Identifying Code Clones having High Possibilities of Containing Bugs

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

***Abstract*—Code cloning has emerged as a controversial term in software engineering research and practice because of its positive and negative impacts on software evolution and maintenance. Researchers suggest managing code clones through refactoring and tracking. Given the huge number of code clones in a software system's code-base, it is essential to identify the most important ones to manage. In our research, we investigate which clone fragments have high possibilities of containing bugs so that such clones can be prioritized for refactoring and tracking to help minimize future bug-fixing tasks. Existing studies on clone bug-proneness cannot pinpoint code clones that are likely to experience bug-fixes in the future.**

According to our analysis on thousands of revisions of four diverse subject systems written in Java, change frequency of code clones does not indicate their bug-proneness (i.e., does not indicate their tendencies of experiencing bug-fixes in future). Bug-proneness is mainly related with change recency of code clones. In other words, more recently changed code clones have a higher possibility of containing bugs. Moreover, for the code clones that were not changed previously we observed that clones that were created more recently have higher possibilities of experiencing bug-fixes. Thus, our research reveals the fact that bug-proneness of code clones mainly depends on how recently they were changed or created (for the ones that were not changed before). It invalidates the common intuition regarding the relatedness between high change frequency and bug-proneness. We believe that code clones should be prioritized for management considering their change recency or recency of creation (for the unchanged ones).

I. Introduction

Code cloning is a common yet controversial software engineering practice which is often employed by programmers during software development and maintenance for repeating common functionalities. Cloning refers to the task of copying a code fragment from one place of a code-base and pasting it to some other places with or without modifications [40]. The original code fragment (i.e., the code fragment from which the copies were made) and the pasted code fragments become clones of one another. Two exactly or nearly similar code fragments form a clone pair. A group of similar code fragments forms a clone class.

Code clones are of great importance from the perspectives of software maintenance and evolution. A great many studies [1], [2], [10], [11], [13], [18], [19], [21], [22], [23], [27], [28], [39], [49], [25], [47], [16], [45], [51], [33], [35], [52] have already been conducted on the impacts of clones on the evolution and maintenance of software systems. While some of these studies [1], [11], [13], [19], [21], [22], [23] identify some positive impacts of code clones, a number of studies [2], [18], [27], [10], [28], [25], [47], [16] have shown empirical evidence of strong negative impacts of code clones such as hidden bug propagation [25], late propagation [2], unintentional inconsistencies [2], [10], and high instability [28]. Because of these negative impacts, code clones are considered to be the number one bad smell in the code-base of a software system.

Motivation. When considering the negative impacts of code clones, researchers recommend managing code clones by refactoring [50] and tracking [9]. However, a software system may contain a large number of code clones, and it is impractical to consider all these clones for refactoring or tracking [20]. Clone refactoring is time consuming, and clone tracking is resource intensive. In such a situation, prioritizing code clones for refactoring and tracking is important. Bug-proneness of code clones should be taken into proper consideration when prioritizing code clones. Code clones with high possibilities of containing bugs should be given a high priority for management. Refactoring or tracking of such clones can help us minimize future bug-fix effort in the following ways:

(1) If the clone fragments in a clone class have a possibility of containing bugs, then it is better to merge (i.e., refactor) the fragments in that class into a single fragment. Any future bug-fixing change will then need to be implemented in just a single fragment instead of implementing and propagating the same fixing change to multiple fragments in the clone class.

(2) If merging the clone fragments in a bug-prone clone class is impossible, then considering them for tracking with simultaneous editing support [9] can help us implement any future bug-fixing change to the fragments with minimal effort.

There are a number of studies [25], [26], [47], [5], [16], [14], [54], [2], [10] on clone bug-proneness, however, none of these existing studies can identify which of the clone fragments residing in a software system's code-base have high possibilities of containing bugs (i.e., have high possibilities of experiencing bug-fixes in future). Focusing on this, in our research we investigate which clone fragments have high possibilities of containing bugs. In particular, we analyze whether and how we can infer bug-prone clones (i.e., clones having possibilities of experiencing bug-fixes in future) from the clone evolution history of our candidate software system. We answer four important research questions listed in Table I in order to discover how change-proneness of code clones can be related to their bug-proneness.

Experiment and Findings. We detect the bug-fix changes in the three major types of clones (Type 1, Type 2, and Type 3) residing in thousands of revisions of four open-source subject systems written in Java. We determine the change-proneness of code clones during system evolution, and analyze whether and how their change-proneness can be related to the occurrence of bug-fixes to them. According to our experimental results and analysis we have the following findings:

Although code clones that were changed in the past have a significantly higher probability of containing bugs (i.e., have a significantly higher probability of experiencing bug-fixes) compared to code clones that were not changed before, *we discover that change frequency of code clones is not related to their bug-proneness. In other words, high change frequency of code clones is not an indication of clone bug-proneness. Bug-proneness of code clones is mainly related to their change recency (i.e., how lately a clone fragment was changed). More recently changed clones have higher tendencies of experiencing bug-fixes in future. Moreover, for the clone fragments that were not changed in the past we observe that the more recently created ones have higher probabilities of experiencing bug-fixes. Thus, our research invalidates the common belief regarding the relatedness of high change frequency with bug-proneness, and reveals the fact that clone bug-proneness mainly depends on how recently the clone fragments were modified or created (for the unchanged clone fragments). Our findings are supported by statistical significance tests.*

According to our findings we suggest that code clones that have been created or modified recently should be given high priorities for management (refactoring and tracking), because such code clones have high possibilities of containing bugs.

Our prototype tool can order code clones according to their recency of creation and modification, and thus, it can help us prioritize code clones for management from the perspective of clone bug-proneness.

A number of studies [55], [4], [24] have investigated scheduling for clone refactoring activity. Given a number of clone fragments for refactoring, these studies aim to find an optimal schedule for refactoring tasks so that refactoring gain is maximized and refactoring effort is minimized. However, these studies cannot identify which of the huge number of code clones in a software system should be given a high importance for refactoring. Our research objective is to prioritize code clones for refactoring or tracking considering their bug-proneness. Thus, we believe that our study can complement the existing clone scheduling studies and techniques by pin pointing the most important ones to be scheduled.

The rest of the paper is organized as follows: Section II describes the terminology, Section III elaborates on the experimental steps, Section IV presents and analyzes the experimental results, Section V presents a discussion of our findings, Section VI discusses the related work, Section VII mentions the possible threats to validity, and Section VIII concludes the paper by mentioning the future work.

II. Terminology

A. Clone-types

We investigate both identical (Type 1) and near-miss code clones (Type 2 and Type 3 clones) in our research. We provide definitions of these three types of code clones in the following way according to the literature [41], [40].

- **Type 1 Clones.** The identical code fragments residing in a software system's code-base are called Type 1 clones. More elaborately, if two or more code fragments in a code-base are exactly the same disregarding their comments and indentations, then we call these code fragments identical clones or Type 1 clones of one another.
- **Type 2 Clones.** Syntactically similar code fragments residing in a software system's code-base are known as Type 2 clones. Type 2 clones are generally created from Type 1 clones because of renaming identifiers and/or changing data types.
- **Type 3 Clones.** Type 3 clones, also known as gapped clones, are generally created from Type 1 or Type 2 clones because of additions, deletions, or modifications of lines in these clones.

TABLE I. Research Questions

SL	Research Question
RQ 1	What is the probability that a clone fragment that was changed (or was not changed) will experience a bug-fix change?
RQ 2	Do clone fragments that were changed more frequently in the past have higher possibilities of experiencing bug-fixes?
RQ 3	Do the clone fragments that were changed more recently in the past have higher possibilities of experiencing bug-fixes?
RQ 4	Which of the clone fragments that were not changed in the past have higher possibilities of experiencing bug-fixes?

B. Clone Fragment

We frequently use the term 'clone fragment' in our paper. A clone fragment is a particular code fragment which is exactly or nearly similar to one or more other code fragments in a code-base. In the introduction we defined a 'clone-pair' and a 'clone class'. Each member in a clone class or a clone-pair is a clone fragment.

C. Clone Genealogy

We detect clone genealogies [20] for the purpose of our investigations. We define a clone genealogy in the following way. Let us assume that a clone fragment was created in a particular revision and was alive in a number of consecutive revisions. Thus, each of these revisions contains a snapshot of the clone fragment. The genealogy of this clone fragment 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 using the SPCP-Miner [32] tool. In our research, by examining the genealogy of a clone fragment we determine which commit operation(s) made changes to it.

III. Experimental Steps

We perform our investigation on four Java systems downloaded from an on-line SVN repository [37]. We list these systems in Table II. We select these systems focusing on

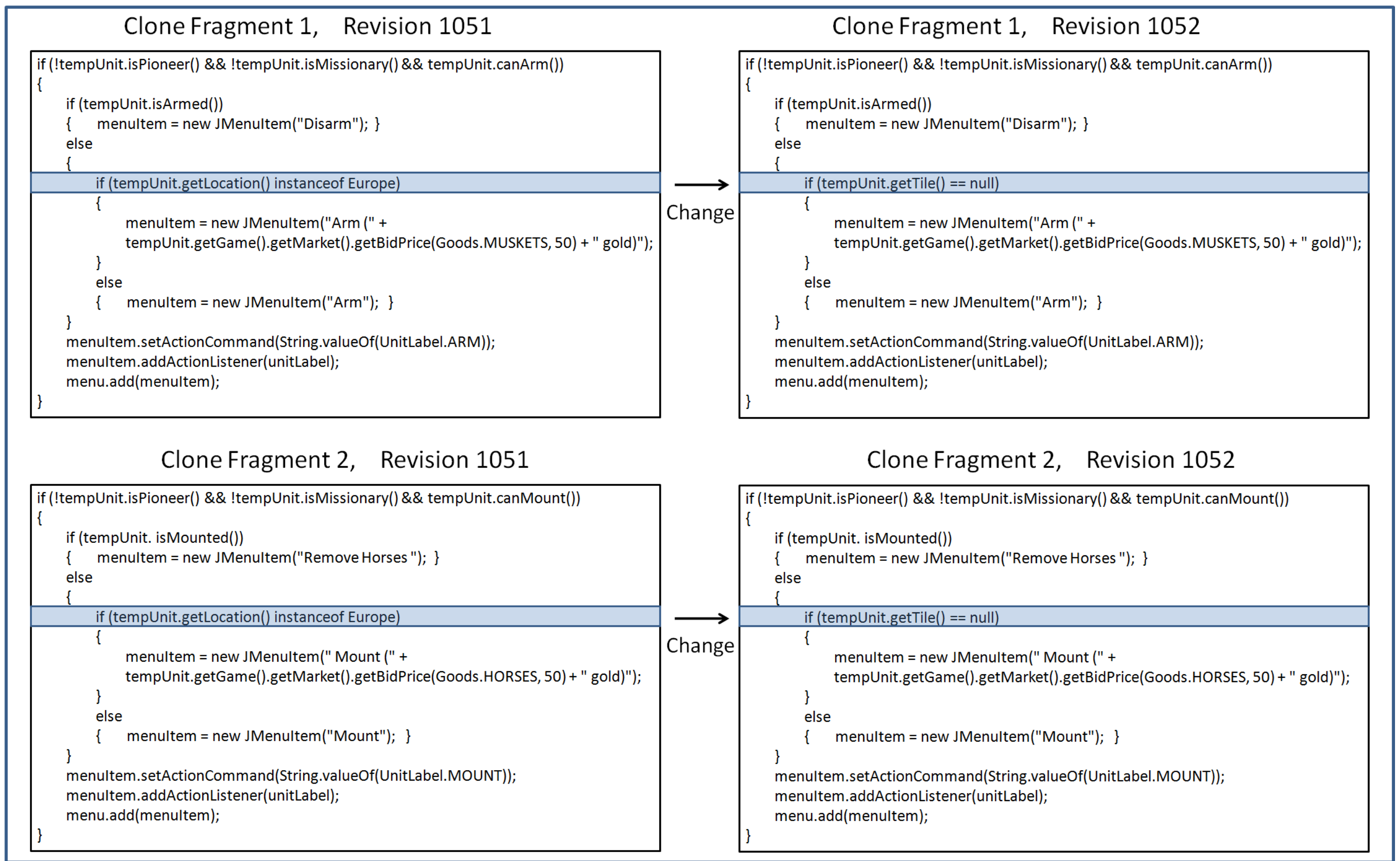

Fig. 1. The commit operation that was applied on revision 1051 of our subject system Freecol was detected as a bug-fix commit by our described bug detection technique. Two clone fragments residing in revision 1051 were changed in this commit operation. The figure shows the snapshots of these clone fragments both in revision 1051 and 1052. The changes that occurred to these clone fragments have also been highlighted. The commit message says, '*Fixed the bug reported by: [1261640] Equipping boarded European units - v0.41cvs*'. From the commit message we understand that the commit was applied for fixing a reported bug. We see that each of the clone fragments was changed in the same way to fix the bug.

TABLE II. SUBJECT SYSTEMS

Systems	Lang.	Domains	LLR	Revs
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Management	45,515	1545

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

TABLE III. NICAD SETTINGS FOR THREE CLONE-TYPES

Detection Parameter	Type 1	Type 2	Type 3
Identifier Renaming	none	blindrename	blindrename
Dissimilarity Threshold	0%	0%	20%
Clone Granularity	block	block	block

their diversity in sizes and revision history lengths. Table II also shows that the systems belong to three different application domains. Before investigating the bug-proneness of code clones in these systems we sequentially perform the following preliminary steps.

- Extracting all revisions (mentioned in Table II) of each candidate system from the online SVN repository.
- Detecting and extracting methods from each of the revisions using CTAGS [7].
- Detecting three types (Type 1, Type 2, and Type 3) of code clones from each revision by applying the NiCad [6] clone detector.
- Detecting changes between every two consecutive revisions using UNIX *diff*.
- Mapping these changes to the already detected methods and clones of the corresponding revisions by using the file paths and the starting and ending line numbers of the changes, methods, and clones.
- Mapping the code clones detected from each revision to the methods of that revision by using the file paths and the starting and ending line numbers of the methods and clones.
- Detecting method genealogies considering all revisions using the technique proposed by Lozano and Wermelinger [28].
- Detecting clone genealogies considering each clone-type separately by identifying the propagation of each clone fragment through a method genealogy.
- Detecting bug-fix changes in code clones.

We use the tool called SPCP-Miner [32] to automatically perform all these steps except the last one. The last step will be discussed later in this section.

A. Clone Detection

We detect three types (Type 1, Type 2, and Type 3) of code clones using the NiCad [6] clone detector. NiCad can separately detect these clone-types with high precision and recall [43], [44]. We setup NiCad to detect block clones of Type 1, Type 2, and Type 3 with a minimum size of 10 LOC. The detection parameters for three clone-types are shown in Table III. According to a recent study [48], these settings provide us with better clone detection results in terms of both precision and recall. We should note that before using the NiCad outputs for Type 2 and Type 3 cases, we pre-processed them in the following way.

- Each Type 2 clone class that exactly matched any Type 1 clone class was excluded from Type 2 outputs.
- Each Type 3 clone class that exactly matched any Type 1 or Type 2 class was excluded from Type 3 outputs.

We performed these steps because we wanted to investigate bug-proneness of each of the three types of clones separately.

B. Detecting Clone Genealogies of Different Clone-Types

SPCP-Miner first applies the NiCad clone detector to detect code clones of each clone-type separately from each revision of a candidate system. Then, considering each clone-type separately it performs origin analysis of the clone fragments of that type detected from different revisions and builds the genealogies. Thus, each clone genealogy contains instances of one particular clone-type only. Here, an instance refers to the snapshot of a clone fragment in a particular revision. We finally get three separate sets of clone genealogies for three different clone-types of a subject system. By considering the genealogies in each set we analyze the bug-proneness of the corresponding clone-type.

C. Handling Clone-Mutations

According to a study of Xie et al.[54], a clone fragment may change its clone-type during evolution. This phenomenon is called clone-mutation. If a clone fragment is considered of different clone-types during different periods of evolution, then SPCP-Miner builds a separate genealogy for this fragment for each of these periods. Thus, even in presence of clone-mutations, we can distinguish which bug-fixes were experienced by which clone-types.

D. Detecting Bug-fixes in Code Clones

We detect bug-fix changes in code clones using the following two steps:

Step 1. We obtain the commit messages of a subject system using the 'SVN log' command. We automatically examine the commit messages, and identify the bug-fix commits using the heuristic proposed by Mockus and Votta [31].

Step 2. We determine which of the bug-fix commits (identified in **Step 1**) made changes to clones. If one or more clone fragments get affected (i.e., are changed) by a particular bug-fix commit for fixing a bug, then it implies that the clone fragment(s) previously contained the bug and modification of these fragment(s) was necessary for fixing the bug.

Fig. 1 shows the bug-fix changes that occurred to two clone fragments in revision 1051 of our subject system Freecol. The figure caption contains necessary descriptions regarding the figure, and mentions the bug-fix commit message.

TABLE IV. NUMBER OF CLONE GENEALOGIES AND BUG-FIX COMMITS

	Type 1		Type 2		Type 3	
System	CG	NBFC	CG	NBFC	CG	NBFC
jEdit	7398	37	399	10	2688	42
Freecol	239	7	162	10	752	46
Carol	415	8	211	8	682	22
Jabref	483	6	228	6	1363	23

CG = No. of clone genealogies
NBFC = No. of bug-fix commits where clone fragments were changed

Previously Barbour et al. [2] used the same technique that we have used for detecting bug-fix commits. They investigated whether late propagation in code clones is related to bugs. Their study involved only the late propagation clones (i.e., the clones that experienced late propagations), and the bug-fixes experienced by these late propagation clones. However, our study is not limited to late propagation clones only. We investigate bug-proneness of all code clones of a software system. Barbour et al.[2] did not consider Type 3 clones in their study. We investigate Type 3 clones in our study. Moreover, we investigate prioritizing code clones for management considering their bug-proneness. Barbour et al.[2] did not perform such an investigation.

IV. EXPERIMENTAL RESULTS

In this section we describe our experiments and answer each of the four research questions (Table I) in our discussion of the experimental results. Table IV shows, for each system, the total number of clone genealogies (**CG**) and the total number of bug-fix commits (**NBFC**), by clone type. The clone genealogies are determined for each system's entire evolution period. Section II defines a clone genealogy. In Section III we discussed that we automatically build clone genealogies from the clone fragments in different revisions using the SPCP-Miner [32] tool. Clone genealogies help us analyze how clone fragments were modified during evolution. **NBFC** is the number of those bug-fix commits that modified one or more clone fragments. A bug-fix commit may affect clone fragments of more than one clone-type, and this is reflected in the number of bug-fix commits (**NBFC**) for each clone type. That is, if a single bug-fix commit changes clone fragments of more than one clone type, the **NBFC** count for each of these clone types will be incremented.

There is a one-to-one relationship between revisions and commits. When a particular revision experiences a commit operation, the next revision is created. Consider the set of bug-fix commits that affected (i.e., made changes to) code clones of a particular clone-type. Each of these commits was applied to a particular revision, and one or more clone fragments in that revision were changed. For this set of bug-fix commits we identify a corresponding set of revisions that experienced these commits. We refer to this set as **BR** (see Table V) and use it in our investigations described in the following subsections. We use r to represent a member of the set **BR**.

RQ 1: What is the probability that a clone fragment that was changed (or not changed) will experience a bug-fix change?

Motivation. The primary goal of our research is to investigate which clone fragments in different clone-types have high possibilities of containing bugs (i.e., high possibilities of experiencing bug-fixes). In RQ 1 we automatically analyze

TABLE V. DEFINITIONS

Term	Definition
BR	The set of all those revisions that experienced bug-fix commits that affected code clones (i.e., clone fragments) of a particular clone-type. That is, a particular revision *r* in the set *BR* experienced a bug-fix commit, and one or more clone fragments of a particular clone-type residing in revision *r* were changed in this bug-fix commit.
r	A member of the set *BR*. That is, a revision that experienced a bug-fix commit that changed one or more clone fragments in this revision.

whether the clone fragments that were changed in the past have higher possibilities of experiencing bug-fixes compared to the clone fragments that were not changed before. Answer to RQ 1 can help us understand whether and to what extent change-proneness of code clones is related to clone bug-proneness.

Methodology. We first determine the bug-fix commit operations following the procedure described in Section III-D, and then, determine the set *BR* (defined in Table V) for a particular clone-type of a particular subject system. Let us consider a particular revision *r* in the set *BR*. The bug-fix commit operation which was applied on revision *r* is BFC_r. One or more clone fragments residing in revision *r* were changed by this bug-fix commit operation. We determine the following four measures considering revision *r*:

- $\mathbf{NCC_r}$: This is the number of clone fragments, each of which satisfies the following two conditions: **(1)** it (i.e., the clone fragment) resides in revision *r*, and **(2)** it was changed in the past (i.e., it was changed in any of the commit operations that occurred before the occurrence of the bug-fix commit BFC_r). Whether a clone fragment residing in revision *r* was changed in the past is determined by examining its genealogy.
- $\mathbf{NCN_r}$: This is the number of clone fragments, each of which obeys the following two conditions: **(1)** it resides in revision *r*, and **(2)** it was not changed in the past (i.e., it was not changed in any of the commit operations that occurred prior to the occurrence of the bug-fix commit operation BFC_r).
- $\mathbf{NCCB_r}$: This is the number of clone fragments, each of which fulfills the following three conditions: **(1)** it resides in revision *r*, **(2)** it was changed in the past, and **(3)** it was changed in the bug-fix commit BFC_r which was applied on revision *r*.
- $\mathbf{NCNB_r}$: This is the number of clone fragments, each of which satisfies the following three conditions: **(1)** it resides in revision *r*, **(2)** it was not changed in the past, and **(3)** it was changed in the bug-fix commit BFC_r which was applied on revision *r*.

If we divide $\mathbf{NCCB_r}$ by $\mathbf{NCC_r}$, then we get the probability that a clone fragment that was changed previously will be changed in the bug-fix commit BFC_r (i.e., will experience a bug-fix change). $\mathbf{NCC_r}$ is the number of clone fragments that were changed previously, and $\mathbf{NCCB_r}$ is the number of previously changed clone fragments that were also changed in the bug-fix commit BFC_r. In the same way, if we divide $\mathbf{NCNB_r}$ by $\mathbf{NCN_r}$, then we get the probability that a clone fragment that was not changed previously will be changed in the bug-fix commit BFC_r (i.e., will experience a bug-fix change). We determine these two probabilities in percentage considering all the revisions in the set *BR* according to the following two equations.

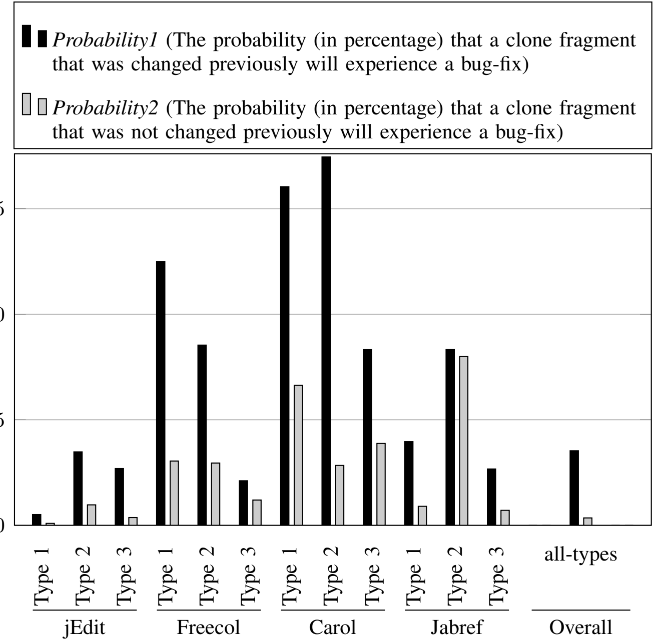

Fig. 2. Comparison of the probabilities that a clone fragment that was changed or not changed previously will experience a bug-fix change.

$$Probability1 = \frac{100 \times \sum_{r \; \epsilon \; BR} NCCB_r}{\sum_{r \; \epsilon \; BR} NCC_r} \quad (1)$$

$$Probability2 = \frac{100 \times \sum_{r \; \epsilon \; BR} NCNB_r}{\sum_{r \; \epsilon \; BR} NCN_r} \quad (2)$$

The sums for $\mathbf{NCC_r}$, $\mathbf{NCN_r}$, $\mathbf{NCCB_r}$, and $\mathbf{NCNB_r}$ are shown in Table VI in the columns NCC, NCN, NCCB, and NCNB respectively. *Probability1* is the probability that a clone fragment that was changed in the past will experience a bug-fix change. *Probability2* is the probability that a clone fragment that not was changed in the past will experience a bug-fix change. We show these probabilities in the graph of Fig. 2. From the graph it is clear that *Probability1* is always higher than *Probability2*. We wanted to determine whether *Probability1* is significantly higher than *Probability2*. Our test details are given below.

Statistical Significance Tests. From Fig. 2 we see that for each clone-type of each of the subject systems we get two probabilities: *Probability1* and *Probability2*. If we consider all the clone-types (i.e., three clone-types) of all the subject systems we get 12 cases (4 systems × 3 clone-types) in total. We perform Wilcoxon Signed-Rank Test [29] to determine whether the 12 values of *Probability1* (i.e., from the 12 cases) are significantly different than the 12 values of *Probability2*. Wilcoxon Signed-Rank Test is non-parametric, and thus it does not require the samples to be normally distributed [30]. We apply this test because the data in our two samples are paired (i.e., from each of the 12 cases we obtain two probability values: *Probability1* and *Probability2*). In our research, we perform this test considering a significance level of 5%. According to our test, the values of *Probability1* are significantly different than the values of *Probability2* with a *p-value* of 0.002 for the 2-tailed test case and 0.001 for the 1-tailed test case. As

TABLE VI. CODE CLONE MEASURES FOR INVESTIGATION REGARDING RQ 1

	Type 1				Type 2				Type 3			
Subject System	**NCC**	**NCN**	**NCCB**	**NCNB**	**NCC**	**NCN**	**NCCB**	**NCNB**	**NCC**	**NCN**	**NCCB**	**NCNB**
jEdit	592	75497	3	69	230	1652	8	16	4802	33164	129	121
Freecol	24	394	3	12	82	305	7	9	2039	7259	43	87
Carol	106	226	17	15	149	282	26	8	1033	1702	86	66
Jabref	101	444	4	4	72	100	6	8	712	2395	19	17

NCC = From all revisions in the set *BR* (defined in Table V), **NCC** is the total number of clone fragments that were changed in the past.

NCN = From all revisions in the set *BR*, **NCN** is the total number of clone fragments that were not changed in the past.

NCCB = From all revisions in the set *BR*, **NCCB** is the total number of clone fragments that were changed in the bug-fix commits, and also, were changed in the past.

NCNB = From all revisions in the set *BR*, **NCNB** is the total number of clone fragments that were changed in the bug-fix commits, but were not changed previously.

Probability1 is always greater that *Probability2* (Fig. 2) we can say that *Probability1* is significantly higher than *Probability2*. In other words, *the probability that a clone fragment that was changed previously will experience a bug-fix change is significantly higher than the probability that a clone fragment that was not changed previously will experience a bug-fix.*

Answer to RQ 1: The clone fragments that were changed in the past have a significantly higher possibility of experiencing bug-fix changes compared to the clone fragments that were not changed in the past.

Our finding implies that when taking clone management decisions we should primarily focus on the code clones that were changed in the past. From our manual observation on the bug-fix changes that occurred to the code clones of Freecol and Jabref we experience that bug-fix changes are often related to method calls in the clone fragments. The bug-fix changes in Fig. 1 are also related to method calls. In future we will manually analyze and classify the bugs and bug-fix changes in different types of code clones.

Although our answer to RQ 1 implies that code clones that were changed in the past have high possibilities of experiencing bug-fixes, it is still unknown whether change frequency of code clones is related to their bug-proneness. The common intuition is that frequent changes in a program entity are likely to introduce bugs in that entity. However, there is no empirical evaluation regarding this. We investigate this in our second research question (RQ 2).

RQ 2: Do clone fragments that were changed more frequently in the past have higher possibilities of experiencing bug-fixes?

Motivation. From the answer to RQ 1 we understand that the clone fragments that were changed in the past have higher possibilities of experiencing bug-fix changes compared to the clone fragments that were not changed in the past. However, we still do not know whether high change frequency is the primary reason behind clone bug-proneness. We investigate this in RQ 2. We perform our investigation considering the clone fragments that were changed in the past. We investigate whether the clone fragments that were more frequently changed in the past have higher probabilities of experiencing bug-fix changes. Intuitively, program entities with higher change frequency have higher possibilities of being related to bugs. We evaluate this common intuition for code clones. Our method of investigation has been described below.

Methodology. For each clone-type of each of the subject systems we determine the set *BR* (defined in Table V). We consider a particular revision *r* in this set. One or more clone fragments in revision *r* were changed by the bug-fix commit which was applied on revision *r*. Considering the clone fragments in revision *r*, we determine the following two sets:

- $\mathbf{SCCB_r}$**:** This set contains clone fragments, each of which satisfies the following three conditions: **(1)** it resides in revision *r*, **(2)** it was changed in the past (i.e., it was changed in any of the commits that occurred before the occurrence of the bug-fix commit on revision *r*), and **(3)** it was changed in the bug-fix commit which was applied on revision *r*.
- $\mathbf{SCCN_r}$**:** This set contains clone fragments, each of which fulfills the following three conditions: **(1)** it resides in revision *r*, **(2)** it was changed in the past, and **(3)** it was not changed in the bug-fix commit which was applied on revision *r*.

For each clone fragment in these two sets we determine its change frequency in the past. Change frequency is the number of times (i.e., the number of commits) a particular clone fragment was changed in the past.

For each revision *r* in the set *BR* (defined in Table V), we determine the above two sets, $\mathbf{SCCB_r}$ and $\mathbf{SCCN_r}$, and the change frequencies of the clone fragments in these sets. Considering all the clone fragments in all the $\mathbf{SCCB_r}$ sets we determine the average change frequency per clone fragment. We call this average frequency **ACF-SCCB**. We understand that **ACF-SCCB** is the average change frequency of the clone fragments that were changed in the past, and also, were changed in the bug-fix commits. In the same way we determine **ACF-SCCN** which is the average change frequency of the clone fragments that were changed in the past, but were not changed in the bug-fix commits. In Fig. 3 we show these two averages for each clone-type of each of the subject systems.

From Fig. 3 we understand that the change frequency of a clone fragment in the past does not necessarily influence its possibility of experiencing bug-fixes. We see that for six cases (for example, Type 2 case of Freecol) the clone fragments that were changed more frequently in the past did not experience bug-fix changes. The opposite is true for five cases (for example, Type 3 case of jEdit). For Type 1 case of Freecol, the two average frequencies are the same.

Statistical Significance Tests. We also wanted to determine whether the two frequencies **ACF-SCCB** and **ACF-SCCN** are significantly different. We performed the Wilcoxon Signed-Rank Test [29], [30] in the same way as we did in RQ 1. According to our test result, the difference between these two frequencies is not statistically significant.

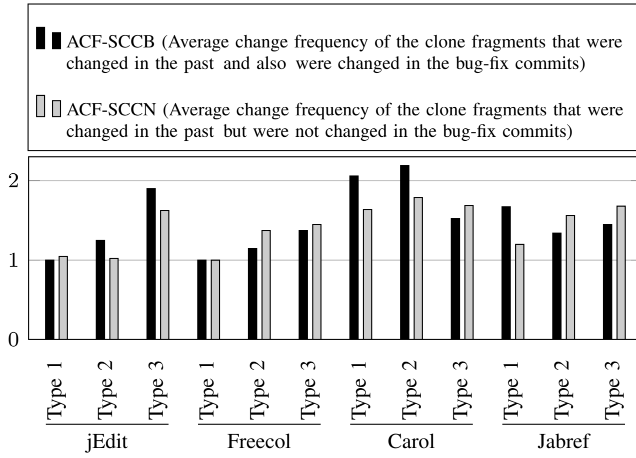

Fig. 3. Comparing the change frequencies of the clone fragments that experienced or did not experience bug-fix changes.

> **Answer to RQ 2.** According to our experimental results, change frequencies of code clones do not necessarily indicate their possibilities of experiencing bug-fixes.

From our answer to RQ 1 we realized that the clone fragments that were changed in the past are highly likely to experience bug-fixes in future. However, the change frequencies of such clone fragments cannot be used for prioritizing or ranking them according to their likeliness of experiencing bug-fixes. Such a finding invalidates the common belief regarding the relatedness of high change frequency with bug-proneness. In RQ 3 we investigate whether change recency of clones can indicate their bug-proneness during system evolution.

RQ 3: Do the clone fragments that were changed more recently in the past have higher possibilities of experiencing bug-fixes?

Motivation. From the answer to RQ 2 we understand that change frequency of a clone fragment cannot be a good indicator of the possibility that it will experience a bug-fix. We wanted to investigate whether change recency of the clone fragments can help us understand further details regarding the possibilities of bug-fixes in code clones.

Methodology. We proceed in the same way as we did in RQ 2. For each clone-type of each of the subject systems we determine the set *BR* (defined in Table V). For each revision *r* in *BR*, we determine the two sets: $\mathbf{SCCB_r}$ (i.e., the set of all those clone fragments in revision *r* that were changed in the past and were also changed in the bug-fix commit which was applied on revision *r*), and $\mathbf{SCCN_r}$ (i.e., the set of all those clone fragments in revision *r* that were changed in the past but were not changed in the bug-fix commit applied on revision *r*). For each clone fragment in these two sets we determine how recently it was changed in the past. For this purpose we determine the oldness of change for each clone fragment.

Let us assume that the bug-fix commit that was applied on revision *r* is BFC_r. A clone fragment CF in revision *r* belongs to the set $\mathbf{SCCB_r}$ or $\mathbf{SCCN_r}$. The clone fragment CF might change a number of times in the past. The last commit where it was changed before the occurrence of BFC_r is $C_{previous}$. This last commit was applied on revision $r_{previous}$. The oldness of change (**OC**) of the clone fragment CF with respect to *r* is calculated using the following equation.

$$OC_{CF} = r - r_{previous} \tag{3}$$

From the equation we understand that the smaller the oldness of change (**OC**) for a clone fragment with respect to *r*, the more lately it was changed in the past. Considering all the clone fragments in all the $\mathbf{SCCB_r}$ sets obtained from all the revisions in the set *BR* (i.e., from all the revisions that experienced bug-fix commits affecting code clones), we determine the average oldness of change per clone fragment. We call this average **AOC-SCCB**. We understand that **AOC-SCCB** is the average oldness of change of the clone fragments that were changed in the past and also were changed in the bug-fix commits. In the same way we also determine **AOC-SCCN**. In Fig. 4 we plot these two averages for each clone-type of each of the subject systems.

From Fig. 4 we see that for most of the cases (10 cases out of 12, excluding Type 2 case of Freecol, and Type 1 case of Jabref), **AOC-SCCB** is smaller than **AOC-SCCN**. In other words, the clone fragments that experienced bug-fixes were changed more recently in the past compared to others that did not experience bug-fixes. Thus, more recently changed clone fragments have higher possibilities of experiencing bug-fixes.

Statistical Significance Tests. From Fig. 4 we find that **AOC-SCCB** is generally smaller than **AOC-SCCN**. We also wanted to see whether **AOC-SCCB** is significantly smaller than **AOC-SCCN**. The values of **AOC-SCCB** and **AOC-SCCN** are paired as demonstrated in Fig. 4. For this reason we perform the Wilcoxon Signed-Rank Test [29] as we did in RQ 1. Our test result is significant for one-tailed test case with a *p-value* of 0.041 (< 0.05). Such a result implies that **AOC-SCCB** is significantly smaller than **AOC-SCCN**.

> **Answer to RQ 3.** From our investigation and analysis we can say that the clone fragments that were more recently changed in the past have higher possibilities of experiencing bug-fixes in future evolution. Our finding is significant according to our statistical significance test.

From our answer to RQ 3 we realize that change recency of the clone fragments that were changed before can be used for prioritizing them according to their likeliness of experiencing bug-fix changes in future. In RQ 4 we investigate the possibilities of occurring bug-fixes in the clone fragments that were not changed in the past.

RQ 4: Which of the clone fragments that were not changed in the past have higher possibilities of experiencing bug-fixes?

From our investigations regarding RQ 1 we realize that bug-fix changes can also occur to the clone fragments that were not changed in the past. In RQ 4 we investigate which of such clone fragments (i.e., the previously non-changed ones) have higher possibilities of experiencing bug-fix changes. In particular, we analyze whether the more recently created clone fragments have higher possibilities of experiencing bug-fixes. We perform our investigation in the following way.

Methodology. We first identify the set *BR* (defined in Table V) considering each clone-type of each of the subject systems. Let us consider a particular revision *r* in this set. One or more

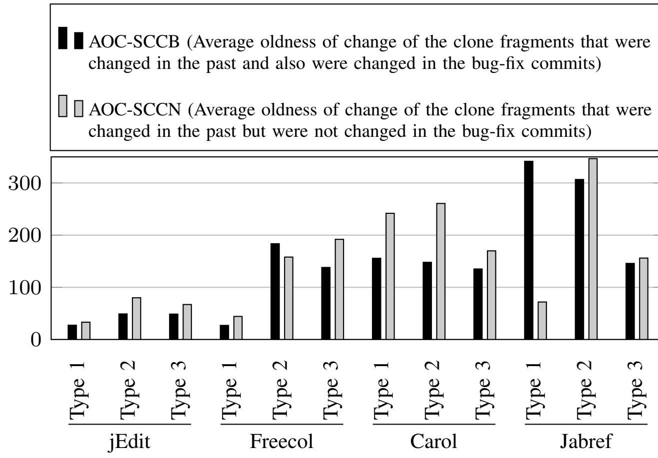

Fig. 4. Comparing the change recency of the clone fragments that experienced or did not experience bug-fix changes.

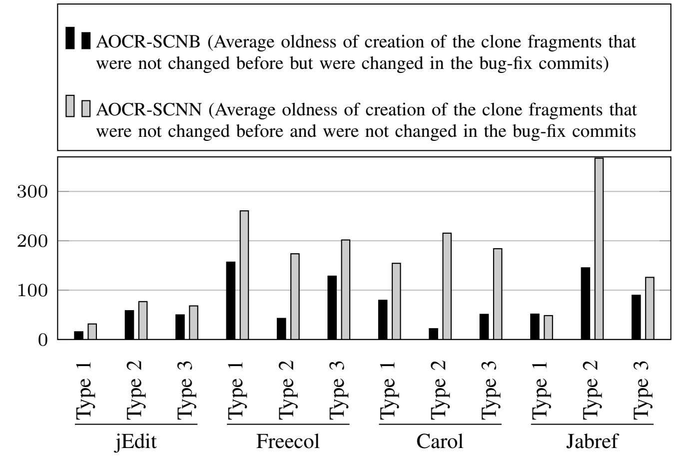

Fig. 5. Comparison of the average oldness of creation of the clone fragments that were not changed in the past.

clone fragments in revision *r* were changed by the bug-fix commit which was applied on revision *r*. Considering the clone fragments in revision *r*, we determine the following two sets:

- **$\mathbf{SCNB_r}$:** This set contains clone fragments, each of which satisfies the following three conditions: **(1)** it resides in revision *r*, **(2)** it was not changed in the past (i.e., it was not changed in any of the commits that occurred before the occurrence of the bug-fix commit on revision *r*), and **(3)** it was changed in the bug-fix commit which was applied on revision *r*.
- **$\mathbf{SCNN_r}$:** This set contains clone fragments, each of which fulfills the following three conditions: **(1)** it resides in revision *r*, **(2)** it was not changed in the past, and **(3)** it was not changed in the bug-fix commit which was applied on revision *r*.

We understand that the above two sets are disjoint, and from their union we can get all those clone fragments in revision *r* that were not changed in the past. For each clone fragment in each of these two sets we determine the revision in which it was created (i.e., added). Let us consider a clone fragment CF in the set $\mathbf{SCNB_r}$ or $\mathbf{SCNN_r}$. This clone fragment was created in revision $r_{created}$ where $r_{created} < r$. Using the following equation we determine how recently the fragment CF was created with respect to *r*.

$$OCR_{CF} = r - r_{created} \tag{4}$$

Here, OCR_{CF} is the ***oldness of creation*** of the clone fragment CF with respect to the revision *r*. For each of the clone fragments in the two sets, $\mathbf{SCNB_r}$ and $\mathbf{SCNN_r}$, we determine its ***oldness of creation*** with respect to *r*. From Eq. 4 we see that the ***oldness of creation*** of a particular clone fragment with respect to a particular revision *r* depends on $r_{created}$ where the clone fragment was created. The smaller the value of $r_{created}$ is (i.e., the older the $r_{created}$ is), the higher is the oldness of creation of the clone fragment (i.e., the older is the clone fragment).

For each revision *r* in the set *BR* (Table V), we determine the sets $\mathbf{SCNB_r}$ and $\mathbf{SCNN_r}$. For each clone fragment in the sets $\mathbf{SCNB_r}$ and $\mathbf{SCNN_r}$ we determine its ***oldness of creation***. Considering all the clone fragments in all the $\mathbf{SCNB_r}$ sets we determine the average oldness of creation per clone fragment. We call this average **AOCR-SCNB**. Thus, **AOCR-SCNB** is the average oldness of creation of the clone fragments that were changed in the bug-fix commits but were not changed previously. In the same way we also determine **AOCR-SCNN** considering all $\mathbf{SCNN_r}$ sets. Fig. 5 shows these two averages for each clone-type of each of the systems.

Fig. 5 shows that **AOCR-SCNB** (i.e., the average oldness of creation of the clone fragments in the $\mathbf{SCNB_r}$ sets) is always smaller than corresponding **AOCR-SCNN** (i.e., the average oldness of creation of the clone fragments in the $\mathbf{SCNN_r}$ sets) except for Type 1 case of Jabref. In other words, the clone fragments in the set $\mathbf{SCNB_r}$ are generally more recently created compared to the clone fragments in the corresponding $\mathbf{SCNN_r}$ set. Thus, more recently created clone fragments are more likely to experience bug-fix changes.

Statistical Significance Tests. In Fig. 5 we see that there are 12 cases (4 subject systems $\times$ 3 clone-types) in total, and for each case we show two values: AOCR-SCNB and AOCR-SCNN. We wanted to determine whether the 12 values of AOCR-SCNB are significantly different than the 12 values of AOCR-SCNN. As the values of AOCR-SCNB and AOCR-SCNN are paired, we again perform the Wilcoxon Signed-Rank Test [29], [30] considering a significance level of 5% as we did in RQ 1. According to our tests, the 12 values of AOCR-SCNB are significantly different than the 12 values of AOCR-SCNN with a *p-value* of 0.0028 for two-tailed test case and 0.0014 for the one-tailed test case. Thus, the values of AOCR-SCNB are significantly smaller than the values of AOCR-SCNN. In other words, regarding the clone fragments that were not changed before we see that the clone fragments that experienced bug-fixes were created significantly more recently compared to the other clone fragments that did not experience bug-fix.

> **Answer to RQ 4.** According to our experiment regarding the clone fragments that were not changed before the bug-fix commits we find that the clone fragments that are more recently created have higher possibilities of experiencing bug-fix changes.

From our answer to RQ 4 we realize that recency of creation of the clone fragments that were not changed before

can be used for prioritizing them according to their likeliness of experiencing bugs in future evolution.

V. Discussion regarding our findings

We have answered four research questions through our investigation. The answers imply whether and how change-proneness of code clones can be related to their bug-proneness and how code clones can be prioritized for managing considering their bug-proneness.

Significance of our findings. While our finding regarding RQ 1 complies with the intuition that code clones that were changed previously exhibit a higher bug-proneness than the code clones that were not changed in the past, our findings from RQ 2 and RQ 3 are interesting. Our finding from RQ 2 goes against the common belief regarding the relatedness between high change frequency and bug-proneness. We realize that change frequency of clone fragments cannot imply their tendencies of containing bugs (i.e., experiencing bug-fixes in future). Our investigation in RQ 3 discovers the fact that bug-proneness of code clones is primarily related to their change recency. More recently changed code clones have a higher possibility of containing bugs. Our finding from RQ 4 regarding the previously unchanged code clones complies with the finding from RQ 3, and reveals that more recently created clones have higher possibilities of containing bugs. We finally discover that bug-proneness of code clones primarily depend on the recency of their creation (for those that were not modified previously) or modification (for those that were modified in the past).

Implication of our findings. Our findings are important for prioritizing code clones for refactoring and tracking from the perspective of clone bug-proneness. We suggest that code clones should be prioritized according to their recency of creation or modification so that the most recent one gets the highest priority. Our implemented prototype tool can rank code clones according to their bug-proneness at any point of evolution of a software system. Thus, it can help us in better management of code clones.

A number of factors such as clone size, location, and complexity can have important impacts on the bug-proneness of code clones. However, in our research we wanted to perform an in-depth investigation regarding how change-proneness of code clones affects their bug-proneness. We plan to investigate the effects of such factors in future.

VI. Related Work

We discuss the existing studies related to our research from the following perspectives.

A. Bug detection in code clones

Li and Ernst [25] developed a tool called CBCD (Cloned Buggy Code Detector) that detects code clones of a given piece of buggy code by applying graph isomorphism over the PDG (program dependency graph) of the source code of a software system. CBCD was evaluated on three software systems, and was found to be effective in detecting clones of buggy code fragments. Li et al. [26] developed CP-Miner with the capability of detecting bugs related to inconsistencies introduced during copy/paste activities. Steidl and Göde [47] applied a machine learning technique for identifying instances of incompletely fixed bugs in near-miss code clones. Göde and Koschke [10] investigated changes that occurred to the code clones of three mature software systems and found that 14.8% of these changes were unintentionally inconsistent. According to a study of Moden et al. [36] on an industrial software system, software modules containing code clones appear to be less maintainable than the modules having no code clones. Chatterji et al. [5] investigated bug localization in code clones through a user study on a number of programmers. Jiang et al. [16] developed an algorithm for automatically detecting context based inconsistencies related to clones. They applied their algorithm on two software systems and detected previously unknown bugs. Inoue et al. [14] investigated a mobile software system, and detected bugs related to inconsistent changes to the identifiers in code clones by developing a tool named 'CloneInspector'. Xie et al. [54] analyzed clone mutation and migration considering Type 3 clones of three open-source software systems, and observed that mutation of code clones to Type 2 and Type 3 is risky.

We see that none of the studies discussed above can help us understand which of the clone fragments residing in a software system's code-base have high possibilities of experiencing bug-fixes in future. Thus, these studies cannot help us prioritize code clones on the basis of their bug-proneness.

B. Scheduling for clone refactoring

Zibran and Roy [55] proposed a conflict aware optimal scheduling algorithm for clone refactoring on the basis of constraint programming. They showed that their scheduling algorithm is superior to the other algorithms those are based on genetic algorithm approaches, greedy approaches and linear programming. Bouktif et al. [4] considered the clone refactoring problem as a constrained knapsack problem where the knapsack consists of all the clones to be refactored. They found an optimal schedule for refactoring the clones in the knapsack by applying a genetic algorithm. Lee et al. [24] also proposed a genetic algorithm based technique for scheduling clone refactoring activities.

We see that while these existing clone scheduling studies [4], [55], [24] can find an optimal schedule for refactoring a given set of clone fragments, these studies cannot determine which of the huge number of code clones in a code-base should be given a higher priority for refactoring or tracking. We investigate clone bug-proneness for identifying which clone fragments are likely to experience bug-fixes in future. We suggest prioritizing such code clones for management such as refactoring and tracking. Thus, our study can complement the clone scheduling studies by identifying which code clones should be given high priorities for scheduling.

C. Analyzing bug-proneness of code clones

Rahman et al. [38] compared bug-proneness of clone and non-clone code residing in four subject systems using the clone detector DECKARD [17], and found non-clone code to be more bug-prone than clone code. However, their investigation was based on monthly snapshots (revisions) of their subject 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 from the beginning one. Thus, we do not miss any bug-fix commits. Also, our goal in this study is different. We investigate how bug-proneness of code clones is affected by their change-

proneness and how to prioritize clones considering their bug-proneness, whereas they compared bug-proneness of clone and non-clone code in their study.

Selim et al. [46] investigated defect-proneness of code clones using Cox hazard models and found that their defect-proneness is system dependent. Their investigation was based only on Type 1 and Type 2 method clones. However, we consider all major types (Type 1, 2, and 3) of block clones in our study. They studied only two subject systems, whereas we conduct our investigation on four systems. Also, we investigate clone prioritization considering clone bug-proneness. Selim et al. [46] did not perform this investigation.

Existing studies have also investigated whether clone bug-proneness is related to late propagation. Aversano et al.[1] reported that late propagation in code clones can be directly related to bugs on the basis of their investigation on two subject systems. Barbour et al.[2] mined and analyzed eight late propagation patterns in Type 1 and Type 2 clones of three subject systems and identified the highly bug-prone patterns.

Two existing studies [15], [34] compared bug-replication and bug-densities in three types of code clones: Type 1, Type 2, and Type 3. However, these studies did not investigate how change-proneness of code clones can affect their possibilities of containing bugs. Thus, our research is significantly different from these existing studies [15], [34].

The tool called SPCP-Miner [32] introduced by Mondal et al. can identify the important code clones for refactoring and tracking on the basis of their evolutionary coupling. However, it does not consider clone bug-proneness for ranking code clones. We believe that clone bug-proneness should also be considered for prioritizing code clones for management. Our study presented in this paper considers bug-proneness of code clones. More specifically, we investigate whether and how bug-proneness of code clones can be related with their change-proneness. None of the existing studies on clone bug-proneness investigated this matter.

Hasan and Holt [12] investigated predicting bugs in open source systems and found that subsystems that were more frequently modified were more bug-prone compared to the more recently modified ones. In our investigation we analyze bug-proneness considering code fragment level granularity which is a finer granularity compared to subsystem level granularity in Hasan and Holt's study. More specifically, we analyze the bug-proneness of code clones in our candidate systems. We found that more recently changed code clones have higher tendencies of containing bugs.

D'Ambros et al. [8] compared a number of bug-prediction approaches and found that their proposed approaches: WCHU and LDHH show promising prediction efficiencies. However, they investigated bug-proneness considering the class level granularity. We analyze bug-proneness considering code fragment level granularity which is a finer granularity compared to class level granularity. Moreover, we investigate the bug-proneness of code clones disregarding the non-clone code fragments. We find that change recency of code clones can be a good predictor of bugs in them.

We see that existing studies have detected and investigated bug-proneness of code clones in different ways. However, none of these studies focus on whether and how change-proneness of code clones can be related to their bug-proneness. Thus, these existing studies cannot identify which of the clone fragments residing in the code-base of a software system are more likely to contain bugs. Focusing on this we perform an in-depth investigation on the bug-proneness of code clones in this research. Our experimental results are promising and provide useful implications for ranking code clones according to their likeliness of containing bugs.

VII. Threats to Validity

We conduct our investigations by detecting code clones using the NiCad clone detector [6]. While all clone detectors suffer from the *confounding configuration choice problem* [53] and might provide different clone detection results for different settings of the tools, the settings that we have used for NiCad are considered standard [42]. According to a recent study [48] NiCad can detect code clones with high precision and recall with these settings. Thus, we believe that our findings in this experiment are important for prioritizing code clones considering their bug-proneness.

We detect bug-fix commits in our experiment. The technique that we used for detecting such commits is similar to that followed by Barbour et al. [3]. Such a technique proposed by Mocus and Votta [31] can sometimes detect 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.

VIII. Conclusion

In this research we investigate the clone evolution history from thousands of revisions of four subject systems written in Java, and analyze how change-proneness of code clones can be related with their possibilities of containing bugs. We answer four important research questions by analyzing our experimental results and find that code clones that were changed in the past have a significantly higher possibility of containing bugs (i.e., experiencing bug-fixes) compared to the code clones that were not changed previously. While such a finding complies with the common intuition, our research discovers that change frequencies of clone fragments do not indicate their possibilities of containing bugs. This finding invalidates the common belief regarding the relatedness of bug-proneness of with high change frequency. We discover that bug-proneness of code clones is primarily related with their change recency. More recently changed clones are more likely to experience bug-fixes. Regarding the clone fragments that were not changed previously we find that the more recently created ones have higher tendencies of experiencing bug-fixes. Our research establishes the fact that bug-proneness of code clones is mainly influenced by their recency of creation (for clones that were not previously modified) or modification (for clones that were previously modified). We suggest that code clones should be prioritized for management (refactoring and tracking) considering their recency of creation and modification from the perspective of their bug-proneness. In the future we plan to investigate the effects of a number of factors such as clone size, location, and complexity on the bug-proneness of code clones.

Acknowledgments: This work is supported in part by the Natural Science and Engineering Research Council of Canada (NSERC).

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] S. Bouktif, G. Antoniol, E. Merlo, M. Neteler, "A Novel Approach to Optimize Clone Refactoring Activity", Proc. *GECCO*, 2006, pp. 1885 – 1892.

[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] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches", Proc. *MSR*, 2010, pp. 31 – 41.

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

[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] A. E. Hassan, R. C. Holt, "The top ten list: Dynamic fault prediction", Proc. *ICSM*, 2005, pp. 263 – 272.

[13] 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.

[14] 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.

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

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

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

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

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

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

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

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

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

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

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

[26] 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.

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

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

[29] Wilcoxon Signed-Rank Test On-line: http://www.socscistatistics.com/tests/signedranks/Default2.aspx

[30] Wilcoxon signed-rank test: https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

[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, pp. 484 - 488.

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

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

[35] M. Mondal, C. K. Roy, and K. A. Schneider. "A comparative study on the intensity and harmfulness of late propagation in near-miss code clones", *Software Quality Journal*, 24(4): 883 – 915.

[36] A. Monden, D. Nakae, T. Kamiya, S. Sato, K. Matsumoto, "Software Quality Analysis by Code Clones in Industrial Legacy Software", Proc. *METRICS*, 2002, pp. 87 – 94.

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

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

[39] D. C. Rajapakse and S. Jarzabek, "Using Server Pages to Unify Clones in Web Applications: A Trade-off Analysis", Proc. *ICSE*, 2007, pp. 116 – 126.

[40] 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.

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

[42] 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.

[43] 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.

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

[45] R. K. Saha, C. K. Roy, K. A. Schneider, "An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies", Proc. *ICSM*, 2011, pp.293 – 302.

[46] 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.

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

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

[49] 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.

[50] N. Tsantalis, D. Mazinanian, G. P. Krishnan, "Assessing the Refactorability of Software Clones", *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 2015, 41(11): 1055 – 1090.

[51] R. D. Venkatasubramanyam, S. Gupta, H. K. Singh, "Prioritizing code clone detection results for clone management", Proc. *IWSC*, 2013, pp. 30 - 36.

[52] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, H. Mei, "Predicting Consistency-Maintenance Requirement of Code Clones at Copy-and-Paste Time", Proc. *IEEE Transactions on Software Engineering*, 2014, 40(8): 773 – 794.

[53] 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.

[54] 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.

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