67.74%

CGBF = Number of Clone Genealogies related to a Bug-fix.
BFCS = Percentage of Bug-fix Clone genealogies that experienced similarity preserving co-change in bug-fix commits.

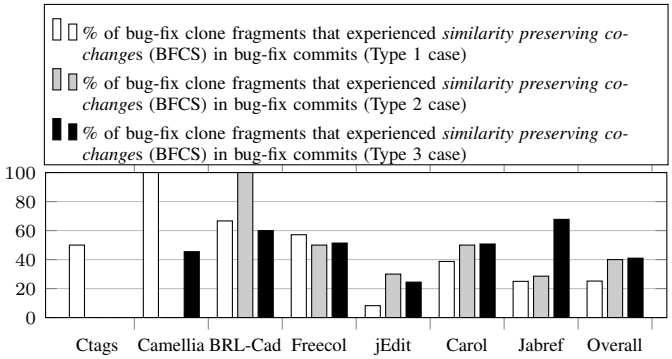

Fig. 3. Comparison regarding the percentage of bug-fix clones that experienced *similarity preserving co-change*s during bug-fix commits.

fragments were really modified consistently (i.e., whether the clone fragments were modified in the same way). According to our manual analysis in each case of similarity preserving co-change, the clone fragments were changed together consistently. Fig. 4 shows an example of similarity preserving co-change of two Type 3 clone fragments in the bug-fix commit operation applied to revision 1075 of Freecol. We show the instances of these two clone fragments in revisions 1075 and 1076 and highlight the changes that occurred to them. We see that the clone fragments changed together consistently (i.e., in the same way) in the bug-fix commit operation. The commit log as stated by the programmer is "*Fixes a bug relating to giving units equipment while onboard a carrier in Europe*". We see that the bug-description is relevant to the context. Fig. 4 shows that both the clone fragments contained the same bug and were fixed in the same way. The example reveals the fact that unification of these two clone fragments into a single one could help us fix future bugs with reduced effort.

During our manual investigation of the bug-fixes that occurred to code clones, the categories of bug-fixes that appeared frequently are as follows: fixing the same semantically incorrect implementation in multiple clone fragments from the same class, addition of the same missing implementations in multiple clone fragments of the same class, and fixing the same GUI related error in multiple clone fragments.

Answer to RQ 2. Our investigation results show that *clone fragments from the same clone class have a tendency of co-changing (i.e., changing together) consistently during the bug-fix commit operations*. Considering all of the subject systems, bug-fix clones of Type 1 exhibit the lowest tendency. The tendencies regarding both Type 2 and Type 3 cases are higher compared to Type 1 case. According to our findings, we should possibly prioritize Type 3 and Type 2 clones over Type 1 clones when making refactoring or tracking decisions.

Through our investigation of this research question (*RQ 2*) we suggest to consider higher priorities for managing those clones that experienced similarity preserving co-changes during bug-fixes. Our findings are important for ranking clones for both refactoring and tracking. However, we require further investigations of the evolution histories of the bug-fix clones because of the following two issues.

Issue 1. The clone fragments that experienced *similarity preserving co-changes* in bug-fix commits might evolve independently afterwards. In that case we should possibly not consider these clone fragments important for management.

Issue 2. A clone fragment that was changed in a bug-fix commit without experiencing a *similarity preserving co-change* might co-evolve consistently with the other fragments in its class afterwards. In that case this clone fragments should be considered important for management.

In order to address these two issues, we need to investigate the entire evolution histories of the bug-fix clones to analyze whether they co-evolved with the other clone fragments in their respective clone classes following a *similarity preserving change pattern* which we called SPCP in our previous studies [33], [34]. We perform such an investigation in RQ 3.

RQ 3: *What proportion of the clone fragments that experienced bug-fixing changes are SPCP clones?*

Rationale. From our discussion at the end of *RQ 2* we realize that it is important to analyze whether the clone fragments that experienced bug-fixes also have the tendencies of evolving following a *similarity preserving change pattern* called *SPCP* (defined in Section II). As the bug-fix clones have tendencies of experiencing *similarity preserving co-change*s (revealed from *RQ 2*), we suspect that they might have tendencies of following SPCP too. In other words, bug-fix clones might also be regarded as SPCP clones. In our previous studies [33], [34] we empirically showed that SPCP clones are important candidates for refactoring or tracking. The clone fragments that do not follow SPCP either evolve independently or are rarely changed during evolution. Thus, the non-SPCP clones should not be considered important for clone management.

To address the research question we investigate which of the bug-fix clones are also SPCP clones. Such clone fragments (i.e., the SPCP clones that experienced bug-fixes) should be given the highest priorities for management. In our previous studies [33], [34] we ranked the SPCP clones on the basis of their co-change tendencies. We did not consider the bug-proneness of the SPCP clones. We believe that bug-proneness should also be considered for ranking the SPCP clones. However, ranking of SPCP clones considering both bug-proneness and co-change tendencies is not our main focus in this research. We focus on investigating whether bug-fix

Fig. 4. An example of a similarity preserving co-change of two Type 3 clone fragments (i.e., Clone Fragment 1, and Clone Fragment 2) of Freecol in a bug-fix commit operation applied to revision 1075. Each of these two clone fragments is a method clone (i.e., the whole method is a clone fragment). The figure shows that they were changed consistently in the bug-fix commit and were again considered as Type 3 clones of each other in revision 1076.

clones also have the possibility of following an SPCP, and if so, how this possibility differs across different clone-types.

A clone fragment that experienced a bug-fix (whether through a *similarity preserving co-change* or not) might not evolve following an SPCP afterwards (related to **Issue 1** stated in *RQ 2*). In this case we understand that the particular clone fragment evolved independently and thus, is not important from the perspectives of clone management.

Methodology. Considering each clone-type of each of the subject systems we determine the SPCP clones using SPCP-Miner [30]. We also determine those clone fragments that experienced bug-fixes following the procedure described in Section III. Then we identify which of these bug-fix clones also appear in the list of SPCP clones. Finally, we determine the percentage of bug-fix clones that have also been selected as the SPCP clones. We determine the following four measures for each clone-type of each candidate system and show these measures in Table VI.

- **Measure 1:** The total number of bug-fix clones (The column **CGBF** in Table VI).
- **Measure 2:** The total number of SPCP clones (The column **CGSPCP** in Table VI).
- **Measure 3:** The total number of bug-fix clones which have also been selected as SPCP clones (The column **CGBFSPCP** in Table VI).
- **Measure 4:** The total number of bug-fix clones which have been selected as SPCP clones and are alive in the last revision (The column **CGBFSPCPL** in Table VI). We determine and present this measure because while making refactoring or tracking decisions we are primarily concerned with those clone fragments that are alive in the last revision (i.e., the most recent revision) of the system.

It might be the case that only a single clone fragment from a clone class got changed in a bug-fix commit operation however, the clone fragment later co-evolved with the other clone fragments in its class by preserving similarity and thus, can be selected as an SPCP clone fragment (related to **Issue 2** stated in *RQ 2*). Such examples are evident in Type 3 case of Ctags. From Table V we see that the bug-fix clone fragments (14 in total) of Type 3 case of Ctags did not experience similarity preserving co-changes. However, Table VI shows that some of these clone fragments (6 in total) evolved following SPCPs (similarity preserving change patterns). If we compare Table V and VI we can discover some other examples of such cases.

We also determine the following two percentages from the above four measures considering each clone-type of each of the candidate system.

(1) The percentage of the bug-fix clones that are selected as SPCP clones. This percentage (Measure 3 * 100 / Measure 1) is shown in Fig 5.

(2) The percentage of the bug-fix clones that have been selected as SPCP clones and are also present in the last revision with respect to all bug-fix clones. This percentage (Measure 4 * 100 / Measure 1) is shown in Fig. 6.

From Fig. 5 we see that for most of the subject systems except Ctags and Camellia, the percentages regarding Type 2 and Type 3 cases are higher compared to the percentage regarding Type 1 case. The overall percentages for the three clone-types also reflect this. From these overall percentages

TABLE VI
NO. OF BUG-FIX CLONES THAT EVOLVED FOLLOWING AN SPCP (SIMILARITY PRESERVING CHANGE PATTERN)

	Type 1				Type 2				Type 3			
Systems	CGBF	CGSPCP	CGBFSPCP	CGBFSPCPL	CGBF	CGSPCP	CGBFSPCP	CGBFSPCPL	CGBF	CGSPCP	CGBFSPCP	CGBFSPCPL
Ctags	4	20	4	2	4	27	0	0	14	85	6	1
Camellia	2	4	2	2	0	2	0	0	11	36	10	0
BRL-Cad	3	42	2	2	2	8	2	2	10	41	8	4
Freecol	14	43	9	3	12	49	10	2	107	331	80	10
jEdit	73	50	0	0	20	63	11	2	184	614	157	96
Carol	31	82	16	0	32	73	20	3	134	325	117	22
Jabref	8	104	5	2	14	51	11	0	31	293	25	10

CGBF = Total number of Clone Genealogies (i.e., clones) that are related to a Bug-fix.
CGSPCP = Total number of Clone Genealogies that followed an SPCP (*Similarity Preserving Change Pattern*).
CGBFSPCP = Total number of bug-fix clones (i.e., the clones that were changed in bug-fix commits) that followed an SPCP.
CGBFSPCPL = Total number of bug-fix clones that followed an SPCP and are also alive in the last revision.

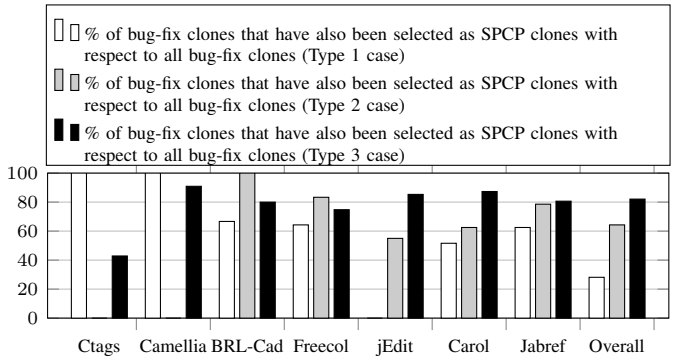

Fig. 5. Comparison regarding the percentage of clone fragments that have experienced bug-fixes and have also been selected as SPCP clones.

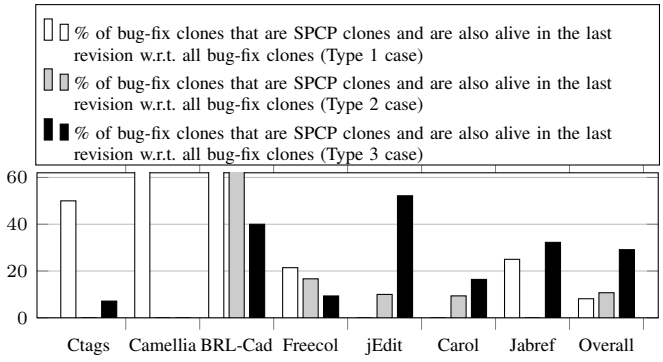

Fig. 6. Comparison regarding the percentage of bug-fix clones that have been selected as SPCP clones and are also present in the last revision.

we can see that the bug-fix clones of the Type 3 case have the highest possibility of evolving following an SPCP (Similarity Preserving Change Pattern). The possibility regarding the Type 1 case is the lowest among the three cases. Such an overall scenario can also be observed in Fig. 6.

Answer to RQ 3. From our investigations we can state that *a considerable proportion of the clone fragments that experienced bug-fixing changes have a tendency of evolving by following a similarity preserving change pattern (SPCP) and thus, are the most important candidates for refactoring or tracking*. We also observe that the bug-fix clones of the Type 3 case generally have the highest probability of following an SPCP. Thus, we again infer that Type 3 clones should be given the highest priority for management.

Our findings from Fig. 6 also imply that for most of the subject systems a considerable proportion of the bug-fix clones that evolve following a *similarity preserving change pattern* remain alive in the last revision (i.e., the most recent revision) of the subject systems. Such clones should be given the highest importance for management, because programmers are mostly concerned with the last revision of the code-base (i.e., the working copy). The findings from this research question and also, from the previous one are important for ranking clones considering their bug-proneness. In future, on the basis of these findings we would like to propose a clone ranking mechanism considering both the co-change tendencies and bug-proneness of code clones.

V. THREATS TO VALIDITY

We used the NiCad clone detector [5] for detecting clones. For different settings of NiCad, the statistics that we present in this paper might be different. Wang et al. [52] defined this problem as the *confounding configuration choice problem* and conducted an empirical study to ameliorate the effects of the problem. However, the settings that we have used for NiCad are considered standard [44] and with these settings NiCad can detect clones with high precision and recall [45], [46], [49]. Thus, we believe that our findings on the bug-proneness of code clones are of significant importance.

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique followed by Barbour et al. [3]. Such a technique proposed by Mocus and Votta [29] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [3] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

In our experiment we did not study enough subject systems to be able to generalize our findings regarding the comparative bug-proneness of clone-types. However, our candidate systems were of diverse variety in terms of application domains, sizes and revisions. Thus, we believe that our findings are important

from the perspectives of clone management and can help us in better ranking of code clones for refactoring and tracking.

VI. Related Work

Bug-proneness of code clones has already been investigated by a number of studies. Li and Ernst [22] performed an empirical study on the bug-proneness of clones by investigating four software systems and developed a tool called CBCD on the basis of their findings. CBCD can detect clones of a given piece of buggy code. Li et al. [23] developed a tool called CP-Miner which is capable of detecting bugs related to inconsistencies in copy-paste activities. Steidl and Göde [48] investigated on finding instances of incompletely fixed bugs in near-miss code clones by investigating a broad range of features of such clones involving machine learning. Göde and Koschke [9] investigated the occurrences of unintentional inconsistencies to the code clones of three mature software systems and found that around 14.8% of all changes occurred to the code clones are unintentionally inconsistent. Chatterji et al. [4] performed a user study to investigate how clone information can help programmers localize bugs in software systems. Jiang et al. [13] performed a study on the context based inconsistencies related to clones. They developed an algorithm to mine such inconsistencies for the purpose of locating bugs. Using their algorithm they could detect previously unknown bugs from two open-source subject systems. Inoue et al. [12] developed a tool called 'CloneInspector' in order to identify bugs related to inconsistent changes to the identifiers in the clone fragments. They applied their tool on a mobile software system and found a number of instances of such bugs. Xie et al. [53] investigated fault-proneness of Type 3 clones in three open-source software systems. They investigated two evolutionary phenomena on clones: (1) mutation of the type of a clone fragment during evolution, and (2) migration of clone fragments across repositories and found that mutation of clone fragments to Type 2 or Type 3 clones is risky.

Rahman et al. [40] found that bug-proneness of cloned code is less than that of non-cloned code on the basis of their investigation on the evolution history of four subject systems using DECKARD [14] clone detector. However, they considered monthly snap-shots (i.e., revisions) of their systems and thus, they have the possibility of missing buggy commits. In our study, we consider all the snap-shots/revisions (i.e., without discarding any revisions) of a subject system as mentioned in Table II from the beginning one. Thus, we believe that we are not missing any bug-fix commits. Moreover, our goal in this study is different. We compare the bug-proneness of different types of code clones.

Selim et al. [47] used Cox hazard models in order to assess the impacts of cloned code on software defects. They found that defect-proneness of code clones is system dependent. However, they considered only method clones in their study. We consider block clones in our study. While they investigated only two subject systems, we consider seven diverse subject systems in our investigation. Also, we compare the bug-proneness of different types of clones. Selim et al. [47] did not perform a type centric analysis in their study.

A number of studies have also been done on the late propagation in clones and its relationships with bugs. Aversano et al. [1] investigated clone evolution in two subject systems and reported that late propagation in clones is directly related to bugs. Barbour et al. [2] investigated eight different patterns of late propagation considering Type 1 and Type 2 clones of three subject systems and identified those patterns that are likely to introduce bugs and inconsistencies to the code-base.

We see that different studies have investigated clone related bugs in different ways and have developed different bug detection tools. However, none of these studies make a comparison of the bug-proneness of different types of code clones. Comparing the bug-proneness of different clone-types is important from the perspectives of clone management. The clone-type with a higher bug-proneness can be given a higher priority when making clone management decisions. Focusing on this issue we make a comparison of the bug-proneness of the major types (Type 1, Type 2, Type 3) of clones from different perspectives and identify which types of clones have a higher bug-proneness and thus, should be given a higher priority for management. None of the existing studies made such a comparison. Our study also provides useful implications regarding ranking of code clones for refactoring and tracking.

VII. Conclusion

In this paper we present an empirical study on the comparative bug-proneness of different types of code clones. According to our investigation on the major types of code clones: Type 1 (Exact clones), Type 2 (Near-miss clones), and Type 3 (Near-miss clones) in thousands of revisions of seven diverse subject systems written in two different programming languages (C, and Java) we can state that:

(1) Type 3 clones exhibit the highest bug-proneness among the three clone-types. The bug-proneness of Type 3 clones is significantly higher than that of Type 1 clones.

(2) Also, Type 3 clones have the highest likeliness of co-changing (i.e., changing together) consistently during the bug-fixing changes.

(3) Moreover, the bug-fix clones of Type 3 exhibit the highest tendencies of evolving following a *similarity preserving change pattern* (SPCP). The existing studies [33], [34] show that the SPCP clones (i.e., the clone fragments that evolve following a similarity preserving change pattern) are important for refactoring and tracking.

Our experimental results imply that Type 3 clones should be given the highest priority when making clone management decisions. Our findings regarding the consistent co-change of bug-prone clones and also, regarding their tendencies of following SPCP can be considered for ranking code clones for refactoring and tracking. We plan to investigate such a ranking as future work using our technologies from this research. We also plan to investigate classifying the bugs that occurred to the code clones.

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] 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.

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

[6] CTAGS: http://ctags.sourceforge.net/

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

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

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

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

[11] 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.

[12] 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.

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

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

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

[16] 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, pp. 16 - 20.

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

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

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

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

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

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

[23] Z. Li, S. Lu, S. Myagmar, Y. Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code", Proc. *OSDI*, 2004, pp. 20 – 20.

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

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

[26] Mann-Whitney-Wilcoxon Test. http://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test

[27] Mann-Whitney-Wilcoxon Test Online. http://elegans.som.vcu.edu/~leon/stats/utest.cgi

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

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

[30] 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).

[31] M. Mondal, C. K. Roy, K. A. Schneider, "Late Propagation in Near-Miss Clones: An Empirical Study", *Electronic Communications of the EASST*, 63(2014):1 – 17.

[32] M. Mondal, C. K. Roy, K. A. Schneider, "Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems", Proc. *CASCON*, 2012, pp. 205 – 219.

[33] 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.

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

[35] 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.

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

[37] M. Mondal, C. K. Roy, K. A. Schneider, "Prediction and Ranking of Co-change Candidates for Clones", Proc. *MSR*, 2014, pp. 32 – 41.

[38] Nonparametric Tests. http://sphweb.bumc.bu.edu/otlt/MPH-Modules/BS/BS704_Nonparametric/mobile_pages/BS704_Nonparametric4.html

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

[40] F. Rahman, C. Bird, P. Devanbu, "Clones: What is that Smell?", Proc. *MSR*, 2010, pp. 72 – 81.

[41] D. Rattan, R. Bhatia, M. Singh, "Software Clone Detection: A Systematic Review", *Information and Software Technology*, 2013, 55(7): 1165 – 1199.

[42] 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.

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

[44] 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.

[45] 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.

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

[47] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, Y. Zou, "Studying the Impact of Clones on Software Defects", Proc. *WCRE*, 2010, pp. 13 - 21.

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

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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