Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2013
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1597

An empirical study of faults in late propagation clone genealogies

Liliane Barbour[1], Foutse Khomh[2,*,†] and Ying Zou[1]

[1]*Department of Electrical and Computer Engineering, Queen's University, Kingston, ON Canada*
[2]*SWAT, École Polytechnique de Montréal, Québec, Canada*

SUMMARY

Two similar code segments, or clones, form a clone pair within a software system. The changes to the clones over time create a clone evolution history. In this work, we study late propagation, a specific pattern of clone evolution. In late propagation, one clone in a clone pair is modified, causing the clone pair to diverge. The code segments are then reconciled in a later commit. Existing work has established late propagation as a clone evolution pattern and suggested that the pattern is related to a high number of faults. In this study, we examine the characteristics of late propagation in three long-lived software systems using the SIMIAN (Simon Harris, Victoria, Australia, http://www.harukizaemon.com/simian), CCFINDER, and NICAD (Software Technology Laboratory, Queen's University, Kingston, ON, Canada) clone detection tools. We define eight types of late propagation and compare them to other forms of clone evolution. Our results not only verify that late propagation is more harmful to software systems but also establish that some specific types of late propagations are more harmful than others. Specifically, two types are most risky: (1) when a clone experiences diverging changes and then a reconciling change without any modification to the other clone in a clone pair; and (2) when two clones undergo a diverging modification followed by a reconciling change that modifies both the clones in a clone pair. We also observe that the reconciliation in the former case is more prone to faults than in the latter case. We determine that the size of the clones experiencing late propagation has an effect on the fault proneness of specific types of late propagation genealogies. Lastly, we cannot report a correlation between the delay of the propagation of changes and its faults, as the fault proneness of each delay period is system dependent. Copyright © 2013 John Wiley & Sons, Ltd.

Received 12 February 2012; Revised 21 January 2013; Accepted 15 March 2013

KEY WORDS: clone genealogies; late propagation; fault proneness

1. INTRODUCTION

A code segment is labeled as a code clone if it is identical or highly similar to another code segment. Two similar code segments form a clone pair. Groups of clone pairs are known as 'clone classes'. Clones can be introduced into systems deliberately (e.g., 'copy and paste' actions) or inadvertently by a developer during development and maintenance activities. Like all code segments, code clones are not immune to change. Large software systems undergo thousands of commits over their life cycles. Each commit can involve modifications to code clones. As the clones are modified, a change evolution history, known as a clone genealogy [1], is generated. In this paper, we study the change evolution history of clone pairs.

Two types of evolutionary changes can affect a clone pair: a diverging change or a reconciling change. A change is reconciling if after the change the clone pair relationship exists, whether or not the clone pair existed before the change. In a diverging change, one or both of the clones evolves independently, destroying the clone pair relationship. Diverging changes can occur deliberately, such as when code is copied and pasted and then subsequently modified to fit the

*Correspondence to: Foutse Khomh, SWAT, École Polytechnique de Montréal, Québec, Canada.
†E-mail: foutse.khomh@polymtl.ca

new context. For example, if a driver is required for a new printer model, a developer could copy the driver code from an older printer model and then modify it. Diverging changes can also occur accidentally. A developer may be unaware of a clone pair and cause a divergence by changing only one half of the pair. This diverging change can result in a software fault. If a fault is found in one clone and fixed, but not propagated to the other clone in the clone pair, the fault remains in the system. For example, a fault might be found in the old printer driver code and fixed, but the fix is not propagated to the new printer driver. For these reasons, a previous study [1] has argued that accidental diverging changes make code clones more prone to faults.

Late propagation occurs when a clone pair undergoes one or more diverging changes followed by a reconciling change [2]. The reconciliation of the code clones indicates that the divergence was accidental. Because accidental changes are considered risky [3], the presence of late propagation in clone genealogies can be an indicator of risky, fault-prone code.

Many studies have been performed on the evolution of clones. A few (e.g., [2, 3]) have studied late propagation and indicated that late propagation genealogies are more fault-prone than other clone genealogies. Thummalapenta *et al*. [3] began the initial work in examining the characteristics of late propagation. The authors considered the delay between a diverging change and a reconciling change and related the delay to software faults. In our work, we examine more characteristics of late propagation to determine if only a subset of late propagation genealogies are at risk of faults. In our case study, we found that late propagation genealogies account for between 8–21% of all clone genealogies that experience at least one change. If all late propagation genealogies are considered equally prone to faults, this means that as much as a fifth of all genealogies must be monitored for faults, which is resource intensive. Developers are interested in identifying which clones are most at risk of faults. Our goal is to support developers in their allocation of limited code testing and review resources towards the most risky late propagation genealogies. Most previous clone evolution studies examine groups of clone pairs known as 'clone classes'. However, clone pairs within the same clone group are not equally fault prone. By studying clone pairs, we identify the most risky clone pairs within each clone class.

In this paper, we extend our previous work published at the 27th IEEE International Conference on Software Maintenance [4], titled 'Late Propagation in Software Clones' [4], which studies the characteristics of late propagation genealogies and estimates the likelihood of faults in two software systems. In addition to adding a new subject system and clone detection tool, in this paper, we address three other questions that investigate the following three relationships: (1) the effect of the size of cloned code on the fault proneness of late propagation genealogies; (2) the impact of the time interval between a diverging change and a reconciling change on fault proneness; and (3) the fault proneness of reconciling changes in late propagation genealogies. Using clone genealogies from three open-source software systems, that is, ArgoUML (Jason E. Robbins, California, USA),[‡] Ant (The Apache Software Foundation, Forest Hill, MD, USA),[§] and JBoss (Red Hat, Raleigh, NC, USA),[¶] we address the following six research questions:

- *RQ1: Are there different types of late propagation?* Late propagation has been defined by several researchers [2, 5] as a diverging change followed by a reconciling change. We perform an exploratory study to examine several late propagation patterns and investigate whether diverged clone pairs are ever reconciled without a change propagation occurring.
- *RQ2: Are some types of late propagation more fault prone than others?* Previous researchers have determined that late propagation is more prone to faults than other clone genealogies [2]. Using the classification of late propagation clone genealogies described in Section 3, we evaluate late propagation in greater depth and examine if the risk of faults is the same for all types of late propagation.

[‡]http://argouml.tigris.org/
[§]http://ant.apache.org/
[¶]http://www.jboss.org/

DOI: 10.1002/smr

- *RQ3: Which type of late propagation experiences the highest proportion of faults?* In the previous question, we determine if some types of late propagation are more prone to faults than others. For this question, we examine the overall number of faults across each late propagation type to determine which type of late propagation is responsible for the most faults.
- *RQ4: Does the size of cloned code affect the fault proneness of late propagation genealogies?* We want to determine if for late propagation types, the size of cloned code (in lines of code (LOC)) has an impact on the fault proneness. It is expected that a smaller clone will be less complex and less prone to faults.
- *RQ5: For a clone pair experiencing late propagation, does the time interval between diverging and reconciling changes affect the fault proneness?* In a previous study on clone genealogies, Thummalapenta *et al*. [3] examined the fault proneness of late propagation. The authors checked clone genealogies that were reconciled within 1 day, separately from other late propagation genealogies. Overall, they found that both experienced a higher proportion of faults compared with other types of clone genealogies. In this question, we examine late propagation delay at a finer level of granularity and determine if the time interval between the divergence and reconciliation phases influences the fault proneness of late propagation clones. It is believed that a long time interval between changes will lead a developer to become unfamiliar with the code, causing an increase in the number of faults.
- *RQ6: Are late propagation types reconciled because of fault-fixing activities?* In this question, we investigate if reconciling changes on late propagation clone pairs are fault-fixing changes.

The results of this study can be used to identify clone pairs at risk of faults. This helps determine where testing and review efforts should be focused.

The rest of this paper is organized as follows. Section 2 summarizes related studies on clones and late propagation. Section 3 discusses different types of late propagation. Section 4 explains the design of our study. Section 5 describes the study results. Section 6 lists threats to the validity of the study. Finally, Section 7 concludes the paper and explores future work.

2. RELATED WORK

2.1. Definition of clone genealogies

The first study on code clone evolution was by Kim *et al*. [1] who analyzed clone classes (i.e., a group of similar clone pairs) and defined patterns of clone evolution. Through a case study on two Java systems using the CCFINDER clone detection tool, they observed that the majority of clones in systems are very volatile, with at least half of the clones being eliminated within eight check-ins after their creation. They stressed the need for a better understanding of clone genealogies to better support code clones. Our work strives for a deeper understanding of one specific type of clone genealogy, late propagation, to help developers efficiently focus their maintenance efforts.

2.2. Analysis of clone genealogies

Several studies have analyzed clone genealogies. They focus on the reconciling and diverging changes experienced by clones.

Krinke [6] performed a study on five open-source systems to examine reconciling and diverging changes to code clones. He observed the systems over a 200-week period, using a time interval of 1 week between system snapshots. He used the SIMIAN clone detection tool, but examined only identical clones. He found that clone pairs experience reconciling changes about half of the time and that late propagation occurs very infrequently. He also found that during late propagation, the reconciling change usually occurred within a week of the diverging change. This observation, coupled with the fixed time interval between system snapshots (1 week) suggests that a more fine-grained time interval is necessary to fully understand late propagation. In our study, we use a time interval of one commit, so all changes to each clone are considered when generating clone genealogies.

DOI: 10.1002/smr

Saha *et al*. [7] looked at 17 open-source systems written in four programming languages and performed an empirical study of clone genealogies at the release level. Overall, they found that around 67% of clones were unchanged across releases. They also found that a majority of clones still remained in the system through the final release. When studying reconciling changes to clones, they found that on average, 24% of changes reconciled the clones.

Göde *et al*. [8] repeated and extended another of Krinke's studies [9] on the stability of cloned code. Similar to our work, they examined clones at the interval of one commit. They used a token-based clone detection tool and experimented with different clone lengths. Overall, they confirmed Krinke's findings that cloned code is more stable than non-cloned code. They also confirmed that cloned code experiences more changes involving code deletion than non-cloned code. Göde *et al*. found that varying the parameters of the clone detection tool can significantly influence the results. To mitigate this risk in our work, we use three different clone detection tools.

Göde *et al*. [10] studied changes to code clones. Looking at three subject systems, they found that over half of the clones were never modified once a pair was formed. Only about 12% of the clones experienced more than one change. They concluded that these clones were the most relevant for developers, because they required additional maintenance effort. This stresses the importance of understanding the behavior of late propagation genealogies, because of their large number of changes. In our work, we study late propagation in more detail to identify precisely which clones are most relevant for maintenance purposes.

Göde *et al*. [11] examined consecutive changes to code clones. They identify four different patterns of consecutive changes, consisting of combinations of reconciling and diverging changes. In a study of three subject systems, they concluded that the majority of clones never experienced more than one change, if they changed at all. They also concluded that the majority of diverging changes in clones were intentional. In a separate study on identical clones, Göde [5] determined that the ratio of reconciling to diverging changes was system dependent. However, overall most diverging changes never experienced a reconciling change, so late propagation was rare in identical clones.

Overall, these studies show that although many clones are stagnant, some clones experienced further changes after they are created. In our work, we focus on clones that experience late propagation, which means that they undergo multiple changes, both reconciling and diverging. We suggest that due to the higher amount of churn experienced by these clones, they are more susceptible to faults.

2.3. Faults in clones

Static information about clones has been used to identify faulty clones in software systems. Jiang *et al*. [12] examine the context of clones to locate faults. They assume that when code is copied and pasted into a new context, faults can be introduced when the code is not properly modified to suit the new context. They validated their approach using LINUX and ECLIPSE, and showed that the context of a clone could be used to locate faults in a software system. Their work is limited to identifying clones caused by context-related issues. Similarly, Li *et al*. [13] used their clone detection tool, CP-Miner, to detect faults in software systems. Their tool located inconsistently renamed identifiers in clones. In a case study, they were able to identify 49 faults in a version of LINUX, and 32 faults in FREEBSD, many of which were previously unreported. In our work, we examine if dynamic clone information can be used to identify fault-prone late propagation genealogies.

2.4. Faults in clone genealogies

Studies have examined clone genealogies, which considers the history of clones. Bakota *et al*. [14] argued that to isolate risky clones, clones should be seen as dynamic instead of static. Clones that experience many changes during their evolution may help locate faults in the system. They presented several cases where the analysis of changes to clones highlighted risky clones. For example, they found that clones that evolved independently decreased the maintainability of the system due to the lost connection between the clones. Bakota *et al*. performed a case study on 12 revisions of Mozilla Firefox. They found that the use of evolutionary information can be successful in locating faults in the system. In our work, we examine the evolutionary characteristics of late propagation genealogies to identify the most fault-prone late propagation genealogies.

DOI: 10.1002/smr

Thummalapenta *et al*. [3] performed a study on four open-source C and Java systems, including ARGOUML. They found that late propagation occurred in a maximum of 16% of code clone genealogies. They also observed that clones exhibiting late propagation were more prone to faults, concluding that late propagation was a risky cloning behavior. Aversano *et al*. [2] also showed that late propagation was a risky clone genealogy and occurred in about 18% of all clones. Thummalapenta *et al*. also defined a specific type of late propagation, called delayed propagation, which occurs when the reconciling change is made within 24 hours of a diverging modification. In our work, we extend this idea. We examine more periods of delay between a diverging and reconciling change to determine if specific periods of delay are more prone to faults.

Bettenburg *et al*. [15] argue that clones should be analyzed at the release level. They suggested that clones are highly volatile, so we should examine only clones that affect the end user. In a study of two open-source software systems, they found that only 1–3% of diverging changes caused faults at the release level. However, in a study of diverging changes in three industrial software systems, Juergens *et al*. [16] found that 23% of diverged clones contained a fault. In this paper, we examine late propagation genealogies, which experienced diverging changes. We determine if specific combinations of changes to one or more clones in a clone pair make the pair more prone to faults.

3. CLASSIFICATION OF LATE PROPAGATION GENEALOGIES

In the current state of the art, late propagation is defined as a clone pair that experiences one or more diverging changes followed by a reconciling change [3]. For example, consider two clones that call a method. A developer modifies the actual parameters of the method call and updates one of the clones to reflect the change. This causes the clone pair to diverge. Later, she discovers the divergence, possibly because of a bug report, and propagates the change to the other clone. The clones are now reconciled.

We define a reconciling change as a change that results in the existence of a clone pair relationship so that the clones belong to the same clone class [2, 3]. The relationship may or may not exist immediately prior to the reconciling change. Thummalapenta [3] more specifically defines a reconciling change as a situation where the resulting clones do not differ by more than a threshold of the textual and functional similarity of the code segments. In our work, we use the threshold of similarity of type 1 and type 2 clones. Two code segments that are textually identical except for variations in whitespace, layout and comments are considered to be type 1 clones. When code segments are syntactically identical except for variations in identifiers, literals, types, whitespace, layout, and comments, they are considered to be type 2 clones. Therefore, a reconciling change to each clone is not necessarily an identical change. The clones can differ because of variable renaming or different literals.

A diverging change destroys the cloning relationship, so that one or both of the clones no longer belong to the containing clone class [2]. In other words, they differ by more than a fixed threshold [3]. However, if one or both clones that make up a clone pair are deleted, the genealogy terminates. We do not define a deletion of the clone pair as a diverging change.

We analyze all the possible sequences of late propagation based on the following three phases:

- Clones modified in diverging change: either one or both of the clones is modified independently, causing the divergence.
- Clones modified during period of divergence: one, both, or neither of the clones experiences changes so that the fragments are still different
- Clones modified during reconciling change: either one or both of the clones is modified, reconciling the clone pair.

Using all combinations of the three phases, we identify eight possible types of late propagation genealogies. All late propagation genealogies can be classified as one of the eight types. The characteristics of the individual types are described in Table I. The clones in the table are interchangeable, so the patterns A-AB-B and B-BA-A are both classified as LP2.

DOI: 10.1002/smr

Table I. Description of late propagation types for clone pair A and B.

Propagation category	LP type	Clones modified in diverging change	Clones modified during period of divergence	Clones modified in reconciling change
Propagation always occurs	LP1	A	A	B
		A	—	B
	LP2	A	B	B
		A	A and B	B
	LP3	A	—	A and B
		A	A	A and B
Propagation may or may not occur	LP4	A	B	A
		A	A and B	A
	LP5	A	B	A and B
		A	A and B	A and B
	LP6	A and B	—	A
		A and B	A	A
		A and B	B	A
		A and B	A and B	A
	LP7	A and B	—	A and B
		A and B	A	A and B
		A and B	A and B	A and B
Propagation never occurs	LP8	A	—	A
		A	A	A

LP, late propagation.

Because the late propagation types are not extracted from existing systems, the types may not appear in all the systems in our study. It describes the possible sets of modifications to two clones, Clones A and B, in a clone pair. We divide the types of late propagation into three categories:

1. A propagation always occurs.
2. A propagation may or may not occur.
3. A propagation never occurs.

A propagation occurs when changes from one clone are applied to the other clone in a clone pair. We observe that late propagation does not necessarily involve any propagation when the reconciling change is a reverting change. In this study, we consider this factor and examine if the cases that always involve propagation (i.e., LP1, LP2, and LP3) or never involve propagation (i.e., LP8) are more prone to faults than other types of late propagation.

```
//Clone A, Commit 595
addField(new UMLComboBox(typeModel),1,0,0);

//Clone B, Commit 595
addField(new UMLComboBox(classifierModel),2,0,0);

//Diverging Change: Clone A, Commit 602
addField(new UMLComboBoxNavigator(this,"NavClass",
  new UMLComboBox(typeModel)),1,0,0);

//Reconciling Change: Clone B, Commit 604
addField(new UMLComboBoxNavigator(this,"NavClass",
  new UMLComboBox(classifierModel)),2,0,0);
```

Listing 1: An example of a LP1 genealogy from ARGOUML

As listed in Table I, LP1, LP2, and LP3 belong to the first category, because a change must be always propagated between the clones in the clone pair to reconcile them. For example, in LP1, Clone A is modified, diverging Clones A and B. Clone A can experience further changes during the period of divergence. Finally, all changes are propagated to Clone B, reconciling the clone pair. Listing 1 is an example of an LP1 genealogy taken from ARGOUML using CCFINDER as the clone

DOI: 10.1002/smr

detection tool. As shown in Listing 1, two clones (i.e., Clones A and B) form a clone pair in commit 595. In commit 602, the parameter in the method call is updated, modifying Clone A. In commit 604, this change is propagated to Clone B, reconciling the clone pair.

Listing 2 is an example of an LP2 genealogy. Both clones are reconciled in commit 274,858. In commit 306,781, Clone A is modified so that the parameters are changed in two method calls. The clones are now diverged. While the clones are diverged, they both undergo an additional change in commit 432,728 that adds an additional line of code to both clones ('Throwable cause = null;'). However, the clones remain diverged. Finally, in commit 551,986, the parameter change in Clone A is propagated to Clone B, reconciling the clone pair.

```
//Clone A, Commit 274858
if (p == null) {
    systemDefault = System.getProperty("ant.regexp.regexpimpl");
} else {
    systemDefault = p.getProperty("ant.regexp.regexpimpl");
}
if (systemDefault != null) {
    return createInstance(systemDefault);
}
//Clone B, Commit 274858
if (p == null) {
    systemDefault = System.getProperty("ant.regexp.regexpimpl");
} else {
    systemDefault = p.getProperty("ant.regexp.regexpimpl");
}
if (systemDefault != null) {
    return createRegexpInstance(systemDefault);
}
//Diverging Change: Clone A, Commit 306781
if (p == null) {
    systemDefault = System.getProperty(MagicNames.REGEXP_IMPL);
} else {
    systemDefault = p.getProperty(MagicNames.REGEXP_IMPL);
}
if (systemDefault != null) {
    return createInstance(systemDefault);
}
//Change to Both Clone A and B During Period of Divergence, Commit 432728
//Clone A
if (p == null) {
    systemDefault = System.getProperty(MagicNames.REGEXP_IMPL);
} else {
    systemDefault = p.getProperty(MagicNames.REGEXP_IMPL);
}
if (systemDefault != null) {
    return createInstance(systemDefault);
}
Throwable cause = Null;

//Clone B
if (p == null) {
    systemDefault = System.getProperty("ant.regexp.regexpimpl");
} else {
    systemDefault = p.getProperty("ant.regexp.regexpimpl");
}
if (systemDefault != null) {
    return createRegexpInstance(systemDefault);
}
Throwable cause = Null;

//Reconciling Change: Clone B, Commit 551986
if (p == null) {
    systemDefault = System.getProperty(MagicNames.REGEXP_IMPL);
} else {
    systemDefault = p.getProperty(MagicNames.REGEXP_IMPL);
}
if (systemDefault != null) {
    return createRegexpInstance(systemDefault);
}
Throwable cause = Null;
```

Listing 2: An example of a LP2 genealogy from ANT

LP8 is the only clone type in the third category, as shown in Table I. In LP8, Clone A is modified, diverging the clone pair. The change is later reverted, reconciling the clone pair. Listing 3 is an example of an LP8 genealogy taken from ANT using CCFINDER as the clone detection tool. As shown in listing 3, two clones (i.e., Clones A and B) form a clone pair in commit 270,250. In commit 270,264, Clone A is modified so that the string 'm_' is added to the beginning of each variable name. In commit 271,109, this change is reverted, reconciling the clone pair. For space reasons, the listings discussed in this section contain only the interesting lines of code extracted from bigger clones.

DOI: 10.1002/smr

```
//Clone A, Commit 270250
if( destFile == null )
{
  destFile = new File( destDir, file.getName() );
}
//Clone B, Commit 270250
if (destFile == null) {
  destFile = new File(destDir, file.getName());
}
//Diverging Change: Clone A, Commit 270264
if( m_destFile == null )
{
  m_destFile = new File( m_destDir, m_file.getName() );
}
//Reconciling Change: Clone A, Commit 271109
if (destFile == null) {
  destFile = new File(destDir, file.getName());}
```

Listing 3: an example of a LP8 genealogy from ANT

In the second category, changes to one clone may or may not be propagated between the clones. In LP4 and LP5 shown in Table I, only one clone is modified during the initial diverging change. In LP6 and LP7, also shown in Table I, both Clones A and B are modified during the diverging change. For all four types, both Clones A and B experience diverging changes, so it is possible that changes are propagated in both directions. For example, in LP4 without propagation, during the period of divergence, partially reconciling changes could be applied to both clones, but because of the initial diverging change, they remain diverged. If the initial diverging change is reverted on Clone A, the clones are reconciled without a propagation occurring. However, if the period of divergence experiences bidirectional propagations between Clones A and B followed by a reconciling propagation from Clones B to A, LP4 is said to experience propagation. Therefore, because of changes experienced by both clones during the period of divergence, it is unclear if propagation occurs for all types in the second category.

For the remainder of this paper, we use the abbreviations 'LP' for late propagation and 'Gen' for genealogy.

4. EXPERIMENTAL SETUP

The *goal* of our study is to investigate the fault proneness of clone pairs that undergo LP. The *quality focus* is the increase in maintenance effort and cost due to the presence of late propagated clone pairs in software systems. The *perspective* is that of researchers, interested in studying the effects of LP on clone pairs. The results may also be of interest to developers, who perform development or maintenance activities. The results will provide insight in deciding which code segments are most at risk for faults and in prioritizing the code for testing.

The *context* of this study consists of the change history of three software systems, ARGOUML, APACHE ANT, and JBOSS, which have different sizes and belong to different domains.

ARGOUML is a UML-modeling application that supports forward and reverse code engineering activities. It provides a user with a set of views and tools to model systems using UML diagrams, to generate the corresponding code skeletons and to reverse engineer diagrams from existing code. The project started in January 1998 and is still active. It has over 3.1M LOC and 18k commits in its software repository. We consider an interval of observation ranging from January 1998 to November 2010. ARGOUML has been used in previous studies on code clone evolution [2].

APACHE ANT is a Java library and tool that enables the user to compile, assemble, test and run Java, C, and C++ applications. The project started in January 2000 and is currently active. It has over 2.3M LOC and 1.0M commits in its commit history. We study code snapshots in an interval of observation ranging from January 2000 to November 2010.

JBOSS Application Server (or JBOSS) is an open-source Java EE-based application server, which is usable on any operating system that supports Java. The project started in 1999 and is still active. It has over 1.7M LOC and 110K commits in its software repository. We consider an interval of observation ranging from April 2000 to December 2010.

DOI: 10.1002/smr

Table II. Characteristics of the systems.

System	# LOC	# Commits	# Gen CCFinder	# LP CCFinder	# Gen Simian	# LP Simian	# Gen NiCad	# LP NiCad
Ant	2.3M	1.0M	30k	6k	461	103	3616	832
ArgoUML	3.1M	18k	14k	2k	111	23	4123	833
JBoss	1.6M	109k	59k	2k	771	12	152	20

LOC, lines of code; Gen, genealogy; LP, late propagation.

Clone detection is performed using three existing clone detection tools, Simian,[‖] CCFinder[17], and NiCad[18]. Simian is a string-based clone detection tool. For Simian we select a minimum clone length of five LOC. Simian outputs a list of clones where each clone is identified by its location (i.e., start and end line numbers) in a specific file. It identifies clones between code fragments that are identical (i.e., type 1 clones) or where some identifier names and literals have been changed (i.e., type 2 clones). CCFinder is a token-based clone detection tool with a default minimum clone length of 50 tokens. Like Simian, it identifies types 1 and 2 clones. Additionally, due to a normalization step during preprocessing of the code, it identifies some clones with gaps [17]. We use the most recent versions of CCFinder, CCFinderX and NiCad, NiCad3. NiCad uses a hybrid approach to detect clones. We use the default settings of NiCad3 to detect clones greater than 10 LOC. We detect identical clones and clones where the variable names are different.

We select these three clone detection tools because they are fast and consume very little memory. Abstract syntax-based tools, such as CloneDR,** require extensive memory and computation powers. Thus, they have limitations when scanning the entire history of a system.

Although other studies [6, 8] have investigated the impact of the minimum clone length on the percentage of reconciling changes between clones in software systems, their results may be influenced by their choice of clone detection tool. To allow our study to be replicated using any clone detection tool, we use the default settings for both Simian and CCFinder. To build the clone genealogies for our experiment, we require the line numbers of each clone. The CCFinder output describes a clone by its start and end token numbers within a file, so we post-process the results to map the token numbers to the line numbers. We use a tool that analyzes the token files created during CCFinder's tokenizing step, minimizing any distortion when mapping tokens to line numbers.

The validity of the genealogies is based on the precision of the clone detection tool. We repeat our study using two clone detection tools that use different clone detection techniques. This increases the validity of our study.

Table II reports descriptive statistics on the three systems. The size of each system is reported as the cumulative number of LOC across all versions of Java files within each system's Subversion (SVN). In this work, we study all clone genealogies that contain at least one reconciling modification beyond the creation of the clone pair. We filter out clones that diverge immediately and remain diverged for the remainder of their genealogy because they are likely to be false positive clones.

4.1. Data extraction

This section gives an overview of our approach to collect and process clone data to build clone genealogies. Figure 1 shows an overview of our approach. First, we use the tool J-Rex [19] to mine the code repository of each subject system. J-Rex identifies the commits that modify each Java file and outputs a snapshot of the file at those commits. Commits corresponding to fault fixes are marked during the process. Next, clone detection is executed to detect clones in the entire subject system. By using the clone detection results, the clones are mapped across their commits to create

[‖]http://www.harukizaemon.com/simian/

**http://www.semdesigns.com/Products/Clone/

DOI: 10.1002/smr

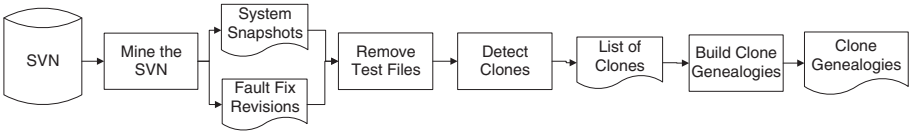

Figure 1. Overview of the analysis process.

clone genealogies. Each clone genealogy is examined to identify instances of LP. Lastly, the LP genealogies are categorized by the types of LP. We now describe each of the major steps in detail.

4.1.1. Mining Subversion to identify faults. We use J-REX [19] to identify fault fixes within the clone genealogies. J-REX enables source code extraction, evolutionary analysis, and fault fix identification. To perform source code extraction and evolutionary analysis, J-REX first extracts a snapshot of the subject system at each commit. It then breaks each snapshot into component methods and flags any methods that have been modified since the last commit.

J-REX analyzes each commit message for Java systems to identify the reason for a commit, such as a fault fix. It performs the analysis using the heuristics proposed by Mockus *et al*. [20] and used in prior fault studies [21, 22]. For example, if a commit message contains the word 'bug', it is classified as a fault fix. Using heuristics can lead to false positive 'faulty' commits. For example, in ARGOUML, commit number 828 has the commit message 'Removed debugging line'. J-REX would misclassify this commit as a fault fix because of the word 'debugging'. We could not remove all the false positives from our study because of the large number of commit messages. To provide a context for our results, we performed a statistical sampling to estimate the accuracy of J-REX. We manually evaluated a sample of commit messages from the three subject systems based on a confidence level of 95% with a confidence interval of 5. This translates to between 377 and 384 samples per system. To avoid a bias of the J-REX results from a single evaluator, we recruited four graduate students to manually evaluate the samples. All four students had experience using an SVN system and had industrial software development experience. The authors reviewed the evaluations of the students. Overall, the accuracy of J-REX was found to be 87%. Measuring precision only gives an indication of how many commits identified as faulty are actually faulty. We determine the accuracy of J-REX to also evaluate how many non-faulty commits are correctly identified as non-faulty. The accuracy is similar to a finding by Hassan in an evaluation of C-REX [23], which analyzes the commit messages for C systems. J-REX is based on C-REX and uses the same approach and heuristics to identify fault-fixing commits. Hassan compared the output of C-REX to a manual evaluation of commit messages by six professional developers from industry and found that the inter-developer correlation was as high as the correlation between C-REX and the developers ($\sigma > 0.8$). These results show that C-REX's ability to recognize fault fixes is comparable with that of a professional developer. We assume that J-REX has a similar performance to C-REX because both tools use the same set of heuristics. Although J-REX will misclassify some of the commits, because all our data is extracted from the SVN repositories using J-REX, the accuracy of J-REX will affect the experimental and control groups of our experiments equally.

Existing studies build clone genealogies between system snapshots taken at fixed intervals (e.g., 1 week). The interval chosen can affect the creation of clone genealogies because any changes that occur between system snapshots are lost. Therefore, we examine clones between each commit, the minimum interval obtainable from an SVN repository.

Due to hardware limitations and the large number of clones in ARGOUML, for CCFINDER we limit our study of ARGOUML to the period before release 0.12 (October 2002), or the first 2576 commits.

4.1.2. Removing test files. All our subject systems contain files that are not used during normal execution of the system. Such files are used during the development of the system to test the different functionalities. By their nature, they can contain incomplete and even syntactically incorrect code to test the failure modes of the system. Because test files are frequently copied and modified to test a different case, they can contain many clones. Therefore, we remove the test files from our study. To remove the test files, we perform a search on each system for files and folders

DOI: 10.1002/smr

with a filename containing the word 'test'. We then manually verify each file before removing it from the study to prevent the automatic removal of a non-test file, such as a file with the name 'upda**teSt**ate.java'.

4.1.3. Detecting clones. We detect all of the clones in our subject systems. To build the clone genealogies, we are interested in clones within the same commit of a software system. To identify the clones, we first perform clone detection on the entire system. We then post-process the results to identify any clones that co-exist within the same commit. Further details about our approach can be found in our study on the impact of software clones [24].

In the first step, we perform clone detection on all the Java file snapshots from the source repository. Before executing the clone detection, we pre-process the Java file snapshots to extract the methods. We do this for reasons similar to Göde *et al*. [8]. First, we exclude package and import statements, as they add no value to the study and may include many false positive clones. Second, clones can begin in one method and end in another, creating syntactically incorrect clones. By forcing hard boundaries between the methods, these clones are eliminated. We wrap each method snapshot in an individual file and submit it for clone detection.

Table II describes the number of clone genealogies and the number of LP Gens for each subject system using all three clone detection tools. There is a large discrepancy (in orders of magnitude) between the number of clones in CCFinder and Simian. We identify two reasons for this discrepancy: the clone detection technique and the minimum clone length of each tool. Simian uses a text-based clone detection technique, whereas CCFinder uses a token-based technique. CCFinder converts each Java file into a series of tokens during a preprocessing phase to normalize code structures, such as variable names and strings. In a manual examination of the CCFinder clone detection results, we identify many cases of 'false positive' clones due to this normalization. Several methods have a large number of false positive clones. For example, a method containing a list of variables being assigned values (e.g., $x = 0;$) may report tens of thousands of clones. We created a tool to automatically locate methods with a large number of clone pairs. After manually examining each method, we filtered the results to remove them from the study. A large number of false positive clones are not as apparent in the Simian clone detection results. Overall, we found CCFinder to have a high recall but low precision. A high recall means that CCFinder does not miss many clones during clone detection. The clone detection tool parameters also contribute to the discrepancy. CCFinder specifies a minimum number of tokens, whereas Simian specifies a minimum number of lines. To have the same minimum clone size, each Java file must have an average of around eight tokens per line. If the code has a much higher average number of tokens per line, CCFinder will have a smaller minimum number of lines and will therefore detect more clones than Simian.

In addition to these two tools, we extend our previous work [4] by repeating the study using NiCad. NiCad was selected because it uses a different approach compared with our other two tools; it is fast and has been reported to be accurate [18].

4.1.4. Building clone genealogies. To build the clone genealogies, we map clones from the clone list across commits. Both the line numbers and the size of the clones can change over time. To determine the changes to a clone pair over time and to assign new line numbers to each clone in the clone pair, we query the SVN of each studied system using *diff*. *diff* is a utility that compares files and generates a list of differences between them. When building clone genealogies, we only note changes that modify one or both of the clones in the clone pair. This is because changes that occur outside of the clone boundaries affect the line numbers, but not the contents of the clone. For example, if a clone starts on line 14 of a method, and three lines of code are inserted at line 3, then the clone start line and end line increases by three. The change does not affect the consistency of the clone pair.

For each clone pair, we query the J-Rex output for a list of all the commits where the methods containing the clones are modified. As mentioned previously, not all of these commits modify the cloned code, but this step reduces the number of commits that must be checked for changes. Using the commit number of the clone pair as a starting commit (i.e., the 'reference clone'), we execute *diff* on the SVN of the subject system to create a list of changes between the current version and the next commit in the commits list. We update the line numbers of the clone pair as needed to create

DOI: 10.1002/smr

an updated reference clone and determine if the clones themselves are modified during the commit. If they are modified, we need to determine if the change was diverging or reconciling.

A clone detection tool is used to determine if a change is divergent. Using the existing clone list obtained during the clone detection step, we identify a clone pair in the same snapshot of the system that contains the same start and end line numbers as the updated reference clones. If no matching clone pair is found in the clone list, we flag it as a diverging change in the clone genealogy. If the clone pair is found, the change is marked in the genealogy as a reconciling change. We repeat the entire process for each commit in the commit list, until each possible commit has been visited or the clone pair is removed. A clone pair is considered to be removed when either the file is deleted or the lines of code containing the clone are deleted.

In our work, a change is reconciling if the changed clone pair is identified as a types 1 or 2 clones using a clone detection tool. Changes that go beyond simple identifier renaming or changes to literals must be propagated in order for a change to be reconciling. Therefore, if a clone pair becomes a type 3 clone, where a line or more of code has been inserted (excluding comments and whitespace), the change will be labeled as diverging.

A clone detection tool may find a clone larger than the updated reference clone, so we allow a clone in the list to *contain* the updated reference clone. Even if we identify a clone larger than our clone pair of interest, we continue to build the genealogy using the updated reference clone. This is because the updated reference clone can be contained in a larger clone for only one commit, and yet it can continue to be modified in future commits.

Our approach for generating clone genealogies is similar to the approaches used in other studies [5, 6]. Both Göde and Krinke track clones over time by acquiring a list of changes from the source code repositories of the subject systems. They then query a clone detection tool with the updated clone pair to determine if changes caused the clones to diverge. Unlike these authors, we create an overall list of clones before creating clone genealogies, instead of calling a clone detection tool during the genealogy building process. Like Krinke [6], we use existing clone detection tools, SIMIAN, CCFINDER, and NICAD, to detect reconciling and diverging changes. In his work, Krinke made several assumptions when updating line numbers of clones between commits. We use the same assumptions in our study:

1. If a change occurs before the start of the clone, or after the end of the clone, the clone is not modified.
2. If an addition occurs starting at the first line number of a clone, the clone shifts within the method but is not modified.
3. If a deletion occurs anywhere within the clone boundaries, the clone is modified and its size shrinks.
4. If a deletion followed by an addition overlaps the clone boundaries, we assume that the clone size shrinks because of the deletion, and the new lines do not make up part of the clone.

In the last assumption, it is possible that there exists a clone containing both our updated reference clone and the newly added lines. We use the strictest assumption that the new lines are not included. When determining reconciling and diverging changes, we look for clones in the clone list that contain our updated reference clone. Therefore, this scenario would still be considered a reconciling change.

4.2. Analysis method

Three analysis methods are used in this study: the odds ratio, chi-square test, and the Kruskal–Wallis test. In this section, we describe each of these tests in more detail.

The odds ratio (OR) indicates the likelihood of an event to occur. It is defined as the ratio of the odds $p/(1-p)$ of an event (i.e., a fault-fixing change) occurring in one sample (i.e., experimental group) to the odds $q/(1-q)$ of the event occurring in the other sample (i.e., control group): $OR = \frac{p/(1-p)}{q/(1-q)}$, where p is the probability of the event occurring (i.e., a fault-fixing change) for the experimental group, and q is the probability of the event occurring for the control group.

An $OR = 1$ indicates that the event is equally likely in both samples; an $OR > 1$ shows that the event is more likely in the experimental group, whereas an $OR < 1$ indicates that it is more likely in the control group.

DOI: 10.1002/smr

In our results, each control group is assigned an OR of 1, because it is being compared with itself. In Table IV, the second and third columns contain data of the control group, that is, the genealogies without LP. The fifth and sixth columns contain data of the experimental group. Using the data from the first row of the table, the probability of a fault-fixing change occurring in the control group is $q = \frac{10,064}{10,064+18,706} = 0.349$. Hence, the odds of a fault-fixing change occurring in the control group is $\frac{0.349}{1-0.349} = 0.537$. Similarly the probability of a fault-fixing change occurring in the experimental group is $p = \frac{3437}{3437+3018} = 0.532$. The odds of a fault-fixing change occurring in the experimental group is therefore $\frac{0.532}{1-0.532} = 1.138$. Consequently, the OR for ANT using CCFINDER is $OR = \frac{1.138}{0.537} = 2.12$.

The chi-square test is a statistical test used to determine if there are non-random associations between two variables. We use the test to validate our odds ratios. The control group and the experimental groups of the odds ratios must all have the same two categorical variables (e.g., the number of the members of the group that contain faults and the number of the members of the group that contain no faults). If the p-value of the chi-square test is less than 0.01 then the OR values are statistically significant. Otherwise, we must assume that the OR could be due to chance variations in the data set.

The Kruskal–Wallis test is a non-parametric test. Given a set of experimental groups, it determines if the number of faults in each group is independent.

5. CASE STUDY RESULTS

This section reports and discusses the results of our study.

5.1. RQ1: Are there different types of late propagation?

Motivation

In Table I, we see that as much as a fifth of clone pair genealogies contain LP. We suggest that the instances of LP can be classified into eight types of LP. If only specific types of LP are more prone to faults, then they should be prioritized over other types for testing. This question is preliminary to the other questions. It provides the quantitative data on the percentages with which different types of LP occur in our studied systems. The eight types of LP as discussed in Section 3 are hypothetical. In this question, we determine if all eight types appear in software systems and are therefore worth investigating individually.

Approach

We address this question by classifying all instances of LP using the three characteristics described in Section 3. For each type of LP, we report the number of occurrences in the systems. Using the types of LP described in Section 3, we categorize all instances of LP in our studied systems. Table III lists each of the categories and the proportion of occurrences in each system, both as a numerical value and a percentage of the overall number of LP instances for that system. For each system we repeat the experiment using all three clone detection tools. Each set of two rows in Table III summarizes the distribution of LP clone pairs for a specific system (e.g., ARGOUML) using a specific clone detection tool (e.g., SIMIAN).

Results

As summarized in Table III, four types of LP are dominant across all systems using three clone detection tools (i.e., LP1, LP6, LP7, and LP8). The five dominant types represent the three propagation categories. As shown in Table III, the instances of LP2 and LP3 are low. Therefore, the 'propagation always occurs' category (i.e., LP1, LP2, and LP3) does not account for the majority of instances of a diverging change followed by a reconciling change. For all cases, the 'propagation never occurs' category (i.e., LP8) contributes more instances of LP than the 'propagation always occurs' category. As shown in Table III, LP7 occurs in an average of 50% of instances of LP, so it is the most common form of LP across all systems. However, LP7 is also the least understood of the types of LP. Because both clones in LP7 clone pairs are modified during all three steps of LP (i.e., diverging, period of divergence, and reconciliation), it is unclear in which direction changes are propagated during the evolution of the clone pair. A few types of LP (i.e., LP2 and LP4) contribute minutely to the number of LP Gens.

DOI: 10.1002/smr

Table III. Number of clone pairs that underwent a late propagation.

		Propagation always occurs			Propagation may or may not occur				Propagation never occurs
		LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
					CCFinder				
Ant	Number	521	46	121	33	70	166	2482	3016
	%	8.07	0.71	1.87	0.51	1.09	2.57	38.45	46.72
ArgoUML	Number	57	13	226	18	73	129	1242	394
	%	2.65	0.60	10.50	0.85	3.39	5.99	57.71	18.31
JBoss	Number	45	1	55	4	44	70	995	758
	%	2.28	0.05	2.79	0.20	2.23	3.55	50.46	38.44
					Simian				
Ant	Number	21	0	0	0	0	1	29	52
	%	20.39	0.00	0.00	0.00	0.00	0.97	28.15	50.49
ArgoUML	Number	0	0	1	2	0	5	6	9
	%	0.00	0.00	4.35	8.70	0.00	21.74	26.09	39.13
JBoss	Number	0	0	0	0	1	0	8	3
	%	0.00	0.00	0.00	0.00	8.33	0.00	66.67	25.00
					NiCad				
Ant	Number	37	4	1	3	1	1	516	269
	%	4.45	0.48	0.12	0.36	0.12	0.12	62.02	32.33
ArgoUML	Number	41	13	51	4	32	53	498	141
	%	4.92	1.56	6.12	0.48	3.84	6.36	59.78	16.93
JBoss	Number	0	0	0	0	2	1	12	5
	%	0.00	0.00	0.00	0.00	10.00	5.00	60.00	25.00

Overall, we conclude that there is representation from multiple types of LP and across all categories of LP. In the following research questions, we examine the types in more detail to determine if some types are more risky than others.

5.2. *RQ2: Are some types of LP more fault prone than others?*

Motivation

Previous researchers [3] have studied the relationship between LP and faults. In this research question, we first replicate the earlier studies, and then extend our study to include the different categories of LP. To replicate earlier studies, we determine if LP Gens are more fault prone than non-LP Gens in general. Then, we divide the instances of LP into groups based on their type of LP. We compare the fault proneness of each type of LP compared with non-LP Gens. We investigate if only specific types of LP are at risk of faults. Rather than suggesting that all LP Gens are prone to faults and must be monitored, developers can focus their limited testing efforts only on certain types of LP that are the most fault prone.

Approach

We compute the number of fault-containing and fault-free Gens in each LP category. A Gen is fault-containing if its Gen contains at least one fault-fixing change during its history. We compute the same values for non-LP clone Gens that experience at least one change. For the remainder of this paper, we use the abbreviation 'Non-LP' for clone pairs that experience at least one change but are not involved in any type of LP. We test the following null hypothesis[††] H_{02}: *Each type of LP Gen has the same proportion of clone pairs that experience a fault fix*. For this question, we use the chi-square test [25] and compute the OR [25].

Results

The results for this question are divided into two sections. First, we present the results for LP Gens in general. Then, we examine the fault proneness of each LP type in detail.

[††]There is no H_{01} because RQ1 is exploratory.

DOI: 10.1002/smr

5.2.1. Fault proneness of late propagation. Table IV summarizes the results of the tests described in Section 4.2 for instances of LP compared with non-LP Gens. The first and second columns in the table list the number of non-LP Gens that experience fault fixes and the number that are free of fault fixes. The third and fourth columns show the same data for LP-Gens. The last column of the table lists the OR test results for each system using all three clone detection tools. Except for the cases where ARGOUML and JBOSS are analyzed using SIMIAN, and JBOSS is analyzed using NICAD, all of our results pass the chi-square test with a *p*-value less than 0.01 and are therefore significant. Where there are few data points, we use Fisher's exact test to confirm the results from the chi-square test. The Fisher's exact test is more accurate than the chi-square test when sample sizes are small [25]. In this study, the Fisher test provided the same results as the chi-square test, so we do not present the Fisher test results in the tables.

In all the significant cases, the OR is greater than 1, indicating that LP Gens are more fault prone than non-LP Gens. However, for ARGOUML and JBOSS using SIMIAN, and JBOSS using NICAD, the results of the chi-square test are not statistically significant. This can be explained by the small number of clone Gens obtained with the SIMIAN and NICAD detection tools. Overall, our results agree with previous studies [3] that found that LP is more at risk of faults.

5.2.2. Fault proneness of late propagation types. We repeat the previous tests, dividing the instances of LP into their respective LP types. We compare each type of LP to Gens with no LP. Tables V, VI, and VII summarizes the results obtained from CCFINDER, SIMIAN, and NICAD, respectively. For each type of LP, the table lists the number of instances that experience a fault fix, the number that never experience a fault fix, the result from the chi-square test, and the OR using the control group.

The chi-square test results for ARGOUML and JBOSS using SIMIAN in Table VI and JBOSS using NICAD are greater than 0.01, so they are insignificant. Therefore, they are excluded from consideration.

An examination of the significant cases in Tables V, VI, and VII reveals that the ORs are greater than 1, so each type of LP is more fault prone than non-LP Gens. There are six exceptions to this observation within ANT: LP6 in Table VI, LP5 and LP7 in Table V, and LP4, LP6, and LP7 in Table VII. Additionally, JBOSS using CCFINDER is less fault prone than non-LP Gens for LP1 and LP7. All exceptions, except LP1, belong to the 'propagation may or may not occur' category. LP1 belongs to the 'propagations always occur' category.

Comparing Tables V, VI, and VII to Table III, we conclude that there are many types that make up a small proportion of LP instances and have a very high OR. Thus, when one of these LP types occurs, it is very likely to contain a fault fix. For example, LP2 has a high OR (e.g., 26.64 in ANT using CCFINDER in Table III), but accounts for less than 1% of all LP instances in Table III.

The two most common LP types in the previous research question, LP7 and LP8, in general have low ORs in Table III. This indicates that although they occur frequently, they are less fault prone than other less common LP types (e.g., LP2).

Table IV. Contingency table and chi-square test results for clone genealogies with and without late propagation.

Releases	No LP – faults	No LP – no faults	LP – faults	LP – no faults	*p*-values	OR
			CCFINDER			
ANT	10,064	18,706	3437	3018	<0.01	2.12
ARGOUML	8755	9869	1398	754	<0.01	2.10
JBOSS	5684	8268	1020	952	<0.01	1.56
			SIMIAN			
ANT	124	251	68	35	<0.01	3.93
ARGOUML	44	45	12	11	1	1.12
JBOSS	5	8	4	8	0.88	0.8
			NICAD			
ANT	797	1987	357	475	<0.01	1.87
ARGOUML	1891	1399	655	178	<0.01	2.72
JBOSS	53	79	9	11	0.87	1.21

LP, late propagation; OR, odds ratio.

DOI: 10.1002/smr

Table V. CCFINDER – contingency tables with the chi-square test.

		Propagation always occurs				Propagation may or may not occur				Propagation never occurs
		No LP	LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
ANT	Faults	10,064	282	43	50	27	19	109	814	2093
	No faults	18,706	239	3	71	6	51	57	1668	923
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	1	2.19	26.64	1.31	8.36	0.69	3.55	0.91	4.21
ARGOUML	Faults	8755	41	11	175	13	57	96	712	293
	No faults	9869	16	2	51	5	16	33	530	101
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	1	2.88	6.2	3.87	2.93	4.02	3.28	1.51	3.27
JBOSS	Faults	5684	7	1	25	2	19	44	356	566
	No faults	8268	38	0	30	2	25	26	639	192
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	< 0.01
	OR	1	0.27	Infinite	1.21	1.45	1.11	2.46	0.81	4.29

LP, late propagation; OR, odds ratio.

Table VI. SIMIAN – contingency tables with the chi-square test.

		Propagation always occurs				Propagation may or may not occur				Propagation never occurs
		No LP	LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
ANT	Faults	124	8	0	0	0	0	0	10	50
	No Faults	251	13	0	0	0	0	1	19	2
	p-value	<0.01	<0.01	n/a	n/a	n/a	n/a	<0.01	<0.01	<0.01
	OR	1	1.25	n/a	n/a	n/a	n/a	0	1.07	50.6
ARGOUML	Faults	44	0	0	0	2	0	2	3	5
	No Faults	45	0	0	1	0	0	3	3	4
	p-value	0.65	n/a	n/a	0.65	0.65	n/a	0.65	0.65	0.65
	OR	1	n/a	n/a	0	Infinite	n/a	0.68	1.02	1.28
JBOSS	Faults	10,064	0	0	0	0	5	0	2	2
	No Faults	18,706	0	0	0	0	8	0	6	1
	p-value	0.52	n/a	n/a	n/a	n/a	0.52	n/a	0.52	0.52
	OR	1	n/a	n/a	n/a	n/a	0	n/a	0.53	3.2

LP, late propagation; OR, odds ratio; n/a, not applicable.

Table VII. NICAD – contingency tables with the chi-square test.

		Propagation always occurs				Propagation may or may not occur				Propagation never occurs
		No LP	LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
ANT	Faults	797	22	4	1	0	1	0	133	196
	No faults	1987	15	0	0	3	0	1	383	73
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	1	3.66	Infinite	Infinite	0	Infinite	0	0.87	6.69
ARGOUML	Faults	1891	32	9	43	3	28	45	408	87
	No faults	1399	9	4	8	1	4	8	90	54
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	1	2.63	1.66	3.98	2.22	5.18	4.16	3.35	1.91
JBOSS	Faults	53	0	0	0	0	0	1	5	3
	No faults	79	0	0	0	0	2	0	7	2
	p-value	0.46	n/a	n/a	n/a	n/a	0.46	0.46	0.46	0.46
	OR	1	n/a	n/a	n/a	n/a	0	Infinite	1.06	2.24

LP, late propagation; OR, odds ratio; n/a, not applicable.

DOI: 10.1002/smr

Overall, each type of LP has a different level of fault proneness. Thus, we reject H_{02} in general.

5.3. RQ3: Which type of late propagation experiences the highest proportion of faults?

Motivation

In the previous question (RQ2), we examine which types of LP are the most prone to faults. In this question, we examine which types of LP contribute the most faults to each system. In other words, we examine if, when faults occur, do they occur in large numbers? This question examines the impact of a fault-prone Gen, because a clone pair that experiences 10 faults may require more effort than a clone pair that only experiences one fault.

Approach

We want to identify which type of LP experiences the highest proportion of faults. Therefore, we test the following null hypothesis H_{03}: *Different types of LP have the same proportion of clone pairs that experience a fault fix*. For each type of LP, we calculate the sum of all faults experienced by instances of that type of LP. We use the non-parametric Kruskal–Wallis test to investigate if the number of faults for the different types of LP are identical.

Results

Table VIII presents the distribution of faults for different types of LP. The 'Total' row represents the total numbers of faults over all LP Gens. For example, for ANT using CCFINDER, there are 5566 fault fixes across all Gens. These 5566 faults are spread over 1104 commits marked by J-REX as a fault fix. This is because multiple clone pairs are modified during a fault-fixing commit.

To validate the results, we perform the non-parametric Kruskal–Wallis test, which compares the distribution of faults between groups of different types of LP. Table IX summarizes the results of the Kruskal–Wallis test. Globally, we observe a statistically significant difference between the distribution of faults across all the groups of LP types. From Table IX, only ARGOUML and JBOSS using SIMIAN, and JBOSS using NICAD are not statistically significant.

Examining the results in Table VIII for almost all the significant cases, we see that in general, LP7 and LP8 contribute to a large proportion of the faults. In the previous question, LP7 and LP8 have lower ORs. Although they are less prone to faults, when they do experience faults, fault-fixing

Table VIII. Proportion of faults for each type of late propagation.

		Propagation always occurs			Propagation may or may not occur				Propagation never occurs
		LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
					CCFINDER				
ANT	Number of Faults	484	79	68	56	25	197	999	3658
	%	8.70	1.42	1.22	1.01	0.45	3.54	17.95	65.72
ARGOUML	Number of Faults	72	25	311	17	108	134	1121	500
	%	3.15	1.09	13.59	0.74	4.72	5.86	48.99	21.85
JBOSS	Number of Faults	7	1	40	2	26	47	486	851
	%	0.48	0.07	2.74	0.14	1.78	3.22	33.29	58.29
					SIMIAN				
ANT	Number of Faults	9	0	0	0	0	0	12	106
	%	7.09	0.00	0.00	0.00	0.00	0.00	9.45	83.47
ARGOUML	Number of Faults	0	0	0	4	0	2	6	6
	%	0.00	0.00	0.00	22.22	0.00	11.11	33.33	33.33
JBOSS	Number of Faults	0	0	0	0	0	0	4	3
	%	0.00	0.00	0.00	0.00	0.00	0.00	57.14	42.86
					NICAD				
ANT	Number of Faults	32	8	1	0	2	0	140	289
	%	6.78	1.70	0.21	0.00	0.420	0.00	29.66	61.23
ARGOUML	Number of Faults	126	39	89	5	65	97	676	168
	%	9.96	3.08	7.04	0.40	5.14	7.67	53.44	13.28
JBOSS	Number of Faults	0	0	0	0	0	2	5	6
	%	0.00	0.00	0.00	0.00	0.00	15.38	38.46	46.15

DOI: 10.1002/smr

Table IX. Results of the Kruskal–Wallis tests.

System	Kruskal–Wallis p-values
ANT – CCFINDER	<0.01
ANT – SIMIAN	<0.01
ANT – NICAD	<0.01
ARGOUML – CCFINDER	<0.01
ARGOUML – SIMIAN	0.24
ARGOUML – NICAD	<0.01
JBOSS – CCFINDER	<0.01
JBOSS – SIMIAN	0.47
JBOSS – NICAD	1

commits are likely to occur in large numbers. When a developer is reconciling these divergent clones due to a fault fix, he should be careful to check for other faults in the clones. In the case of ANT, LP8 contributes over half of all fault fixes, but is one of the least fault-prone types compared with other LP types. The change causing the divergence may lead to faults in the system, which may be why the change is reverted instead of being propagated to the other clone in the clone pair.

The remaining results are system dependent. For example, in the case of ARGOUML using CCFINDER in Table VIII, the category where propagation always occurs contributes over two times the amount of faults than the category where propagation never occurs. This trend does not hold across all systems in our study.

Overall, we can conclude that types LP7 and LP8 are the most dangerous, with the other types being system dependent in their fault proneness. The proportion of faults for each type of LP are therefore very different. Thus, we reject H_{03}.

5.4. *RQ4: Does the size of cloned code affect the fault proneness of late propagation genealogies?*

Motivation

In RQ2, we investigate the fault proneness of the different types of LP Gens described in Section 3. In this question, we examine the relationship between the size of the cloned code and fault proneness. It is expected that smaller clones will be less prone to faults, as they contain less code and would be easier to comprehend. For each type of LP, we investigate if smaller clones have a different odds ratio than bigger clones of the same type. If the fault proneness is size dependent, then fault-prone LP types can be further filtered by their size to highlight the most risky clones.

Approach

For each LP clone pair, we measure the number of lines of cloned code. Using this value, we classify each clone pair as either small or large. It is classified as 'LARGE' if the size is greater than 10 LOC, or 'SMALL' if it is smaller than or equal to 10 LOC. We obtain the cutoff by examining the distribution of clone sizes. For example, Figure 2 shows the distribution of clone sizes for ANT using CCFINDER. The figure shows that almost all of the clones are smaller than 100 LOC. A large proportion of the clones are smaller than 10 LOC. The number of occurrences of clones of a specific size decreases as the size increases. Using the clone size distribution information, we select our cut-off of 10 LOC for small clones.

We test the following null hypothesis H_{04}: *There is no relationship between the size of a clone experiencing LP and its fault proneness*. We group the clone pairs based on their size (i.e., LARGE and SMALL). For each type of LP, we compute the number of small and big fault-containing and small and big fault-free Gens. Therefore, there are two experimental groups for each type of LP, a BIG group and a SMALL group. We also calculate the number of small and large fault-containing and fault-free non-LP clone Gens. We compute the ORs and perform the chi-square test.

Results

We determine the control group for this question by calculating the ORs between the small non-LP Gens and the big non-LP Gens. The p-value of the chi-square test and the OR of the small non-LP

DOI: 10.1002/smr

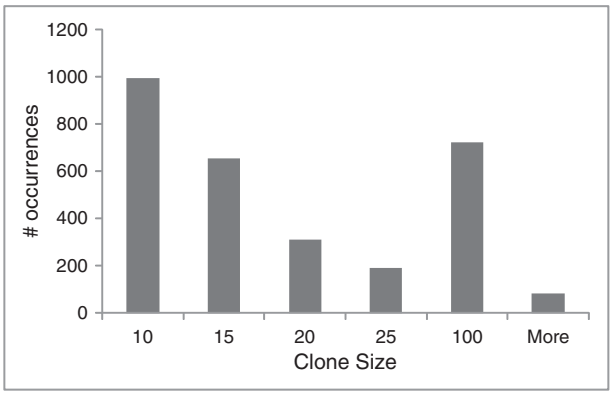

Figure 2. Clone size distribution of ANT using CCFINDER.

Gens are shown in the first column of data in Tables X, XI, and XII. We use the group of non-LP Gens with big sizes as the control group.

The ORs for each LP type and size and their corresponding chi-square test values are listed in Tables X, XI, and XII. Each type of LP is divided into two different groups, big and small. The first column of data in the big rows describes the control group, which consists of all big non-LP Gens. In Table XI, only ANT passes the chi-square test. Therefore, we exclude ARGOUML and JBOSS using SIMIAN from our discussion. Similarly, when using the NICAD clone detection tool, JBOSS fails the chi-square test. Therefore, we exclude it from our discussion.

From Tables X, XI, and XII, examining all systems, we notice that for LP4, the small clones are more fault prone than big clones. However, except for LP2 in ARGOUML in Table XII, for LP2,

Table X. CCFINDER – contingency tables with the chi-square test for the fault proneness of late propagation types grouped by size.

			Propagation always occurs				Propagation may or may not occur				Propagation never occurs
			No LP	LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
ANT	BIG	Faults	6326	167	26	31	17	8	54	517	1453
		No faults	11776	79	1	32	4	36	16	1226	635
		p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
		OR	1	3.93	48.39	1.80	7.91	0.41	6.28	0.78	4.25
	SMALL	Faults	3738	115	17	19	10	11	55	297	640
		No faults	6930	160	2	39	2	15	41	442	288
		p-value	0.88	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
		OR	1.01	1.33	15.82	0.91	9.3	1.36	2.49	1.25	4.13
ARGOUML	BIG	Faults	3692	18	8	81	3	18	41	267	114
		No faults	2866	5	0	12	2	3	6	138	33
		p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
		OR	1	2.79	Infinite	5.24	1.16	4.65	5.30	1.50	2.68
	SMALL	Faults	5063	23	3	94	10	39	55	445	179
		No faults	7003	11	2	39	3	13	27	392	68
		p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
		OR	0.56	1.62	1.16	1.87	2.58	2.33	1.58	0.88	2.04
JBOSS	BIG	Faults	1692	4	1	13	0	8	27	211	209
		No faults	4608	22	0	11	0	15	5	292	69
		p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
		OR	1	0.5	Infinite	3.22	n/a	1.45	14.71	1.97	8.24
	SMALL	Faults	3992	3	0	12	2	11	17	145	357
		No faults	3660	16	0	19	2	10	21	347	123
		p-value	<0.01	<0.01	n/a	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
		OR	2.97	0.51	n/a	1.72	2.72	2.99	2.20	1.13	7.90

LP, late propagation; OR, odds ratio; n/a, not applicable.

DOI: 10.1002/smr

Table XI. SIMIAN – contingency tables with the chi-square test for the fault proneness of late propagation types grouped by size.

			Propagation always occurs				Propagation may or may not occur				Propagation never occurs
			No LP	LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
ANT	BIG	Faults	89	6	0	0	0	0	0	10	35
		No faults	159	6	0	0	0	0	1	16	2
		p-value	<0.01	<0.01	n/a	n/a	n/a	n/a	<0.01	<0.01	<0.01
		OR	1	1.78	n/a	n/a	n/a	n/a	0	1.12	31.26
	SMALL	Faults	35	2	0	0	0	0	0	0	15
		No faults	92	7	0	0	0	0	0	3	0
		p-value	0.13	<0.01	n/a	n/a	n/a	n/a	n/a	<0.01	<0.01
		OR	0.68	0.51	n/a	n/a	n/a	n/a	n/a	0	Infinite
ARGOUML	BIG	Faults	17	0	0	0	1	0	0	0	3
		No faults	22	0	0	0	0	0	1	0	3
		p-value	0.79	n/a	n/a	n/a	0.79	n/a	0.79	n/a	0.79
		OR	1	n/a	n/a	n/a	Infinite	n/a	0	n/a	1.29
	SMALL	Faults	27	0	0	0	1	0	2	3	2
		No faults	23	0	0	1	0	0	2	3	1
		p-value	0.44	n/a	n/a	0.79	0.79	n/a	0.79	0.79	0.79
		OR	1.52	n/a	n/a	0	Infinite	n/a	1.29	1.29	2.58
JBOSS	BIG	Faults	2	0	0	0	0	0	0	0	2
		No faults	4	0	0	0	0	0	0	0	1
		p-value	0.52	n/a	n/a	n/a	n/a	n/a	n/a	n/a	0.52
		OR	1	n/a	n/a	n/a	n/a	n/a	n/a	n/a	4
	SMALL	Faults	3	0	0	0	0	0	0	2	0
		No faults	4	0	0	0	0	1	0	6	0
		p-value	0.82	n/a	n/a	n/a	n/a	0.52	n/a	0.52	n/a
		OR	1.50	n/a	n/a	n/a	n/a	0	n/a	0.67	n/a

LP, late propagation; OR, odds ratio; n/a, not applicable.

LP3, and LP6, the opposite is true. For these types, big clones are more fault prone than small clones. In most cases for LP8, the ORs of the big and small clones are similar in value. For the other types of LP (i.e., LP1, LP5, and LP7), the fault proneness of big and small clones is different. Thus, in general, we reject H_{04}. Additionally, which of the big or small clones is more fault prone is system dependent.

5.5. RQ5: For a clone pair experiencing LP, does the time interval between the diverging change and the reconciling change affect fault proneness?

Motivation

In a previous study on code clone Gens [3], Thummalapenta *et al*. categorized LP Gens by whether or not a reconciling change occurs within the first day. In this research question, we investigate the different periods of delay between the diverging and reconciling changes. We want to evaluate if a smaller period of delay will be less prone to faults because a developer will be more familiar with the code. If certain periods of delay are more fault prone, then those Gens should be prioritized for testing.

Approach

We establish if the time interval between the divergent phase and the reconciling phase of the clone pair impacts the fault proneness of a clone pair experiencing LP. We classify clone pairs experiencing LP by the time interval between the diverging and the reconciling changes. We divide them into five groups corresponding to the following five time period: 1 day, 1 week, 1 month, 1 year, and more than 1 year. Using the 'More than a Year' group as the control group, we calculate the ORs between the control group and the other time period groups (i.e., the experimental groups) and perform the chi-square test. We select the 'More than a Year' group as our control group because we suggest that a longer period of delay will be the most fault prone. A developer may be unfamiliar with the code and could be more likely to introduce faults into the system. We test the following null

DOI: 10.1002/smr

Table XII. NiCad – contingency tables with the chi-square test for the fault-proneness of late propagation types grouped by size.

			Propagation always occurs				Propagation may or may not occur				Propagation never occurs
			No LP	LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
Ant	BIG	Faults	775	22	4	1	0	1	0	132	195
		No faults	1889	12	0	0	3	0	1	383	72
		p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
		OR	1	4.47	Infinite	Infinite	0	Infinite	0	0.84	6.60
	SMALL	Faults	22	0	0	0	0	0	0	1	1
		No faults	98	3	0	0	0	0	0	0	1
		p-value	0.01	<0.01	n/a	n/a	n/a	n/a	n/a	<0.01	<0.01
		OR	0.68	0.51	n/a	n/a	n/a	n/a	n/a	Infinite	2.44
ArgoUML	BIG	Faults	1186	19	6	36	2	23	31	309	76
		No faults	1087	9	4	4	1	4	3	61	47
		p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
		OR	1	1.93	1.37	8.25	1.83	5.27	9.47	4.64	1.48
	SMALL	Faults	705	13	3	7	1	5	14	99	11
		No faults	312	0	0	4	0	0	5	29	7
		p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
		OR	2.07	Infinite	Infinite	1.60	Infinite	Infinite	2.57	3.13	1.44
JBoss	BIG	Faults	52	0	0	0	0	0	1	5	3
		No faults	72	0	0	0	0	2	0	7	2
		p-value	0.48	n/a	n/a	n/a	n/a	0.48	0.48	0.48	0.48
		OR	1	n/a	n/a	n/a	n/a	0	Infinite	0.99	2.08
	SMALL	Faults	1	0	0	0	0	0	0	0	0
		No faults	7	0	0	0	0	0	0	0	0
		p-value	0.20	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		OR	0.2	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

LP, late propagation; OR, odds ratio; n/a, not applicable.

hypothesis H_{05}: *The time interval between the diverging and the reconciling of a LP clone pair has no relationship with its fault proneness*.

Results

The results for this research question are shown in Tables XIII, XIV, and XV. The control group, the delay before the reconciling change is greater than a year, is shown in the last column in each table.

In all cases that passed the chi-square test, we see that compared with LP Gens that reconcile after more than a year, the fault proneness of each delay period is different. Therefore, we reject H_{05}. However, the fault proneness of each delay period is not consistently greater than 1 or less than 1 across all systems using both clone detection tools, so the fault proneness of each delay period is system dependent. For example, the OR of LP Gens that reconcile within 1 day is almost zero in three cases, however it is infinite in the three other cases. We cannot say that overall the fault proneness of a LP increases or decreases as the time because the divergence increases.

5.6. *RQ6: Are the clone pairs in late propagation reconciled because of fault-fixing activities?*

Motivation

In this question, we determine if the reconciling change following a diverging change (RC) in a LP Gen is a fault-fixing change. A divergent clone pair becomes reconciled when either changes are propagated between two clones or the diverging changes are reverted. Because a fault-fixing reconciling change eliminates the divergence, it is possible that the divergence was accidental. It is also possible that the diverging change introduced the fixed fault in the system. Therefore, if a high proportion of the reconciling changes of a type of LP are fault-fixing changes, we recommend that developers monitor that type of LP more carefully. However, we cannot claim that LP caused the fixed faults, because these faults could have been introduced in the system prior to the LP occurring.

DOI: 10.1002/smr

Table XIII. CCFinder – contingency tables with the chi-square test for the delay between a diverging change and a reconciling change and faults.

		Within 1 day	Within 1 week	Within 1 month	Within 1 year	More than a year
Ant	Faults	37	128	542	1130	1600
	No faults	75	413	849	969	712
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	0.22	0.14	0.28	0.52	1
ArgoUML	Faults	2	67	93	920	316
	No faults	26	17	62	477	172
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	0.04	2.15	0.82	1.05	1
JBoss	Faults	18	14	253	642	93
	No faults	0	14	88	751	99
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	Infinite	1.06	3.06	0.91	1

OR, odds ratio.

Table XIV. Simian – contingency tables with the chi-square test for the delay between a diverging change and a reconciling change and faults.

		Within 1 day	Within 1 week	Within 1 month	Within 1 year	More than a year
Ant	Faults	7	0	10	38	13
	No faults	0	5	6	19	5
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	Infinite	0	0.64	0.77	1
ArgoUML	Faults	0	0	3	3	6
	No faults	0	0	2	6	3
	p-value	n/a	n/a	0.34	0.34	0.34
	OR	n/a	n/a	0.75	0.25	1
JBoss	Faults	0	0	0	3	1
	No faults	0	0	3	5	0
	p-value	n/a	n/a	0.17	0.17	n/a
	OR	n/a	n/a	0	0	n/a

OR, odds ratio; n/a, not applicable.

Table XV. NiCad – contingency tables with the chi-square test for the delay between a diverging change and a reconciling change and faults.

		Within 1 day	Within 1 week	Within 1 month	Within 1 year	More than a year
Ant	Faults	6	22	63	116	150
	No faults	8	117	180	153	17
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	0.09	0.02	0.04	0.09	1.00
2 ArgoUML	Faults	2	19	37	366	231
	No faults	0	17	22	88	51
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	Infinite	0.25	0.37	0.92	1.00
JBoss	Faults	0	2	1	6	0
	No faults	0	0	1	4	6
	p-value	n/a	0.04	0.04	0.04	0.04
	OR	n/a	Infinite	Infinite	Infinite	1.00

OR, odds ratio; n/a, not applicable.

Approach

We test the following null hypothesis H_{06}: *the proportion of reconciling changes that follow a diverging change that are fault fixes is equal to the proportion of other changes that are fault fixes*. For

DOI: 10.1002/smr

each type of LP, we calculate the number of RC changes that were a fault fix and the number that were not a fault fix. We use all other changes within the LP clone Gens as our control group and determine the number that contained a fault fix. A non-RC change is any change that modifies the clone pair. This includes all diverging changes and reconciling changes that are immediately preceded by a reconciling change. We calculate the OR between the non-RC changes and the RC changes from each type of LP. We validate our results using the chi-square test.

Results

The results of our study are shown in Tables XVI, XVII, and XVIII. The first column of data in the tables contains the non-RC changes, which is the control group in our study. ARGOUML and JBOSS using SIMIAN fail the chi-square test, so we exclude them from our discussion.

In five of the seven significant cases, the OR of LP8 is greater than 1, indicating that the reconciling change in an LP8 Gen is more likely to be a fault-fixing change. LP8 belongs to the 'Propagation never occurs' category, and in this type, the diverging change is reverted without the other clone in a clone pair being modified. This indicates that when a change is reverted to reconcile the clone pair, it has a high likelihood of being a fault-fixing change.

The OR of LP7 is less than 1 in all significant cases, except for JBOSS using NICAD. This indicates that in LP Gens where both clones are modified during all three phases (i.e., diversion, period of divergence, and reconciliation) as described in Section 3, the reconciliation is less likely to be due to a fault fix. All other changes in a LP Gen are more likely to be a fault-fixing change.

The results for all other types of LP are both system and clone detection tool dependent. However, overall, the fault proneness of the reconciling change for each LP type is different, so we reject H_{06}.

6. THREATS TO VALIDITY

We now discuss the threats to validity of our study, following the guidelines for case study research [26].

Construct validity threats concern the relation between theory and observation. In this study, the threats are mainly due to measurement errors possibly introduced by our chosen clone detection tools. To reduce the possibility of misclassification of code fragment as clones, we chose three clone detection tools that have been used in previous studies and repeat the study for all three tools.

All clone detection tools in this study can detect identical (i.e., type 1) and near-identical clones (i.e., type 2). CCFINDER can detect some gaps in type 1 and 2 clones. An additional LOC within a clone (excluding whitespace and comments) cannot be detected by the tools. The addition or deletion of an LOC to one clone segment and not the other is a diverging change. Thus, if we were to use a clone detection tool that detects multiple lines between clone segments, our Gens would be

Table XVI. CCFINDER – contingency tables with the chi-square test for the likelihood that reconciling changes are fault-fixing changes.

		Propagation always occurs				Propagation may or may not occur				Propagation never occurs
		Non-RS	LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
ANT	Faults	4433	139	7	3	15	3	24	190	680
	No faults	21,093	382	39	117	18	67	142	2293	2336
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	1	1.73	0.85	0.12	3.97	0.21	0.8	0.39	1.39
ARGOUML	Faults	1836	14	6	63	1	11	14	224	98
	No faults	7778	43	7	163	17	62	115	1018	296
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	1	1.38	3.63	1.64	0.25	0.75	0.52	0.93	1.40
JBOSS	Faults	1175	0	0	11	0	6	3	116	123
	No faults	5900	45	1	43	4	38	67	880	635
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	1	0	0	1.28	0	0.79	0.22	0.66	0.97

OR, odds ratio; Non-RS, Non-resynchronizing.

DOI: 10.1002/smr

Table XVII. SIMIAN – contingency tables with the chi-square test for the likelihood that reconciling changes are fault-fixing changes.

		Propagation always occurs				Propagation may or may not occur				Propagation never occurs
		Non-RS	LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
ANT	Faults	95	1	0	0	0	0	0	1	23
	No faults	368	20	0	0	0	0	1	28	29
	p-value	<0.01	<0.01	n/a	n/a	n/a	n/a	<0.01	<0.01	<0.01
	OR	1	0.19	n/a	n/a	n/a	n/a	0	0.14	3.07
ARGOUML	Faults	17	0	0	0	0	0	0	0	2
	No faults	83	0	0	0	2	0	5	7	7
	p-value	0.55	n/a	n/a	n/a	0.55	n/a	0.55	0.55	0.55
	OR	1	n/a	n/a	n/a	0	n/a	0	0	1.39
JBOSS	Faults	5	0	0	0	0	0	0	2	0
	No faults	36	0	0	0	0	0	0	7	3
	p-value	0.57	n/a	n/a	n/a	n/a	n/a	n/a	0.57	0.57
	OR	1	n/a	n/a	n/a	n/a	n/a	n/a	2.06	0

OR, odds ratio.

Table XVIII. NICAD – contingency tables with the chi-square test for the likelihood that reconciling changes are fault-fixing changes.

		Propagation always occurs				Propagation may or may not occur				Propagation never occurs
		Non-RS	LP1	LP2	LP3	LP4	LP5	LP6	LP7	LP8
ANT	Faults	324	8	1	0	0	1	0	26	55
	No faults	3357	29	3	1	3	0	1	490	214
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	1	2.86	3.45	0	0	Infinite	0	0.55	2.66
ARGOUML	Faults	1082	9	3	19	0	9	10	100	19
	No faults	2855	32	10	32	4	23	43	398	122
	p-value	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
	OR	1	0.74	0.79	1.57	0	1.03	0.61	0.66	0.41
JBOSS	Faults	7	0	0	0	0	0	1	2	3
	No faults	54	0	0	0	0	2	0	10	2
	p-value	<0.01	n/a	n/a	n/a	n/a	<0.01	<0.01	<0.01	<0.01
	OR	1	n/a	n/a	n/a	n/a	0	Infinite	1.54	11.57

OR, odds ratio.

inaccurate. It is possible in our current approach that the clone detection tool may detect the two clone segments as new clone pairs, but only if the clone segments meet the minimum clone size requirements. These new clone pairs would create Gens in parallel with the larger clone Gen. However, they would only experience LP if the main clone Gen experienced LP a second time, and it must occur within one of the clone segments. One of the authors manually inspected the Gens and could not find such occurrences. In future work, we plan to consider type 3 clone Gens.

Another construct validity threat is the software evolution tool J-REX, which uses heuristics to identify fault-fixing changes [19]. The results of this study are dependent on the accuracy of the results from J-REX. However, we are confident in the results from J-REX as it implements the same algorithm used previously by Hassan *et al*. [27] and Mockus *et al*. [20]. Additionally, we perform an evaluation of J-REX and determine that the accuracy of J-REX is 87%. We do not remove the false positives from our study, due to the large number of commit messages that would need to be manually examined.

DOI: 10.1002/smr

J-REX can distinguish between methods with the same name by their parameters. However, a renamed method is treated as a deletion of the previous method and creation of a new method.

Threats to *internal validity* do not affect this study, as it is an exploratory study [26]. Although we cannot claim causation, we do identify, in RQ2 and RQ3, a relation between LP and fault proneness for clone pair Gens. Furthermore, we have provided some qualitative explanation of our results on the basis of the inspection of the source code of our studied systems.

Conclusion validity threats concern the relation between the treatment and the outcome. We pay attention to the assumptions of the statistical tests. Also, we mainly use non-parametric tests that do not require a normal distribution of the data.

Threats to *external validity* concern the possibility of generalizing our results. We examine Java systems, some that use a plug-in architecture. Although the selected systems have different sizes and belong to different domains, further validation on more systems should be performed. Additionally, this study should be repeated on systems that use other programming languages, such as C and C++.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the details needed to replicate our study. Also, the source code and SVN repositories of the studied systems are publicly available.

7. CONCLUSION

In this paper, we extend previous studies on clone Gens to examine LP in more detail. These studies identified LP as a fault-prone type of clone Gen. We first confirm that LP is more risky than other clone Gens. We then identify eight types of LP and study them in detail to identify which contribute most to faults in LP. Overall, we find that two types of LP, LP7, and LP8 are riskier than the others, in terms of their fault proneness and the magnitude of their contribution toward faults. LP8 involves no propagation at all and occurs when a clone diverges and then reconciles itself without changes to the other clone in a clone pair (i.e., any modifications that diverge the clone pair are reverted). The reconciling change in LP8 has a higher likelihood of being a fault-fixing change, indicating that the period of divergence within an LP8 Gen may be prone to faults. LP7 occurs when both clones are modified, causing a divergence and then both are modified to reconcile the clone pair. The contribution of other types of LP is found to be system dependent. We also investigated the effect of size and the delay between the diverging change and reconciling change in a clone pair. For both of these characteristics, we find that the fault proneness varies with the size and period of delay. However, it is system dependent. Overall, we show that instances of LP can be filtered by their types to identify the most fault prone clone pairs. These types of LP can be further filtered by their size and the period of delay, but these two characteristics should be evaluated to determine their relationship with faults within a specific software system.

Currently our Gen generation framework supports Java systems. In the future, we plan to expand our framework to evaluate other languages. We will also examine the effect of the design phase or maturity of a project on LP. Lastly, we will increase the scope of our work to study more Gen patterns.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers of JSME for their valuable feedback that helped us to improve the quality of the paper. We would also like to thank Weiyi Shang at Queen's University, for his assistance in the extraction of evolutionary change and fault fix information from the software repositories, as well as Hao Yuan, Feng Zhang, Bipin Upadhyaya, Shuai Xie, and Shaohua Wang at Queen's University, for their assistance in evaluating J-REX.

REFERENCES

1. Kim M, Sazawal V, Notkin D, Murphy G. An empirical study of code clone genealogies. Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, ACM: New York, NY, USA, 2005; 187–196.

DOI: 10.1002/smr

2. Aversano L, Cerulo L, Di Penta M. How clones are maintained: an empirical study. *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, 2007; 81–90.
3. Thummalapenta S, Cerulo L, Aversano L, Di Penta M. An empirical study on the maintenance of source code clones. *Empirical Software Engineering* 2010; **15**:1–34.
4. Barbour L, Khomh F, Zou Y. Late propagation in software clones. *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, 2011; 273–282. DOI: 10.1109/ICSM.2011.6080794.
5. Göde N. Evolution of type-1 clones. *Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, 2009; 77–86. DOI: 10.1109/SCAM.2009.17.
6. Krinke J. A study of consistent and inconsistent changes to code clones. *Proceedings of the Working Conference on Reverse Engineering* 2007; **0**:170–178.
7. Saha R, Asaduzzaman M, Zibran M, Roy C, Schneider K. Evaluating code clone genealogies at release level: an empirical study. *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM '10, 2010; 87–96. DOI: 10.1109/SCAM.2010.32.
8. Göde N, Harder J. Clone stability. *15th European Conference on Software Maintenance and Reengineering*, 2011.
9. Krinke J. Is cloned code more stable than non-cloned code? *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, 2008; 57–66.
10. Göde N, Koschke R. Frequency and risks of changes to clones. *Proceedings of the 33rd International Conference on Software Engineering*, 2011.
11. Göde N, Harder J. Oops! . . . i changed it again. Proceedings of the 5th International Workshop on Software Clones, IWSC '11, ACM: New York, NY, USA, 2011; 14–20. DOI: http://doi.acm.org/10.1145/1985404.1985408. URL: http://doi.acm.org/10.1145/1985404.1985408.
12. Jiang L, Su Z, Chiu E. Context-based detection of clone-related bugs. Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE '07, ACM: New York, NY, USA, 2007; 55–64. DOI: http://doi.acm.org/10.1145/1287624.1287634. URL: http://doi.acm.org/10.1145/1287624.1287634.
13. Li Z, Lu S, Myagmar S, Zhou Y. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, USENIX Association: Berkeley, CA, USA, 2004; 20–20. URL: http://dl.acm.org/citation.cfm?id=1251254.1251274.
14. Bakota T, Ferenc R, Gyimothy T. Clone smells in software evolution. *Proceedings of the IEEE International Conference on Software Maintenance*, 2007; 24–33.
15. Bettenburg N, Shang W, Ibrahim W, Adams B, Zou Y, Hassan AE. An empirical study on inconsistent changes to code clones at release level. Proceedings of the 16th Working Conference on Reverse Engineering, WCRE '09, IEEE Computer Society: Washington, DC, USA, 2009; 85–94. DOI: http://dx.doi.org/10.1109/WCRE.2009.51. URL: http://dx.doi.org/10.1109/WCRE.2009.51.
16. Juergens E, Deissenboeck F, Hummel B, Wagner S. Do code clones matter? Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society: Washington, DC, USA, 2009; 485–495. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070547. URL: http://dx.doi.org/10.1109/ICSE.2009.5070547.
17. Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 2002; **28**:654–670.
18. Roy C, Cordy J. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, 2008; 172–181. DOI: 10.1109/ICPC.2008.41.
19. Shang W, Jiang ZM, Adams B, Hassan A. MapReduce as a general framework to support research in Mining Software Repositories (MSR). *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, 2009; 21–30.
20. Mockus A, Votta L. Identifying reasons for software changes using historic databases. *Proceedings of the International Conference on Software Maintenance*, 2000.
21. Shihab E, Ihara A, Kamei Y, Ibrahim W, Ohira M, Adams B, Hassan A, Matsumoto K. Predicting re-opened bugs: a case study on the eclipse project. *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 2010; 249–258. DOI: 10.1109/WCRE.2010.36.
22. Hassan A. Predicting faults using the complexity of code changes. *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, 2009; 78–88. DOI: 10.1109/ICSE.2009.5070510.
23. Hassan AE. Predicting faults using the complexity of code changes. Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society: Washington, DC, USA, 2009; 78–88. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070510. URL: http://dx.doi.org/10.1109/ICSE.2009.5070510.
24. Selim GM, Barbour L, Shang W, Adams B, Hassan AE, Zou Y. Studying the impact of clones on software defects. *Proceedings of the Working Conference on Reverse Engineering*, 2010; 13–21.
25. Sheskin D. Handbook of Parametric and Nonparametric Statistical Procedures (4th edn). Chapman & All, 2007.
26. Yin RK. Case Study Research: Design and Methods (3rd edn). SAGE Publications: Boca Raton, FL, 2002.
27. Hassan AE, Holt RC. Studying the evolution of software systems using evolutionary code extractors. Proceedings of the 7th International Workshop on Principles of Software Evolution, IEEE Computer Society: Washington, DC, USA: Thousand Oaks, CA, 2004; 76–81.

DOI: 10.1002/smr

AUTHORS' BIOGRAPHIES

Liliane Barbour is a Developer and QA Specialist at Namzak Labs. She received her BSc in Electrical Engineering from Queen's University in 2009 and her MASc in Software Engineering from Queen's University in 2012. Her research interests include clone genealogies and mining software repositories.

Foutse Khomh is an Assistant Professor at the Ecole Polytechnique de Montreal (Canada). He received a Ph.D in Software Engineering from the University of Montreal in 2010. His research interests include software maintenance and evolution, cloud engineering, service-centric software engineering, empirical software engineering, and software analytics. He also received a Master's degree in Software Engineering from the National Advanced School of Engineering (Cameroon) and a D.E.A (Master's degree) in Mathematics from the University of Yaoundé I (Cameroon). He has experience as Software Engineer at different companies doing research, system design, and project management. He has published several papers in international conferences and journals. He is a member of IEEE and IEEE Computer Society.

Ying Zou is an Associate Professor in Department of Electrical and Computer Engineering at Queen's University. She is also cross appointed to the School of Computing at Queen's University. She is the Canada Research Chair Tier II in Software Evolution at Queen's University in Canada. She is an IBM visiting scientist and was awarded twice IBM Faculty awards. Her research interests include: software engineering, software maintenance, software analytics, service oriented architecture, and business process management.

DOI: 10.1002/smr