Bug Replication in Code Clones: An Empirical Study

Judith F. Islam Manishankar Mondal Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Canada
{judith.islam, mshankar.mondal, chanchal.roy}@usask.ca

***Abstract*—Code clones are exactly or nearly similar code fragments in the code-base of a software system. Existing studies show that clones are directly related to bugs and inconsistencies in the code-base. Code cloning (making code clones) is suspected to be responsible for replicating bugs in the code fragments. However, there is no study on the possibilities of bug-replication through cloning process. Such a study can help us discover ways of minimizing bug-replication. Focusing on this we conduct an empirical study on the intensities of bug-replication in the code clones of the major clone-types: Type 1, Type 2, and Type 3.**

According to our investigation on thousands of revisions of six diverse subject systems written in two different programming languages, C and Java, a considerable proportion (i.e., up to 10%) of the code clones can contain replicated bugs. Both Type 2 and Type 3 clones have higher tendencies of having replicated bugs compared to Type 1 clones. Thus, Type 2 and Type 3 clones are more important from clone management perspectives. The extent of bug-replication in the buggy clone classes is generally very high (i.e., 100% in most of the cases). We also find that overall 55% of all the bugs experienced by the code clones can be replicated bugs. Our study shows that replication of bugs through cloning is a common phenomenon. Clone fragments having *method-calls* and *if-conditions* should be considered for refactoring with high priorities, because such clone fragments have high possibilities of containing replicated bugs. We believe that our findings are important for better maintenance of software systems, in particular, systems with code clones.

I. Introduction

If two or more code fragments in a software system's code-base are exactly or nearly similar to one another we call them code clones [44], [45]. A group of similar code fragments forms a clone class. Code clones are mainly created because of the frequent copy/paste activities of the programmers during software development and maintenance. Whatever may be the reasons behind cloning, code clones are of great importance from the perspectives of software maintenance and evolution [44].

A great many studies [1], [2], [10]–[12], [14], [16], [18], [20]–[23], [25], [26], [36], [37], [51], [53] have been conducted on discovering the impact of cloning on software maintenance. While a number of studies [1], [11], [12], [18], [20]–[22] have revealed some positive sides of code cloning, there is strong empirical evidence [2], [10], [14], [16], [23], [25], [26], [36], [37], [51] of negative impacts of code clones too. These negative impacts include higher instability [36], late propagation [2], and unintentional inconsistencies [10]. Existing studies [2], [39] show that code clones are related to bugs in the code-base. Also, it is suspected that cloning is responsible for replicating bugs [44]. If a particular code fragment contains a bug and a programmer copies that code fragment to several other places in the code-base without the knowledge of the existing bug, the bug in the original fragment gets replicated. Fixing of such replicated bugs may require increased maintenance effort and cost for software systems. However, although cloning is suspected to be responsible for replicating bugs, there is no study on the possibilities of bug-replication through cloning. Such a study can provide us helpful insights for minimizing bug-replication as well as for prioritizing code clones for refactoring or tracking. Focusing on this we conduct an in-depth empirical study regarding bug-replication in the code clones of the major clone-types: Type 1, Type 2, Type 3.

We conduct our empirical study on thousands of revisions of six diverse subject systems written in two different programming languages (Java and C). We detect code clones from each of the revisions of a subject system using the NiCad [6] clone detector, analyze the evolution history of these code clones, and investigate whether and to what extent they contain replicated bugs. We answer four important research questions (Table I) regarding the intensity and cause of bug-replication through our investigation. According to our investigation involving rigorous manual analysis we can state that:

(1) A considerable percentage of the code clones can be related to bug-replication. According to our observation up to 10% of the code clones in a software system can contain replicated bugs.

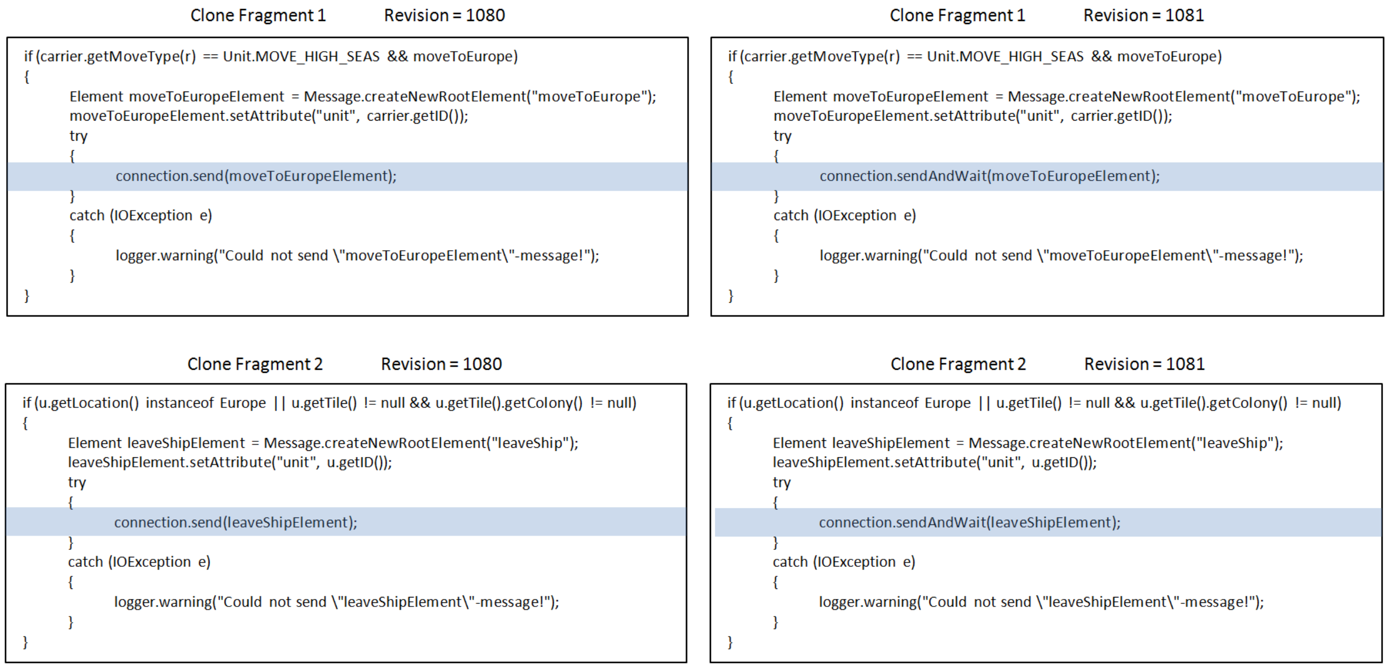
(2) Both Type 2 and Type 3 clones have higher possibilities of containing replicated bugs compared to Type 1 clones. Thus, Type 2 and Type 3 clones should be given higher priorities for management.

(3) A considerable proportion (around 55%) of the bugs occurred in code clones can be replicated bugs.

(4) Most of the replicated bugs are related to the *method-calls* and *if-conditions* residing in the clone fragments. Thus, clone fragments containing *method-calls* and/or *if-conditions* should be considered for refactoring or tracking with high priorities.

Our findings imply that bug-replication tendencies of code clones should be taken in proper consideration when making clone management decisions. The findings from our study are important for better management of code clones as well as for better maintenance of software systems.

The rest of the paper is organized as follows. Section II contains the terminology, Section III discusses the experimental steps, Section IV describes the process of identifying the


Clone Fragment 1 Revision = 1080
Clone Fragment 1 Revision = 1081
Clone Fragment 2 Revision = 1080
Clone Fragment 2 Revision = 1081

```
if (carrier.getMoveType(r) == Unit.MOVE_HIGH_SEAS && moveToEurope)
{
        Element moveToEuropeElement = Message.createNewRootElement("moveToEurope");
        moveToEuropeElement.setAttribute("unit", carrier.getID());
        try
        {
                connection.send(moveToEuropeElement);
        }
        catch (IOException e)
        {
                logger.warning("Could not send \"moveToEuropeElement\"-message!");
        }
}
```

```
if (carrier.getMoveType(r) == Unit.MOVE_HIGH_SEAS && moveToEurope)
{
        Element moveToEuropeElement = Message.createNewRootElement("moveToEurope");
        moveToEuropeElement.setAttribute("unit", carrier.getID());
        try
        {
                connection.sendAndWait(moveToEuropeElement);
        }
        catch (IOException e)
        {
                logger.warning("Could not send \"moveToEuropeElement\"-message!");
        }
}
```

```
if (u.getLocation() instanceof Europe || u.getTile() != null && u.getTile().getColony() != null)
{
        Element leaveShipElement = Message.createNewRootElement("leaveShip");
        leaveShipElement.setAttribute("unit", u.getID());
        try
        {
                connection.send(leaveShipElement);
        }
        catch (IOException e)
        {
                logger.warning("Could not send \"leaveShipElement\"-message!");
        }
}
```

```
if (u.getLocation() instanceof Europe || u.getTile() != null && u.getTile().getColony() != null)
{
        Element leaveShipElement = Message.createNewRootElement("leaveShip");
        leaveShipElement.setAttribute("unit", u.getID());
        try
        {
                connection.sendAndWait(leaveShipElement);
        }
        catch (IOException e)
        {
                logger.warning("Could not send \"leaveShipElement\"-message!");
        }
}
```

Fig. 1. This figure demonstrates an example of Similarity Preserving Co-change (SPCO) of two Type 3 clone fragments (Clone Fragment 1, and Clone Fragment 2) in the commit operation applied on revision 1080. We take this example from our subject system Freecol. The snapshots of each clone fragment in the two revisions 1080 and 1081 are shown in the figure, and the changes are highlighted. We can easily understand that in case of each clone fragment, a method named 'send' was replaced by a method named 'sendAndWait'. We see that the clone fragments were changed in the same way. The comment from the programmer regarding this change says that *Fixed a potential synchronization bug. "sendAndWait" should be used instead of "send" in order to ensure that the server has completed handling the request before we make any modifications to the model. This is a necessary because the server and the AI share the same model.* From the changes to the clone fragments and programmer comments we realize that this SPCO is an example of fixing a replicated bug.

TABLE I. RESEARCH QUESTIONS

SL	Research Question
RQ 1	What percentage of the clone fragments in different clone-types takes part in bug-replication?
RQ 2	What is the extent of bug-replication in the buggy clone classes of different types of clones?
RQ 3	What percentage of the bugs that were experienced by the code clones of different clone-types are replicated bugs?
RQ 4	Which types of statements are highly related to bug-replication?

replicated bugs, Section V answers the research questions by presenting and analyzing the experimental results, Section VI discusses the related work, Section VII mentions the possible threats to validity, and Section VIII concludes the paper by mentioning possible future work.

II. TERMINOLOGY

A. Types of Clones

We conduct our experiment considering both exact (Type 1) and near-miss clones (Type 2 and Type 3 clones) [44], [45]. The clone-types have been defined below.

Type 1 Clones. If two or more code fragments in a particular code-base are exactly the same disregarding the comments and indentations, these code fragments are called exact clones or Type 1 clones of one another.

Type 2 Clones. Type 2 clones are syntactically similar code fragments in a code-base. In general, Type 2 clones are created from Type 1 clones because of renaming identifiers and/or changing data types.

Type 3 Clones. Type 3 clones are mainly created because of additions, deletions, or modifications of lines in Type 1 or Type 2 clones. Type 3 clones are also known as *gapped clones*.

B. Similarity Preserving Co-change (SPCO) of two or more clone fragments

Let us consider two code fragments, CF1 and CF2, which are clones of each other in revision R of a subject system. A commit operation was applied on revision R and both of these two fragments were changed (i.e., the clone fragments co-changed) in such a way that they were again considered as clones of each other in the next revision R+1 (i.e., created because of the commit). In other words, the clone fragments preserved their similarity even after experiencing changes in the commit operation. Thus, we call this co-change of clone fragments (i.e., change of more than one clone fragment together) a Similarity Preserving Co-change (SPCO).

In a previous study [34] we showed that in a similarity preserving co-change (SPCO) more than one clone fragment from the same clone class are changed together consistently (i.e., the clone fragments are changed in the same way). Fig. 1 shows an example of SPCO of two Type 3 clone fragments from our subject system Freecol. The figure caption includes the details regarding the example. The figure demonstrates that the two clone fragments were changed together consistently in a particular commit operation for fixing a replicated-bug.

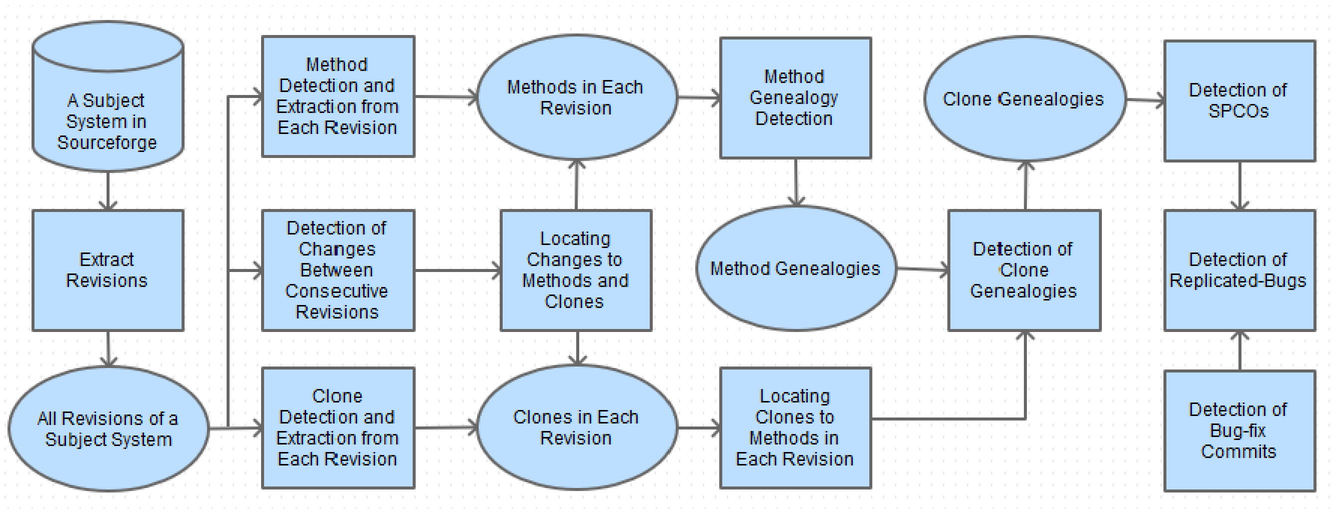

A Subject System in Sourceforge
Extract Revisions
All Revisions of a Subject System
Method Detection and Extraction from Each Revision
Detection of Changes Between Consecutive Revisions
Clone Detection and Extraction from Each Revision
Methods in Each Revision
Locating Changes to Methods and Clones
Clones in Each Revision
Method Genealogy Detection
Method Genealogies
Locating Clones to Methods in Each Revision
Clone Genealogies
Detection of Clone Genealogies
Detection of SPCOs
Detection of Replicated-Bugs
Detection of Bug-fix Commits

Fig. 2. The execution flow diagram of the experimental steps in detecting replicated bugs in code clones. The rectangles demonstrate the steps.

TABLE II. SUBJECT SYSTEMS

Systems	Lang.	Domains	LLR	Revisions
Ctags	C	Code Def. Generator	33,270	774
Camellia	C	Image Processing Library	89,063	170
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

C. Late propagation in code clones

Late propagation is defined as the occurrence of one or more inconsistent changes of a clone pair followed by a re-synchronizing change. The re-synchronization of the code clones indicates that the inconsistent changes were unintentional [2].

III. EXPERIMENTAL STEPS

We perform our investigation on six subject systems (Table II) downloaded from Sourceforge [41]. Fig. 2 shows the work procedure of our experimental steps.

A. Preliminary Steps

As demonstrated in Fig. 2, we perform the following experimental steps for detecting replicated bugs: **(1)** Extraction of all revisions (as mentioned in Table II) of each of the subject systems from the online SVN repository; **(2)** Method detection and extraction from each of the revisions using CTAGS [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; **(9)** Detection of SPCOs by analyzing clone change patterns; **(10)** Detection of bug-fix commit operations; and **(11)** Detection of replicated-bugs in code clones. For completing the first nine steps we use the tool SPCP-Miner [31]. For the details of these steps we refer the interested readers to our earlier work [33]. We will describe the detection of bug-fix commits later in this section. In Section IV we will describe how we detect replicated bugs.

TABLE III. NICAD SETTINGS FOR THREE TYPES OF CLONES

Clone Types	Identifier Renaming	Dissimilarity Threshold
Type 1	none	0%
Type 2	blindrename	0%
Type 3	blindrename	20%

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 [47], [48]. 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. NiCad settings for detecting three clone-types (Type 1, Type 2, and Type 3) are mentioned in Table III. These settings are explained in detail in our earlier work [33]. For different settings of a clone detector the clone detection results can be different and thus, the findings regarding bug-replication in 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. Moreover, code clones with a minimum size of 10 LOC are more appropriate from maintenance perspectives [4], [44], [49]. Before using the NiCad outputs of Type 2 and Type 3 cases, we processed them in the following way.

(1) Every Type 2 clone class that exactly matched any Type 1 clone class was excluded from Type 2 outputs.

(2) Every Type 3 clone class that exactly matched any Type 1 or Type 2 class was excluded from Type 3 outputs.

We processed NiCad clone detection results in the mentioned ways because we wanted to investigate bug-replication in three types of clones separately.

Clone Genealogies of Different Clone-Types. SPCP-Miner [31] 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 [34]. As we obtain three separate sets of clone genealogies for three different clone-types, we can easily determine and compare the bug-replication tendencies 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.

B. Bug-proneness Detection Technique

For a particular subject 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 [30] 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 [33].

The way we detect the bug-fix commits was also previously followed by Barbour et al. [2]. They 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 different in the sense that we investigate the bug-replication tendencies of different types of code clones. Also, Barbour et al. [2] did not investigate the most important clone type, the Type 3. Generally, the number of Type 3 clones in a system is the highest among the three clone-types. We consider Type 3 clones in our bug-replication study. While in one of our recent studies [39], we studied the bug-proneness of different types of clones, our focus in this paper is to study to what extent bugs are replicated in clone code.

IV. Replicated Bugs in Code Clones

In Section II we defined SPCO (Similarity Preserving Co-change) of two or more clone fragments from the same clone class. In a previous study [34] we found that in an SPCO the participating clone fragments (i.e., from the same clone class) are likely to be changed consistently (i.e., each of the participating clone fragments is likely to be changed in the same way). We detect replicated bugs in code clones on the basis of this finding.

A. Identifying replicated bugs in code clones

If two or more clone fragments of a clone class experience an SPCO in a bug-fix commit operation (i.e., if two or more clone fragments are changed together consistently, that means in the same way, for fixing a bug), then we understand that those clone fragments contained the same bug, and for fixing that bug each of the fragments was changed in the same way. Thus, this case is an example of fixing a replicated bug in code clones.

B. Steps in identifying replicated bugs in code clones

By analyzing the clone evolution history of a software system we identify the cases of fixing replicated bugs in code clones. Our identification consists of the following two steps.

Step 1: In this step we detect all the bug-fix commit operations of a subject system. The procedure of detecting bug-fix commits was discussed in Section III-B.

Step 2: Considering each of the bug-fix commits detected in **Step 1** we determine whether clone fragments of any clone-type experienced an SPCO in this commit. Let us consider a bug-fix commit BFC which was applied on revision R of a subject system. If we see that two or more clone fragments from a clone class in revision R experienced an SPCO in the commit BFC, then this is a case of fixing a replicated bug (i.e., fixing the same bug in each of the clone fragments that experienced an SPCO), because in an SPCO the participating clone fragments are likely to be changed together in the same way (i.e., consistently). We detect SPCOs of clone fragments of different clone-types using our tool SPCP-Miner [31].

We identify all the cases of fixing replicated bugs in each of the three types (i.e., Type 1, Type 2, and Type 3) of code clones by analyzing the clone evolution history of each of our candidate systems listed in Table II.

C. Manual analysis of the SPCOs in the bug-fix commits

After detecting the SPCOs in the bug-fix commits, we manually investigate the changes that occurred to the participating clone fragments in each SPCO to check whether the clone fragments were actually changed in the same way

TABLE IV. NUMBER OF SPCOS EXPERIENCED BY DIFFERENT TYPES OF CODE CLONES DURING BUG-FIX COMMIT OPERATIONS

	Ctags	Camellia	jEdit	Freecol	Carol	Jabref
Type 1	1	1	5	6	10	1
Type 2	0	0	5	5	14	5
Type 3	0	4	35	37	40	17

(i.e., whether the clone fragments were changed consistently). Table IV shows the number of SPCOs that occurred in the bug-fix commits considering each clone-type of each of the subject systems. From our manual investigation on each of these 186 SPCOs (in total) we can state that *in each SPCO, two or more clone fragments from a particular clone class were changed together consistently (i.e., in the same way).* Fig. 1 shows an SPCO of two Type 3 clone fragments in the bug-fix commit operation applied on revision 1080 of our subject system Freecol. In RQ 4 (i.e., the fourth research question) we investigate the change-types experienced by the clone fragments in the SPCOs that occurred in the bug-fix commits.

V. EXPERIMENTAL RESULTS AND ANALYSIS

In this section we answer the research questions listed in Table I by presenting and analyzing our experimental results.

A. Answering the first research question (RQ 1)

RQ 1: *What percentage of the clone fragments in different clone-types takes part in bug-replication?*

Motivation. Answering this question is important since a clone-type with a higher tendency of replicating bugs should be given a higher priority compared to the other clone-types when making clone management decisions. We conduct our study considering the major clone-types: Type 1, Type 2, and Type 3. We should note that any code fragment in the code-base can contain a bug. However, cloning of this fragment is responsible for replicating/propagating that bug in several other places. Without studying bug-replication in code clones we cannot understand the real impact of cloning in propagating bugs.

Methodology. For answering RQ 1 we identify bug-replication in three types (Type 1, Type 2, Type 3) of code clones by mining the clone evolution history of each of our subject systems. Section IV elaborates on the procedure we follow for detecting replicated bugs in code clones.

We automatically detect the bug-fix commits and then identify the similarity preserving co-changes (SPCOs) of clones in these commits. Table IV shows the number of SPCOs considering each clone-type of each of our subject systems. If an SPCO of two or more clone fragments occurs in a bug-fix commit, then it is an indication of fixing of a replicated bug. Table V shows the following four measures concerning bug-replication in each clone-type.

Measure CG: This is the total number of clone genealogies created during the whole period of evolution (the columns in Table V with the heading **CG**).

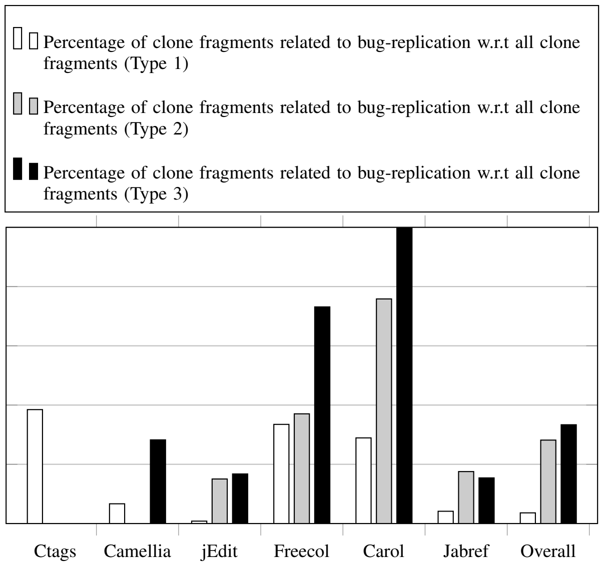


Fig. 3. Percentage of clone fragments related to bug-replication with respect to all clone fragments.

Measure CGBF: This is the total number of clone genealogies related to bug-fix (the columns with the heading **CGBF** in Table V).

Measure CGBR: This is the total number of clone genealogies related to bug-replication (the columns with the heading **CGBR** in Table V).

Measure PCGBR: This is the percentage of clone genealogies related to bug-replication with respect to all clone genealogies (The columns with the heading **PCGBR** in Table V). We calculate this percentage for a particular clone-type of a particular subject system using the following equation.

$$PCGBR = \frac{100 \times CGBR}{CG} \quad (1)$$

We also calculate the overall percentage (i.e., considering all subject systems) of clone fragments related to bug-replication considering each clone-type. This overall percentage was calculated using the following equation.

$$OPCGBR_{T_i} = \frac{100 \times \sum_{all\ systems} CGBR_{T_i}}{\sum_{all\ systems} CG_{T_i}} \quad (2)$$

Here, $OPCGBR_{T_i}$ is the overall percentage of clone fragments related to bug-replication with respect to all clone fragments of clone-type T_i (i = 1, 2, or 3).

Fig. 3 shows the percentage **PCGBR** for each clone-type of each subject system. It also shows the overall percentage **OPCGBR** for each clone-type. Looking at the percentages (PCGBRs) we understand that although the percentage of clone fragments containing replicated bugs is low for some subject systems (such as: jEdit, Jabref), this percentage can sometimes be considerable (for example, the percentages regarding our subject systems: Freecol, and Carol). We also see that for most of the subject systems (i.e., for four systems out of six) the

TABLE V. NUMBER OF CLONE GENEALOGIES RELATED TO BUG-REPLICATION

	Type 1				Type 2				Type 3			
Subject Systems	**CG**	**CGBF**	**CGBR**	**PCGBR**	**CG**	**CGBF**	**CGBR**	**PCGBR**	**CG**	**CGBF**	**CGBR**	**PCGBR**
Ctags	52	4	2	3.85%	88	4	0	0.00%	155	14	0	0.00%
Camellia	300	2	2	0.67%	48	0	0	0.00%	177	11	5	2.82%
jEdit	7398	73	6	0.08%	399	20	6	1.50%	2688	184	45	1.67%
Freecol	239	14	8	3.35%	162	12	6	3.70%	752	107	55	7.31%
Carol	415	31	12	2.89%	211	32	16	7.58%	682	134	68	9.97%
Jabref	483	8	2	0.41%	228	14	4	1.75%	1363	31	21	1.54%

CG = Number of Clone Genealogies **CGBF** = Number of Clone Genealogies Related to Bug-Fix
CGBR = Number of Clone Genealogies Related to Bug-Replication
PCGBR = Percentage of Clone Genealogies Related to Bug-Replication with respect to all clone genealogies

percentage regarding Type 3 case is the highest. From Table V we see that for the Type 2 and Type 3 cases of Ctags and Type 2 case of Camellia we did not get any clone fragments related to bug-replication.

We also wanted to investigate what percentage of code clones that are related to bugs contain replicated bugs. We calculate this percentage using the following equation.

$$PCGBRBF = \frac{100 \times CGBR}{CGBF} \quad (3)$$

Here, **PCGBRBF** is the percentage of clone fragments containing replicated bugs with respect to all buggy clone fragments of a particular clone-type of a particular subject system. We also calculate the overall value (i.e., considering all system) of this percentage for each clone-type using an equation which is similar to Eq. 2. We show these percentages (**PCGBRBF**, and the overall one) in Fig. 4.

From Fig. 4 we see that for three clone-types: Type 1, Type 2, and Type 3 overall respectively 24%, 39%, and 40% of the buggy clone fragments contain replicated bugs. From this graph we again see that Type 3 clones have the highest possibility of containing replicated bugs. The percentage regarding Type 2 clones is also very near to Type 3 clones.

Answer to RQ 1. For most of our subject systems a small percentage of all clone fragments can contain replicated bugs. However, this percentage can sometimes be considerable. According to our subject systems, the percentages for the three clone-types, Types 1, 2 and 3 can be up to 3.85%, 7.58%, and 9.97% respectively. We also see that around 24%, 39%, and 40% of the buggy clone fragments in Type 1, Type 2, and Type 3 case can contain replicated bugs. According to our observation, the percentages of code clones containing replicated bugs in Type 2 and Type 3 case are higher than in Type 1 case.

From our investigation and analysis in RQ 1 we realize that bugs in code fragments can often get replicated through cloning process. These replicated bugs can cause higher instability as well as increased maintenance efforts and costs for the software systems. Also, as Type 2 and Type 3 clones have higher possibilities of containing replicated bugs compared to Type 1 clones, we should possibly consider refactoring or tracking of Type 2 and Type 3 clones with high priorities.

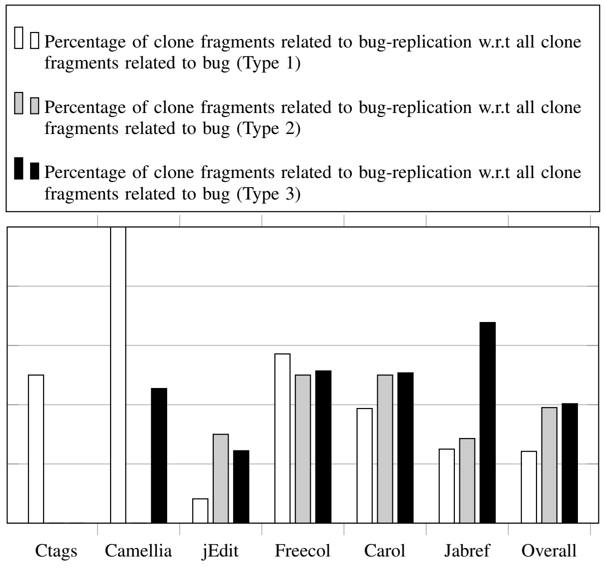


Fig. 4. Percentage of clone fragments related to bug-replication with respect to all clone fragments related to bug-fix.

B. Answering the second research question (RQ 2)

RQ 2: *What is the extent of bug-replication in the buggy clone classes of different types of clones?*

Motivation. From our answer to the previous research question (RQ 1) we can understand what percentage of clone fragments from which clone-type contain replicated bugs. However, we still do not know the extent to which the clone fragments in a clone class of a particular clone-type can contain replicated bugs. Intuitively, a clone-type with a higher extent of bug-replication should be considered more harmful compared to the other clone-types. We answer RQ 2 in the following way.

Methodology. We first detect the bug-fix commits following the procedure described in Section III-B. Considering each of these commits we determine whether a clone class was affected in that commit (i.e., whether one or more clone fragments from the clone class were changed in that commit for fixing a bug). Let us consider a bug-fix commit BFC which was applied on a particular revision R. We determine all the clone classes in this revision, and then identify which of these clone classes were affected by the commit BFC. Let us consider a clone class CC which was affected by BFC. We determine whether two or more clone fragments from

CC experienced a similarity preserving co-change (SPCO) in BFC. If this is true, then we select this clone class for our investigation in RQ 2. We call such a clone class an *Eligible Clone Class* (ECC). According to our previous discussions we can easily understand that an ECC contains a replicated bug. An affected clone class which is not ECC did not experience a similarity preserving co-change, and thus, this class does not contain replicated bugs.

Considering each of the eligible clone classes (ECCs) we determine the following two measures:

CF: The total number of clone fragments in the eligible clone class (i.e., in the ECC).

CFRB: The number of clone fragments that experienced similarity preserving co-change (SPCO). In other words, this is the number of clone fragments that contained replicated bugs.

From the above two measures of a particular ECC (eligible clone class) we can determine the extent of bug-replication in that ECC in the following way.

$$EBR = \frac{100 \times CFRB}{CF} \tag{4}$$

Here, EBR is the extent of bug-replication for a particular ECC. Considering all the ECCs of a particular clone-type from all the bug-fix commits of a particular subject system we determine the above two measures, and then calculate the overall extent of bug-replication using the following equation.

$$OEBR = \frac{100 \times \sum_{all\ ECCs} CFBR}{\sum_{all\ ECCs} CF} \tag{5}$$

Here, OEBR is the overall extent of bug-replication in the code clones of a particular clone-type of a subject system. We show the measure OEBR for the three clone-types of each of our subject systems in Fig. 5. The figure shows that the extent of bug-replication in the buggy clone classes of Type 1 case is 100% for five out of six subject systems (except jEdit). Also, for three subject systems (jEdit, Freecol, and Jabref) the extent of bug-replication in the buggy clone classes of Type 2 case is 100%. Bug-replication extent is also very high in the Type 3 buggy clone classes of three subject systems: Freecol, Carol, and Jabref. We also show the overall scenario (i.e., considering all subject systems) of the extent of bug-replication in the buggy clone classes of three clone-types. We see that while the buggy clone classes of Type 2 case show the highest extent of bug replication, Type 3 case shows the lowest extent. However, according to our candidate systems, the extent of bug replication in the buggy clone classes is very high in case of each of the clone-types.

Our answer to RQ 2 implies that in case of each clone-type, most of the clone fragments in a buggy clone class contain the same bug. For many cases the extent of replication is 100%. This is expected. If a particular code fragment contains a bug, then all other copies of that fragment will also contain the same bug. However, in case of each of the subject systems we find that there are some buggy clone classes where the extension of bug-replication is not 100%. Only some of the clone fragments in such a class (i.e., with partial replication of bug) were consistently modified for fixing bug leaving the other clone fragments in the class as they are. We were interested to investigate why some of the clone fragments in a partially buggy clone class were not considered buggy. We manually investigated such clone classes from each of the subject systems and had the following findings.

Answer to RQ 2. The extent of bug replication in the buggy clone classes of each of Type 1 and Type 2 cases is higher than the extent in Type 3 case. However, the overall extent of bug-replication in each of these three clone-types is very high (i.e., more than 70%). From such a finding we believe that there should be automatic support for finding clone classes with replicated bugs. When refactoring clone classes of any clone-type we should primarily focus on the clone classes that contain replicated bugs.

For most of the cases, one or more clone fragments were created by copy/pasting an original code fragment making a clone class consisting of the original one as well as the newly created ones. However, some of the newly created clone fragments were not properly changed to meet the contextual requirements. Eventually, a bug-fix commit only modified those clone fragments (i.e., the clone fragments that were not previously modified properly) leaving the other clone fragments in the class as they are. As an example of such a case we can mention the commit operation applied on revision 318 of our subject system Jabref. The commit message as was indicated by the developer says, "Fixed cut/copy/paste focus bug". Through further investigation we found that a clone class in revision 318 contained two clone fragments. The commit operation on this revision changed one of these two fragments by commenting an *if-statement* in that fragment. We infer that the *if-statement* that was commented was not appropriate for the context where the copied fragment was pasted.

We finally state that although some of the eligible clone classes (ECCs) do not exhibit a complete replication (i.e., a bug-replication extent of 100%) of the bug, the overall extent of bug-replication is very high for each clone-type. Thus, bug-replication tendencies of code clones should be given importance when taking clone management decisions.

C. Answering the third research question (RQ 3)

RQ 3: *What percentage of the bugs that were experienced by the code clones of different clone-types are replicated bugs?*

Motivation. Even after answering the previous two research questions it is still unknown whether most of the bugs experienced by the code clones are replicated bugs or not. If most of the bugs in code clones are replicated bugs then we understand that bug-replication is a common phenomenon during cloning, and in that case the programmers should be suggested to be aware that the code fragments they are going to copy are bug-free. We answer RQ 3 by investigating three clone-types of each of the subject systems.

Methodology. We detect the bug-fix commits of a candidate subject system. We sequentially examine each of these commits and update the following two counters considering each clone-type.

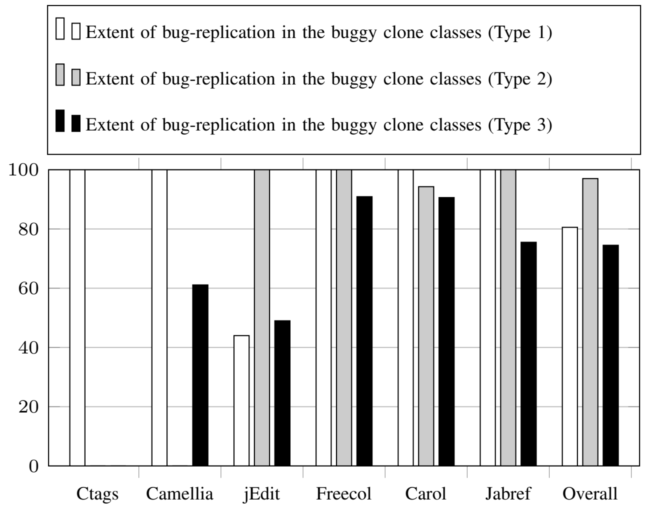


Fig. 5. Extent of bug-replication in the buggy clone classes.

NBC (The number of bugs experienced by the code clones): Let us consider a bug-fix commit BFC which was applied on the revision R of a subject system. If one or more of the clone classes residing in this revision were affected by the commit BFC, then we increase the counter NBC by one.

NBRC (The number of bugs that were replicated in the code clones): Let us consider that a bug-fix commit was applied on revision R of a candidate system and one or more clone classes in this revision were affected by the commit. If any of these affected clone classes experienced a similarity preserving co-change in this commit, then we increase the counter NBRC by one. We should again note that according our former discussions, experiencing a similarity preserving co-change (SPCO) in a bug-fix commit operation is an indication of the existence of replicated bugs in the code clones.

The percentage of replicated bugs with respect to all bugs experienced by the code clones of a particular clone-type of a particular subject system is determined using Eq. 6.

$$\% \ of \ replicated \ bugs = \frac{100 \times NBRC}{NBC} \qquad (6)$$

In Table VI we show the two measures: NBC and NBRC for each clone-type of each subject system. We also show the percentage of replicated bugs for three clone-types of each system in Fig. 6. We see that this percentage is considerable for most of the cases. From Table VI and Fig. 6 we realize that all the eight bugs experienced by the Type 2 clones of Carol were replicated bugs. Fig. 6 also shows the overall percentage of replicated bugs in the three clone-types considering all subject systems. This overall percentage was calculated using an equation similar to Eq. 2. For Type 1, Type 2, and Type 3 clones the overall percentages are respectively 30.65%, 55.26%, and 46.3%. While the overall percentage of replicated bugs is the highest in Type 2 clones, this percentage is the lowest in Type 1 clones.

TABLE VI. NUMBER OF BUGS THAT WERE REPLICATED IN CLONES

	Type 1		Type 2		Type 3	
Systems	**NBC**	**NBRC**	**NBC**	**NBRC**	**NBC**	**NBRC**
Ctags	3	1	4	0	11	0
Camellia	1	1	0	0	5	3
jEdit	37	5	10	5	42	15
Freecol	7	6	10	4	46	23
Carol	8	5	8	8	22	14
Jabref	6	1	6	4	23	14

NBC = Number of Bugs experienced by Clones
= The number of bug-fix commits experienced by the code clones of a particular clone type.

NBRC = Number of Bugs that were Replicated in Clones
= The number of bug-fix commits having SPCOs experienced by the code clones of a particular clone-type

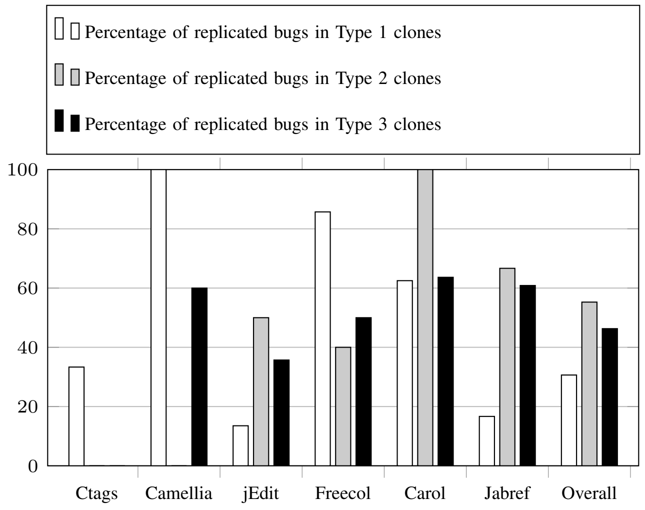


Fig. 6. Percentage of replicated bugs in different types of code clones.

Answering RQ 3. A considerable percentage of the bugs experienced by the code clones of each clone-type can be replicated bugs. This percentage can sometimes be very high (i.e., up to 100%). The overall percentage of replicated bugs in both Type 2 and Type 3 clones are higher compared to Type 1 clones.

Replication of bugs can cause increased maintenance effort and cost for software systems. Thus, it is important to investigate which code statements in the clone fragments primarily contain replicated bugs. We perform such an investigation in RQ 4.

D. Answering the fourth research question (RQ 4)

RQ 4: *Which code statements are highly related to bug-replication?*

Motivation. In RQ 4 we investigate which code statements in the clone fragments have high possibilities of containing replicated bugs. Programmers might need to be careful while copying code fragments containing those statements. The clone classes containing such statements (i.e., that contain replicated

TABLE VII. MOST FREQUENT CHANGE-TYPES DURING FIXING REPLICATED BUGS IN CODE CLONES

	Change Types	Ctags	Camellia	jEdit	Freecol	Carol	Jabref	Total
1	Addition of if-else blocks	4	2	17	12	17	7	59
2	Modification of if Condition			21	14	9	2	46
3	Deletion of if-else blocks			9	5	4	1	19
4	Modification of Parameters in the Called Method	2	2	23	8	26	7	68
5	Addition of Method Call			17	9	7	10	43
6	Replacement of Old Method Call by New Method Call			12	7	14		33
7	Deletion of Method Call			5	2	1	2	10

bugs) could be given higher priorities for refactoring and tracking. The existing studies did not investigate the bug-replication possibilities of clone fragments. Such an investigation can provide us helpful insights regarding minimizing bug-replication. We perform our investigation for answering RQ 4 in the following way.

Methodology. As we did before we detect the bug-fix commits and then identify cases where the bugs replicated in the clone fragments were fixed. Fixing of replicated bugs were identified by detecting the similarity preserving co-changes (SPCOs) of clone fragments in the bug-fix commits. We should again note that if two or more clone fragments from a particular clone class undergo an SPCO in a particular commit operation, then it is very much likely that those clone fragments were changed consistently (i.e., each of the clone fragments was changed in the same way) in that commit.

We manually check each of the bug-fix commits where the clone fragments experienced SPCOs. We identify the changes in the SPCOs that occurred in a bug-fix commit, and then categorize these changes. In Table VI we have already reported the number of bug-fix commits having SPCOs for each clone-type of each subject system. The column named 'NBRC'in this table reports this number. Table IV shows the number of SPCO cases for each clone-type of a subject system. There can be more than one SPCO in a bug-fix commit. In other words, more than one clone classes can experience similarity preserving co-change in a bug-fix commit. The frequent change categories that we observed during our manual analysis of the SPCO cases are shown in Table VII. A particular clone fragment can experience changes of more than one of these categories during fixing replicated bugs.

From Table VII we realize that the most frequent change experienced by the clone fragments during fixing replicated bugs is 'Modification of Parameters in the Called Method'. We suggest programmers to be more careful when copying code fragments containing *method-calls*. Before copying the programmers should ensure that the code fragment she is going to copy is bug-free. We should also prioritize refactoring of clone fragments that contain *method-calls*. Intuitively, a change in the name or in the parameters of a particular method will affect all the code fragments that call that particular method. Refactoring of clone fragments containing the same method calls can considerably minimize future software maintenance effort and cost. We suggest that clone fragments containing *if-conditions* should also be given a high priority for refactoring, because we observe a high frequency of the occurrence of modifying *if-conditions*.

The other frequent change-types according to Table VII are: addition of if-else blocks, addition of method calls, and replacement of an old method call by a new method call. We suggest that clone fragments with the possibilities of experiencing such types of changes should also be considered for management. If such clone fragments are not suitable for refactoring, then they should be tracked. We also observed a number of infrequent change-types during our investigation. These change-types include: replacement of C preprocessor by method call, addition, deletion or modification of loops, and modification of try-catch blocks.

Answer to RQ 4. According to our manual investigation it seems that clone fragments having *method calls* and *if-conditions* have high possibilities of containing replicated bugs. We suggest that such clone fragments should be considered for maintenance (i.e., refactoring or tracking) with high priorities.

We should note that as Type 1 clones are exactly the same code fragments, refactoring is possibly the best option for maintaining such clones. Tracking is the most suitable maintenance technique for the other two types (Type 2, and Type 3) of code clones.

VI. RELATED WORK

Bug-proneness of code clones has been investigated by a number of existing 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.

We see that none of the studies discussed above investigated bug-replication in code clones. In a previous study [39] we investigated and compared the bug-proneness of three types of code clones. However, that study does not focus on the bug-replication tendencies of code clones.

Rahman et al. [42] 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 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 investigate and compare the bug-replication tendencies of different types of code clones whereas they only focused on the bug-proneness of clones.

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 six diverse subject systems in our investigation. Also, we investigate the bug-replication possibilities 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 the late propagation in clones and its relationships with bugs. Aversano et al. [1] investigated clone evolution in two subject systems and reported that late propagation in clones is directly related to bugs. Barbour et al. [2] investigated eight different patterns of late propagation considering Type 1 and Type 2 clones of three subject systems and identified those patterns that are likely to introduce bugs and inconsistencies to the code-base.

We see that a number of studies have been conducted on the bug-proneness of code clones. However, none of these studies focus on the bug replication tendencies of different clone-types. We believe that without studying bug-replication in code clones we cannot realize their actual impact on software maintenance and evolution. Focusing on this we perform an in-depth investigation on bug-replication in code clones in this research. Our experimental results are promising and provide useful implications for better maintenance of software systems through minimizing bug-replications.

VII. Threats to Validity

We used the NiCad clone detector [6] for detecting clones. While all clone detections tools suffer from the *confounding configuration choice problem* [55] and might give different results for different settings of the tools, the setting that we used for NiCad for this experiment are considered standard [46] and with these settings NiCad can detect clones with high precision and recall [47], [48], [52]. 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 [30] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [3] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

In our experiment we did not study enough subject systems to be able to generalize our findings regarding the comparative bug-proneness of different types of clones. 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.

VIII. Conclusion

In this paper we present our empirical study on bug-replication in the major three types of code clones: Type 1, Type 2, and Type 3. Although a number of existing studies have investigated bug-proneness of code clones, none of these studies focus on analyzing the bug-replication tendencies of different clone-types. Without studying bug-replication tendencies of code clones we cannot understand the real impact of cloning on software maintenance and evolution. Focusing on this we conduct an in-depth empirical study on the three types of code clones residing in thousands of revisions of six diverse subject systems to investigate whether and to what extent bug-replication occurs in different clone-types.

According to our investigation, bug replication through code cloning is a common phenomenon during software maintenance and evolution. Around 55% of the bugs occurred in code clones can be replicated bugs. Also, up to 10% of the code clones can contain replicated bugs. Tendencies of bug-replication is higher in Type 2 and Type 3 clones than in Type 1 clones. It implies that code clones of Type 2 and Type 3 should be considered for management (such as refactoring or tracking) with high priorities. From our manual analysis we observe that *method-calls* and *if-conditions* residing in the clone fragments exhibit high tendencies of being related to bug-replication. From this we suggest that clone fragments containing *method-calls* and/or *if-conditions* should be prioritized for refactoring and tracking to minimize bug-replication. We finally believe that the outcomes of our bug-replication study are important for better management of code clones as well as for better maintenance of software systems. In future we would like to investigate ranking of code clones for management considering their bug-replication tendencies.

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] I. D. Baxter, M. Conradt, J. R. Cordy, R. Koschke, "Software clone management towards industrial application (dagstuhl seminar 12071)", *Dagstuhl Reports*, 2012, 2(2):21 – 57.

[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. 96 – 105.

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

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

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

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

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

[39] 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.

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

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

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

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

[44] C. K. Roy, M. F. Zibran, R. Koschke, “The Vision of Software Clone Management: Past, Present, and Future (Keynote paper)”, Proc. *CSMR-WCRE*, 2014, pp. 18 – 33.

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

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

[47] C. K. Roy, J. R. Cordy, R. Koschke, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach”, *Science of Computer Programming*, 2009, 74 (2009): 470 – 495.

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

[49] C. K. Roy, J. R. Cordy, “A Survey on Software Clone Detection Research”, *Technical Report 2007-541*, 2007, School of Computing, Queen’s University, 115 pp.

[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.