Bug-proneness and Late Propagation Tendency of Code Clones: A Comparative Study on Different Clone Types

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 an empirical 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 nine 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. Our investigation on the relatedness between bug-proneness and late propagation in code clones implies that bug-proneness of code clones is not primarily related with late propagation. The possibility that a bug-fix

experienced by a clone fragment will be related with late propagation is only 1.4%. Moreover, for only 10.76% of the cases, a late propagation experienced by clone fragments can be related with a bug. Thus, late propagation contributes to a very little proportion of the bugs in code clones. We believe that our study provides useful implications for ranking clones for management such as refactoring and tracking.

Keywords: Code clones, Clone-types, Bug-proneness, Late propagation

1. 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 [45]. 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 four types (Type 1, Type 2, Type 3, and Type 4) according to the literature [45]. While clone fragments in a Type 1 clone class exhibit exact textual similarity, clone fragments in a Type 2 clone class exhibit syntactic similarity. Type 3 clones get created from Type 1 or Type 2 clones because of addition or deletion of lines (i.e., program statements). We provide detailed descriptions for these three clone types in Section 2. Finally, code fragments that perform the same task but were implemented in different ways are known as Type 4 clones or semantic clones.

Code clones are of great importance from the perspectives of software maintenance and evolution. A great many studies [1, 2, 10, 11, 12, 16, 18, 20, 21, 22, 25, 26, 37, 38, 53, 23, 51, 14] have already been conducted on the impacts of clones on the evolution and maintenance of software systems. While some of these studies [1, 11, 12, 18, 20, 21, 22] identify some positive impacts of code

clones, a number of studies [2, 16, 25, 10, 26, 37, 38, 23, 51, 14] have shown empirical evidence of strong negative impacts of code clones such as hidden bug propagation [23], late propagation [2], unintentional inconsistencies [2, 10], and high instability [38]. 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 [23, 24, 51, 5, 14, 13, 56, 2, 10], 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 we investigate the comparative bug-proneness of the major types of code clones: Type 1, Type 2, and Type 3 (defined in Section 2). We also investigate the relatedness of clone bug-proneness with late propagation in code clones. Existing studies [2, 3, 1] report that code clones that experienced late propagation have high possibilities of experiencing bug-fixes. However, we still do not know whether bug-proneness of code clones is mostly (i.e., primarily) related with late propagation. Investigating what proportion of the bugs in code clones can be related with late propagation might help us realize the extent of harmfulness of late propagation. We answer six important research questions listed in Table 1. According to our in-depth investigation on thousands of revisions of nine diverse subject systems written in three different programming languages (C, Java, and C#) 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 [35, 36], SPCP clones (i.e., clones that evolve following a Similarity Preserving Change Pattern) are the most important ones to consider for clone management.

(4) For only a very little proportion of the bug-fix clones (i.e., code clones that experience bug-fixes), the bug-fixes were related with late propagation. According to our subject systems, this proportion can be at most 1.4%. Moreover, for only a little proportion of the late propagation clones (i.e., code clones that experienced late propagation), the occurrences of late propagation were related with bug-fix. For our subject systems, this proportion can be at most 10.76%.

Our experimental results stated above have the following useful implications:

Implication 1. 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 [35, 36] 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.

Implication 2. Bug-fixes in code clones are rarely related with late propagation. The probability that a bug-fix experienced by a clone fragment will be related with late propagation is only 0.014 (1.4%). Thus, late propagation does not seem to be primary reason behind clone bug-proneness. Such a finding is supported by our statistical significance tests. Moreover, experiencing bug-fixes during late propagation is not a common scenario for the late propagation clones. The possibility that a late propagation experienced by a clone fragment will be related with a bug is only 0.107 (10.76%). Thus, late propagation does not appear to be strongly related with clone bug-proneness.

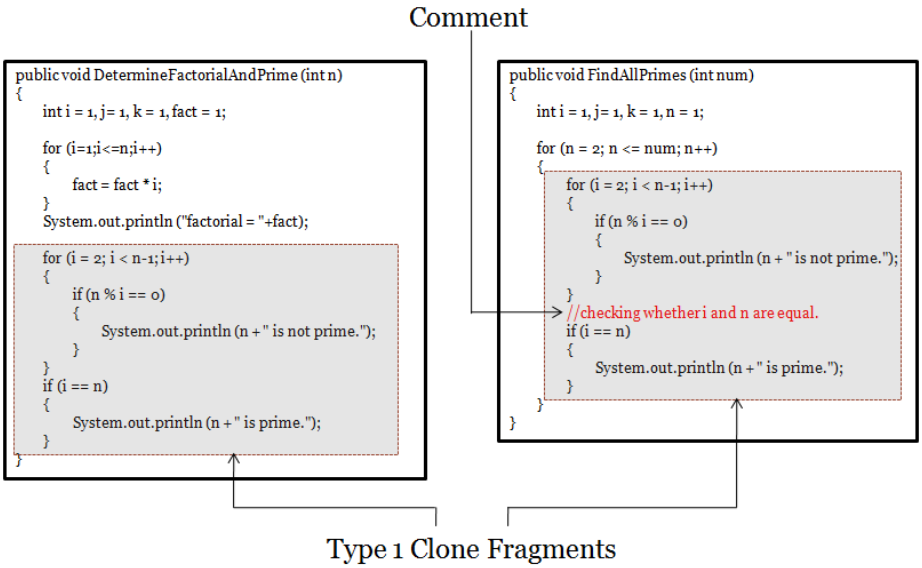

Figure 1: Type 1 (identical) clone pair

The rest of the paper is organized as follows: Section 2 describes the terminology, Section 3 discusses the experimental steps, Section 4 answers the research questions by presenting and analyzing the experimental results, Section 5 mentions the possible threats to validity, Section 6 discusses the related work, and finally, Section 7 concludes the paper by mentioning possible future work.

Our study presented in this paper is a significant extension of our earlier work [40] on clone bug-proneness. In our earlier work [40] we did not investigate any C# system. In this extended study we investigate two additional subject systems written in C#. In our previous work [40] we did not investigate late propagation in code clones. However, in this extended work we automatically detect late propagation in code clones, and then analyze the relatedness of late propagation with clone bug-proneness. From this analysis we answer three additional research questions: RQ 4, RQ 5, and RQ 6 which were not answered in our earlier work [40].

Table 1: 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?
RQ 4	Do bug-fix changes mainly occur to the late propagation clones (i.e., to the clone fragments that experienced late propagation)?
RQ 5	Do most of the late propagation clones experience bug-fix changes?
RQ 6	What proportion of the clone fragments experienced bug-fixes that are associated with late propagation?

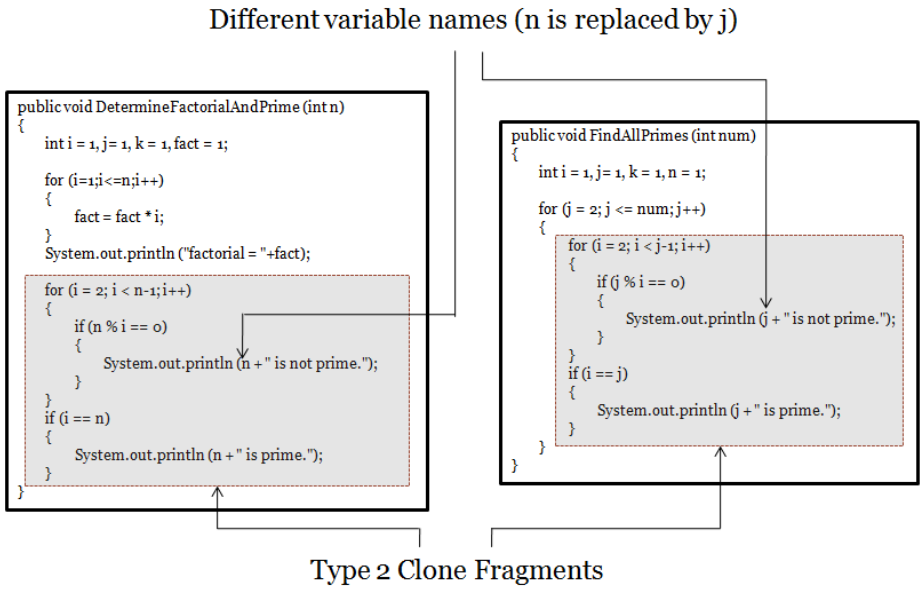

Figure 2: Type 2 clone pair

2. Terminology

2.1. 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 [46, 45], 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. Fig. 1 shows a Type 1 clone-pair. One fragment of the pair resides in the method named 'DetermineFactorialAndPrime', and the other fragment resides in the method 'FindAllPrimes'. The two fragments have been shown in the light gray boxes. We see that the fragment at the right hand side contains a comment. If we disregard this comment, then the two fragments become identical (i.e., Type 1 clones).

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

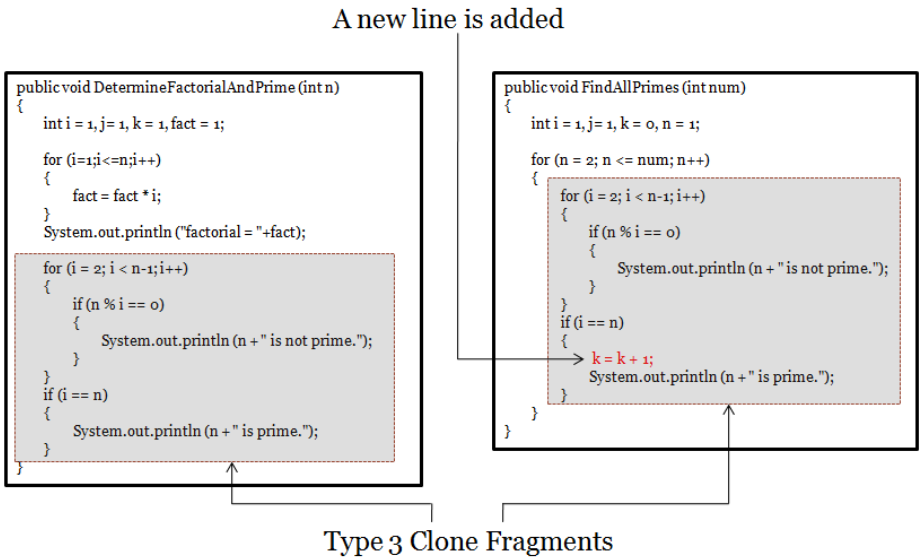

Figure 3: Type 3 clone pair

data types. Fig. 2 shows a Type 2 clone pair where the two fragments in the pair reside in two methods 'DetermineFactorialAndPrime' and 'FindAllPrimes'. The clone fragments have also been highlighted in the methods. We see that the fragment at the left hand side contains a variable called n. The fragment at the right hand side the corresponding variable has been named as j. Because of this variable renaming, these two fragments make a Type 2 clone pair.

Type 3 clones are mainly created because of additions, deletions, or modifications of lines in Type 1 or Type 2 clones. Fig. 3 contains an example of a Type 3 clone pair. The two fragments in the clone pair again reside in the two methods 'DetermineFactorialAndPrime' and 'FindAllPrimes'. We see that the fragment at the right hand side contains a line 'k=k+1' for counting the number of primes. However, this line is absent in the fragment at the left hand side. Thus, these two clone fragments make a Type 3 clone pair.

2.2. Clone Genealogy

Our research requires the detection of clone genealogies. A clone genealogy can be defined in the following way. Let us assume that a clone fragment, CF, was created in a particular revision of a software system and was alive in a number of consecutive revisions. Thus, each of these revisions contains a snapshot of CF. The genealogy of CF consists of the set of its consecutive snapshots from the consecutive revisions where it was alive. Each clone fragment in a particular revision belongs to a particular clone genealogy. In other words, a particular clone fragment in a particular revision is actually a snapshot in a particular clone genealogy. By examining the genealogy of a clone fragment we can determine how it changed during evolution. We automatically detect clone genealogies by using the SPCP-Miner tool [32]. The procedure for detecting clone genealogies will be described in Section 3. Before detecting clone genealogies, we need to detect method genealogies. We define a method genealogy in the following way.

2.3. Method Genealogy

Let us assume that a particular method was created in a particular revision of a software system and was alive in a number of consecutive revisions. Each of these consecutive revisions contains a snapshot of the method. The genealogy of the method consists of the set of consecutive snapshots of it from the consecutive revisions where it was alive. In our research, we detect method genealogies for the purpose of detecting clone genealogies. The process of detecting clone genealogies from method genealogies will be described in Section 3.

2.4. Similarity Preserving Change Pattern (SPCP).

In our previous studies [35, 36] 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*. We describe these two types of changes in the following paragraphs.

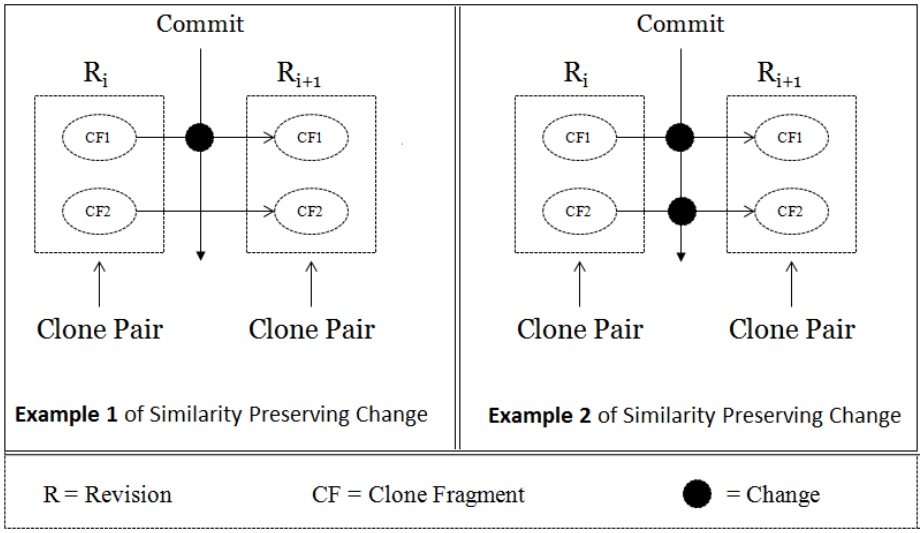

Figure 4: Two examples of similarity preserving change

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.

Fig. 4 shows two examples of similarity preserving change. In the first example, we see a clone pair (CF1, CF2) in revision R_i. The commit operation on R_i modified one (c.f., CF1) of these two fragments. However, in revision R_{i+1}, they still remain as a clone pair. That means, although there were some changes to one clone fragment, the two clone fragments preserved their similarity. So, this is a similarity preserving change. In the second example, we see that both the clone fragments changed in the commit operation. However, even after these changes, the fragments preserved their similarity (i.e., remained as a clone pair).

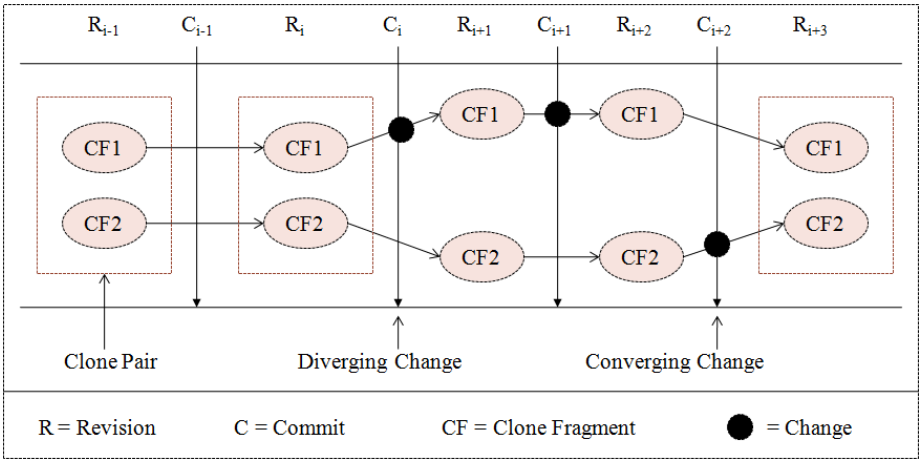

Figure 5: An example of re-synchronizing change (i.e., an example of late propagation)

So, this is also a similarity preserving change.

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*.

Fig. 5 shows an example of re-synchronizing change experienced by a clone pair (CF1, CF2). The commit C_i applied on revision R_i modified CF1 and as a result, CF1 and CF2 diverged. However, in commit operation C_{i+2}, the fragment CF2 changed and CF1 and CF2 got re-synchronized in revision R_{i+3} (i.e., CF1 and CF2 again became clones of each other in revision R_{i+3}).

Table 2: Subject Systems

Systems	Lang.	Domains	LLR	Revisions
MonoOSC	C#	Formats and Protocols	18,991	355
SqlBuddy	C#	Editor for SQL	16,116	945
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

2.5. Late Propagation in Clones

Let us consider a pair of clone fragments. If this clone pair experiences a re-synchronizing change during evolution, we say that the clone pair experienced a *late propagation* [2]. In other words, each occurrence of re-synchronizing change in a clone pair is also termed as a *late propagation*. A particular clone pair may experience late propagation more than once during evolution. Fig. 5 shows an example of late propagation.

3. Experimental Steps

We perform our investigation on nine subject systems (Table 2) downloaded from an on-line SVN repository [42] called SourceForge.

3.1. Preliminary Steps

We perform the following preliminary steps before analyzing bug-proneness: **(1)** Extraction of all revisions (as mentioned in Table 2) of each of the subject

systems from the online SVN repository; **(2)** Method detection and extraction from each of the revisions using CTAGS [7]; **(3)** Detection and extraction of code clones from each revision by applying the NiCad [6] 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 [26]; **(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 [32]. For the details of these steps we refer the interested readers to our earlier work [34]. In the following paragraphs, we first provide a short description for our method detection tool CTAGS, and then, we discuss detecting code clones using the NiCad clone detector.

CTAGS: We used CTAGS [7] in our experiment for detecting methods. CTAGS [7] is a widely used tool that, when applied on the source code of a program, generates a list of source code definitions such as variables, methods, classes, class members, macros etc depending on the language the program was written in. CTAGS currently supports 41 programming languages in total. We should note that CTAGS is one of our subject systems (c.f., Table 2) as well.

Clone detection using NiCad: We use NiCad [6] for detecting clones because it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [48, 49]. 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 [34]. 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 [52] Svajlenko and Roy show that these settings provide us with better clone detection results in terms of both precision and recall. We detect clone genealogies using the SPCP-Miner tool [32]. The detection procedure has been described below.

Detecting Clone Genealogies. As we have already mentioned, we detect methods from each revision using CTAGS [7]. Then we detect method genealogies by applying the technique proposed by Lozano and Wermelinger [26]. This technique considers method movement, renaming and changing when detecting method genealogies. After detecting method genealogies, we detect code clones of block granularity from each revision using the NiCad [6] clone detector. The detected clones are then mapped inside the methods in each revision. As we have already detected method genealogies, we can easily detect the propagation of clone fragments through the method genealogies. During the evolution of a software system it might be seen that a particular code fragment, CF, was considered as a clone fragment in revision r. However, in the next several revisions, it was not considered as a clone fragment. Our clone genealogy detector tracks the evolution of CF even in the period when it did not appear as a clone fragment. We track the evolution in such a period because CF might again appear as a clone fragment after this period. If a clone fragment gets deleted after a particular revision and reinserted in the code-base after a certain period of time (i.e., after a number of commits), we start a new clone genealogy for the clone fragment after reinsertion. We believe that this is a reasonable consideration. As the clone fragment got deleted for a period of time, we cannot keep track of the fragment during that period of disappearance. After reappearance, we should consider it a new clone fragment because we stopped tracking it.

Clone Genealogies of Different Clone-Types. SPCP-Miner [32] 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 [35]. 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.[56] 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.

3.2. 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 [31] 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 [34].

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.

3.3. Detecting Late Propagation in Code Clones

We detect late propagation considering code clones of each clone-type separately. Let us consider that two code fragments, CF1 and CF2, were initially considered as a clone-pair in revision R. We want to determine whether this clone pair experienced a late propagation. We analyze the genealogies of these two clone fragments. We previously mentioned that we detect clone genealogies using our tool SPCP-Miner [32]. By examining the genealogies of the two clone fragments in the pair we determine whether they evolved by following the resynchronizing change pattern mentioned below:

- The clone fragments diverged after a particular revision $R_{diverge}$ ($R_{diverge} >= R$) because of the changes that occurred to the fragments in the commit operation which was applied on $R_{diverse}$, and
- The fragments again converged in a later revision $R_{converge}$ ($R_{converge} > R_{diverge}$) because of the changes that occurred to the fragments in the commit operation which created the revision $R_{converge}$.

If a clone-pair evolved by following the above pattern, we consider the clone-pair as a late propagation clone-pair. We mark the clone fragments CF1 and CF2 as late propagation clone fragments. A particular clone-pair may experience late propagation more than once. However, our goal is to determine whether

a clone-pair experienced late propagation at least once during evolution. Late propagation detection technique has been elaborated in our earlier work [41].

We detect late propagation for each clone-type separately as we did in our earlier work [41]. Previously we investigated late propagation by detecting code clones of a minimum size of 5 LOC [41]. However, in our study presented in this paper we detect code clones using a minimum threshold of 10 LOC (we have already discussed why we select such a setting).

4. 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 1.

4.1. Answering RQ 1

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 [38] 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 2). 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.

$$PCB = \frac{NBC \times 100}{TNC} \tag{1}$$

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 3 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. 6 to get a visual understanding regarding their comparison.

From Fig. 6 we see that for seven out of nine subject systems (i.e., except Camellia, and MonoOSC) the percentage of bug-fix changes is the lowest for the Type 1 case. For six systems (MonoOSC, SqlBuddy, 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 (BRL-CAD, jEdit, and Jabref), the 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, and is the lowest in Type 1 case. We calculate the overall percentages using the following equation.

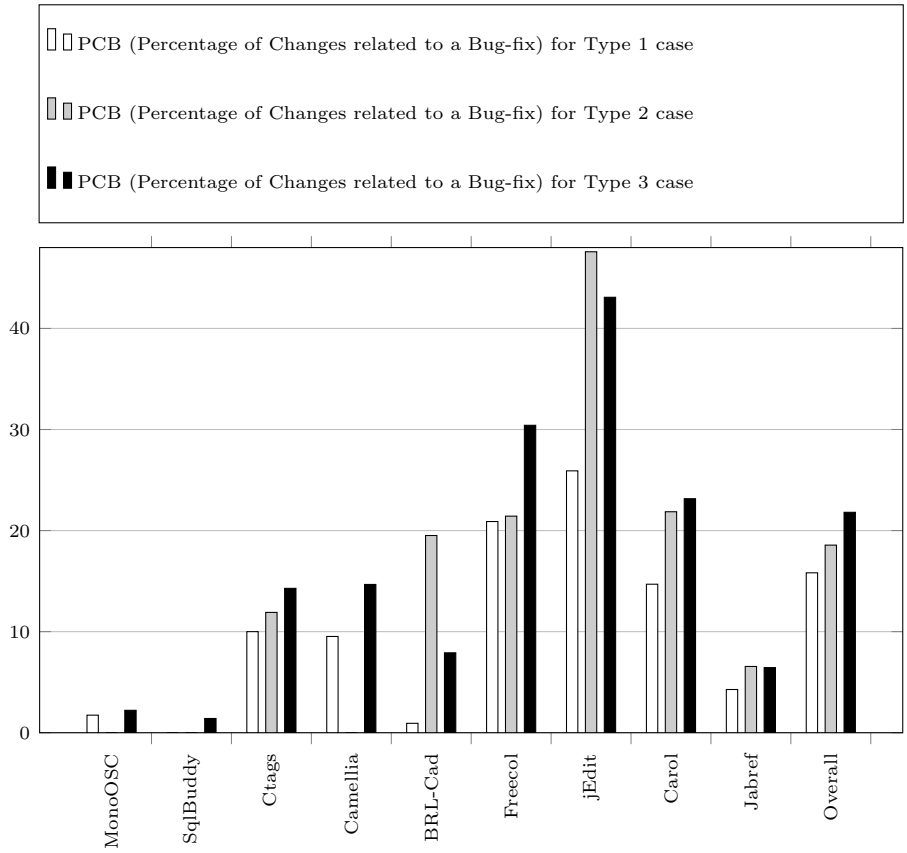

Figure 6: Comparison regarding the percentage of bug-fix changes that occurred to the clone fragments.

Table 3: Percentage of changes related to bug-fix

	Type 1		Type 2		Type 3	
Systems	TNC	PCB	TNC	PCB	TNC	PCB
MonoOSC	516	1.74%	40	0%	270	2.22%
SqlBuddy	35	0%	50	0%	284	1.41%
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.

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

$OP_{Type\ i}$ 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 3) 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 equation Eq. 3.

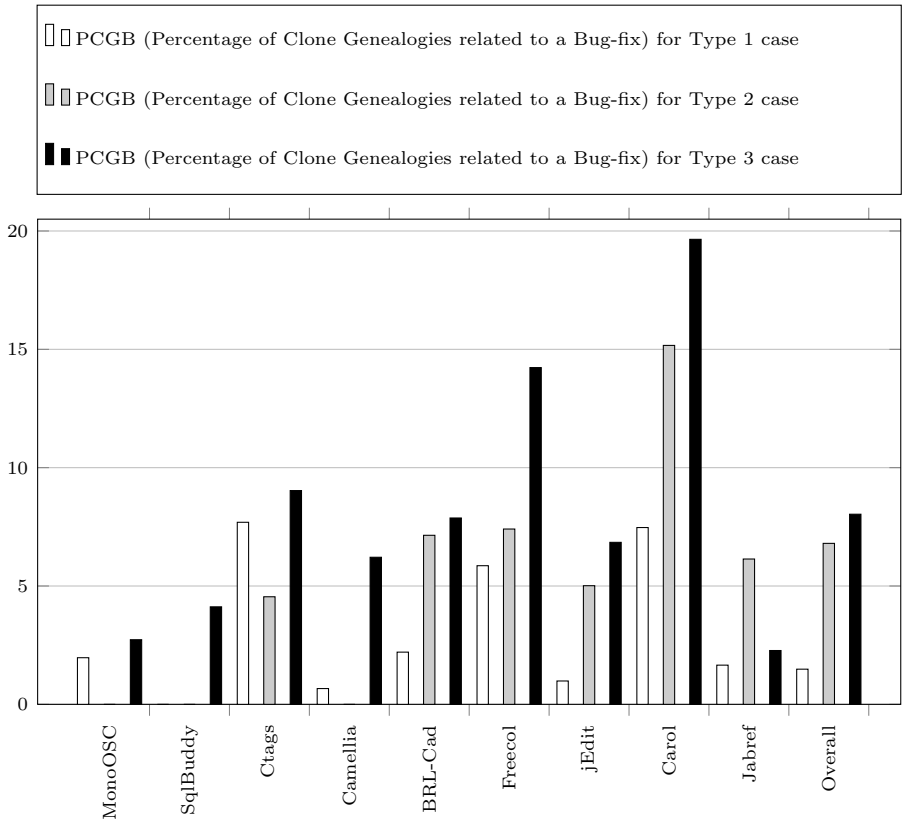

Figure 7: Comparison regarding the percentage of clone fragments that experienced bug-fixing changes.

Table 4: Percentage of clones related to bug-fix

	Type 1		Type 2		Type 3	
Systems	TNCG	PCGB	TNCG	PCGB	TNCG	PCGB
MonoOSC	152	1.97%	40	0%	183	2.73%
SqlBuddy	112	0%	31	0%	97	4.12%
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.

$$PCGB = \frac{NCGB \times 100}{TNCG} \tag{3}$$

For a particular clone-type, $TNCG$ is the total number of clone genealogies created during evolution, $NCGB$ is the number of clone genealogies that experienced bug-fixes, and $PCGB$ is the percentage of clone genealogies that experienced bug-fixes during evolution. Table 4 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. 7 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 Eq. 4.

$$OPCGB_{Type\ i} = \frac{100 \times \sum_{for\ all\ systems} NCGB_{Type\ i}}{\sum_{for\ all\ systems} TNCG_{Type\ i}} \tag{4}$$

For a particular clone-type, *Type i* (i = 1, 2, or 3), $OPCGB$ is the overall percentage (considering all subject systems) of clone genealogies that experienced bug-fixes, $NCGB$ is the number of clone genealogies that experienced bug-fixes, and $TNCG$ is the total number of clone genealogies that were created during entire evolution. From Fig. 7 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 MonoOSC, Ctags, and Camellia. The overall percentages of the bug-fix clones in the three clone-types provide such implications. Finally, the graph in Fig. 7 implies that Type 3 clones generally have a 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, 28] considering the percentages of the bug-fix clone genealogies of the three cases (Type 1, Type 2, and Type 3) as recorded in Table 4. We should note that MWW test is nonparametric, and thus, it does not require the samples to be mormally distributed [29, 27]. This test can be applied to both small and large data samples [29].

Our test results are shown in Table 5. 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 (Table 5) implies that *the percentages regarding Type 3 case are significantly higher than the percentages regarding Type 1 case with a* p-value *of 0.017 (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 nonparametric and does not require the samples to be normally distributed [27]. This test can be applied to both small and large sample sizes. 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*

Table 5: Mann-Whitney-Wilcoxon test result for comparing different clone types

Comparison between Type 1 and Type 3	
Sample Size	9
Significance level	5%
U-value	13
Probability value (i.e., *p-value*) for two tailed test case	0.017
Probability value (i.e., *p-value*) for one tailed test case	0.0085
The two samples are significantly different (*p-value*s are less than 0.05).	

Comparison between Type 2 and Type 3	
Sample Size	9
Significance level	5%
U-value	27
Probability value (i.e., *p-value*) for two tailed test case	0.25
Probability value (i.e., *p-value*) for one tailed test case	0.125
The two samples are not significantly different (*p-value*s are greater than 0.05).	

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 4 (except SqlBuddy, 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.

4.2. Answering RQ 2

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 [35] 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. The second example (i.e., Example 2) in Fig. 4 demonstrates a similarity preserving co-change.

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 6 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. 8 to do a visual comparison of the percentages regarding different clone-types.

From Fig. 8 we see that there are no vertical bars for Type 2 case of MonoOSC, Type 1 and Type 2 cases of SqlBuddy, 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 the column **BFCS** in Table 6. From the overall percentages (Fig. 8) 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 oc-

Table 6: 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
MonoOSC	3	66.67%	0	0%	5	40%
SqlBuddy	0	0%	0	0%	4	50%
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.

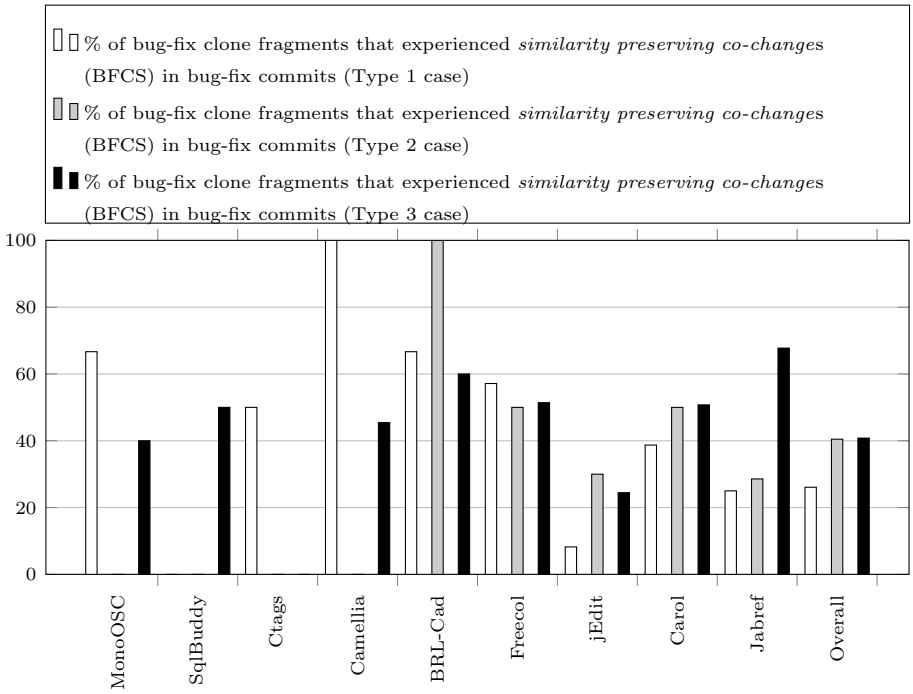

Figure 8: Comparison regarding the percentage of bug-fix clones that experienced *similarity preserving co-change*s during bug-fix commits.

curred to the bug-fix clones of each clone-type of Freecol during the bug-fix commits to see whether the clone 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. 9 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. 9 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, we mostly observed the following bug-fixing categories: 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.* Consdering all the subject systems, bug-fix clones of Type 1 exhibit the lowest tendency (c.f., Fig. 8). 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

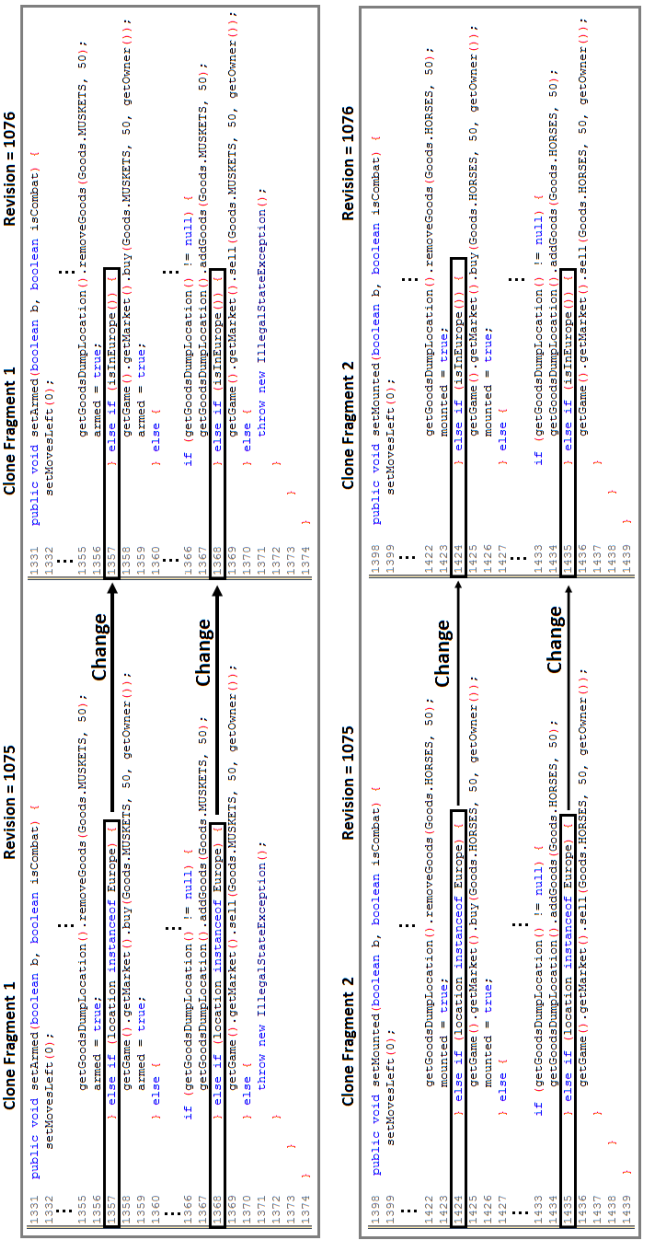

Figure 9: An example of a similarity preserving co-change of two Type 3 clone fragments (i.e., Clone Fragment 1, and Clone Fragment 2) of Freecol in a bug-fix commit operation applied to revision 1075. Each of these two clone fragments is a method clone (i.e., the whole method is a clone fragment). The 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.

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 [35, 36]. We perform such an investigation in RQ 3.

4.3. Answering 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 2). As the bug-fix clones have tendencies of experiencing *similarity preserving co-changes* (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 [35, 36] 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 answer this 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 [35, 36] 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 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 [32]. We also determine those clone fragments that experienced bug-fixes following the procedure described in Section 3. 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 7.

- **Measure 1:** The total number of bug-fix clones (The column **CGBF** in Table 7).
- **Measure 2:** The total number of SPCP clones (The column **CGSPCP** in Table 7).
- **Measure 3:** The total number of bug-fix clones which have also been

selected as SPCP clones (The column **CGBFSPCP** in Table 7).

- **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 **CGBF-SPCPL** in Table 7). 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 6 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 7 shows that some of these clone fragments (6 in total) evolved following SPCPs (similarity preserving change patterns). If we compare Table 6 and 7 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 systems.

(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 10.

(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. 11.

From Fig. 10 we see that for most of the subject systems, 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 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

Table 7: 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
MonoOSC	3	16	2	2	0	14	0	0	5	103	5	5
SqlBuddy	0	2	0	0	0	17	0	0	4	47	4	0
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.

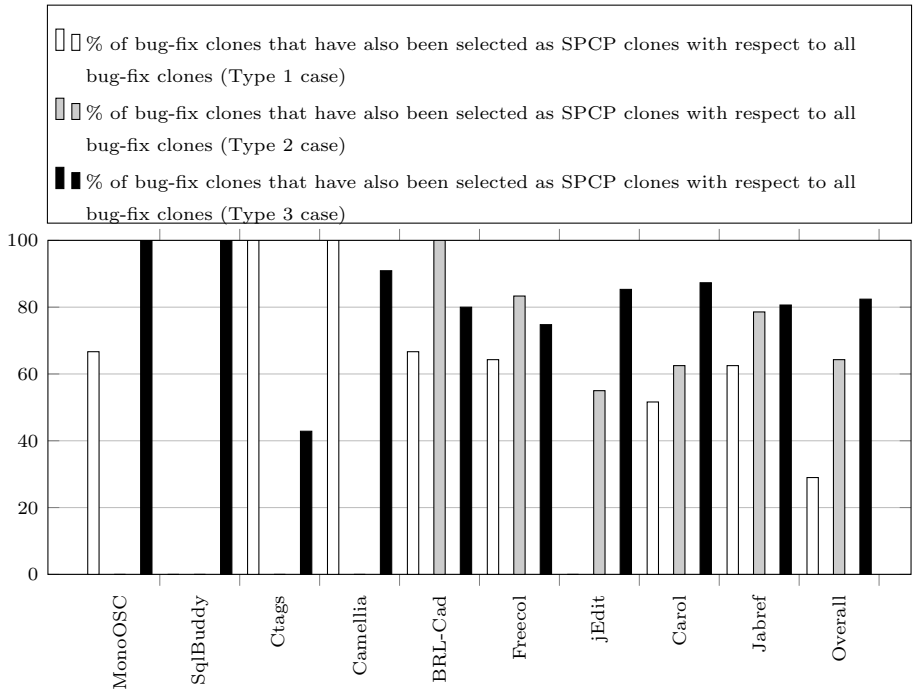

Figure 10: Comparison regarding the percentage of clone fragments that have experienced bug-fixes and have also been selected as SPCP clones.

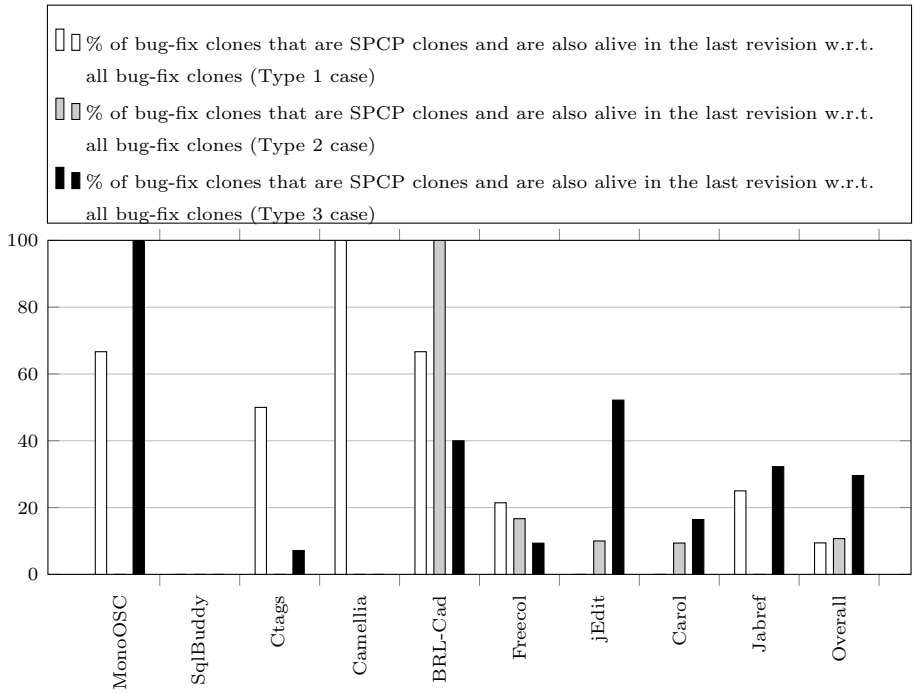

Figure 11: Comparison regarding the percentage of bug-fix clones that have been selected as SPCP clones and are also present in the last revision.

case is the lowest among the three cases. Such an overall scenario can also be observed from the bar-graph in Fig. 11.

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. 11 also imply that for more than 50% 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.

4.4. Answering RQ 4

RQ 4: *Do bug-fix changes mainly occur to the clone fragments that experienced late propagation?*

Rationale. In a previous study Barbour et al.[3] investigated late propagation in code clones. They showed that the code clones that experience late propagation are sometimes related to bug-fix. However, it is still unknown whether bug-fixing changes mainly occur to the late propagation clones or not. We investigate this matter in this research question.

Methodology. Considering each clone-type of each of the subject systems we determine two sets of code clones (i.e., clone genealogies). One set contains all those clones that experienced bug-fixing changes during their evolution. The

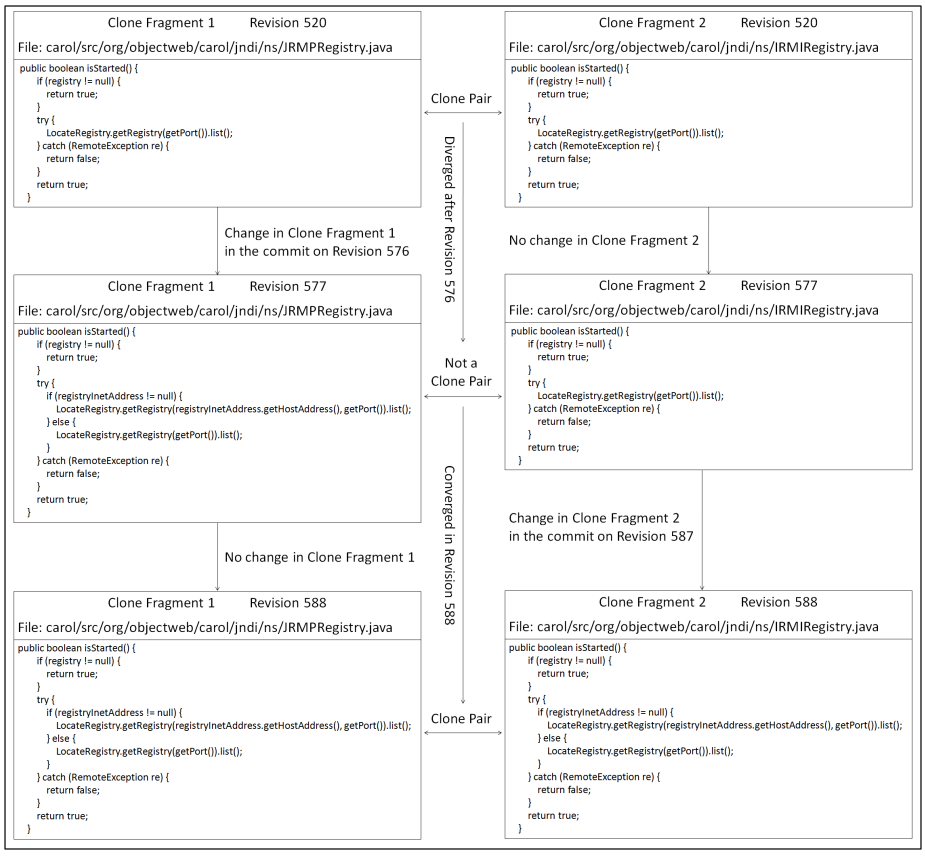

Figure 12: An example of late propagation of two Type 1 clone fragments (i.e., Clone Fragment 1, and Clone Fragment 2) in the code-base of our subject system Carol has been shown here. These two fragments made a clone-pair in revision 520. The commit operation that was applied on revision 576 made a change to Clone Fragment 1. Because of this change, the two fragments diverged. We see that the fragments did not make a clone-pair in revision 577. The period of divergence ended in revision 588 because of the change that occurred to Clone Fragment 2 in the commit operation on revision 587. We see that the two fragments again made a clone-pair in revision 588.

other set contains each of those clone fragments that experienced late propagation(s). Then, we identify which code clones experienced both bug-fixing change(s) and late propagation(s) during their evolution from the intersection of the two sets. Finally, we determine what proportion of the bug-fix clones (i.e., the clones that experienced bug-fixing changes) also experienced late propagations. We detect late propagations following the procedure described in Section 3.3. We also manually analyzed each of the occurrences of late propagations and confirm that in each case, the two participating clone fragments in the clone pair actually experienced a late propagation. Fig. 12 shows an example of late propagation experienced by a Type 1 clone-pair from our subject system Carol. The figure caption contains the details regarding late propagation. Table 8 contains the following measures for each clone-type of each of the subject systems.

- The number of code clones that experienced bug-fixes (The column named **CGBF** in Table 8),
- The number of code clones (i.e., clone genealogies) that experienced late propagations (The column named **CGLP** in Table 8),
- The number of code clones that experienced both bug-fixing changes and late propagations (The column named **CGBFLP** in Table 8).
- The percentage of bug-fix clones that also experienced late propagations with respect to all bug-fix clones (The column named **PBFLP** in Table 8). We calculate PBFLP using the following equation.

$$PBFLP = (CGBFLP \times 100)/CGBF \tag{5}$$

We also calculate the overall value of the percentage (**PBFLP**) considering all subject systems using the following equation.

$$PBFLP_{overall} = \frac{100 \times \sum_{s \epsilon S} CGBFLP_s}{\sum_{s \epsilon S} CGBF_s} \tag{6}$$

Table 8: Percentage of bug-fix clones that also experienced late propagations

	Type 1					Type 2					Type 3				
Systems	CGBF	CGLP	CGBFLP	PBFLP	PLPBF	CGBF	CGLP	CGBFLP	PBFLP	PLPBF	CGBF	CGLP	CGBFLP	PBFLP	PLPBF
MonoOSC	3	0	0	0%	0%	0	0	0	0%	0%	5	2	2	40%	100%
SqlBuddy	0	0	0	0%	0%	0	2	0	0%	0%	4	3	0	0%	0%
Ctags	4	2	0	0%	0%	4	5	0	0%	0%	14	3	0	0%	0%
Camellia	2	0	0	0%	0%	0	0	0	0%	0%	11	0	0	0%	0%
BRL-CAD	3	2	0	0%	0%	2	0	0	0%	0%	10	2	0	0%	0%
Freecol	14	0	0	0%	0%	12	0	0	0%	0%	107	20	0	0%	0%
jEdit	73	3	0	0%	0%	20	8	1	5%	12.5%	184	17	2	1.09%	11.76%
Carol	31	5	3	9.67%	60%	32	0	0	0%	0%	134	9	5	3.73%	55.56%
Jabref	8	6	1	12.5%	16.67%	14	6	0	0%	0%	31	9	3	9.67%	33.33%

CGBF = Total number of Clone Genealogies that experienced Bug-fixing change.

CGLP = Total Number of Clone Genealogies (i.e., clones) that experienced late propagation.

CGBFLP = Total number of Clone Genealogies that experienced both Bug-fixing change(s) and Late Propagation(s).

PBFLP = Percentage of Clone Genealogies that experienced both bug-fixing change(s) and late propagation(s) with respect to all clones that experienced Bug-fixing changes.
PBFLP = CGBFLP * 100 / CGBF

PLPBF = Percentage of Clone Genealogies that experienced both bug-fixing change(s) and late propagation(s) with respect to all clones that experienced late propagations.
PBFLP = CGBFLP * 100 / CGLP

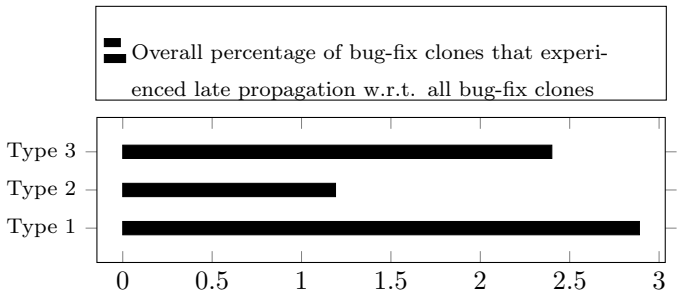

Figure 13: Overall percentage of bug-fix clones that experienced late propagation with respect to all bug-fix clones

Here, S is the set of all subject systems, and s denotes a particular system in this set. We have already defined the terms **CGBFLP**, **CGBF**, and **CGLP**.

If we look at the percentage PBFLP (Percentage of bug-fix clones that also experienced late propagation) in Table 8 we realize that it is generally very low. The percentage is 0% for most of the cases. For only seven cases (for example Type 1 case of Carol), this percentage is greater than zero. If we look at the overall percentages for each clone-type in Fig. 13 we realize that a very little proportion of the code clones that experience bug-fix changes also experience late propagation. This proportion is the highest (2.89%) in Type 1 case and lowest (1.19%) in Type 2 case. In other words, bug-fix clones rarely experience late propagations. Most of the code clones that experienced bug-fix changes have never experienced late propagations.

Statistical significance test. From our previous discussions and analysis we understand that bug-fix clones generally do not experience late propagations. We also wanted to investigate whether the number of clone fragments that experienced both bug-fixes and late propagations is significantly smaller than the number of clone fragments that experienced bug-fixes but did not experience late propagations. Table 9 shows these two numbers for each clone-type of each of the subject systems. From Table 9 we see that there are 27 cases (9 subject systems $\times$ 3 clone-types) in total, and for each case there are two numbers, **CGBFLP** and **CGBFNLP**:

Table 9: Number of bug-fix clones that experienced or did not experience late propagations

	Type 1		Type 2		Type 3	
Systems	CGBFNLP	CGBFLP	CGBFNLP	CGBFLP	CGBFNLP	CGBFLP
MonoOSC	3	0	0	0	3	2
SqlBuddy	0	0	0	0	4	0
Ctags	4	0	4	0	14	0
Camellia	2	0	0	0	11	0
BRL-Cad	3	0	2	0	10	0
Freecol	14	0	12	0	107	0
jEdit	73	0	19	1	182	2
Carol	28	3	32	0	129	5
Jabref	7	1	14	0	28	3

CGBFLP = Number of Clone Genealogies that experienced both Bug-fixing change(s) and Late Propagation(s).

CGBFNLP = Number of Clone Genealogies that experienced Bug-fixing changes(s) but did not experience late propagations.

Table 10: Mann-Whitney-Wilcoxon test result for CGBFLP and CGBFNLP values from Table 9

Sample Size	27
Significance level	5%
U-value	85
Probability value (i.e., *p-value*) for two tailed test case	< 0.001
Probability value (i.e., *p-value*) for one tailed test case	< 0.001
The two samples are significantly different (*p-value*s are less than 0.05).	
Thus, CGBFNLP values are significantly different than the CGBFLP values.	

- **CGBFLP:** The number of clone genealogies (i.e., clone fragments) that experienced both bug-fix changes and late propagations.
- **CGBFNLP:** The number of clone fragments that experienced bug-fix changes but did not experience late propagations.

Here, the summation of these two numbers for a particular case in Table 9 equals the CGBF value for that particular case in Table 8. We perform the Mann-Whitney-Wilcoxon (MWW) test [27, 28] to determine whether the 27 values of CGBFLP are significantly different than the 27 values of CGBFNLP. We perform the test considering a significance level of 5%. The test details have been reported in Table 10. We should note that MWW test is non-parametric [27]. It does not require the samples to be normally distributed [27].

From the test result reported in Table 10 we realize that the **CGBFNLP** values in Table 9 are significantly different than the **CGBFLP** values in the same table. Also, as **CGBFLP** values are always smaller than the corresponding **CGBFNLP** values, we can state that **CGBFLP** is significantly smaller than **CGBFNLP**. In other words, *the number of clone fragments that experi-*

enced both bug-fix changes and late propagations is significantly smaller than the number of clone fragments that experienced bug-fix changes but did not experience late propagations. Thus, bug-proneness of code clones does not seem to be primarily related with late propagation.

Answer to RQ 4. According to our observations *only a very little proportion of the code clones that experience bug-fixing changes can experience late propagations. Our statistical significance test result indicates that the number of code clones that experience bug-fixes without experiencing late propagations is significantly higher compared to the number of code clones that experienced both bug-fixes and late propagations. Thus, bug-proneness of code clones does not seem to be primarily (i.e., strongly) related with late propagation.*

Although we realize that bug-proneness of code clones is not strongly related with late propagation, we still do not know whether late propagation in code clones is primarily related with clone bug-proneness. We investigate this in RQ 5.

4.5. Answering RQ 5

RQ 5: *Do most of the late propagation clones experience bug-fix changes?*

Rationale. In RQ 4 we investigated what percentage of the bug-fix clones experienced late propagations. We found that only a very little proportion of the bug-fix clones can experience late propagation. However, we still do not know whether most of the late propagation clones experience bug-fix changes or not. If most of the late propagation clones experience bug-fixes, then it implies that we should be careful of the occurrences of late propagations. For example, programmers can be automatically notified for each diverging change to the clone fragments. We mentioned that Barbour et al. [3] studied late propagation in code clones. We also have a previous study [41] on late propagation. However, these existing studies do not necessarily answer our fifth research question (i.e., RQ 5). Also, our previous study [41] was performed considering code clones of at least 5LOC. However, in this study we consider code clones of at least 10LOC, because Svajlenko and Roy [52] show that these settings provide us with better

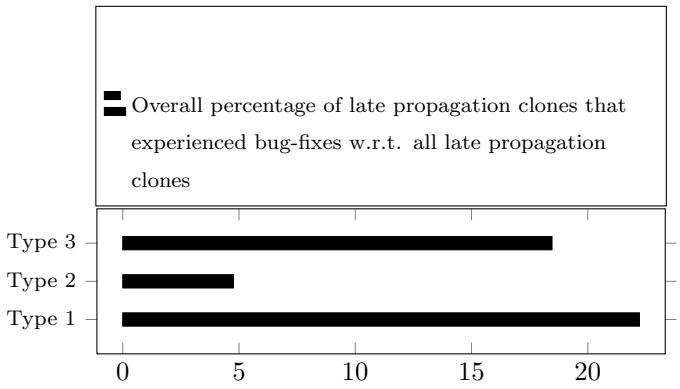

Figure 14: Overall percentage of late propagation clones that experienced bug-fixes with respect to all late propagation clones

clone detection results in terms of both precision and recall. We answer RQ 5 in the following way.

Methodology. For answering RQ 5, we proceed in the same way as we did in our previous research question (RQ 4). From the two measures, CGLP and CGBFLP, recorded in Table 8 we determine the percentage of late propagation clones that experienced bug-fixes with respect to all late propagation clones. We call this percentage PLPBF (Percentage of Late Propagation clones that experienced Bug Fixes) and calculate it in the following way.

$$PLPBF = (CGBFLP \times 100)/CGLP \tag{7}$$

In Table 8, we show this percentage for each clone-type of each of the subject systems. We also calculate the overall value of this percentage considering all subject systems using the following equation.

$$PLPBF_{overall} = \frac{100 \times \sum_{s \epsilon S} CGBFLP_s}{\sum_{s \epsilon S} CGLP_s} \tag{8}$$

From the percentage PLPBF (Percentage of late propagation clones that experienced bug-fixes) in Table 8 we understand that sometimes a considerable proportion of the late propagation clones can experience bug-fixes. If we look at the overall values of the percentage ($PLPBF_{overall}$) in Fig. 14 we can see

that the overall percentage for Type 1 clones is the highest (22.22%) and that of Type 2 clones is the lowest (5.26%). From such a scenario we realize that most of the late propagation clones do not experience bug-fixes. In other words, experiencing bug-fixes is not a common scenario for the late propagation clones.

Statistical significance test. We perform statistical significance tests to determine whether the number of late propagation clones that experience bug-fixes is significantly smaller compared to the number of late propagation clones that did not experience bug-fixes. In Table 11 we record these two numbers (CGLPNBF and CGBFLP) for each clone-type of each of the subject systems. We define CGLPNBF and CGBFLP in the following way.

- **CGLPNBF:** The number of clone genealogies that experienced late propagations but did not experience bug-fixes.
- **CGBFLP:** As we defined previously, it is the number of clone genealogies that experienced both late propagations and bug-fixes.

The summation of these two numbers for a particular case (i.e., for a particular clone-type of a particular subject system) is equal to the CGLP value for that case in Table 8. There are 27 cases (9 subject systems $\times$ 3 clone-types) in total. As we did in RQ 4, we perform MWW tests [27, 28] to determine whether the 27 values of CGBFLP are significantly different than the 27 values of CGLPNBF. We perform the tests considering a significance level of 5%. The test details are shown in Table 12. From the test results we realize that CGLPNBF values are significantly different CGBFLP values. As CGBFLP values are most of the time smaller than the corresponding CGLPNBF values, we can say that CGBFLP is significantly smaller than CGLPNBF. In other words, the number of late propagation clones that experience bug-fixes is significantly smaller than the number of late propagation clones that did not experience bug-fixes. Thus, late propagation clones do not generally experience bug-fixes.

Answer to RQ 5. According to our investigation and analysis, overall only a small proportion of the late propagation clones experience bug-fixes. Most of the late propagation clones do not experience bug-fix changes.

Table 11: Number of late propagation clones that experienced or did not experience bug-fixes

	Type 1		Type 2		Type 3	
Systems	CGLPNBF	CGBFLP	CGLPNBF	CGBFLP	CGLPNBF	CGBFLP
MonoOSC	0	0	0	0	0	2
SqlBuddy	0	0	2	0	3	0
Ctags	2	0	5	0	3	0
Camellia	0	0	0	0	0	0
BRL-Cad	2	0	0	0	2	0
Freecol	0	0	0	0	20	0
jEdit	3	0	7	1	15	2
Carol	2	3	0	0	4	5
Jabref	5	1	6	0	6	3

CGBFLP = Number of Clone Genealogies that experienced both Bug-fixing change(s) and Late Propagation(s).

CGLPNBF = Number of Clone Genealogies that experienced Late Propagations but did not experience bug-fix changes.

Table 12: Mann-Whitney-Wilcoxon test result for CGLPNBF and CGBFLP values from Table 11

Sample Size	27
Significance level	5%
U-value	215
Probability value (i.e., *p-value*) for two tailed test case	0.00988
Probability value (i.e., *p-value*) for one tailed test case	0.00494
The two samples are significantly different (*p-value*s are less than 0.05).	
Thus, CGLPNBF values are significantly different than the CGBFLP values.	

From our answer to RQ 5 we again realize that late propagation is not primarily related with bugs in code clones. However, sometimes a considerable proportion of the late propagation clones can experience bug-fixes. For Type 1 case of our subject system Carol, 3 out of 5 late propagation clones (60% of the late propagation clones) experienced bug-fixes. Thus, it is better not to ignore late propagations. We should take proper measures for minimizing late propagations that might occur because of unconsciousness of the programmers.

4.6. Answering RQ 6

RQ 6: *What proportion of the clone fragments experienced bug-fixes that are associated with late propagation?*

Rationale. From our answer to *RQ 4* we realize that only a very little proportion of the clone fragments that experience bug-fixes also experience late propagation. However, we still do not know whether the bug fixes experienced by this little proportion of clone fragments are really associated with late propagations occurred in those fragments. A clone fragment can experience both a bug-fix and a late propagation during evolution. However, if a bug-fix does not

occur during the diverging period of late propagation, then the bug-fix cannot be considered associated with late propagation. In order to realize the associativity of bug-fixes and late propagations, we need to perform time analysis of the occurrences of bug-fixes and late propagations considering those clone fragments that experienced both late propagations and bug-fixes. We perform such an analysis in RQ 6. The details of our analysis have been given below.

Methodology. In Table 8 we see that the number of clone fragments that experienced both bug-fixes and late propagations has been reported in the column named CGBFLP. This column contains zero for most of the cases except for Type 1 case of Carol and Jabref, Type 2 case of jEdit, and Type 3 case of MonoOSC, jEdit, Carol, and Jabref. We manually analyze the evolution of each of the clone fragments in each of these non-zero cases and determine whether the late propagations experienced by these clone fragments are really associated with bug-fixes. We investigate 17 clone genealogies (calculated from CGBFLP column of Table 8) in total. Table 13 shows our manual analysis results. We see that for each clone genealogy that experienced both bug-fixes and late propagation we have recorded the followings:

- The commit operation where the clone fragment experienced a bug-fix.
- The starting and ending commit operations of the period of late propagation that the clone fragment experienced. The starting commit is the commit where the divergence occurred and the ending commit is the one where the convergence occurred.
- Our decision regarding whether the bug-fix change in the clone fragment is related with the late propagation that it experienced.

From the first row of Table 13 we understand that a Type 1 clone fragment (c.f., Clone Genealogy 1) of our subject system, Carol, experienced a bug-fix in commit operation 430. However, this clone fragment experienced late propagation between commits 576 and 588. Such a scenario clearly implies that the bug-fix experienced by the clone fragment cannot be related with the late

propagation experienced by it, because the bug-fix occurred long before the occurrence of late propagation. However, if we consider the Type 1 clone fragment of subject system, Jabref, we see that the fragment experienced a bug-fix during the period of late propagation. The bug-fix commit, 77, in this example falls in the late propagation period (from commit 76 to 78). In such a scenario, we manually analyze the bug-fix commit message, and the changes that occurred to the clone fragment during the late propagation period. From our manual investigation, we decide whether the bug-fix is really related with late propagation. Our decision is recorded in the last column of Table 13. The last column of this table corresponding to the row for Type 1 case of Jabref we realize that the bug-fix experienced by the Type 1 clone fragment is associated with the late propagation that it experienced. From Table 13 we again see that a Type 3 clone fragment (c.f., Clone Genealogy 1) of the subject system, Carol, experienced bug-fixes in three commit operations: 182, 430, and 730. We manually analyze whether the bug-fixes occurred in commits 430 and 730 are related with late propagation, because these two bug-fix commits occurred in the late propagation period. From our analysis we found that the bug-fix change that occurred in commit 730 is related with late propagation.

By analyzing the last column in Table 13 we realize that for only 8 out of 17 clone genealogies, the bug-fixes are really associated with late propagation. Considering all the bug-fix clone genealogies (the column CGBF in Table 8) in all the subject systems we determine what proportion of these clone genealogies experienced bug-fixes that are really associated with late propagation. This percentage for each of the three clone-types (Type 1, Type 2, and Type 3) is shown in the graph of Fig. 15. In a similar way, by considering all the late propagation clone genealogies from all the subject systems we determine what proportion of these clone genealogies experienced late propagations that are really associated with bug-fixes. This percentage for each clone-type is shown in Fig. 16. Each of the two graphs demonstrates that the percentage regarding Type 2 is zero. From Table 8 we see that only one Type 2 clone genealogy of our subject system, jEdit, experienced both a bug-fix and a late propagation.

However, from Table 13 we realize that the bug-fix experienced by this clone genealogy is not related with the late propagation experienced by it. No other Type 2 clone genealogy of any other subject system experienced both a bug-fix and a late propagation. As a result, the percentage regarding Type 2 case in each of the two graphs, Fig. 15 and Fig. 16, becomes zero. From these two graphs we can state the followings:

- The bug-fixes in a very little proportion of the bug-fix clone genealogies can be associated with late propagation. This proportion is the lowest (i.e., zero) for Type 2 case. For Type 3 clones, this proportion is the highest which is only 1.4%.

- The late propagations occurred in a very little proportion of the late propagation clone genealogies can be associated with bug-fixes. While this proportion is the lowest (zero) for Type 2 clones, Type 3 clones exhibit the highest proportion which is only 10.76%.

Answer to RQ 6. Our findings imply that bug-proneness in code clones is not primarily related with late propagation. Moreover, only a little proportion of the occurrences of late propagation can be associated with bug-fix. Thus, it seems that late propagation is not strongly related with bug-proneness of code clones.

5. Threats to Validity

We used the NiCad clone detector [6] for detecting both exact and near-miss clones. We detected near-miss clones considering a dissimilarity threshold of 20% with blind renaming of identifiers. For different settings of NiCad, the statistics that we present in this paper might be different. Wang et al. [55] 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 [47] and with these settings NiCad can detect clones with high precision and recall [48, 49, 52].

Table 13: Description of the cases where a clone fragment experienced both a bug-fix and late propagation

Cases	CGBFLP	Clone Genealogies	Experienced bug-fix in commit	Experienced Late propagation between commits	Is Bug-fix associated with late propagation?
Type 1 case of Carol	3	Clone genealogy 1	430	576 and 588	No
		Clone genealogy 2	422	217 and 277	No
		Clone genealogy 3	422	217 and 277	No
Type 1 case of Jabref	1	Clone genealogy 1	77	76 and 78	Yes
Type 2 case of jEdit	1	Clone genealogy 1	3928	3826 and 3860	No
Type 3 case of jEdit	2	Clone genealogy 1	3960	3959 and 3963	Yes
		Clone genealogy 2	3846	3893 and 3900	No
Type 3 case of Carol	5	Clone genealogy 1	182, 430, 730	397 and 730	Yes
		Clone genealogy 2	397, 430	396 and 398	Yes
		Clone genealogy 3	430, 464	396 and 398	No
		Clone genealogy 4	398, 430	397 and 730	Yes
		Clone genealogy 5	1597	1522 and 1578	No
Type 3 case of Jabref	3	Clone genealogy 1	881, 883	880 and 883	Yes
		Clone genealogy 2	881, 883	880 and 883	Yes
		Clone genealogy 3	881, 883	880 and 883	Yes
Type 3 case of MonoOSC	2	Clone genealogy 1	302	312 and 352	No
		Clone genealogy 2	302	312 and 352	No

CGBFLP = Number of Clone Genealogies that experienced both Bug-fixing change(s) and late propagation(s)

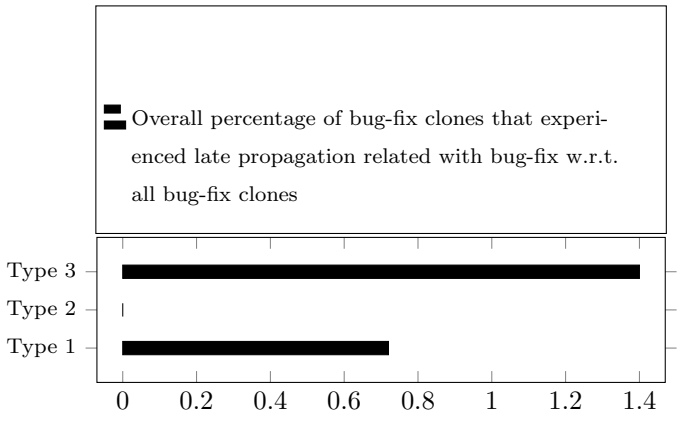

Figure 15: Overall percentage of bug-fix clones that experienced late propagation which is related with bug-fix with respect to all bug-fix clones

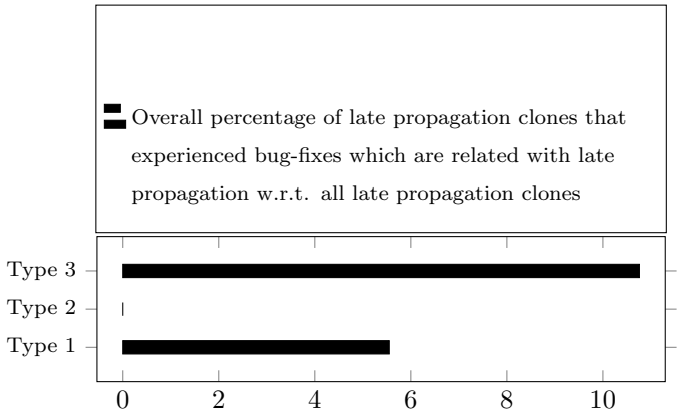

Figure 16: Overall percentage of late propagation clones that experienced bug-fixes which are associated with late propagation with respect to all late propagation clones

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 [31] 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.

We detect bug-fix commits by automatically analyzing the commit messages. Issue tracking system could be an option for identifying bugs. However, we did not use issue tracking system for a number of reasons. First, we were interested about the bugs that got fixed. Generally, an issue tracking system contains a lot of bug reports. However, many of these bugs are open for a long time. Also, existing research shows that fixing priorities for these reported bugs are often assigned without properly considering their importance or severity. This might also be the case that a particular bug has been fixed but the corresponding entry in the issue tracking system has not been updated. Thus, an issue tracking system might not always reflect the latest information on bug-fixing. Second, during our manual analysis of the bug-fixing commits we realize that many bugs get fixed without a reported bug-id. It implies that either those bugs were not reported in the issue tracking system or those bugs were reported but the developer did not mention the bug-id (i.e., issue id) in the commit message after fixing. If we only rely on the bugs reported in the issue tracking or bug tracking system, we will miss bug-fixes without any bug-id. Third, by manually examining the commit messages of the all the bug-fixes that occurred during late propagation periods we see that none of these commit messages report a bug-id or issue-id. However, the commit messages report which bugs were fixed in which ways. Thus, for our subject systems, if we rely on the issue tracking system, we will not get any bug-id or issue-id related with late propagation. We finally believe that our decision of not including issue tracking system in our research is reasonable.

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.

6. Related Work

Bug-proneness of code clones has already been investigated by a number of studies. Li and Ernst [23] 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. [24] developed a tool called CP-Miner which is capable of detecting bugs related to inconsistencies in copy-paste activities. Steidl and Göde [51] 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 [10] 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.[5] performed a user study to investigate how clone information can help programmers localize bugs in software systems. Jiang et al.[14] 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.[13] 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.[56] 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.

In a previous study [41] we only investigated late propagation and late propagation related bugs in code clones. However, in our study presented in this paper we identify all the bugs that occurred to code clones (i.e., not only the late propagation related bugs) and make a comparison of the bug-proneness of the three clone-types. Such a comparison was not done in our previous study [41]. In this study we also investigate whether bug-proneness of code clones is primarily related with late propagation by answering three research questions: RQ 4, RQ 5, and RQ 6. Such an investigation was not done in our previous study. From this investigation with manual verification we find that a very little proportion (1.4%) of the bug-fix clones experienced bug-fixes that are related with late propagation. Such a finding is a new addition to clone research. None of the existing studies reveals this finding. This finding indicates that late propagation contributes to only a very little proportion of bugs in code clones. In our previous study we considered code clones of at least 5LOC. However, in this study we consider code clones of at least 10LOC, because Svajlenko and Roy [52] found that with such a setting NiCad can provide better clone results in terms of precision and recall.

Rahman et al. [43] 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 [15] 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 2 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.

Barbour et al. [4] investigated evolution of code clones in four subject systems with the goal of identifying factors that can affect the bug-proneness of

code clones. According to their findings, clone size can be directly related to the bug-proneness of clone pairs. They also found that time interval between consecutive changes to clone pairs does not have a considerable impact on clone bug-proneness. Our study is different than Barbour et al.'s study. We investigate the evolution history of code clones from nine subject systems in order to make a comparison among the bug-proneness of different clone-types. We also investigate the relatedness of late propagation in code clones with clone bug-proneness. Barbour et al. [4] did not perform such investigations.

Selim et al. [50] 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 nine diverse subject systems in our investigation. Also, we compare the bug-proneness of different types of clones. Selim et al. [50] did not perform a type centric analysis in their study.

A number of studies have also been done on late propagation in code clones and its relationships with bugs.

Barbour et al. [2, 3] investigated different types of late propagation in code clones. They reported that clone genealogies that experienced late propagation have a higher possibility of experiencing bug-fixes compared to the clone genealogies that did not experience late propagations. In their experiment, they first identified the late propagation clone genealogies, and then determined which ones of these genealogies experienced bug-fixes. However, they did not determine whether the late propagation clone genealogies experienced bug-fixes during their late propagation periods. A particular clone genealogy might experience both late propagation and bug-fix. An occurrence of late propagation can be related to a bug-fix, only if the bug-fix occurs during the late propagation period. Barbour et al. did not investigate whether the late propagation genealogies experienced bug-fixes during the period of late propagation. Thus, their investigation does not report how many of the clone genealogies experienced bug-fixes that are really associated late propagation. In our research, we

report this statistic by incorporating time analysis and manual investigations. Moreover, Barbour et al. investigated using Simian and CCFinder clone detectors with a minimum clone size of 6 lines and 50 tokens (equivalent to 5 lines) respectively. We have used NiCad clone detector because NiCad is a better choice according to a recent study [52] of Svajlenko and Roy. We considered a minimum clone size of 10 lines as was suggested by Svajlenko and Roy.

By investigating two subject systems, Aversano et al. [1] reported that around 18% of the code clones undergo late propagation during evolution. They also observed that around 41% of the clone related bug-fixes occurred during late propagation. While such a finding establishes the fact that late propagation is risky, it does not imply whether bug-proneness of code clones is mostly related with late propagation. In our study, we investigate thousands of revisions of 9 diverse subject systems and analyze whether bug-proneness of code clones is mostly related with late propagations or not.

From an investigation on four open source subject systems written in C and Java, Thummalapenta et al. [53] reported that code clones that experience late propagations are more prone to faults. However, their experiment does not report what percentage of bug-fixes experienced by code clones were directly related with late propagation. In our research we quantify late propagation related bugs from two directions: (i) by determining what proportion of the bug-fixes experienced by code clones are related to late propagation, and (2) by determining what proportion of the late propagation clone genealogies experienced bug-fixes. From these measures we try to determine whether late propagation is the primarily related to bugs in code clones.

Our finding regarding the proportion of late propagation clone genealogies is consistent with the findings from existing studies [53, 1, 2, 3]. We support that clone genealogies experiencing late propagations sometimes have high possibilities of containing bugs (i.e., high possibilities of experiencing bug-fixes). However, we additionally investigate whether bug-proneness of code clones is mostly associated with late propagation or not. Such an investigation was not done by the existing studies.

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.

7. Conclusion

In this paper we present an empirical study on the comparative bug-proneness of different types of code clones. We also investigate whether clone bug-proneness is mainly related to late propagation in code clones. According to our investigation on the major clone-types: Type 1 (Exact clones or identical clones), Type 2 (Near-miss clones), and Type 3 (Near-miss clones) in thousands of revisions of nine diverse subject systems written in three different programming languages (C, Java, and C#) 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 [35, 36] show that the SPCP clones (i.e., the clone fragments that evolve following a similarity preserving change pattern) are important for refactoring

and tracking.

(4) Finally, it appears that bug-proneness in code clones is not primarily related with late propagation. The possibility that a bug-fix change experienced by a clone fragment will be associated with late propagation is only 1.4%. We also find that late propagation in code clones is not strongly related with clone bug-proneness. The probability that a late propagation experienced by a clone fragment will be related with a bug is only 10.76%. According to our statistical significance tests, the number of late propagation clones that experienced bug-fixes is significantly smaller than the number of non-late-propagation clones that experienced bug-fixes.

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. Our research also implies that late propagation in code clones is not primarily related with bug-proneness of code clones. Such a finding helps us realize the extent to which late propagation contributes to clone bug-proneness.

In future, we plan to investigate clone ranking for refactoring and tracking considering our findings 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] L. Barbour, L. An, F. Khomh, Y. Zou, and S. Wang, "An investigation of the Fault-proneness of clone evolutionary patterns", *Software Quality Journal*, 2017, pp. 1 - 36.

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

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

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

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

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

[12] 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.

[13] 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.

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

[15] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones", Proc. *ICSE*, 2007, pp. 96105.

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

[17] 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.

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

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

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

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

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

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

[24] 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.

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

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

[27] H. B. Mann, D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other", *Annals of Mathematical Statistics*, 1947, 18(1): 5060.

[28] Mann-Whitney-Wilcoxon Test Online. http://www.socscistatistics.com/tests/mannwhitney/Default2.aspx

[29] Mann-Whitney-Wilcoxon Test Details. https://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test

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

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

[32] 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).

[33] 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.

[34] 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.

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

[40] 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.

[41] M. Mondal, C. K. Roy, and K. A. Schneider, "A comparative study on the intensity and harmfulness of late propagation in near-miss code clones", *Software Quality Journal*, pp. 1 – 33.

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

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

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

[45] 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.

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

[47] 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.

[48] 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.

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

[50] 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.

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

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

*Biography

Manishankar Mondal

Manishankar Mondal is an Assistant Professor in the Computer Science and Engineering Discipline of Khulna University, Bangladesh. He completed his M.Sc. in Software Engineering from the Computer Science Department of the University of Saskatchewan, Canada by working under the supervision of Dr. Chanchal K. Roy and Dr. Kevin A. Schneider. During M.Sc. studies, he received the Best Paper Award from the 27th Symposium On Applied Computing (ACM SAC 2012) in the Software Engineering Track. He also received his PhD in February, 2017 from the same department by working under the same advisors. His research interests are software maintenance and evolution including clone detection and analysis, program analysis, empirical software engineering and mining software repositories. He has been a reviewer of a number of software engineering conferences and journals. He has served as the web and publicity co-chair of ICPC 2014 and as a program committee member of IWSC 2016. He has also served as a research associate in the software research laboratory of the Computer Science Department of the University of Saskatchewan.

Chanchal K. Roy

Chanchal Roy is an associate professor of Software Engineering/Computer Science at the University of Saskatchewan, Canada. While he has been working on a broad range of topics in Computer Science, his chief research interest is Software Engineering. In particular, he is interested in software maintenance and evolution, including clone detection and analysis, program analysis, reverse engineering, empirical software engineering and mining software repositories. He served or has been serving in the organizing and/or program committee of major software engineering conferences (e.g., ICSE, ICSME, SANER, ICPC, SCAM, CASCON, and IWSC). He has been a reviewer of major Computer Science journals including IEEE

Transactions on Software Engineering, International Journal of Software Maintenance and Evolution, Science of Computer Programming, Journal of Information and Software Technology and so on. He received his Ph.D. at Queen's University, advised by James R. Cordy, in August 2009.

Kevin A. Schneider

Kevin Schneider is a Professor of Computer Science, Special Advisor ICT Research and Director of the Software Engineering Lab at the University of Saskatchewan. Dr. Schneider has previously been Department Head (Computer Science), Vice-Dean (Science) and Acting Chief Information Officer and Associate Vice-President Information and Communications Technology.

Before joining the University of Saskatchewan, Dr. Schneider was CEO and President of Legasys Corp., a software research and development company specializing in design recovery and automated software engineering. His research investigates models, notations and techniques that are designed to assist software project teams develop and evolve large, interactive and usable systems. He is particularly interested in approaches that encourage team creativity and collaboration.