Late Propagation in Software Clones

Liliane Barbour, Foutse Khomh, Ying Zou
Department of Electrical and Computer Engineering
Queen's University
Kingston, ON
{*l.barbour, foutse.khomh, ying.zou*}*@queensu.ca*

***Abstract*—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 the clone pair is modified, causing the clone pair to become inconsistent. The code segments are then re-synchronized in a later revision. 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 two long-lived software systems using the Simian and CCFinder clone detection tools. We define 8 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 cases of late propagations are more harmful than others. Specifically, two cases are most risky: (1) when a clone experiences inconsistent changes and then a re-synchronizing change without any modification to the other clone in a clone pair; and (2) when two clones undergo an inconsistent modification followed by a re-synchronizing change that modifies both the clones in a clone pair.**

***Keywords*-clone genealogies; late propagation; fault-proneness.**

I. Introduction

A code segment is labeled as a code clone if it is identical or highly similar to another code segment. Similar code segments form a clone pair. Clone pairs 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 revisions over their lifecycles. Each revision can involve modifications to code clones. As the clones in a clone pair are modified, a change evolution history, known as a clone genealogy [1], is generated.

In a previous study on clone genealogies, Kim *et al.* [1] define two types of evolutionary changes that can affect a clone pair: a consistent change or an inconsistent change. During a consistent change, both clones in a clone pair are modified in parallel, preserving the clone pair. In an inconsistent change, one or both of the clones evolves independently, destroying the clone pair relationship. Inconsistent changes can occur deliberately, such as when code is copied and pasted and then subsequently modified to fit the 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. Inconsistent changes can also occur accidentally. A developer may be unaware of a clone pair, and cause an inconsistency by changing only one half of the clone pair. This inconsistency could cause 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 inconsistent changes make code clones more prone to faults.

Late propagation occurs when a clone pair undergoes one or more inconsistent changes followed by a re-synchronizing change [2]. The re-synchronization of the code clones indicates that the gap in consistency is accidental. Since accidental inconsistencies 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.* began the initial work in examining the characteristics of late propagation. The authors measured the delay between an inconsistent change and a re-synchronizing 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 defects, 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.

In this paper, we study the characteristics of late propagation genealogies and estimate the likelihood of faults. Using clone genealogies from two open source systems, ArgoUML[1] and Ant[2], we address the following three research questions:

- *RQ1: Are there different types of late propagation?* Late propagation has been defined by several researchers [2], [4] as an inconsistent change followed by a re-synchronizing change. We perform an exploratory study to examine several late propagation patterns and investigate whether inconsistent clone pairs ever re-synchronize without a 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 III, we evaluate late propagation in greater depth, and examine if the risk of faults remains consistent across all types of late propagation.
- *RQ3: Which type of late propagation experiences the highest proportion of faults?* In the previous question, we determine if some types 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.

The rest of this paper is organized as follows. Section II summarizes related studies on clones and late propagation. Section III discusses different types of late propagation. Section IV explains the design of our study. Section V describes the study results. Section VI lists threats to the validity of the study. Finally, Section VII concludes the paper and explores future work.

II. Related Work

A. Clone Genealogies

The first study on code clone evolution was by Kim *et al.* [1] who analyzed clone classes (*i.e.*, 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.

B. Analysis of Clone Genealogies

Krinke [5] performed a study on five open source systems to examine consistent and inconsistent changes to code clones. He observed the systems over a 200 week period, using a time interval of one week between system snapshots. He used the Simian clone detection tool, but only examined identical clones. He found that clone pairs are changed consistently about half of the time, and that late propagation occurs very infrequently. He also found that during late propagation, the consistent change usually occurred within a week of the inconsistent change. This observation, coupled with his fixed time interval between system snapshots (one week) suggests that a more fine-grained time interval is necessary to fully understand late propagation.

Göde *et al.* [6] repeated and extended another of Krinke's studies [7] on the stability of cloned code. Similar to our work, they examined clones at the interval of one revision. 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 two different clone detection tools.

In an unrelated study, Göde *et al.* [8] 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, since 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.

Aversano *et al.* [2], examined clone genealogies to investigate how clones are maintained. They selected a specific stable snapshot of each of their two studied systems (one of which was ArgoUML), and traced these clones over time. They found that about 18% of the clones exhibited late propagation behavior. Out of the 17 instances of bug fixes, 7 occurred during late propagation, suggesting that late propagation is risky.

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. The authors also defined a specific type of late propagation, called delayed propagation, which occurs when the re-synchronizing change is made within 24 hours of a diverging modification.

[1]http://argouml.tigris.org/
[2]http://ant.apache.org/

Table I
DESCRIPTION OF LATE PROPAGATION TYPES.

Propagation Category	LP Type	Clone Pair	Clones Modified in Diverging Change	Clones Modified During Period of Divergence	Clones Modified During Re-synchronizing Change
Propagation Always Occurs	LP1	$< A, B >$	A	A	B
	LP2	$< A, B >$	A	A, B	B
	LP3	$< A, B >$	A	A	A, B
Propagation May or May Not Occur	LP4	$< A, B >$	A	A, B	A
	LP5	$< A, B >$	A	A, B	A, B
	LP6	$< A, B >$	A, B	A, B	$A or B$
	LP7	$< A, B >$	A, B	A, B	A, B
Propagation Never Occurs	LP8	$< A, B >$	A	A	A

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

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

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

//Re-synchronizing Change: Clone B, Revision 604
addField(new UMLComboBoxNavigator(this,"NavClass",
   new UMLComboBox(classifierModel)),2,0,0);
```

Listing 1. An example of a LP1 genealogy from ArgoUML.

III. 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 inconsistent changes followed by a re-synchronizing change [3]. For example, consider two clones that call a method. A developer modifies the call parameters of the method, and updates one of the clones to reflect the change. This causes the clone pair to become inconsistent. Later, she discovers the inconsistency, possibly because of a bug report, and propagates the change to the other clone. The clones are now re-synchronized.

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: either one, both, or neither of the clones experiences additional changes.
- Clones Modified During Re-synchronizing Change: either one or both of the clones is modified, re-synchronizing the clone pair.

Using all combinations of the three phases, we identify eight possible types of late propagation genealogies. Since they are not extracted from existing systems, the types may not appear in all the systems in our study. The characteristics of the individual types are described in Table I. It describes the possible sets of modifications to two clones, Clone A and Clone 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

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

Listing 2. An example of a LP8 genealogy from Ant.

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 re-synchronizing 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.

As listed in Table I, LP1, LP2, and LP3 belong to the first category, since a change must be always propagated between the clones in the clone pair to re-synchronize them. For example, in LP1 Clone A is modified, creating an inconsistency between Clone A and Clone B. Clone A can experience further changes during the period of divergence. Finally, all changes are propagated to Clone B, re-synchronizing the clone pair. Listing 1 is an example of an LP1 genealogy taken from ARGOUML using CCFINDER as the clone detection tool. As shown in Listing 1, two clones (*i.e.*, Clone A and Clone B) form a clone pair in revision 595. In revision 602, the parameter in the method call is updated, modifying Clone A. In revision 604, this change is propagated to Clone B, re-synchronizing the clone pair.

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, re-synchronizing the clone pair. Listing 2 is an example of an LP8 genealogy taken from ANT using CCFINDER as the clone detection tool. As shown in Listing 2, two clones (*i.e.*, Clone A and Clone B) form a clone pair in revision 270250. In revision 270264, Clone A is modified so that the string "m_" is added to the beginning of each variable name. In revision 271109, this change is reverted, re-synchronizing the clone pair. For space reasons, the listings discussed in this section contain only the interesting lines of code extracted from bigger clones.

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 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 A and B are modified during the period of divergence so it is possible that changes are propagated in both directions. For example, in LP4 without propagation, during the period of divergence changes could be applied to both clones consistently, but because of the initial diverging change, they remain inconsistent. If the initial diverging change is reverted on Clone A, the clones are re-synchronized without a propagation occurring. However, if the period of divergence experiences bidirectional propagations between Clone A and Clone B followed by a re-synchronizing propagation from Clone B to Clone A, LP4 is said to experience propagation. Therefore, because of changes experienced by both clones during the period of divergence, it is unclear for all types in the second category if a propagation occurs.

For the remainder of this paper, we use the abbreviations "LP" for Late Propagation and "Gen" for Genealogy.

IV. Experimental Setup

The *goal* of our study is to investigate the fault-proneness of clone pairs that undergo late propagation. 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 late propagation 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 two systems, ArgoUML and Apache Ant, which have different sizes and belong to different domains. ArgoUML is a UML-modelling 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 revisions 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 revisions in its revision history. We study code snapshots in an interval of observation ranging from January 2000 to November 2010.

Table II
Characteristics of the systems

System	# LOC	# Revisions	# Gen - CCFinder	# LP - CCFinder	# Gen - Simian	# LP - Simian
ArgoUML	3.1M	18k	14k	1.1k	111	23
Ant	2.3M	1.0M	30k	4.7k	461	80

Clone detection is performed using two existing clone detection tools, Simian[3] and CCFinder[9]. Simian is a string-based clone detection tool with a default minimum clone length of six lines. 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 (type 1 clones) or where some identifiers names and literals have been changed (type 2 clones). CCFinder is a token-based clone detection tool with a default minimum clone length of 50 tokens. Like Simian, it identifies type 1 and 2 clones.

Although other studies [5], [6] have investigated the impact of the minimum clone length on the percentage of consistent 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 two systems. In this work, we study all clone genealogies that contain at least one consistent modification beyond the creation of the clone pair. We filter out clones that diverge immediately and remain diverged for the remainder of their genealogy since they are likely to be false positive clones.

A. 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 [10] to mine the code repository of each subject system. J-Rex identifies the revisions that modify each Java

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

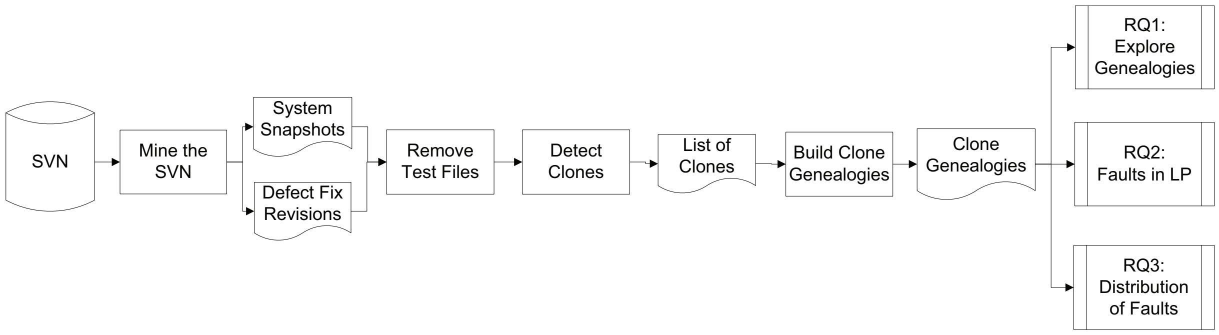

Figure 1. Overview of the analysis process.

file and outputs a snapshot of the file at those revisions. Revisions corresponding to fault fixes are marked during the process. Next, clone detection is executed to detect clones in the entire subject system. Using the clone detection results, the clones are mapped across their revisions to create clone genealogies. Each clone genealogy is examined to identify instances of late propagation. Lastly, the late propagation genealogies are categorized by the types of late propagation. We now describe each of the major steps in detail:

1) Mining the SVN to identify faults: We use J-REX [10] 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 revision. It then breaks each snapshot into component methods and flags any methods that have been modified since the last revision.

J-REX analyzes each commit message to identify the reason for a commit, such as a fault fix. It performs the analysis using the heuristics proposed by Mockus *et al.* [11] and used in prior fault studies [12], [13]. 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'. To ascertain the impact of this risk, one author manually examined all 1.8k commit messages in ARGOUML identified by J-REX to be a fault fix. The precision of J-REX was determined to be just over 85%.

Existing studies build clone genealogies between system snapshots taken at fixed intervals (*e.g.*, one week). The interval chosen can affect the creation of clone genealogies since any changes that occur between system snapshots are lost. Therefore, we examine clones between each revision, 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.

2) Removing Test Files: Both ARGOUML and ANT 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. Since 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.

3) Detecting Clones: We detect all of the clones in ARGOUML and ANT. To build the clone genealogies, we are interested in clones within the same revision 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 revision. Further details about our approach can be found in our study on the impact of software clones [14].

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 the similar reasons as Göde *et al.* [6]. 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 late propagation genealogies for each subject system using both 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, while CCFINDER uses a token-based technique. CCFINDER converts each Java file into a series of tokens during a pre-processing phase in order 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. We filter 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. The clone detection tool parameters also contribute to the discrepancy. CCFINDER specifies a minimum number of tokens, while SIMIAN specifies a minimum number of lines. To have the same minimum clone size, each Java file must have an average of around 8 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.

4) Building Clone Genealogies: To build the clone genealogies, we map clones from the clone list across revisions. 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 assign new line numbers to each clone in the clone pair, we query the SVN of each studied system using *diff*. 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 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 revisions where the methods containing the clones are modified. As mentioned previously, not all of these revisions modify the cloned code, but this step reduces the number of revisions that must be checked for changes. Using the revision number of the clone pair as a starting revision (*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 revision in the revisions list. We update the line numbers of the clone pair as needed to create an updated reference clone, and determine if the clones themselves are modified during the revision. If they are modified, we need to determine if the change was consistent or inconsistent.

A clone detection tool is used to determine if a change is consistent. Using the existing clone list obtained during the clone detection step, we identify a clone pair in the same revision that contains the same start and end line numbers. If no matching clone pair is found in the clone list, we add an inconsistent change to the clone genealogy. If the clone pair is found, the change is marked in the genealogy as a consistent change. We repeat the entire process for each revision in the revisions list, until each possible revision has been visited or the clone pair is removed.

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 revision, and yet it can continue to be modified in future revisions.

Our approach for generating clone genealogies is similar to the approaches used in other studies [4], [5]. 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 an inconsistency between the clones. 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 [5], we use existing clone detection tools, SIMIAN and CCFINDER, to detect consistent and inconsistent changes. In his work, Krinke made several assumptions when updating line numbers of clones between revisions. 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 consistent and inconsistent changes, we look for clones in the clone list that *contain* our updated reference clone. Therefore this scenario would still be considered a consistent change.

B. Analysis Method

RQ1: This question is preliminary to questions RQ2 and RQ3. It provides the quantitative data on the percentages with which different types of late propagation occur in our studied systems.

We address this question by classifying all instances of late propagation using the three characteristics described in Section III. For each type of late propagation, we report the number of occurrences in the systems.

RQ2: We investigate if some types of late propagation are more fault-prone than others. We compute the number of fault-containing and fault-free genealogies in each late propagation category. We compute the same values for non-late propagation clone genealogies that experience at least one change. For the remainder of this paper we use the abbreviation "Non-LP" for clone pairs that experience at

Table III
NUMBER OF CLONE PAIRS THAT UNDERWENT A LATE PROPAGATION

Propagation Category	LP Type	ArgoUML - Simian		ArgoUML - CCFinder		Ant - Simian		Ant - CCFinder	
		number	%	number	%	number	%	number	%
Propagation Always Occurs	LP1	1	4.35 %	31	2.75 %	12	15.00 %	187	3.94 %
	LP2	0	0 %	5	0.44 %	0	0 %	21	0.44 %
	LP3	0	0 %	207	18.37 %	0	0 %	92	1.94 %
Propagation May or May Not Occur	LP4	0	0 %	16	1.42 %	0	0 %	51	1.07 %
	LP5	1	4.35 %	66	5.86 %	0	0 %	46	0.97 %
	LP6	3	13.04 %	56	4.97 %	2	2.50 %	26	0.55 %
	LP7	11	47.83 %	622	55.19 %	35	43.75 %	2237	47.14 %
Propagation Never Occurs	LP8	7	30.43 %	124	11.00 %	31	38.75 %	2086	43.95 %

least one change but are not involved in any type of late propagation. We test the following null hypothesis[4] H_{02}: *Each type of late propagation genealogy has the same proportion of clone pairs that experience a fault fix.*

We use the Chi-square test [15] and compute the *odds ratio* (OR) [15]. The Chi-square test is a statistical test used to determine if there are non-random associations between two categorical variables. The odds ratio indicates the likelihood of an event to occur. It is defined as the ratio of the odds p of an event (*i.e.*, a fault fixing change) occurring in one sample (*i.e.*, experimental group), to the odds q of the event occurring in the other sample (*i.e.*, control group): OR $= \frac{p/(1-p)}{q/(1-q)}$. 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 while an $OR < 1$ indicates that it is more likely in the control group. Specifically, we compute two sets of odds ratios. First, we select the clone pairs that underwent a late propagation as experimental group. Second, we form one experimental group for each type LP_i of late propagation and re-compute the odds ratios. In both cases, we select the non-LP genealogies as the control group.

RQ3: We want to identify which type of late propagation experiences the highest proportion of faults. Therefore we test the following null hypothesis H_{03}: *Different types of late propagation have the same proportion of clone pairs that experience a fault fix*. For each type of late propagation, we calculate the sum of all faults experienced by instances of that type of late propagation. We use the non-parametric Kruskal Wallis test to investigate if the number of faults for the different types of late propagation are identical.

V. CASE STUDY RESULTS

This section reports and discusses the results of our study.

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

Using the types of the late propagation described in Section III, we categorize all instances of late propagation 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 late propagation instances for that system. For each system we repeat the experiment using both clone detection tools. Each column (*e.g.*, ARGOUML - SIMIAN) in Table III summarizes the distribution of late propagation clone pairs for a specific system (*e.g.*, ARGOUML) using a specific clone detection tool (*e.g.*, SIMIAN).

As summarized in Table III, four types of late propagation are dominant across both systems using two clone detection tools (*i.e.*, LP1, LP6, LP7, and LP8). The four 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 an inconsistent change followed by a re-synchronization. For all cases except ARGOUML using CCFinder, the 'propagation never occurs' category (*i.e.*, LP8) contributes more instances of late propagation than the 'propagation always occurs' category. ARGOUML using CCFINDER does not follow this trend; 20% of all late propagation instances always contain a propagation. As shown in Table III, LP7 occurs in an average of 48% of instances of late propagation, so it is the most common form of late propagation across all systems. However, LP7 is also the least understood of the types of late propagation. Since both clones in LP7 clone pairs are modified during all three steps of late propagation (*i.e.*, diverging, period of divergence, re-synchronization), it is unclear in which direction changes are propagated during the evolution of the clone pair. A few types of late propagation (*i.e.*, LP2, LP4, and LP5) contribute minutely to the number of late propagation genealogies.

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

B. RQ2: Are some types of late propagation more fault-prone than others?

Previous researchers [3] have studied the relationship between late propagation and faults. In this research question, we first replicate the earlier studies, and then extend the study to include the different categories of late propagation.

1) Fault-proneness of Late propagation: Table IV summarizes the results of the tests described in Section IV-B for instances of late propagation compared to non-late

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

Table IV
CONTINGENCY TABLE, CHI SQUARE TESTS 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*
Simian						
ArgoUML	36	52	8	15	0.768	0.770
Ant	94	287	30	50	**0.026**	1.831
CCFinder						
ArgoUML	2357	11402	443	684	$<$ **0.01**	3.133
Ant	7028	19301	1931	2815	$<$ **0.01**	1.884

propagation (LP) genealogies. The first and second columns in the table list the number of non-LP genealogies that experience fault fixes and the number that are free of fault fixes. The third and fourth columns show the same data for LP-genealogies. The last column of the table lists the odds ratio test results for each system using both clone detection tools. Except the case where ARGOUML is analyzed using SIMIAN, all of our results pass the Chi-square test with a p-value less than 0.05 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 [15]. In this study, the Fisher test provides the same information as the Chi Square test, so we do not present the Fisher test results in the tables.

In all the significant cases, the odds ratio is greater than 1, indicating that late propagation genealogies are more fault-prone than non-LP genealogies. However, for ARGOUML using SIMIAN, the result of the Chi Square test is not statistically significant. This can be explained by the small number of clone genealogies obtained with the SIMIAN detection tool. Overall, our results agree with previous studies [3] that found that late propagation is more at risk of faults.

2) Fault-proneness of Late Propagation Types: We repeat the previous tests, dividing the instances of late propagation into their respective late propagation types. We compare each type of late propagation to genealogies with no late propagation. Table V and Table VI respectively summarize the results obtained from SIMIAN and CCFINDER. For each type of late propagation, 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 odds ratio using the control group.

The Chi-Square test results for ARGOUML using SIMIAN in Table V are greater than 0.05, so they are insignificant. Therefore, they are excluded from consideration.

An examination of the significant cases in Tables V and VI reveals that the odds ratios are greater than 1, so each type of late propagation is more fault-prone than non-LP genealogies. There are three exceptions to this observation, LP1 in ANT in Table V, LP3 in ANT in Table VI, and LP2 in ARGOUML in Table VI. All three exceptions belong to the 'propagation always occurs' category. Thus, in general, the late propagation types are not more fault-prone than non-LP genealogies.

Comparing Tables V and VI to Table III, we conclude that there are many types that make up a small proportion of LP instances and have a very high odds ratio. Thus, when one of these LP types occurs, it is very likely to contain a fault fix. For example, LP6 has a high odds ratio (*e.g.*, 16.00 in ARGOUML in Table VI), but accounts for less than 5% of all late propagation instances in Table III.

The two most common late propagation types in the previous research question, LP7 and LP8, in general have low odds ratios in Tables V and VI. This indicates that although they occur frequently, they are less fault-prone than other less common late propagation types (*e.g.*, LP6).

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

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

In the previous question (RQ2), we examine which types of late propagation are the most prone to faults. In this question, we examine which types of late propagation contribute the most faults to each system. In other words, we examine if, when faults occur, do they occur in large numbers?

Table VII presents the distribution of faults for different types of late propagation. The 'Total' row represents the total numbers of faults over all late propagation genealogies. For example, for ANT using CCFINDER, there are 3100 fault fixes across all genealogies, as shown in the final row in Table VII. These 3100 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.

In order to validate the results, we perform the non-parametric Kruskal Wallis test which compares the distribution of faults between groups of different types of late propagation. Table VIII 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 late propagation types. From Table VIII, only ARGOUML using SIMIAN is not statistically significant.

Examining the results in Table VII for 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 odds ratios. Although they are less prone to faults, when they do experience faults, the faults are likely to occur in large numbers. In the case of ANT using CCFINDER, LP8 contributes over half of all fault fixes, but is one of the least fault-prone types compared to other late propagation types. The change causing the inconsistency 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.

Table V
SIMIAN - CONTINGENCY TABLES WITH THE CHI SQUARE TEST FOR DIFFERENT LATE PROPAGATION TYPES

Propagation Category		Ant				ArgoUML			
		Faults	No Faults	p-value	OR	Faults	No Faults	p-value	OR
	No LP	94	287	$<$ **0.01**	1	26	52	0.624	1
Propagation Always Occurs	LP1	1	11	$<$ **0.01**	0.277	0	1	0.624	0
	LP2	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	LP3	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Propagation May or May Not Occur	LP4	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	LP5	n/a	n/a	n/a	n/a	0	1	0.624	0
	LP6	1	1	$<$ **0.01**	3.053	0	3	0.624	0
	LP7	9	26	$<$ **0.01**	1.057	5	6	0.624	1.667
Propagation Never Occurs	LP8	19	12	$<$ **0.01**	4.834	3	4	0.624	1.500

Table VI
CCFINDER - CONTINGENCY TABLES WITH THE CHI SQUARE TEST FOR DIFFERENT LATE PROPAGATION TYPES

Propagation Category		Ant				ArgoUML			
		Faults	No Faults	p-value	OR	Faults	No Faults	p-value	OR
	No LP	7028	19301	$<$ **0.01**	1	2357	11402	$<$ **0.01**	1
Propagation Always Occurs	LP1	139	48	$<$ **0.01**	7.953	13	18	$<$ **0.01**	3.494
	LP2	17	4	$<$ **0.01**	11.672	0	5	$<$ **0.01**	0
	LP3	17	75	$<$ **0.01**	0.622	96	111	$<$ **0.01**	4.184
Propagation May or May Not Occur	LP4	23	28	$<$ **0.01**	2.256	10	6	$<$ **0.01**	8.063
	LP5	18	28	$<$ **0.01**	1.765	21	45	$<$ **0.01**	2.258
	LP6	18	8	$<$ **0.01**	6.179	43	13	$<$ **0.01**	16.000
	LP7	710	1527	$<$ **0.01**	1.277	210	412	$<$ **0.01**	2.466
Propagation Never Occurs	LP8	989	1097	$<$ **0.01**	2.476	50	74	$<$ **0.01**	3.269

Table VII
PROPORTION OF FAULTS FOR EACH TYPE OF LATE PROPAGATION.

Propagation Category		Ant - Simian		Ant - CCFinder		ArgoUML - Simian		ArgoUML - CCFinder	
		# of Faults	% of Faults	# of Faults	% of Faults	# of Faults	% of Faults	# of Faults	% of Faults
Propagation Always Occurs	LP1	1	2.17%	281	9.06%	0	0%	27	4.84%
	LP2	n/a	n/a	37	1.19%	n/a	n/a	0	0%
	LP3	n/a	n/a	24	0.77%	n/a	n/a	113	20.25%
Propagation May or May Not Occur	LP4	n/a	n/a	49	1.58%	n/a	n/a	10	1.79%
	LP5	n/a	n/a	31	1.00%	0	0%	30	5.38%
	LP6	1	2.17%	22	0.71%	0	0%	49	8.78%
	LP7	11	23.91%	1010	32.58%	10	71.43%	271	48.57%
Propagation Never Occurs	LP8	33	71.74%	1646	53.10%	4	28.57%	58	10.39%
	TOTAL	46	100.00%	3100	100.00%	14	100.00%	558	100.00%

Table VIII
RESULTS OF THE KRUSKAL WALLIS TESTS

System	Kruskal Wallis p-values
Ant - Simian	$<$ **0.01**
Ant - CCFinder	$<$ **0.01**
ArgoUML - Simian	0.571
ArgoUML - CCFinder	$<$ **0.01**

The remaining results are system-dependent. For example, in the case of ARGOUML using CCFINDER in Table VII, 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 late propagation are therefore very different. Thus, we reject H_{03}.

VI. THREATS TO VALIDITY

We now discuss the threats to validity of our study, following the guidelines for case study research [16].

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 two clone detection tools that have been used in previous studies and repeat the study for both tools.

Both of the clone detection tools in this study can detect identical (*i.e.*, type 1) and near-identical clones (*i.e.*, type 2). An additional line of code within a clone from a clone pair (*i.e.*, a type 3 clone) cannot be detected by the tools. The addition or deletion of a line of code to one clone segment and not the other is an inconsistent change. Thus, if we were to use a clone detection tool that detects type 3 clones, our genealogies would be inaccurate.

Another construct validity threat is the software evolution tool J-REX which uses heuristics to identify fault-fixing changes [10]. 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.* [17] and Mockus *et al.* [11]. Additionally, one author manually reviewed each commit message in ARGOUML identified by J-REX as a fault fix and calculated the precision of J-REX to be just over 85%.

Threats to *internal validity* do not affect this study, as it is an exploratory study [16]. Although we cannot claim causation, we do identify, in RQ2 and RQ3, a relation between late propagation and fault-proneness for clone pair genealogies. Furthermore, we have provided some qualitative explanation of our results based on 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 the normality of the distribution of the data.

Threats to *external validity* concern the possibility of generalizing our results. We examine two Java systems, both that use a plug-in architecture. Although the two selected systems are different sizes and belong to different domains, further validation on more systems should be performed.

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.

VII. Conclusion

In this paper, we extend previous studies to examine late propagation in more detail. We first confirm the conclusion from previous studies that late propagation is more risky than other clone genealogies. We then identify eight types of late propagation and study them in detail to identify which contribute most to faults in late propagation. Overall, we find that two types of late propagation (*i.e.*, LP8 and LP7) are riskier than the others, in terms of their fault-proneness and the magnitude of their contribution towards faults. LP8 involves no propagation at all, and occurs when a clone diverges and then re-synchronizes itself without changes to the other clone in a clone pair. LP7 occurs when both clones are modified, causing a divergence and then both are modified to re-synchronize the clone pair. The contribution of other types of late propagation is found to be system dependent. From this study, we can conclude that the different types of late propagation are inconsistently risky. Only some late propagation genealogies require monitoring. In the future, we plan to expand this study to include more systems.

Acknowledgment

The authors would like to thank Weiyi Shang, of Queen's University, for his assistance in the extraction of evolutionary change and fault fix information from the software repositories.

References

[1] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 187–196.

[2] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *11th European Conference on Software Maintenance and Reengineering*, 2007, pp. 81 –90.

[3] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, pp. 1–34, 2010.

[4] N. Göde, "Evolution of type-1 clones," in *Proceedings of the 9th International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2009, pp. 77–86.

[5] J. Krinke, "A study of consistent and inconsistent changes to code clones," *Working Conference on Reverse Engineering*, vol. 0, pp. 170–178, 2007.

[6] N. Göde and J. Harder, "Clone stability," in *15th European Conference on Software Maintenance and Reengineering*, May 2011.

[7] J. Krinke, "Is cloned code more stable than non-cloned code?" *IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 57–66, 2008.

[8] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *33rd International Conference on Software Engineering*, May 2011.

[9] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, pp. 654–670, 2002.

[10] W. Shang, Z. M. Jiang, B. Adams, and A. Hassan, "Mapreduce as a general framework to support research in mining software repositories (msr)," in *6th IEEE International Working Conference on Mining Software Repositories*, May 2009, pp. 21 –30.

[11] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings. International Conference on Software Maintenance*, 2000.

[12] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. Hassan, and K. Matsumoto, "Predicting reopened bugs: A case study on the eclipse project," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 2010, pp. 249 –258.

[13] A. Hassan, "Predicting faults using the complexity of code changes," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, May 2009, pp. 78 –88.

[14] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the impact of clones on software defects," *Working Conference on Reverse Engineering*, pp. 13–21, 2010.

[15] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[16] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[17] A. E. Hassan and R. C. Holt, "Studying the evolution of software systems using evolutionary code extractors," in *Proceedings of the 7th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 76–81.