Investigating Context Adaptation Bugs in Code Clones

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

***Abstract*—The identical or nearly similar code fragments in a code-base are called code clones. There is a common belief that code cloning (copy/pasting code fragments) can introduce bugs in a software system if the copied code fragments are not properly adapted to their contexts (i.e., surrounding code). However, none of the existing studies have investigated whether such bugs are really present in code clones. We denote these bugs as *Context Adaptation Bugs*, or simply *Context-Bugs*, in our paper and investigate the extent to which they can be present in code clones. We define and automatically analyze two clone evolutionary patterns that indicate fixing of Context-Bugs. According to our analysis on thousands of revisions of six open-source subject systems written in Java, C, and C#, code cloning often introduces Context-Bugs in software systems. Around 50% of the clone related bug-fixes can occur for fixing Context-Bugs. Cloning (copy/pasting) a newly created code fragment (i.e., a code fragment that was not added in a former revision) is more likely to introduce Context-Bugs compared to cloning a preexisting fragment (i.e., a code fragment that was added in a former revision). Moreover, cloning across different files appears to have a significantly higher tendency of introducing Context-Bugs compared to cloning within the same file. Finally, Type 3 clones (gapped clones) have the highest tendency of containing Context-Bugs among the three major clone-types. Our findings can be important for early detection as well as removal of Context-Bugs in code clones.**

Keywords*—*Code Clones, Context Adaptation Bugs, Clone-Types, Software Maintenance

I. Introduction

Code clone has emerged as a controversial term in software maintenance research and practice. Programmers often perform code cloning (copy/pasting) during development for repeating common functionalities. Such an activity causes the existence of identical or nearly similar code fragments in the code-base of a software system. These code fragments are known as code clones in the literature [4], [9]. Two similar code fragments form a clone-pair. A group of similar code fragments forms a clone class or a clone group.

Code clones are of significant importance from the perspectives of software maintenance and evolution. A great many studies [1], [2], [6], [10], [13]–[15], [20]–[23], [26], [28], [30], [32]–[35], [37], [38], [41]–[46], [48], [49] have been conducted on the detection, analysis, and management of code clones. While a number of clone analysis studies [26], [28], [44] have identified some positive impacts of code clones, other studies [1], [2], [15], [37], [41] have shown evidence of serious negative impacts (such as hidden bug propagation [39], unintentional inconsistencies [45], high instability [1], [41]) of code cloning on software evolution. Our study in this paper focuses on a particular type of bug which we call Context Adaptation Bug that can be introduced by code cloning.

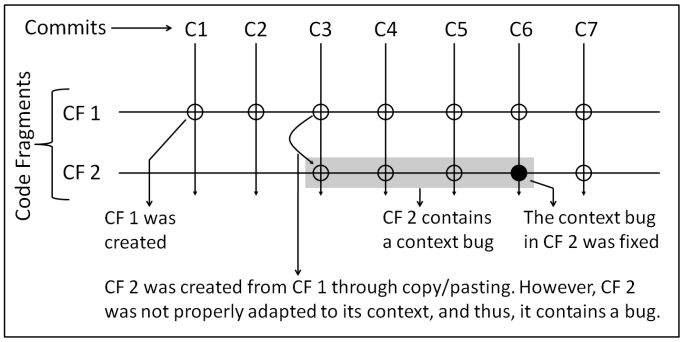

Fig. 1: Explaining context adaptation bug (i.e., context-bug)

Context Adaptation Bug (or simply, Context-Bug). Making copies of existing code fragments is a common activity of the programmers during implementation [6]. It is commonly suspected that a copied code fragment might introduce bugs in a code-base if that fragment is not properly adapted to its context. Such bugs have been called context adaptation bugs (Context-Bugs) in this paper. We should note that Mondal et al. [39] investigated propagated bugs. Bug propagation occurs because of making copies of an already buggy code fragment. However, we investigate context-bugs that get introduced when we copy a non-buggy code fragment from one place of a code-base and paste it to another place without properly adapting the pasted fragment in its context. We should also note that bugs might be introduced because of late-propagation [18], [29] in code clones. Late-propagation related bugs get introduced when changes made to one clone fragment are not immediately propagated to the other fragments in the same clone class. Context bugs are different because these get introduced when copied code fragments are not adapted to their contexts.

Example of a context-bug: Fig. 1 shows an example that explains how a context bug can get introduced to the code-base during evolution. In Fig. 1 we can see the evolution of two code fragments CF1 and CF2 through seven commit operations. CF1 is older and was created in commit C1. CF2 was created from CF1 in commit C3 through copy/pasting. However, CF2 was not properly adapted to its surrounding code and thus, contains a context bug. This context-bug was fixed in commit C6. CF1 was not buggy in its own context, and thus, it did not experience a bug-fix. Fig. 2 demonstrates a real-world example of fixing a context-bug from our subject system Carol. The figure shows that a clone fragment, *Clone Fragment 2*, which was created from *Clone Fragment 1* through copy/pasting was not properly adapted to its context and eventually it (*Clone Fragment 2*) experienced a context bug-fix. Figure caption explains the details.

Clone Fragment 2 (CF2) in Revision = 151			Clone Fragment 2 (CF2) in Revision = 152	
1	**private Object resolveObject(Object o) throws NamingException** {	Bug-fix change	**1**	**private Object resolveObject(Object o, Name name) throws NamingException** {
2	try { if (o instanceof IIOPRemoteReference) {		2	try { if (o instanceof IIOPRemoteReference) {
3	Reference objRef = ((IIOPRemoteReference)o).getReference();		3	Reference objRef = ((IIOPRemoteReference)o).getReference();
4	ObjectFactory objFact = (ObjectFactory)(Class.forName (objRef.getFactoryClassName())).newInstance();		4	ObjectFactory objFact = (ObjectFactory)(Class.forName (objRef.getFactoryClassName())).newInstance();
5	**return objFact.getObjectInstance(objRef,null,null,null);** }	Bug-fix change	**5**	**return objFact.getObjectInstance (objRef, name, this, iiopContext.getEnvironment());** }
6	else if (o instanceof IIOPRemoteResource) {		6	else if (o instanceof IIOPRemoteResource) {
7	return ((IIOPRemoteResource)o).getResource(); }		7	return ((IIOPRemoteResource)o).getResource(); }
8	else {return o;}}		8	else {return o;}}
9	catch(Exception e) {throw new NamingException("" + e); }		9	catch(Exception e) {throw new NamingException("" + e); }

Clone Fragment 1 (CF1) in Revision = 151		Clone Fragment 1 (CF1) was not changed in Revision = 152
1	private Object resolveObject(Object o) {	
2	try {if (o instanceof IIOPRemoteReference) {	**File path of Clone Fragment 1:**
3	Reference objRef = ((IIOPRemoteReference)o).getReference();	carol/src/org/objectweb/carol/jndi/iiop/IIOPContextWrapper.java
4	ObjectFactory objFact = (ObjectFactory)(Class.forName (objRef.getFactoryClassName())).newInstance();	**File path of Clone Fragment 2:** carol/src/org/objectweb/carol/jndi/iiop/IIOPReferenceContextWrapper.java
5	return (Referenceable)objFact.getObjectInstance(objRef,null, null,null);}	
6	else if (o instanceof IIOPRemoteResource) {	Clone Fragment 2 experienced context bug-fix as demonstrated above. However,
7	return ((IIOPRemoteResource)o).getResource(); }	Clone Fragment 1 was not buggy in its own context, and thus, it did not require
8	else {return o;}}	the bug-fix changes.
9	catch (Exception e) {TraceCarol.error ("IIOPContextWrapper.resolveObject()", e); return o; }	

Fig. 2: This figure shows an example of context bug-fix in a clone fragment denoted as Clone Fragment 2 (CF2) from our subject system Carol. The bug-fix commit was applied on revision 151. We can see the snapshots of CF2 both in revision 151 and 152. The changes (in Line 1 and Line 5) have been highlighted. The commit message says "*Bug 283 correction: IIOP Reference binding problem*". The figure contains another clone fragment called Clone Fragment 1 (CF1) in revision 151. Both CF1 and CF2 were created in revision 140 and they make a Type 3 clone-pair. None of these fragments was changed until revision 151 where CF2 experienced the bug-fix change as we have just described. CF1 never experienced a bug-fix. CF1 and CF2 follow the second bug-fix pattern described in Section IV-B. If we carefully look at CF1 and CF2 and the bug-fix changes experienced by CF2 we realize that the bug that was fixed in CF2 is a context-bug according to our definition. If we look at the lines 1 and 5 of CF1 and CF2 in revision 151, we see that the corresponding lines of the clone fragments are similar except that line 1 in CF2 throws an exception which is absent in line 1 of CF1, and line 5 of CF1 contains a type-casting (i.e., *(Referenceable)*) which is not present in line 5 of CF2. However, the bug-fix changes that occurred in CF2 indicate that CF2 was not properly adapted to its context at the time of its creation (possibly through copy/pasting from CF1) and the bug, *Bug 283*, was introduced to it. The bug-fix commit adapted it by adding an extra parameter called *name* of type *Name* to it and making corresponding changes in its fifth line.

TABLE I. Research questions

Serial	Research question
RQ 1	What proportion of clone-related bug fixes are associated with context-bugs?
RQ 2	Which pattern of context bugs is more likely to occur during software evolution?
RQ 3	Which type of code clones have a high possibility of containing context-bugs?
RQ 4	Do the two clone fragments from a clone pair that is involved with context bug generally reside in two different files?

Although a number of existing studies [13], [19], [23], [27], [32], [36], [39] have investigated bug-proneness of code clones in software systems, none of these studies investigated context-bugs introduced by code cloning (copy/pasting). Without investigating context-bugs, we cannot properly assess the impact of code cloning on software maintenance and evolution. Moreover, if it is observed that context-bugs often get introduced to software systems through code cloning, we should have a technique for automatically identifying code clones that are likely to contain such bugs so that we can detect and fix these bugs earlier in evolution.

In our study, we detect and analyze context-bugs introduced by code cloning. We define and automatically examine two clone evolutionary patterns that indicate fixing of context bugs in code clones. To the best of our knowledge, ours is the first study to investigate context-bugs during software evolution. We perform our investigation on thousands of revisions of six software systems written in three programming languages and answer the four research questions listed in Table 1. According to our findings:

- Around 50% of the clone related bug-fixing changes occur for fixing context bugs.
- According to our statistical significance tests, copy/pasting a newly created code fragment (i.e., a code fragment that was not added in a former revision) has a significantly higher possibility of introducing context bugs to the code-base compared to copy/pasting a preexisting code fragment (i.e., a code fragment that was added in a former revision).
- Moreover, cloning across different source code files is significantly more likely to introduce context-bugs compared to cloning within the same file.
- Finally, Type 3 clones exhibit the highest likeliness of containing context bugs among the three clone-types (Type 1, Type 2, and Type 3).

Our second and third findings (mentioned above) can improve developer awareness regarding risky cloning operations (for example, cloning a newly created code fragment and cloning

across different files) so that they can avoid these operations for minimizing context bugs. These findings can also be leveraged when developing a tool for detecting clone fragments that are likely to contain context bugs. We should prioritize refactoring Type 3 clones because such clones are the mostly likely ones to contain context-bugs among the three major clone-types (Type 1, Type 2, and Type 3). We finally suggest that detection and removal of context bugs should be given importance during software evolution and maintenance. Our implementation and data are available on-line [55].

The rest of the paper is organized as follows. Section II discusses the background topics, Section III positions our research in the context of existing studies, Section IV defines and discusses the context bug-fix patterns, Section V describes our experiment setup and steps, Section VI answers our research questions by presenting and analyzing our experiment results, Section VII discusses the implications of our findings, Section VIII discusses the validity threats, Section IX concludes the paper by mentioning future work.

II. Background

Our research involves detection and analysis of code clones of all three major clone-types: Type 1, Type 2, and Type 3. We define these clone-types in the following way according to the literature [4], [9].

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

We automatically detect and analyze context bugs in all these three major types of code clones.

III. Related Work

Bug-proneness of code clones has already been investigated by a number of studies. Li and Ernst [23] performed an empirical study on the bug-proneness of clones by investigating four software systems and developed a tool called CBCD on the basis of their findings. CBCD can detect clones of a given piece of buggy code. Li et al. [54] developed a tool called CP-Miner which is capable of detecting bugs related to inconsistencies in copy-paste activities. They focused on detecting inconsistent mapping of identifiers. Steidl and Göde [13] investigated 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 [45] investigated the occurrences of unintentional inconsistencies to the code clones of three mature software systems and found that around 14.8% of all changes that occurred to the code clones are unintentionally inconsistent. Chatterji et al. [12] performed a user study to investigate how clone information can help programmers localize bugs in software systems. Jiang et al. [32] performed a study on the 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. [27] 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. [50] 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 a number of studies investigated bug-proneness in code clones. However, none of these studies analyze context adaptation bugs (i.e., context-bugs) in code clones. We define two evolutionary patterns that indicate fixing of context-bugs in code clones, mine these patterns from thousands of revisions of six open-source subject systems, and investigate how often code cloning introduces context bugs.

Islam et al. [19] investigated bug-replication in code clones. They identified which of the clone fragments in a clone class contains the same bug. If more than one clone fragment in a clone class contain the same bug, they considered that the bug is a replicated one. Mondal et al. [39] analyzed bug-propagation through code cloning. In particular, they investigated how often a buggy code fragment is copied to several places in the code-base being unaware of the presence of bug in the fragment. According to their investigation on four subject systems, near-miss code clones have a higher tendency of propagating bugs compared to exact code clones. In another study, Mondal et al. [36] compared the bug-proneness of three types of code clones. They investigated which types of clones experience bug-fixes more frequently. However, none of these studies investigate context-bugs in code clones.

Rahman et al. [16] made a comparison of the bug-proneness of clone and non-clone code and found that clone code is less bug-prone than non-clone code. They performed their investigation on the evolutionary history of four subject systems using DECKARD [31] clone detector. However, they did not investigate how often context bugs get introduced to the code-base because of code cloning.

Selim et al. [17] 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, Selim et al. [17] did not investigate context-bugs introduced by code cloning.

A number of studies have also been done on the late propagation in clones and its relationships with bugs. Aversano et al. [28] investigated clone evolution in two subject systems and reported that late propagation in clones is directly related to bugs. Barbour et al. [29], [30] 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. Mui et al. [18] investigated late propagation and bugs related to it by investigating four subject systems.

We see that different studies have investigated clone related bugs in different ways and have developed different bug detection tools [8]. However, none of these studies investigated whether and how often context-bugs get introduced to the code-base through code cloning. Without investigating context-bugs we cannot properly realize the impacts of code cloning on software maintenance and evolution. In our study, we define and analyze two bug-fix patterns that reasonably indicate fixing of context-bugs in code clones. To the best of our knowledge, our study is the first one to investigate context-bugs introduced through code cloning. According to our experiment results and analysis, around 50% of the clone related bug-fixes can occur for fixing context-bugs. We identify risky cloning operations that the programmers should avoid for minimizing context bugs. Our findings can be leveraged to build an automatic tool for finding code clones that have a high possibility of containing context bugs.

IV. Context Bug Fix Patterns

This section defines two bug-fix patterns that reasonably indicate fixing of context adaptation bugs (i.e., context-bugs) in code clones. These patterns are described bellow.

A. Context bug-fix pattern 1

Let us assume that in a particular commit operation, a code fragment CF2 was created by copy/pasting a preexisting (i.e., previously committed) code fragment CF1. However, CF2 was not properly adapted to its context (i.e., surrounding code), and thus, it introduced an inconsistency/a bug in the code-base. This bug is a context bug according to our definition. In a later commit operation, changes occurred in CF2 for fixing the context bug. On the basis of this phenomenon, we provide a formal definition of the context bug-fix pattern 1.

Formal definition of the pattern. Let us assume that the code fragments *CF1* and *CF2* were created in the commit operations C_i and C_j respectively. The fragment, *CF2*, experienced a bug-fix in commit C_k where $C_k > C_j$. This bug-fix is the fixing of a context bug in *CF2*, if the following four conditions hold.

- **Condition 1.** The first condition is, $C_j > C_i$. This condition implies that the code fragment *CF2* was created after the creation of *CF1* (i.e., *CF1* is the older fragment).
- **Condition 2.** The second condition is, *CF1* and *CF2* form a clone-pair. Thus, this condition implies the likeliness that *CF2* was created by copy/pasting *CF1*.
- **Condition 3.** *CF2* did not experience any change during the commits $C_j + 1$ to $C_k - 1$. This condition makes us realize that the bug that *CF2* contained was not introduced to it during the commits $C_j + 1$ to $C_k - 1$. The first change that *CF2* experienced after being created from *CF1* is the bug-fix change in commit C_k. Thus, this third condition along with the previous two conditions implies the likeliness that *CF2* contained a context-bug just after being created because of not properly adapting it to its context and the bug-fix change that it experienced in commit C_k fixed that context-bug.

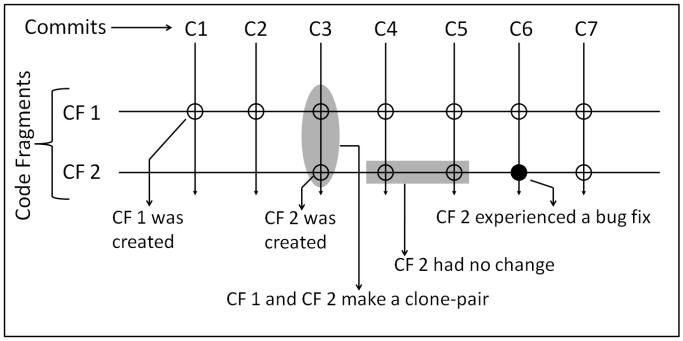

Fig. 3: Context bug-fix Pattern 1

- **Condition 4.** The code fragment *CF1* never experienced a bug-fix. This condition implies that $CF1$ was not buggy. By applying this condition we filter out the cases of propagating already existing bugs and the bugs that are related to late-propagation [18], [29].

Fig. 3 explains this pattern (Pattern 1) with a simple example. We see the evolution of two code fragments CF1 and CF2 through seven commit operations: C1 through C7. CF1 is older than CF2 because they were created in the commits C1 and C3 respectively. Moreover, just after the creation of CF2, the fragments CF1 and CF2 make a clone-pair. CF2 experienced a bug-fix change in the commit operation C6. The figure also indicates that CF2 did not experience any change before experiencing the bug-fix (i.e., in the commits C4 and C5). Moreover, CF1 did not experience a bug-fix. Thus, the evolutionary scenario in the figure (Fig. 3) complies with the four conditions stated above.

B. Context bug-fix pattern 2

This pattern is slightly different than the previous pattern. Let us assume that in a particular commit operation two similar code fragments, CF1 and CF2, were created together. One of the two fragments, for example CF1, was first written by the programmer, and the second one, CF2, was created by copy/pasting the first fragment. However, CF2 was not properly adapted to its context, and it introduced a bug in the code-base. This context bug in CF2 was fixed in a later commit operation by making changes to the fragment CF2. On the basis of this bug-fix phenomenon, we formally define the context bug-fix pattern 2.

Formal definition of the pattern. Let us assume that the code fragments, *CF1* and *CF2*, were created in the commit operation C_i. The fragment *CF2* experienced a bug-fix in the commit operation C_j ($C_j > C_i$). This bug-fix is the fixing of a context bug in *CF2* if the following conditions hold.

- **Condition 1.** *CF1* and *CF2* make a clone-pair and a similar code fragment was not pre-existing. This condition implies the likeliness that one of these two fragments was first written by the developer, and then, she made the second fragment by copy/pasting the first one. Let us assume that *CF1* was created at first.
- **Condition 2.** *CF2* did not experience any change during the commits $C_i + 1$ to $C_j - 1$. This condition

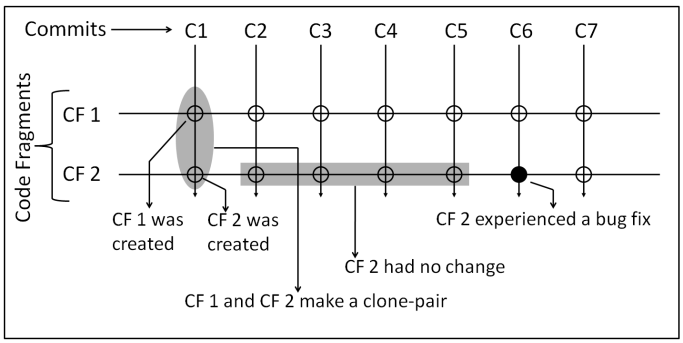

Fig. 4: Context bug-fix Pattern 2

ensures that the first change that *CF2* experienced after being created was the bug-fix change. In this situation it is likely that *CF2* was not properly adapted to its surrounding code when it was created by copy/pasting and a context-bug was introduced to it (*CF2*). The bug-fix change that it experienced in C_j fixed that context-bug.

- **Condition 3.** *CF1* never experienced a bug-fix. Thus, *CF1* was not buggy in its own context.

Fig. 4 explains the second pattern (Pattern 2) using an example. The evolutionary scenario in Fig. 4 is similar to the scenario depicted in Fig. 3 with the exception that both of the fragments CF1 and CF2 in Fig. 4 were created in the same commit operation (i.e., C1 according to Fig. 4). In Fig. 3, CF1 and CF2 were created in different commits. Finally, Fig. 4 presents an evolutionary scenario that complies with the formal definition of the second pattern (Pattern 2).

C. Detection of context-bug fix patterns

We first identify the bug-fix commit operations using the technique proposed by Mockus and Votta [3]. This technique can automatically detect bug-fix commits by analyzing the commit messages. Let us assume that a bug-fix commit operation, BFC, was applied on a particular revision R and the next revision R+1 was created because of the bug-fix. We identify the clone pairs in revision R using NiCad [24] and identify the potential pairs. A potential clone-pair consists of two clone fragments such that one fragment was changed in the commit operation BFC but the other fragment was not. Whether a clone fragment was changed in BFC can be determined by checking the instances of the clone fragment in revisions R and R+1. We should also note that because of the changes in one fragment, the two fragments might not make a clone pair in revision R+1. After detecting all the potential pairs from revision R, we analyze the past evolution of the two clone fragments in each potential pair and determine whether the fragments evolved by following any of the two patterns. The evolution of the clone fragments was analyzed by automatically examining their genealogies. A potential pair is considered to have evolved following the first pattern (pattern1) if the fragment that did not experience the bug-fix in BFC was created earlier than the other fragment that experienced the bug-fix in BFC, and also, the newer fragment did not experience any change before experiencing the bug-fix. A potential pair is considered to have evolved by following the second pattern (Pattern 2) if the two fragments in the pair were created in the same commit operation and the bug-fix change is the first change that one of the two fragments experienced. While detecting each of the patterns we ensured that one of the two fragments forming the pattern never experienced a bug-fix during the whole period of evolution of the software system.

V. Experiment Setup and Steps

We conduct our experiment on six open-source software systems listed in Table III. The systems are available in an on-line SVN repository called SourceForge.net [47]. In Table III, we can see the subject systems along with their application domains, starting revisions, ending revisions, implementation languages, and sizes (LOC). The starting revision for most of the subject systems is 1 except Freecol and jEdit. For these two systems, the starting revisions are respectively 1000 and 3791. The on-line repository does not contain the former revisions for these two systems. For each of the subject systems we perform the following experiment steps.

1) Downloading all the revisions (as indicated in Table III) of the subject system from their repositories.
2) Detecting methods from each of the revisions using the tool called CTAGS [11].
3) Detecting three types of code clones from each of the revisions using the clone detection tool NiCad [24].
4) Detecting changes between every two consecutive revisions using UNIX *diff* operation.
5) Mapping changes to the already detected code clones in each revision by using the starting and ending line numbers of the clones and changes.
6) Mapping the clones to the already detected methods in each revision using the starting and ending line number of the methods and clones.
7) Detecting method genealogies by considering the methods from all the revisions following the technique that was proposed by Lozano and Wermelinger [1].
8) Detecting clone genealogies by tracking the propagation of clone fragments through the methods using the tool called SPCP-Miner [40].
9) Identifying the reported bug-fixes experienced by the code clones by automatically analyzing the commit messages using the technique of Mocus and Votta [3].
10) Detecting and analyzing the context bug-fix patterns.

The procedure for detecting reported bug-fixes experienced by code clones will be discussed later in this section. Section IV-C discusses how we detected the context bug-fix patterns.

Clone Detection. We detect code clones from our subject systems using the NiCad clone detection tool [24]. An existing research [25] shows that NiCad is a promising choice among the modern clone detectors because it shows high precision and recall in detecting the major three clone-types (Type 1, Type 2, and Type 3). We apply NiCad to detect block level code clones of at least 10 lines considering a dissimilarity threshold of 30% with blind-renaming of identifiers. Svajlenko and Roy [25] used these settings and found NiCad to be a promising clone detector in terms of precision and recall. We should note that before using the NiCad outputs for Type 2 and Type 3 cases, we preprocessed them in the following way.

TABLE II. Number of context bug-fixes in different types of code clones

	Ctags			Carol			Freecol			jEdit			Jabref			MonoOSC		
	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3
Count of bug-fixes in clones	2	5	21	9	9	31	11	13	57	19	9	44	10	8	19	2	0	7
Count of context bug-fixes	1	4	8	2	1	13	6	5	29	16	5	32	5	2	5	1	0	2
Count of context bug fixes (Pattern 1)	0	0	5	1	0	3	0	0	5	4	1	2	0	0	1	0	0	1
Count of context bug fixes (Pattern 2)	1	4	3	1	1	10	6	5	24	12	4	30	5	2	4	1	0	1

T1 = Type 1 or exact clones T2 = Type 2 or syntactic clones T3 = Type 3 or gapped clones

TABLE III. Investigated subject systems

Systems	**Lang.**	**Domains**	**LLR**	**SRev**	**LRev**
Ctags	C	Code Definition Generator	33,270	1	774
Carol	Java	Game	25,091	1	1700
Freecol	Java	Game	91,626	1000	1950
jEdit	Java	Text Editor	191,804	3791	4000
Jabref	Java	Reference Management	45,515	1	1545
MonoOSC	C#	Formats and protocols	14,883	1	355

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

(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 performed these preprocessing steps because we wanted to investigate context-bugs in each clone-type separately.

Clone Genealogies of Different Clone-Types. We use SPCP-Miner [40] to detect clone genealogies considering each clone-type (Type 1, Type 2, and Type 3) separately. Considering a particular clone-type this tool first detects all the clone fragments of that particular type from each of the revisions of a 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. As we obtain three separate sets of clone genealogies for three different clone-types, we can detect the context-bug patterns in these clone-types using our technique described in Section IV-C.

Tackling Clone-Mutations. Xie et al. [50] 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 context bug-fixes were experienced by which clone-types.

A. Bug Detection in Code Clones

For each of our subject systems, 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 examine the commit messages using the heuristic proposed by Mockus and Votta [3] to identify those commits that occurred for 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 modifications of those clone fragment(s) were 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 bug-fix commits that affected code clones. The way we detect the bug-fix commits was also previously followed by Barbour et al. [30]. They investigated whether bugs in code clones are related to late propagation. Our study is different because we investigate whether bug-fixes occurred in code clones are related with context-bugs. Barbour et al. [30] did not investigate Type 3 clones. We investigate all three major clone-types (Type 1, Type 2, and Type 3) in our study.

TABLE IV. Total number of bug fixes during entire evolution

	Ctags	Carol	Freecol	jEdit	Jabref	Mono.
No. of bug fixes	300	137	465	58	233	51

VI. Experiment Results and Analysis

We apply our implementation on each of our subject systems and detect the context bug-fix patterns. We analyze these patterns and answer the research questions in Table I.

A. Answering the first research question (RQ 1)

RQ 1. *What proportion of clone-related bug fixes are associated with context-bugs?*

Rationale. As we discussed in the introduction, knowing the answer to RQ 1 is the primary goal of our research. If we see that fixing of context bugs occur frequently during system evolution, then it would be beneficial to take measures towards minimizing such bugs. We perform our investigation for answering RQ 1 in the following way.

Investigation Procedure. We first identify the bug-fix commit operations that affected code clones by following the procedure discussed in Section V-A. Then, we mine the context bug-fix patterns using the technique discussed in Section IV.C. When mining the patterns, we determine the following measures for each of our subject systems.

- Total number of bug-fixes that were experienced by the software system during the whole period of evolution. Table IV contains this number for our subject systems.
- Total number of bug-fixes that were experienced by the code clones of three clone-types (Type 1, 2, and 3).

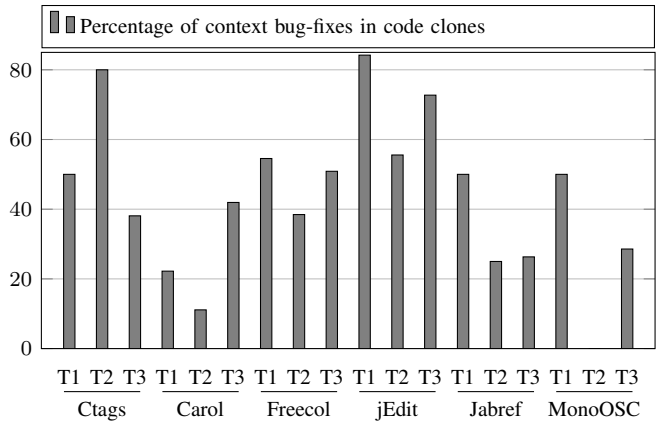

Fig. 5: Percentages of context bug-fixes in different types of code clones of different subject systems

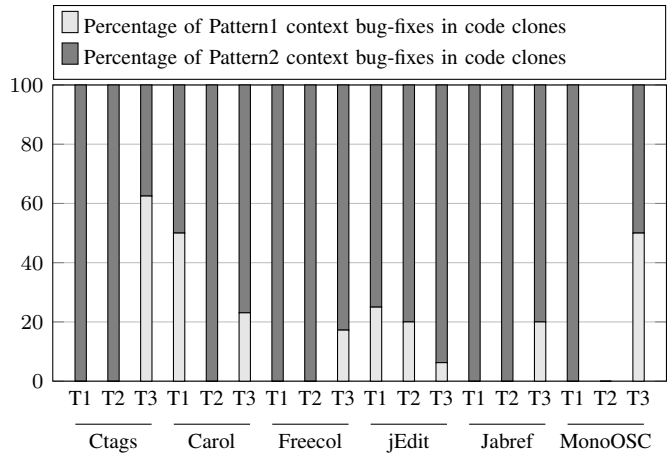

Fig. 6: Percentages of context bug-fixes of two patterns in different types of code clones of different subject systems

- Total number of context bug-fixes experienced by the three types of code clones during evolution.

The last two measures for each subject system are recorded in Table II. A particular bug-fix may affect clone fragments of more than one clone-type, and this is reflected in the number of bug-fixes for each clone type. Fig. 5 shows the percentage of context bug-fixes with respect to all bug-fixes in code clones considering each clone-type of each of our subject systems. We see that Type 1 clones of jEdit shows the highest percentage (84%) of context bug-fixes. According to Table II, out of 19 bug-fixes experienced by the Type 1 clones of jEdit, 16 were context bug-fixes. We also determine the overall percentage of context bug-fixes considering all clone-types of all subject systems. We observe that overall around 50% of the bug-fixes experienced by code clones can be context bug-fixes.

Fig. 2 demonstrates an example of context bug-fix that followed the second pattern (Pattern 2) described in Section IV-B. Our implemented prototype tool mined this bug-fix instance from our subject system Carol. The caption of the figure provides a detailed explain of the context bug-fix.

We manually analyze all 137 context bug-fixes (Table II) detected by our implementation from all clone-types of all subject systems to investigate what type of changes really occur to the clone fragments during fixing context-bugs. We found that the context bugs were mostly introduced to the clone fragments because those were not handling special cases related to their contexts. For example, the commit operation 634 in our subject system called Ctags fixes a context bug with the bug-fix message ”jscript.c was not properly handling escaped quotes”. After investigating the bug-fix changes we found that a method named ’parseString’ was created in file ’jscript.c’ through exact copy/pasting from file ’sql.c’. However, the method in ’jscript.c’ needed to handle escaped quote (\¨) and the bug-fix commit occurred for handling this. The method in ’sql.c’ did not require to handle it. Other instances of the context bug-fixes consist of inserting additional conditions to the *IF-Conditions*, refining parameters of the called methods (as demonstrated in Fig. 2), and inserting or deleting method calls or statements.

Answering RQ 1. According to our experiment results and analysis, a considerable proportion of the bug-fixes experienced by code clones can be context bug-fixes. This proportion can be up to 84% (Fig. 5) according to our subject systems. The overall proportion considering all subject systems and all clone-types is around 50%.

We see that context bugs often get introduced to the code-base through code cloning. Detection and fixing of such bugs can require a considerable amount of maintenance effort and cost. We believe that an automatic support for detecting such bugs at the moments these are introduced can reduce maintenance cost during software evolution.

B. Answering the second research question (RQ 2)

RQ 2. *Which pattern of context bugs is more likely to occur during software evolution?*

Rationale. Our answer to RQ 1 indicates that context bugs often get introduced to code clones during evolution. Such a finding inspires us to investigate which pattern of context bugs has a higher possibility of getting introduced. Finding from RQ 2 might give us insights on vulnerable copy/paste operations during implementation. We perform our investigation for answering RQ 2 in the following way.

Investigation procedure. We described two patterns of context bug-fixes in Section IV. We first mine these two context bug-fix patterns by mining the evolutionary history of our subject systems following the mechanism described in Section IV-C. We then determine how many of the context bug-fixes followed which pattern. Table II shows the counts of Pattern 1 and Pattern 2 bug-fixes for each clone-type of each of our subject systems. Fig. 6 shows a stacked bar graph showing the percentages of Pattern 1 or Pattern 2 context bug-fixes with respect to all context bug-fixes. We see that there are seventeen bars in the figure. For Type 2 case of our subject system, MonoOSc, we did not get any context bug-fixes. For most of the bars in the figure except for three bars (i.e., the bars regarding Type 3 case of Ctags, Type 1 case of Carol, and Type 3 case of MonoOSC), the percentage of Pattern 2 context bug

fixes is much higher than the percentage of Pattern 1 context bug fixes. For Type 3 case of Ctags, the percentage regarding Pattern1 is higher. For the other two cases (Type 1 case of Carol and Type 3 case fo MonoOSC), both patterns show the same percentage (50%). We also determine the overall percentages of Pattern 1 and Pattern 2 bug-fixes considering all the context bug-fixes of all clone-types of all subject systems. These overall percentages for Pattern 1 and Pattern 2 are respectively 16.79% and 83.21%. Thus, around 83.21% of the context bug-fixes follow Pattern 2 during evolution.

Statistical significance test regarding comparing the two patterns. As most of the bars in Fig. 6 indicate a higher proportion of Pattern 2 context bug-fixes, we wanted to investigate whether the proportions regarding Pattern 2 are significantly higher compared to the proportions regarding Pattern 1. We performed Wilcoxon Signed Rank test [52], [53] considering the seventeen cases (regarding the 17 bars) in Fig. 6. We determine whether the percentages regarding Pattern 1 in these cases are significantly different than the percentages regarding Pattern 2. We choose Wilcoxon Signed Rank (WSR) test because our samples are paired. For each of the 17 cases we get two percentages: (i) one for Pattern 1 and (ii) the other for Pattern 2. We should note that WSR test is non-parametric, and thus, it does not require the samples to be normally distributed [52]. This test can be applied to both small and large data samples. We apply this test considering a significance level of 5%. According to our two-tailed test result, the percentages of Pattern 1 bug-fixes are significantly different than those of Pattern 2 bug-fixes with a p-value of 0.0008 which is smaller than 0.05. As the percentages of Pattern 2 bug-fixes are generally higher, we realize that the proportion of context bug-fixes that follow Pattern 2 is significantly higher than Pattern 1.

> **Answer to RQ 2.** According to our investigation and analysis, the context-bug fixes of Pattern 2 are significantly more likely to occur than the context-bug fixes of Pattern 1 (Fig. 6). Thus, we can say that cloning a newly created code fragment (i.e., a code fragment that was not created in a former revision) has a significantly higher possibility of introducing context bugs compared to cloning a preexisting code fragment (i.e., a code fragment that was created in a former revision).

Our findings imply that cloning a preexisting code fragment is less risky than cloning a newly created one.

C. Answering the third research question (RQ 3)

RQ 3. *Which type of code clones have a high possibility of containing context-bugs?*

Rationale. Identifying which type of code clones have a high tendency of containing context-bugs is important for devising a tool for locating and fixing such bugs. A context bug discovery tool can emphasize that particular clone-type when discovering context bugs. Moreover, if a particular clone-type appears to be very likely to contain context-bugs, then we can advise developers to be more careful when making code clones of that particular type. We perform our investigation for answering RQ 3 in the following way.

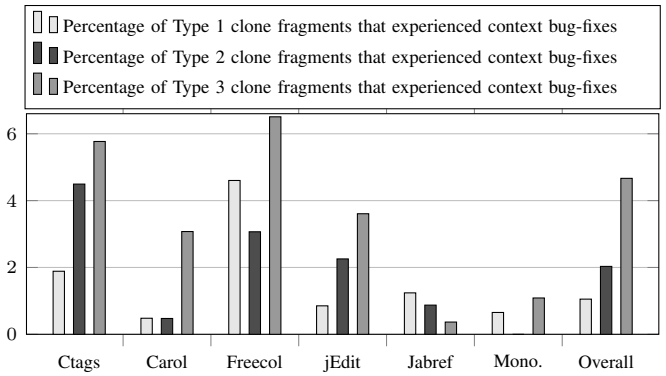

Fig. 7: Percentages of different types of clone fragments that experienced context bug-fixes

Investigation Procedure. For answering RQ 3, we again mine the clone pairs that followed the context bug-fix patterns discussed in Section IV. After identifying these clone pairs for each clone-type of each of our subject systems, we determine which clone fragments experienced the context bug-fixes. From the definition of the patterns we realize that if a clone pair follows a particular context bug-fix pattern then one of the two fragments in the pair contains the context-bug (the other fragment does not). After identifying which clone fragments contain context bugs, we determine the counts of such clone fragments for each clone-type. We then determine what percentage of the code clones in each clone-type contain context-bugs. Table V shows the number of distinct clone-pairs that followed the context bug-fix patterns in each clone-type of our subject systems. This table also shows the number of distinct clone fragments that experienced context bug-fixes. Here, we should note that more than one clone fragment residing in the same clone class might experience fixes for a context bug in a particular bug-fix commit operation.

Fig. 7 shows the percentages of clone fragments in each clone-type that contained context bugs (i.e., that experienced context bug-fixes). According to the figure, Type 3 clones of our subject system Freecol exhibit the highest percentage (6.5%). From Table V we see that among 753 Type 3 clone fragments that were created during the whole period of evolution of Freecol, 49 fragments experienced context bug-fixes. From Fig. 7 we also observe that for most of the subject systems except Jabref, Type 3 clones exhibit the highest percentage. In the case of Jabref, the highest percentage is exhibited by the Type 1 clones. We also determine clone-type wise overall percentages considering all subject systems and show these percentages in Fig. 7. Type 3 clones show the highest overall percentage (around 4.67%) among all clone-types. The percentage regarding Type 1 clones is the lowest.

> **Answer to RQ 3.** According to our investigation and analysis, Type 3 clones generally exhibit the highest tendency of experiencing context bug-fixes during evolution among the three clone-types (Fig. 7).

Our findings imply that a tool for discovering context-bugs should primarily focus on Type 3 clone. Removal of Type 3 clones through refactoring can also help us get rid of context bugs to a considerable extent.

TABLE V. Clone-type centric statistics of context bug-fixes

	Ctags			Carol			Freecol			jEdit			Jabref			MonoOSC		
	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3
CF	53	89	156	416	211	683	239	163	753	7395	399	2689	484	229	1364	153	41	184
CF-CBF	1	4	9	2	1	21	11	5	49	63	9	97	6	2	5	1	0	2
CP-CBFP	1	4	19	2	1	26	12	5	83	69	11	137	6	2	5	2	0	2
CP-CBFP-DF	1	1	15	2	0	15	12	3	72	69	11	133	6	1	4	2	0	1
CP-CBFP-SF	0	3	4	0	1	11	0	2	11	0	0	4	0	1	1	0	0	1

T1 = Type 1 or exact clones T2 = Type 2 or syntactic clones T3 = Type 3 or gapped clones
CF = Total number of clone fragments that were created during the whole period of evolution.
CF-CBF = Number of distinct clone fragments that experienced the context bug-fixes.
CP-CBFP = Number of clone pairs that followed the context bug-fix patterns discussed in Section IV.
CP-CBFP-DF = Number of clone pairs that followed context bug-fix patterns and the two fragments in each pair reside in two different files
CP-CBFP-SF = Number of clone pairs that followed context bug-fix patterns and the two fragments in each pair reside in the same file

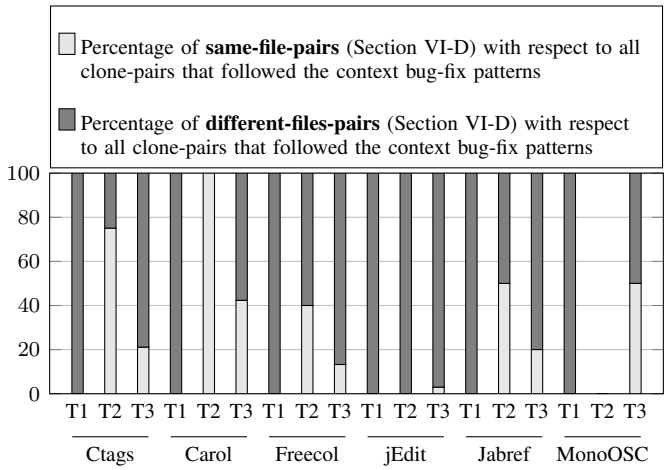

Fig. 8: Percentage of context bug-fix clone pairs containing clone fragments from the same or different files

D. Answering the fourth research question (RQ 4)

RQ 4. *Does cloning within the same file or cloning across different files promote context bugs?*

Rationale. From our answer to RQ 3 we realize that Type 3 clones have the highest possibility of containing context-bugs. However, we still do not know whether cloning within the same file or cloning across different files is likely to introduce context bugs. If a particular cloning operation (cloning within the same file or cloning across different files) seems to be more likely to introduce context bugs, then this information can help us narrow down the search space for a context bug finder. We can emphasize that particular cloning operation when searching possible occurrences of context bugs. We investigate in the following way for answering RQ 4.

Investigation Procedure. As we did in RQ 3, we detect all the clone-pairs that followed the context bug-fix patterns discussed in Section IV. For each clone-pair we determine whether it is a same-file-pair or a different-files-pair. We define these two terms in the following way.

- **Same-file-pair:** A same-file-pair is a clone-pair that followed a context bug-fix pattern (Pattern 1 or Pattern 2) and the two clone fragments in the pair belong to the same source code file.
- **Different-files-pair:** A different-files-pair is a clone-pair that followed a context bug-fix pattern (Pattern 1 or Pattern 2) and the two clone fragments in the pair belong to two different source code files.

Table V shows the number of same-file-pairs and different-files-pairs for each clone type of each of our subject systems. For example, let us consider the Type 3 case of Ctags. According to Table V, 19 Type 3 clone-pairs of Ctags followed the context bug-fix patterns. For 15 of these pairs, the two clone fragments in each pair belong to two different source code files. For each of the remaining 4 pairs, the two clone fragments constituting the pair belong to the same file.

We also determine the percentages of same-file-pairs and different-files-pairs with respect to all clone-pairs that followed the context bug-fix patterns, and show these percentages in the stacked bar-graph of Fig. 8 for each clone-type of each subject system. The figure shows seventeen bars in total. For Type 2 case of MonoOSC, we did not get any clone-pair that followed any of the context bug-fix patterns. The figure does not contain a bar for this case. For most of the other cases (i.e., for 13 out of 17 cases), the percentage of different-files-pairs is higher than the percentage of same-file-pairs. By considering all clone-types of all subject systems we find that the overall percentages of different-files-pairs and same-file-pairs are 89.92% and 10.08% respectively.

Statistical significance test regarding comparing the percentages of same-file-pairs and different-files-pairs. We wanted to investigate whether the percentages regarding different-files-pair are significantly higher than the percentages regarding same-file-pairs. For this purpose we again perform Wilcoxon Signed Rank test [52], [53] considering the percentages for the seventeen cases plotted in Fig. 8. As we did before, we conduct the test considering a significance level of 5%. According to our two-tailed test result, the percentages regarding different-files-pairs are significantly different than the percentages regarding same-file-pairs with a p-value of 0.009 which is less than 0.05. We thus conclude that the percentages regarding different-files-pairs are significantly higher than the percentages regarding same-file-pairs.

Answer to RQ 4. According to our analysis of experiment results, the possibility that a clone-pair that followed a context bug-fix pattern will consist of clone fragments belonging to two different files is significantly higher compared to the possibility that a clone-pair that followed a context bug-fix pattern will consist of clone fragments belonging to the same source code file. According to our

subject systems, 89.92% of the clone-pairs that followed the context bug-fix patterns consisted of clone fragments belonging to different source code files.

Our finding implies that cloning across different source code files is more risky than cloning within the same file. When mining context bugs in code clones, we should primarily focus on those cases where cloning was done across different files.

VII. Discussion regarding the findings

This section summarizes the implications of our findings from all the research questions. Our finding from **RQ 1** implies that context bugs often get introduced to the code-base through code cloning during evolution. Thus, this is important to take measures so that we can minimize context bugs in code clones. Our findings from **RQ 2** and **RQ 4** make us aware of two cloning operations that are highly likely to introduce context bugs. These operations are:

- *Making copies of a newly created code fragment (i.e., a code fragment that was not added in an earlier revision)*. From the development perspective, this operation resembles copying an uncommitted code fragment.
- *Copy/pasting across different files*.

Programmers should avoid these operations whenever possible. A context bug discovery tool can primarily focus on these cloning operations when finding possible occurrences of context bugs. Our answer to third research question (**RQ 3**) implies that Type 3 clones should be prioritized for refactoring. As Type 3 clones have the highest possibility of containing context bugs, minimizing the number of Type 3 clones through refactoring can help us minimize context bugs.

VIII. Threats to Validity

We used the NiCad clone detector [24] for detecting clones. For different settings of NiCad, the statistics that we present in this paper might be different. Wang et al. [51] defined this problem as the *confounding configuration choice problem* and conducted an empirical study to ameliorate the effects of the problem. However, the settings that we have used for NiCad are considered standard [7] and with these settings NiCad can detect clones with high precision and recall [5], [8], [25]. Thus, we believe that our findings on context bugs in 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. [29]. Such a technique proposed by Mocus and Votta [3] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [29] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

Different types of bug-fixes might occur in a code-base. However, our goal is to identify cases where a clone fragment in a code-base experienced a bug-fix because it was not properly adapted to its context at the time of its creation through copy/pasting. Identification of such cases is challenging. An automatic tool for this currently does not exist. However, we defined the patterns emphasizing the evolutionary phenomenons that are likely to occur while fixing a context-bug in a clone fragment. Thus, our findings regarding context-bugs in code clones are important.

In our experiment we investigate six open-source subject systems consisting of thousands of revisions. While these systems are not enough to generalize our findings regarding context bugs in code clones, we select these systems focusing on the diversity of their application domains, sizes, implementation languages (three languages: Java, C, and C#), and revision history lengths. Thus, our findings cannot be attributed to a chance. Our findings regarding context-bugs can be important from the perspectives of clone management.

IX. Conclusion

In this paper, we investigate context adaptation bugs (i.e., context-bugs) in three types of code clones. Context-bugs are those bugs that get introduced to the clone fragments because of not properly adapting those fragments to their respective contexts. Ours is the first study to investigate context bugs in code clones. We define and automatically detect two bug-fix patterns such that the patterns indicate fixing of context-bugs in clone fragments. We mine these patterns from the clone evolutionary history consisting of thousands of revisions of six open-source subject systems written in three different programming languages. We analyze these patterns in three clone-types and find that around 50% of the bug-fixes experienced by code clones can occur for fixing context-bugs. Type 3 clones have the highest possibility of experiencing context bug fixes among the three clone-types (Type 1, Type 2, and Type 3). We also find that cloning (i.e., copy/pasting) a newly added code fragment has a significantly higher likeliness of introducing context-bugs compared to cloning a preexisting code fragment. Moreover, cloning across different files is more likely to introduce context-bugs compared to cloning within the same file. We suggest programmers to be careful about the risky cloning operations (cloning a newly created code fragment, cloning across different source code files) during development. Avoiding such operations can help them minimize context-bugs in code clones. Our findings should be taken into account when developing a tool for automatically identifying code clones that are likely to contain context-bugs. As Type 3 clones have the highest likeliness of containing context bugs among the three clone-types, we suggest to prioritize refactoring of Type 3 clones when making clone refactoring decisions. Removal of Type 3 clones through refactoring can help us minimize context-bugs. As a future work, we plan to investigate context bugs in micro-clones. Our implementation and data are available on-line [55].

Acknowledgement

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by two Canada First Research Excellence Fund (CFREF) grants coordinated by the Global Institute for Food Security (GIFS) and the Global Institute for Water Security (GIWS).

REFERENCES

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

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

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

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

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

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

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

[17] 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.

[18] H. H. Mui, A. Zaidman, M. Pinzger, "Studying Late Propagations in Code Clone Evolution Using Software Repository Mining", *Electronic Communications of the EASST*, 2014, 63(2014):1 - 12.

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

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

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

[26] 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.

[27] 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.

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

[29] 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.

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

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

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

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

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

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

[36] 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.

[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, "Automatic Identification of Important Clones for Refactoring and Tracking", Proc. *SCAM*, 2014, pp. 11 – 20.

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

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

[41] 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.

[42] 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.

[43] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, A. E.Hassan, "An empirical study on inconsistent changes to code clones at the release level", *Science of Computer Programming Journal*, 2012, 77(6): 760 – 776.

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

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

[46] N. Tsantalis, D. Mazinanian, G. P. Krishnan, "Assessing the Refactorability of Software Clones", *IEEE Transactions on Software Engineering*, 41(11):1055 – 1090.

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

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

[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] 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.

[51] 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.

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

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

[54] 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.

[55] Implementation and data regarding our research on context adaptation bugs: https://drive.google.com/open?id=1228I4Bomlv8HRdHn1VRDBNWvm86NwgpR.