EMPIRICAL STUDIES OF CODE CLONE GENEALOGIES

by

Liliane Jeanne Barbour

A thesis submitted to the Department of Electrical and Computer Engineering

In conformity with the requirements for

the degree of Master of Applied Science

Queen's University

Kingston, Ontario, Canada

January, 2012

Abstract

Two identical or similar code fragments form a clone pair. Previous studies have identified cloning as a risky practice. Therefore, a developer needs to be aware of any clone pairs so as to properly propagate any changes between clones. A clone pair experiences many changes during the creation and maintenance of software systems. A change can either maintain or remove the similarity between clones in a clone pair. If a change maintains the similarity between clones, the clone pair is left in a consistent state. However, if a change makes the clones no longer similar, the clone pair is left in an inconsistent state. The set of states and changes experienced by clone pairs over time form an evolution history known as a clone genealogy. In this thesis, we provide a formal definition of clone genealogies, and perform two case studies to examine clone genealogies. In the first study, we examine clone genealogies to identify fault-prone "patterns" of states and changes. We also build prediction models using clone metrics from one snapshot and compare them to models that include historical evolutionary information about code clones. We examine three long-lived software systems and identify clones using Simian and CCFinder clone detection tools. The results show that there is a relationship between the size of the clone and the time interval between changes and fault-proneness of a clone pair. Additionally, we show that adding evolutionary information increases the precision, recall, and F-Measure of fault prediction models by up to 26%. In our second study, 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.

Co-Authorship

The case study in Chapter 5 of this thesis is based on a published paper [1] co-authored with my supervisor, Dr. Ying Zou and Dr. Foutse Khomh, a post-doctoral fellow in our lab. The content of Chapter 3, which describes our approach for both case studies, is also based on this paper. I was the primary author of this paper. The case study in Chapter 4 is a report of a research project supervised by Dr. Ying Zou. I am the primary researcher in this project.

In the two studies described in the paper and project, Dr. Ying Zou supervised all of the research related to the work and polished the paper. Dr. Foutse Khomh also participated in the discussions related to both studies, and provided feedback and suggestions to improve the work. Dr. Foutse Khomh polished the paper and the project report for the case study in Chapter 4.

[1] L. Barbour, F. Khomh, and Z. Ying, "Late Propagation in Software Clones," in *Proc. 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 273-282.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Ying Zou for her wisdom and guidance while supervising my thesis research. Additionally, I would like to thank her for the opportunities to do research and the encouragement that she provided during my undergraduate degree.

Secondly, I am thankful to all the members of the Software Reengineering Research Group and the Software Analysis and Intelligence Lab. In particular, I would like to acknowledge Dr. Foutse Khomh, Dr. Ahmed E. Hassan, Dr. Bram Adams, Weiyi Shang, Hao Yuan, and Gehan Selim for their advice and assistance in producing this thesis.

I would like to thank my committee members, Dr. Jim Cordy, Dr. Ahmad Afsahi, Dr. Keyvan Hashtrudi-Zaad, and Dr. Ying Zou for their valuable feedback on this thesis.

Finally, I am grateful to my friends and family for all their support and encouragement, with special thanks to Jonathan Mash. Lastly, I would like to thank my parents, John Barbour and Francine Contrasty for all their endless love, advice, and support throughout my academic career.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Code duplication is a common practice in software development and maintenance. Duplication can occur either intentionally through the copy-and-paste actions of developers or be accidentally introduced. A duplicated code segment that is identical or highly similar to another segment is called a “code clone”. Two clones can be similar in terms of either their semantics or their structure, and are known as a “clone pair”. Two or more similar clone pairs form a “clone class”. 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 evolutionary history, known as a clone genealogy [2], is generated. In the following sub-sections, we discuss software repositories, and clone genealogies in more detail.

1.1.1 Clones

Previous studies have argued that cloned code reduces the maintainability of a software system and is more prone to faults [3-5]. This is because when developers modify a code segment, they need to be aware of clones in that code segment in order to properly propagate any changes. Failure to propagate changes can introduce or fail to fully eliminate faults in a software system. For example, a developer makes changes to a method to fix a fault described in a bug report. If the developer is not aware of clones of that method, the fix would not be propagated to the clones, so the fault remains in the system.

Table 1-1 is an example of a clone pair from the software system ArgoUML. This example demonstrates that inconsistencies can be introduced if changes are not propagated between clone pairs. All changes to Clone A and Clone B are bolded. As shown in the first row of data, the clone pair is created in revision 595. In revision 602, Clone A is modified, but the change is not propagated to Clone B. Later, in revision 604, Clone B is modified so that the change is propagated from Clone A to Clone B. The clones are now consistent.

Table 1-1: Example of a Code Clone Pair from ArgoUML

Revision Number	Clone A	Clone B
595	`addField(new UMLComboBox(typeModel),1,0,0);`	`addField(new UMLComboBox(classifierModel),2,0,0);`
602	`addField(new` **`UMLComboBoxNavigator(this,"NavClass",new UMLComboBox(typeModel))`**`,1,0,0);`	`addField(new UMLComboBox(classifierModel),2,0,0);`
604	`addField(new UMLComboBoxNavigator(this,"NavClass",new UMLComboBox(typeModel)),1,0,0);`	`addField(new` **`UMLComboBoxNavigator(this,"NavClass",new UMLComboBox(classifierModel))`**`,2,0,0);`

In previous work [3-5], a code clone is described as risky or fault-prone if clones are more likely to experience a fault than non-cloned code from the same software system. In this thesis, we compare different characteristics of code clones to determine if certain code clones are more likely to experience a bug (*i.e*, more fault-prone) than other code clones.

Opposing studies claim that under certain conditions, cloning is justified [6, 7]. In these cases clone management techniques are needed to monitor clones instead of removing them

through refactoring. For example, introducing clones can maintain the stability of the overall system by segregating unstable experimental code.

Previous researchers [8] have identified four types of clones:

- Type 1: The code segments are identical.
- Type 2: The code segments are identical apart from their identifier names or literals.
- Type 3: The code segments are type 2 clones, except lines of code have been added or removed.
- Type 4: The code segments are similar only in terms of their semantics.

Table 1-2: The Four Types of Clones

Type	Clone A	Clone B

1	`for(int i=0; i<10; i++) {`
	`  k = k + i;`
	`  j++;`
	`}`
		`for(int i=0; i<10; i++) {`
		`  k = k + i;`
		`  j++;`
		`}`

2	`for(int i=0; i<10; i++) {`
	`  k = k + i;`
	`  j++;`
	`}`
		`for(int` **`index`**`=0;` **`index`**`<20;`
		`index``++) {`
		`  k = k +` **`index`**`;`
		`  j++;`
		`}`

3	`for(int i=0; i<10; i++) {`
	`  k = k + i;`
	`  j++;`
	`}`
		`for(int` **`index`**`=0;` **`index`**`<10;`
		`index``++) {`
		`  k = k +` **`index`**`;`
		`j = k;`
		`  j++;`
		`}`

4	`for(int i=0; i<10; i++) {`
	`  k = k + i;`
	`  j++;`
	`}`
		`int i = 0;`
		`while(i<10) {`
		`  k = k + i;`
		`  j++;`
		`i++;`
		`}`

Table 1-2 shows an example of each type of clone between two clone segments in a clone pair, Clone A and Clone B. The differences between the two clones are bolded.

If a line is added to a type 1 or 2 clone, it is unclear if the clone becomes a type 3 clone or if the line should be also propagated to the other clone in a clone pair. In this thesis we examine consistent changes between clones in a clone pair. Therefore, we only examine type 1 and type 2 clones.

Clone detection tools are used to locate clones in software systems. Several different techniques exist to detected clones such as text-based, abstract syntax tree-based, program dependency graph-based, and metrics-based tools.

1.1.2 Software Repositories

A software repository acts as a central repository and management system for a software system. It allows multiple developers to make changes to a software system concurrently. A repository stores the master copy of the most recent version of a software system, but also tracks all the historical changes to a project. For example, a developer can query the system to obtain a snapshot of a specific file on April 13, 2004. The repository returns a copy of the file as it appeared on that date.

A software repository allows a developer to work on a local working copy of file and then upload any changes to the central repository. First, a developer sends a request to the repository to acquire a copy of the most recent version of a file. After making changes locally, he uploads his changes to the repository. The act of uploading (a “commit”) merges his changes with the master copy. Typically, the developer is given the option to tag their commit with a comment. For example, when fixing a fault in the system, a developer tags a fix with the comment “fix for bug 3478”.

Three of the most commonly used software repositories are the Concurrent Versioning System (CVS)[1], Subversion (SVN)[2], and Git[3]. The implementation details for each type of software repository are different. For example, SVN allows a user to upload multiple files in one commit. This is different from CVS, which only allows users to upload one file at a time. Most types of software repositories maintain a version number for the system or the files within the system. In a SVN, the version number of the overall system automatically increments after a user performs a commit.

1.1.3 Clone Genealogies

Once created, clones evolve as they are modified during both the development and maintenance phases of software systems. During the lifetime of a software system, a clone pair is either in a consistent state or an inconsistent state. Figure 1-1 is a pictorial representation of a clone genealogy. The arrows in Figure 1-1 represent a change that modifies one or both of the clones in a clone pair. As shown in Figure 1-1, a clone pair always begins in a consistent state. The solid lines between the two clones indicate that the clones are consistent. A clone pair is in a consistent state if the clones are recognized by a clone detection tool as identical or similar. In Figure 1-1, after undergoing a consistent change, the clone pair remains in a consistent state. A clone pair is in an inconsistent state if they are no longer similar. The clones in Figure 1-1 are in an inconsistent state after experiencing an inconsistent change. Clones that become inconsistent can later re-synchronize; and consistent clones can diverge. A diverging change 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. The clones can also diverge

[1] http://savannah.nongnu.org/projects/cvs
[2] http://subversion.apache.org/
[3] http://git-scm.com/

accidentally. A developer may be unaware of a clone pair, and cause an inconsistency by changing only one clone in the clone pair. This inconsistency could cause a software fault. For these reasons, a previous study [2] argued that accidental changes that diverge a clone pair make code clones more prone to faults.

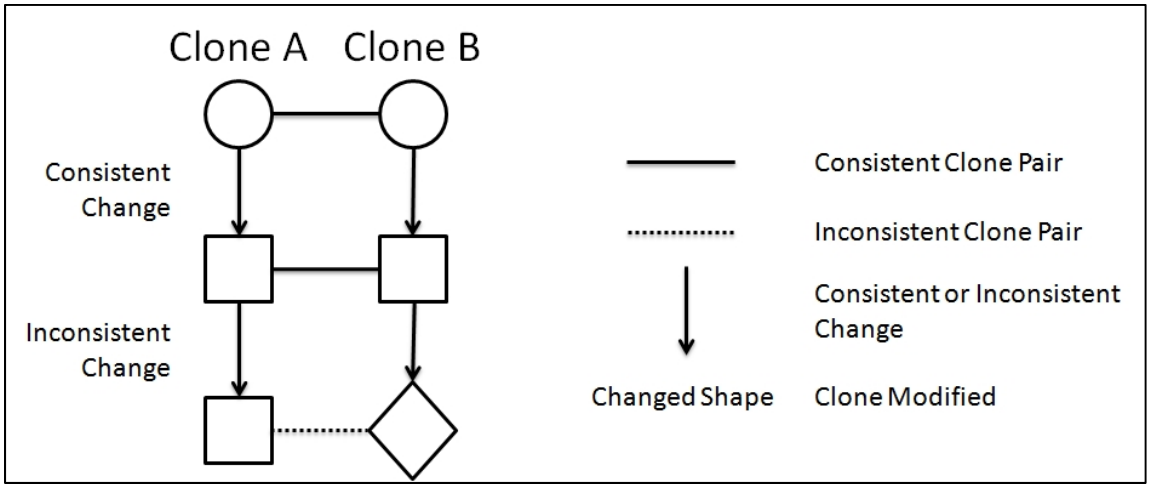

Figure 1-1: An Example of a Clone Genealogy

The set of states and changes between the states experienced by a clone pair across versions of a system is known as a "clone pair genealogy". Furthermore, a clone genealogy can exhibit specific "clone evolutionary patterns" as it evolves over the lifetime of a system. A clone evolutionary pattern defines a specific ordering of states and changes that frequently occur in clone genealogies. For example, a consistent clone pair that diverges to an inconsistent state and then re-synchronizes to a consistent state experiences the late propagation evolutionary pattern [9]. Previous studies [9, 10] have identified late propagation as a fault-prone clone evolutionary pattern.

1.2 Research Statement

Software developers and managers have limited resources for identifying risky code and testing it for faults [11]. Although previous studies have identified clones as fault-prone [12], we cannot assume that all clones are equally risky. The goal of this thesis is to investigate the fault-proneness of clone pairs and identify patterns and metrics that can be used to locate the most fault-prone clones. The results of this study will provide insight in deciding which code segments are the most at risk for faults and in prioritizing the code for testing. In this thesis, we study the following two aspects:

- **Prediction of Fault-prone Clones.** Previous studies on clone genealogies have defined specific clone evolutionary patterns and studied their relationship with faults [9, 10]. Although specific clone evolutionary patterns have been identified as fault-prone, the states and changes within the patterns have not been studied in detail. A genealogy only provides details about the past and cannot inform a developer about whether the current state or the next change is risky. However, the history of a clone pair has also not been considered when predicting faults in clones. In this thesis, we examine clone evolutionary patterns and changes within clone pair genealogies and their relationship with faults. Additionally, we investigate if metrics collected from the clone pair genealogy (*i.e.,* clone evolutionary metrics) can improve the precision and recall when predicting clone pairs that are at a higher risk of faults.
- **Late Propagation and Faults.** A few studies on clone evolution (*e.g.,* [9, 10]) have examined late propagation. They indicated that genealogies exhibiting a late propagation genealogy pattern are more fault-prone than other clone genealogies. Thummalapenta *et al.* [9] began the initial work in examining the characteristics of late propagation. They found that overall, late propagation experiences the highest proportion of fault fixing changes. In this thesis, we find that the late propagation evolutionary pattern accounts for between 2 to

22% of all clone genealogies that experience at least one change. If all instances of late propagation 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. Therefore, we examine more characteristics of late propagation to determine if only a subset of late propagation instances are at risk of faults.

1.3 Organisation

The rest of this thesis is organized as follows:

- Chapter 2: We outline related work in the areas of clone detection, empirical studies on code clones, clone genealogies, and fault prediction. Additionally, we describe the tests and models used in this thesis.
- Chapter 3: We describe our approach for extracting and detecting clones from source code repositories.
- Chapter 4: We present our first case study. The study examines the prediction of faults in code clones.
- Chapter 5: We present our second case study. It studies faults in a specific type of clone genealogy known as late propagation.
- Chapter 6: We summarize and conclude the thesis and discuss future work.

Chapter 2

Background and Literature Review

In this chapter, we discuss previous work in the areas of code clones, clone genealogies, and fault prediction. Additionally, we give an overview of the tests and modeling techniques used in both of our case studies in Chapter 4 and Chapter 5.

2.1 Harmfulness of Clones

Researchers are conflicted about how harmful clones are to software systems. In their book on refactoring, Fowler *et al*. [4] were one of the first to argue that code duplication (*i.e.*, clones), are one of the leading causes of "bad smells" in software systems. A smell is an undesirable flaw in the design or the implementation of the system [13]. Many researchers have argued [3, 5, 10] that clones reduce the maintainability of the code, and that developers should remove them whenever possible.

Lozano *et al.* [5] are among those who argue that cloning is harmful to software systems. They performed a study on the history of clones in the DNSJava[4] software system. They found that methods containing clones changed more often, giving support to the idea that clones lead to more frequent changes in a software system.

Geiger *et al*. [3] also showed that clones negatively impact software maintenance. They examined clones at the file level and assumed that there exists a coupling between cloned files. If one cloned file was modified, then they assumed that the clone of that file should also be modified. Using Mozilla[5] as their subject system, they showed that many cases exist where changes are not propagated, which is harmful to a software system.

---

[4] http://www.dnsjava.org
[5] http://www.mozilla.org

One of the most common approaches of removing clones is through refactoring. Several approaches for refactoring clones have been discussed in the literature [3, 14, 15]. However, more recent studies have indicated that all clones are not equally risky. In many cases, clones are never modified after they are created [16-18], so the effort to remove them does not justify the gain in maintainability.

Kasper and Godfrey [6] performed a case study showing that under certain conditions, cloning does no harm to software systems. They identified eight cloning patterns, similar to design patterns. These patterns formed a classification system for clones in software systems. Overall, they showed that in some cases, cloning is a positive practice, such as when a developer wants to avoid introducing unstable code into a software system.

Kim *et al.* [7] performed a study where they observed programmers and their copy-and-paste behavior. Their overall findings agreed with the cloning patterns proposed by Kasper and Godfrey [6]. For example, they found that the programming language contained limitations that introduced clones in a software system. This was one of the patterns proposed by Kasper and Godfrey. They also found that code was copied and pasted as a template and then customized for the new environment. Kim *et al.* concluded their work by suggesting that the developers require tools to track and support clones. Otherwise, a developer would be responsible for remembering each cloning relationship, and passing this knowledge to others working on the same code base.

Rahman *et al.* [19] examined the relationship between clones and faults. In a study of four systems, they found that clones were actually less fault-prone than non-cloned code. They also failed to find evidence that clone classes with more clones were more fault-prone.

2.1.1 Clone Coverage

Clone coverage refers to the density of clones within a software system. An increase in the number of clones over time can indicate a decline in the structure and maintainability of a software system [8].

Languë *et al*. [20] examined the changes in the clone coverage over six releases of an industrial software system over a three year period. Between 6.4% and 7.5% of the system contained clones. They concluded that although a significant number of clones were removed from the system, overall there was an increase in the number of clones within the system. They suggested that the maintainability of the code could have been improved using monitoring software to reduce the number of clones as they were created.

Antoniol *et al*. [21] investigated clone coverage in 19 releases of the Linux Kernel[6]. A high clone coverage was only detected in a few subsystems. They found that the clone coverage was stable across the releases. They concluded that this indicated that the structure of the system was not harmed by clones, as it remained stable.

Li *et al*. [22] also examined the Linux Kernel, in addition to FreeBSD[7]. They determined that the clone coverage increased from 16.2% to 22.3% between versions 1.0 to 2.6.6 of the Linux Kernel. In FreeBSD, the clone coverage increased from 17.5% to 21.7% between versions 2.0 and 4.10. In more recent versions of both systems, the clone coverage stabilized. In FreeBSD, it stabilized around 21-22%. Like Antoniol *et al*. [21], they found that a few subsystems were responsible for the increase in clone coverage. They suggested that in Linux, the clone coverage increase was due to increased support for similar device drivers.

Göde [23] examined deliberate clone removal from the developer's perspective. In an examination of four software systems, he showed that developers deliberately remove clones

[6] http://www.kernel.org
[7] http://www.freebsd.org

from software systems. Also, developers tended to remove clones that existed within the same source file. Additionally, as the number of developers in a project grew, the removal of clones became less intentional, such as through refactoring. The most common clone removal technique was refactoring using method extraction. Overall, he concluded that approaches need to be developed to identify clones that should be removed from software systems. In another study, Göde [18] examined type 1 clones and showed that overall, the clone coverage decreased over time, and that most clones existed in a system for a period of one year.

Zibran *et al*. [24] performed a study on clone density at the release level of 18 open source systems written in Java, C#, and C. Their study included type 3 clones, which were not generally included in previous studies on clone coverage. They found that clone coverage was language-dependent. Additionally, systems built using object-oriented languages had a higher proportion of type 1 clones. They determined that an increase in the number of functions in a system led to an increase in the number of clones, but showed only a weak correlation with an increase in the clone coverage. Lastly, they found that the clone coverage tended to change frequently in early releases of a software system. The clone coverage was more stable in later releases.

2.1.2 Clone Granularity

The clone granularity is the level of refinement of a clone within a software system. Clones are not limited to the source code of a software system. Researchers have studied clones at higher levels, such as in business process models [25]. Other researchers have studied clones in description languages like WSDL [26], and markup languages like HTML [27].

The level of granularity of clones can vary within a software system. Clone detection executed on software binaries has been used to identify software license violations of third party packages included with a software system [28]. Davis and Godfrey [29] executed clone detection

on generated assembly code to detect semantic (type-4) clones. Santone [30] proposes executing clone detection on Java bytecode to detect semantic clones.

Many studies examine clones consisting of several lines of code or tokens [2, 17, 18, 31-34]. However, researchers have also investigated clones at the function [20, 24, 35, 36], and class or file level [3, 37].

2.2 Clone Detection

There exist many different approaches for locating clones within software systems. Most clone detection tools can be classified into one of five approaches. In this section, we discuss each approach in detail. A more comprehensive summary of clone detection tools can be found in a survey on code detection research by Roy [8].

Text-based. Text-based clone detection tools [38-40] treat the source code as a sequence of strings. The tool compares all the strings within the code and returns sets of matching code fragments. An advantage of text-based approaches is that they are language independent. In order to detect type 2 clones, some form of normalization must occur on the code before clones are detected. Additionally, a minimum clone length (*e.g.*, six lines) is usually specified for each clone segment. In this thesis, we use Simian[8], a commercial text-based clone detection tool.

Token-based. Token-based clone detection tools [12, 22, 41, 42] include a parsing step before executing clone detection. In this parsing step, the code is mapped to a sequence of tokens. Because of the parsing step, token-based clone detection tools are language dependent. The tokenization process normalizes all identifiers and literals [12]. Like with text-based clone detection tools, token-based tools are given a minimum token length for each clone segment. During clone detection, the sequences of tokens are compared to locate matching subsequences. In this thesis, we use the academic token-based CCFinder clone detection tool [12].

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

Tree-based. Tree-based clone detection tools [43-45], like token-based tools, include a parsing step before locating matching clone pairs. During the parsing step, a tree-based tool creates either a parse tree or an abstract syntax tree representation of the source code. Due to the parsing step, tree-based approaches are language dependent. Normalization occurs during the construction of the tree. In order to detect clones, the tree is traversed to locate similar sub-trees.

Program Dependency Graph-based. Program Dependency Graph-based (PDG) clone detection tools [46-49] can be used to identify type-4 (semantic) clones. The source code is abstracted to extract the control flow and data flow graphs. The graphs are then compared to location matching sub-graphs.

Metrics-based. In a metric-based approach [50-53], the source code is divided into smaller units (*e.g.*, one line, one method, one class) and metrics are calculated for each unit. The metrics of each unit are compared and those with the same values are identified as clones. Examples of metrics are the number of function calls within a unit or the cyclomatic complexity of the unit. The type of metrics used by each tool impacts the language dependency of the tool.

2.3 Clone Genealogies

The first study on code clone evolution was by Kim *et al.* [2] who analyzed clone classes (*i.e.*, similar clone pairs) and defined patterns of clone evolution. They analysed groups of clone pairs, known as clone classes, and described the types of changes that can be experienced by a clone class. In this thesis we examine clones at the clone pair level to identify which clone pairs are most at risk of faults. A clone class with dozens of members may only contain a few risky clone pairs.

Through a case study on two Java systems using the CCFinder clone detection tool, Kim *et al.* [2] observed that the majority of clones in systems were 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. In this thesis, we continue to examine clones after they become inconsistent, as they may later re-synchronize. We examine if these clones are more at risk of faults. This thesis also strives for a deeper understanding of one specific type of clone genealogy, late propagation, to help developers efficiently focus their maintenance efforts.

Saha *et al.* [16] 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 to the final release. When studying consistent changes to clones, they found that on average, 24% of changes are consistent.

Krinke [34] 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 (type 1 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. His results may have been affected by the time interval of one week between snapshots. Changes to the clones, including inconsistent changes, may have occurred between snapshots. This suggests that a more fine-grained time interval is necessary to fully understand late propagation.

Göde *et al.* [31] repeated and extended another of Krinke's studies [33] 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 was more stable than non-cloned code. They also

confirmed that cloned code experienced more changes involving code deletion than non-cloned code. They experimented with the parameters of their clone detection tool and showed that the results are impacted by the choice of parameters. To mitigate this risk in this thesis, we use two different clone detection tools.

Göde *et al.* [17] performed a study on code clones that examined three different systems. They found that over half of the clones in the three systems were stagnant. In other words, once they were formed, they were never modified. 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. In this thesis, we explore only clones that experience changes and try to further reduce the number of clones that need to be monitored. We also consider details about the genealogy when predicting faults, so the number of previous changes to a clone pair is considered. In a separate study on type 1 clones, Göde [18] determined that the ratio of consistent to inconsistent changes was system dependent. However, overall most inconsistent changes never experienced a re-synchronizing change, so late propagation was rare in type 1 clones.

Aversano *et al.* [10], 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 and of much as 50% of all clones were consistently changed. Out of the 17 instances of bug fixes, 7 occurred during late propagation, suggesting that late propagation is risky. Surprisingly, of the remaining instances, there were more bug fixes in inconsistent genealogies (6 bug fixes) than in consistently changing genealogies (4 bug fixes). This indicated that inconsistently evolving clones should have continued to be monitored for bugs.

Thummalapenta *et al.* [9] performed a study on four open source C and Java systems, including ArgoUML. It looked at four different types of clone evolutionary patterns within clone classes. They classified their clone classes into consistent evolution, independent evolution, late propagation, and delayed propagation evolutionary patterns. They found that the first two patterns were the most common types. They concluded that each pattern experienced a different proportion of faults within a software system. In this thesis, we examine clones in more detail and define further clone evolutionary patterns. In terms of late propagation, they found that it 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. In this thesis, we extend the concept of a time interval between changes, and examine the impact of time on all types of clone genealogies.

In a recent study, Göde *et al.* [32] examined consecutive changes to code clones. They identify four different patterns of consecutive changes, consisting of combinations of consistent and inconsistent 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 inconsistent changes in clones were intentional. They attempted to find, but were unable to report, a relationship between unwanted inconsistencies and the delay between changes, the number of authors modifying the clone pair, and the location of clones relative to each other (*e.g.*, same file). In this thesis, we examine evolutionary metrics that can be used to find faults, even in inconsistent clone pairs.

Saha *et al.* [54] developed a tool that extracts clone genealogies at the clone class level. Their tool tracks genealogies at multiple granularity levels (*e.g.*, release level or revision level)

and can handle type 3 clones. They validated their tool using three open-source systems and compared it to an existing incremental clone detection tool. Their results indicated that it had high recall and precision. Their tool also supports the identification of clone evolutionary patterns at the clone class level. Like our approach described in Section 3.5, their tool supports the use of different clone detection tools when building clone genealogies.

2.4 Faults in Clones

In this section, we give an overview of the approaches used by previous researchers to locate faults in code clones. We discuss static approaches for fault location, followed by approaches that consider the evolutionary or change history of code clones.

Static Approaches

Jiang *et al.* [55] examine the context of clones to locate faults. They assume that when code is copied and pasted into a new context, that faults can be introduced if the code is not modified properly 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. In this thesis, we investigate other approaches for identifying faults in clones.

Li *et al.* [22] used their clone detection tool CP-Miner to detect faults in software systems. Their tool located inconsistently renamed identifiers in clones. In their case study, they were able to identify 49 faults in a version of Linux, and 32 faults in FreeBSD, many of which were unreported.

Locating Faults in Clones Using Evolutionary Information and Changes

Bettenburg *et al*. [56] argued that clones should be analysed at the release level. They suggested that clones are highly volatile, so only clones that affect the end user should be studied.

In a study of two open-source software systems, they found that only 1-3% of inconsistent changes caused faults at the release level.

Juergens *et al.* [57] analysed inconsistent changes in three industrial software systems. They found that over half of the clones contained inconsistencies. Of these, as much as 23% contained a fault. Overall they concluded that inconsistent clones, especially those introduced unintentionally in the system, were at a high risk of faults. In our case studies, we examine other clone genealogies, such as those that contain consistent changes, to determine if they also require monitoring for faults.

Bakota *et al.* [13] argued that in order to isolate risky clones, clones should be seen as dynamic instead of static. Clones that experienced many changes during their evolution may help locate faults in the system. They presented several cases where dynamic analysis of clones can highlight 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 using evolutionary information can successfully locate faults in the system. In this thesis, we examine evolutionary information in more detail to determine if specific patterns are more fault-prone than others.

2.5 Fault Prediction Models

Fault prediction models are used to identify fault-prone modules within a software system. Using the results of a prediction model, developers can prioritize their limited testing resources [11]. Several fault prediction models have been previously proposed [11, 58-65].

Effort Aware Prediction Models

The effort required to locate faults is different depending on the location of a fault in a software system. In order to allocate limited testing resources, developers would be aided by the

ability to differentiate the effort between possible faults. Some researchers [11, 66] have proposed models that are "effort aware" when predicting faults in a software system.

Process Metrics vs. Product Metrics

Several studies [58-61, 64] have investigated models that include evolutionary information (process metrics) and compared them to models using information available in one snapshot (product metrics). Overall, they found that process metrics are better at predicting fault than product metrics. In this thesis, we examine whether clone-specific evolutionary metrics can be used to increase the precision and recall of fault prediction in code clones.

2.6 Tests and Modeling Techniques

In this section, we discuss the tests and modeling techniques used in our case studies.

2.6.1 Odds Ratio

The odds ratio (OR) is a calculation of the likelihood of the occurrence of an event. It compares two groups, a control group and an experimental group. It then calculates the odds of an event occurring in the experimental group, compared to the control group. In this thesis, the event of interest is a developer performing a change to a clone pair that is a fault fixing change. So given a control group consisting of a sample of clone pairs p, and an experimental group q consisting of a different sample of clone pairs, the odds ratio is calculated as:

$$OR = \frac{p/(1-p)}{q/(1-q)} \qquad (2\text{-}1)$$

Using the results of the odds ratio, we can conclude one of the following:

- if OR = 1 then that the event is equally likely in both samples;
- if OR > 1 then the event is more likely in the experimental group; or
- if OR < 1 then the event is more likely in the control group.

2.6.2 Chi-Square Test

The Chi-Square test is a statistical test used to determine if there are non-random associations between two categorical variables [67]. It examines the "goodness of fit" between what is observed and what is expected. If the results of the Chi-Square test are statistically significant, then our results can be said to be due a reason other than chance variations in the data. For example, we use the test to examine our Odds Ratio calculations. If the Chi-Square test is not statistically significant, then we cannot disregard that the chance is responsible for the difference between the control and experimental groups in our Odds Ratio calculations. Therefore, we perform the Chi-Square test for all Odds Ratio calculations, to confirm their statistical significance. The Chi-Square test is said to be statistically significant if its *p-value* is less than 0.01.

2.6.3 Kruskal-Wallis Test

The Kruskal-Wallis test is a non-parametric test. It is used to evaluate whether groups of data have independent distributions [67]. For example, if the clone genealogies are classified into their different clone evolutionary patterns, we can use the test to determine if the number of faults for each pattern is different compared to each other. They are considered to be different enough from each other if the result of the Kruskal-Wallis is statistically significant. The test is statistically significant if the *p-value* of the test is less than 0.01.

2.6.4 Information Retrieval Metrics

In this section we define the three most common metrics used to evaluate prediction models: precision, recall, and F-Measure [60, 68]. In this thesis, we are predicting faults in clone pairs. The model classifies each clone pair as either faulty or non-faulty. Therefore, we define the three metrics in terms of this classification of clone pairs [69]:

- **Precision.** The number of clone pairs correctly classified as faulty over the number of all clone pairs classified as faulty.
- **Recall.** The number of clone pairs correctly classified as faulty over the total number of actually faulty clone pairs.
- **F-Measure.** The harmonic mean of precision and recall. It is calculated using the following equation:

$$Fmeasure = \frac{2 \times precision \times recall}{precision + recall} \qquad (2\text{-}2)$$

2.6.5 Random Forest Models

In this thesis, we build prediction models using the Random Forest algorithm in the Weka data mining tool[9]. We use Random Forests for both exploring the interactions between predictor variables and for prediction. A predictor is a variable or metric (*e.g*, the number of lines of code in a file) that can be used to predict a specific outcome (*e.g*, a fault in the file). A Random Forest [70] consists of a set of tree-structured prediction models. Each tree uses a classification approach to predict an output value. A pictorial representation of a tree prediction model is shown in Figure 2-1. The tree is constructed using six predictors: A, B, C, D, E, and F. In order to predict the value of the output variable, the tree is traversed according to the values of the six predictor variables. The traversal begins at the root of the model. In Figure 2-1, the root contains predictor A, so the model first evaluates the value of predictor A. If the value is less than or equal to 1000, it then traverses the left branch and evaluates predictor B. If the value is greater than 1000, it traverses the right branch and evaluates the node containing predictor C. This cycle continues until the traversal reaches a leaf of the tree. The leaves of the tree are shown as 'True' and 'False' at the bottom of the tree in Figure 2-1. The leaf contains the predicted value of the output variable

[9] http://www.cs.waikato.ac.nz/ml/weka/

based on the given input. The model only traverses the tree from its root to a leaf. It cannot traverse the branches in reverse. Therefore, only a subset of the predictor variables is used during prediction.

Four predictions examples are described in Table 2-1. The first six columns contain the values of each of the six predictor values. The next column lists the predictor variables used during the evaluation of the predictors when applied to the model in Figure 2-1. The last column contains the predicted output of the model for each example. In the first example, predictor A is less than 1000, so we traverse the model to the node containing predictor B. Since B is equal to 1, we traverse the model to the node containing predictor D, which is equal to 1. Therefore, the model predicts that the output variable to be true, based on the given predictor variables. The model we describe is a simplified model. Prediction trees in a Random Forest can be much bigger and evaluate multiple predictors at each node.

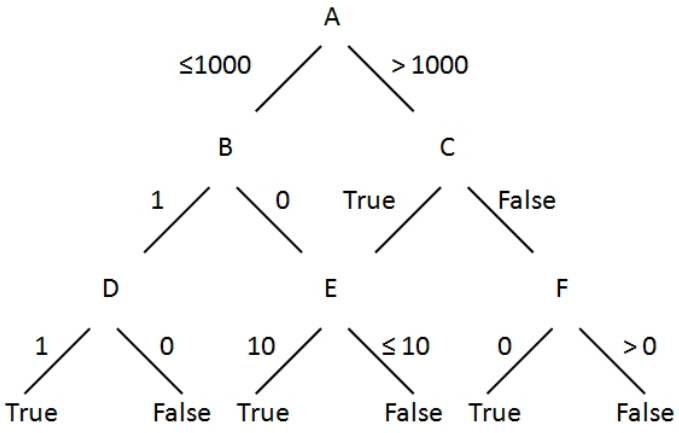

Figure 2-1: Example of a Tree Classification Model

Table 2-1: Example Input Data and Output from Tree Classification Model

A	B	C	D	E	F	Used in Prediction	Output
35	1	True	1	9	0	A, B, D	True
2000	0	True	0	22	1	A, C, E	True
190	0	False	0	4	0	A, B, E	False
4504	0	False	1	37	1	A, C, F	False

We choose the Random Forest classification approach for this work because it is one of the most successful classification methods, with performance on the level of other machine learning techniques, such as boosting and support vector machines. It is fast, robust against noise, and does not overfit [71]. It also accommodates categorical (*e.g.*, Group A, Group B, Group C, etc.) and binary (*i.e.*, true or false) predictor variables.

Given a training data set consisting of *m* predictor variables, the Random Forest algorithm builds a set of decision trees. Each tree is constructed using a different sample from the original training data set. We select one of the variables as the output variable, which will be predicted by the Random Forest. In this study, we assign the output variable to be whether or not a change to a genealogy is a fault fix. The output from our models has one of two possible results, true or false.

Once the model is built, it can be used to predict the output variable for a new sample. Using the sample, each tree predicts the value of the output variable independent from all other trees. Each tree then votes for one of the possible values of the variable. The model returns the winner of the vote.

The Random Forest error rates depend on two factors: the correlation between any two trees in the forest and the strength of each individual tree in the forest. Increasing the correlation increases the forest error rate while increasing the strength of the individual trees decreases the forest error rate. A tree with a low error rate is a strong predictor.

Random Forests can efficiently handle large sets of data, select from among thousands of input variables without eliminating any variables during the tree building process, and identify the most important predictor variables. In our case study, the output variable in our training dataset is much more frequently false than true. The Random Forest can handle this unbalance in the data set and control for it.

We use the permutation accuracy importance measure implemented in an R[10] add-on package [72] to assess the importance of each predictor in our models. The permutation accuracy importance measure is the most advanced variable importance measure available in Random Forests [70].

2.7 Summary

In this chapter, we give an overview of previous work in the area of clones, clone genealogies, and fault prediction. This includes details about the harmfulness of clones, clone detection techniques, and fault prediction models. We also discuss tests and models used in our case studies, including the odds ratio, the Chi-Square test, the Kruskal Wallis test, Information Retrieval metrics, and Random Forest Models.

[10] http://http://cran.r-project.org/

Chapter 3

Mining Clone Genealogies from Software Systems

In this chapter, we present an overview of our approach for building clone genealogies and locating changes that fixed a fault within a software system. More specifically, we discuss our approach for mining a software repository, locating clones within the software system, and building clone genealogies by tracking the clones throughout the history of the software system.

3.1 Overview of Our Approach

Figure 3-1 shows an overview of our approach to collect and process clone data to build clone genealogies. First, we use the tool J-Rex [73] to mine the source code repository of each subject system. J-Rex identifies the revisions that modify each Java file and outputs a snapshot of the file at those revisions. Revisions corresponding to fault fixes are marked during this 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. In the following subsections, we discuss our approach in more detail.

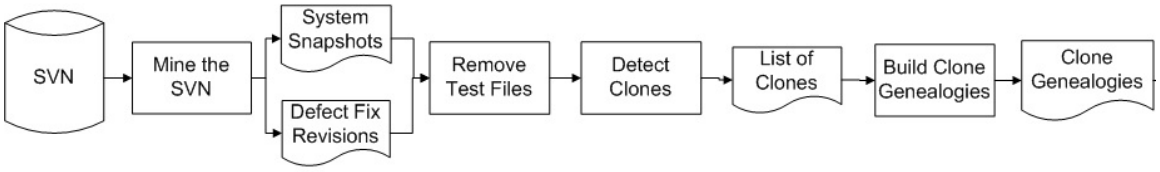

Figure 3-1: Overview of our Approach

3.2 Mining the Subversion Software Repository

We use J-Rex [73] 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 its 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*. [74] and used in prior fault studies [75, 76]. For example, if a commit message contains the word “bug”, it is classified as a fault fix. However, 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, we 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 a software repository.

3.3 Removing Test Files

All our subject systems contain files that are not used during the 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 case studies.

3.4 Detecting Clones

We first detect the clones in our subject systems. 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.

In the first step, we perform clone detection on all the Java file snapshots from the software repository. Before executing clone detection, we pre-process the Java file snapshots to extract the methods. We do this for the similar reasons as Göde *et al*. [31]. 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 all the method snapshots for clone detection. After clone detection, we perform a post-processing step on the clone list to identify clone pairs that existed within the same revision.

In this thesis, we use two existing clone detection tools, Simian[11] and CCFinder [12]. Simian is a string-based clone detection tool that identifies both type 1 and type 2 clones. For Simian, we select a minimum clone length of five lines of code, and set it to ignore literals and balance all parentheses and square brackets. CCFinder is a token-based clone detection tool with a default minimum clone length of 50 tokens. Before detecting clones, CCFinder performs a pre-processing step that tokenizes source code. Following a set of transformation rules, CCFinder inserts and deletes tokens. This leads CCFinder to detect clones that contain gaps of tokens [12]. However, these gaps are not sufficient for CCFinder to detect an entire inserted statement [8],

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

which is a requirement of the definition of type 3 clones. Therefore, CCFinder detects type 1, type 2, and the first two types with gaps. We use the default settings for CCFinder.

3.5 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 software repository of each studied system using *diff*, 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 done 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 in the clone list, 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 software repository 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 of the updated reference clone. 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. Due to the large number of clone pairs in a software system, we automate our clone genealogy building step.

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 done 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. Furthermore, a genealogy is also generated for the larger clone pair, so its genealogy is also considered when training and testing our prediction models in our case study.

Our approach for generating clone genealogies is similar to the approaches used in other studies [18, 34]. Both Göde [18] and Krinke [34] 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 the 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 [34], 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 this thesis:

- If a change occurs before the start of the clone, or after the end of the clone, the clone is not modified.

- If an addition occurs starting at the first line number of a clone, the clone shifts within the method but is not modified.
- If a deletion occurs anywhere within the clone boundaries, the clone is modified and its size shrinks.
- 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.

3.6 Summary

In this chapter, we present an overview of our approach for extracting and processing clone genealogies from software systems. We discuss each step in the approach in detail. First, we mine the software repository using a tool called J-Rex. It outputs snapshots of the system at each commit, and determines if the commit was a fault fix. Using the snapshots, we execute clone detection to locate all clones in the entire history of the software system. Lastly, we map the clones across each commit in a software repository to build clone genealogies.

Chapter 4

An Empirical Study of the Fault-proneness of Clone Evolutionary Patterns

In this chapter, we present the first case study in this thesis. We explain our model which describes clone genealogies as a finite transition system. Using this model, we define six clone evolutionary patterns as paths within the finite transition system. We propose three research questions that build on our model to examine the fault behavior of the clones and whether factors such as the size or time interval between changes has an effect on the fault-proneness of clone pairs. Lastly, we examine whether adding evolutionary information about a clone pair to a fault prediction model increases the precision and recall of the models.

4.1 A Model to Describe Clone Genealogies and Clone Evolutionary Patterns

4.1.1 Clone Pair States

A clone pair can either be in a consistent state (C_s) or an inconsistent state (I_s). We define the set of states of a clone pair as $S = \{C_s, I_s\}$. The two states are shown as circles in Figure 4-1. The consistent and inconsistent states are identified by a clone detection tool. If the tool identifies a clone between the two code segments, the clone pair is in a consistent state. If it does not identify a clone between the code segments, the clone pair is in an inconsistent state. An inconsistent clone pair can transition back to a consistent state (C_s) at a later time, so we continue to study inconsistent clone pairs.

4.1.2 Changes

A change is an input action that modifies the contents of one or both of the clones in a clone pair. It can transition the clone pair between states, or maintain the clone pair's current state. There are four possible changes:

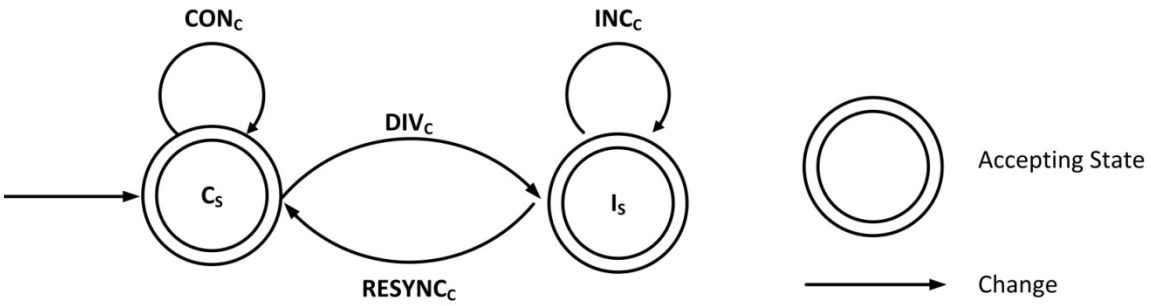

Figure 4-1: Clone Pair States and Changes

- Consistent Change (CON_c): A change modifies one or both clone segments of a clone pair in a consistent state. Such a change is not dramatic, and the clone segments in the clone pair are still detected as clones (*i.e.*, consistent change CON_c is the transition from consistent state C_s to consistent state C_s).
- Inconsistent Change (INC_c): A change modifies one or both clone segments of clone pair in an inconsistent state. The clone segments continue to be undetectable as a clone pair, so the clone pair remains in an inconsistent state. (*i.e.*, inconsistent change INC_c is the transition from inconsistent state I_s to inconsistent state I_s).
- Re-synchronizing Change ($RESYNC_c$): A change modifies one or both clone segments of clone pair in an inconsistent state. The change is sufficient to allow the clone segments to be detected as a clone pair. The clone pair transitions to a consistent state (*i.e.*, re-synchronizing change $RESYNC_c$ is the transition from inconsistent state I_s to consistent state C_s).
- Diverging Change (DIV_c): A change modifies one or both clone segments in a clone pair in a consistent state. The change results in the clone segments no longer being detectable as a clone pair (*i.e.*, diverging change DIV_c is the transition from consistent state C_s to inconsistent state I_s).

4.1.3 Clone Genealogy Model

A clone genealogy describes the evolutionary history of a clone pair. We define a clone genealogy as a finite transition system, $G = \{S, Act, Trans, I_o, A\}$, where:

- The set of states is $S = \{C_s, I_s\}$;
- The set of actions (*i.e.*, changes) is $Act = \{CON_c, INC_c, RESYNC_c, DIV_c\}$;
- The transition relations are

  $Trans = \{(C_s, CON_{c,} C_s), (C_s, DIV_{c,} I_s), (I_s, INC_{c,} I_s), (I_s, RESYNC_{c,} C_s)\}$;
- The set of initial states is $I_0 = \{C_s\}$; and
- The accepting states are $A = \{I_s, C_s\}$

Figure 4-1 is a pictorial representation of the clone genealogy transition system. A genealogy is a finite model, and grows as changes are applied to a clone pair, terminating in either a consistent or an inconsistent state (the accepting states). A clone pair starts from a consistent state when the clone pair can be detected. Therefore, the initial state of a clone genealogy is always a consistent state. The set of actions, shown by as arrows in Figure 4-1 consists of the changes that can be applied to a clone pair. The transition relations describe the starting and ending states when each action is applied.

4.1.4 Clone Evolutionary Patterns

A "clone pair evolutionary pattern" is a path in a graph G. It is a finite sequence of states $P = s_0 s_1 s_2 \ldots s_n$ where $s_0, s_1, s_2, \ldots, s_n$ in $= \{C_s, I_s\}$. The following six evolutionary patterns define all possible paths in graph G, where n is an integer greater than zero:

- Unchanged Pattern (UNC_p): The clone pair is formed, but never experiences any changes (*i.e.*, UNC_p is defined as the path C in graph G).

- Synchronous ($SYNC_p$): The clone pair has experienced one or more changes, but remains in a consistent state (*i.e.*, $SYNC_p$ is defined as the path $C_s C_s^n$ in graph G).
- Inconsistent Pattern (INC_p): After the creation of the clone pair, it transitions to an inconsistent state without ever experiencing any consistent changes (*i.e.*, INC_p is defined as the path $C_s I_s^n$ in graph G).
- Divergent Pattern (DIV_p): The clone pair experiences one or more consistent changes before transitioning to an inconsistent state (*i.e.*, DIV_p is defined as the path $C_s C_s^n I_s^n$ in graph G).
- Late Propagation Pattern (LP_p): the clone pair transitions from a consistent state to an inconsistent state. Later, it experiences a re-synchronizing change that transitions it back to a consistent state (*i.e.*, LP_p is defined as the path $(C_s^n I_s^n)^n C_s^n$ in graph G).
- Late Propagation with Diversion Pattern ($LPDIV_p$): the clone pair undergoes late propagation, but later it experiences a diverging change that brings it back to an inconsistent state (*i.e.*, $LPDIV_p$ is defined as the path $(C_s^n I_s^n)^n C_s^n I_s^n$ in graph G).

Figure 4-2 describes each of these definitions as a diagram. A change is represented by an arrow. Clones in a consistent state are connected by a solid line. As an example, consider a clone pair between code segments A and B. When the clone pair is created it is in a consistent state (C_s). Its genealogy is described by the graph G. It has an unchanged evolutionary pattern, as in Figure 4-2A. If it then experiences a consistent change (CON_c), it will have a synchronous pattern, as shown in Figure 4-2B. If this is followed by an diverging change (DIV_c), then it will have the path $C_s^2 I_s$ in graph G, which belongs to the divergent evolutionary pattern (DIV_p) (Figure 4-2D). If the clone pair then undergoes a re-synchronizing change ($RESYNC_c$), it will have the

path $C_s^2 I_s C_s$ in its graph G. The clone pair is now displaying a late propagation evolutionary pattern (LP_p) as shown in Figure 4-2E.

A clone pair with an unchanged pattern (UNC_p) never changes, and therefore has no evolutionary history. These clone pairs are excluded from our case study.

The inconsistent and divergent evolutionary patterns are similar. However, in a divergent evolutionary pattern (DIV_p), a clone pair must experience at least one consistent change before a diverging change occurs. A clone pair demonstrating an inconsistent evolutionary pattern (INC_p) diverges immediately after the clone pair is formed. Clone pairs exhibiting an inconsistent pattern (INC_p) may be "false positive" clones, since the clone pair never experiences any consistent or re-synchronizing changes. They may also be intentionally transitioned to an inconsistent state. For example, a developer may copy code and then extensively modify it for a new environment [6]. Because clones exhibiting inconsistent and divergent patterns are not able to be identified by a clone detection tool, they are more difficult to monitor, and could be more at risk of faults due to a lack of propagation of changes.

Late propagation (LP_p) occurs much less frequently than other evolutionary patterns [9]. However, previous studies [9] have shown that the late propagation is risky and fault-prone, especially given that the clone pair is later re-synchronized. For example, the diverging change in a late propagation may be accidental. However, accidental changes to clones are considered risky. Therefore, late propagation is considered risky [9]. Late propagation with diversion ($LPDIV_p$) is a special case of the late propagation evolutionary pattern. A clone pair first experiences a late propagation evolutionary (LP_p) pattern (a diverging change later followed by a re-synchronizing change). The clone pair then diverges a second time, creating the late propagation with diversion ($LPDIV_p$) evolutionary pattern. The frequent change of a state in the late propagation with

diversion pattern might indicate that developers have difficulty in monitoring and propagating changes between clone pairs.

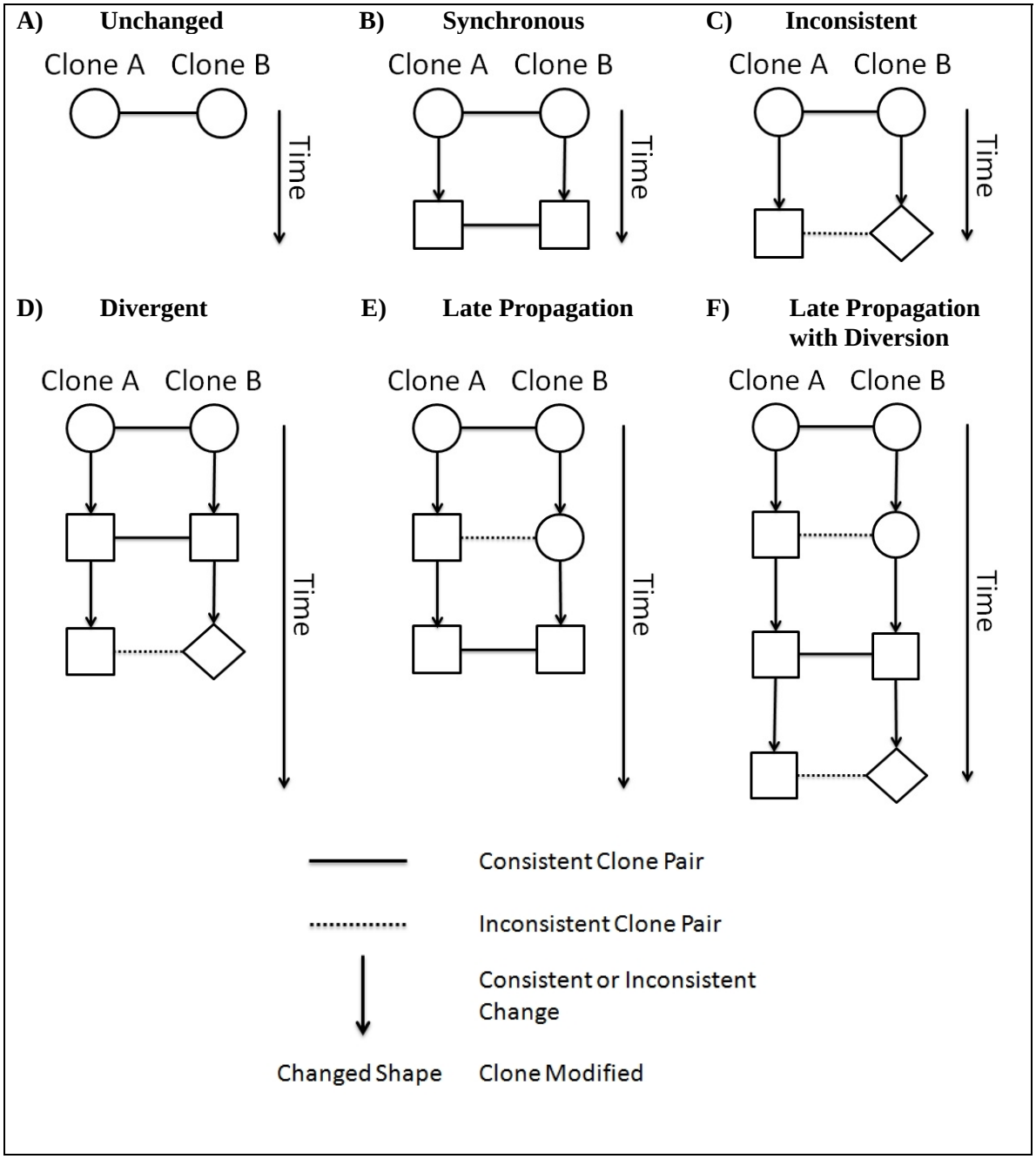

Figure 4-2: Clone Genealogy Evolutionary Patterns

The set of changes of clone genealogy described by a path in graph G can always be described by one of the six evolutionary patterns. However, other patterns are possible. The unchanged and synchronous patterns are the most basic. The synchronous pattern consists of only consistent changes. The divergent and inconsistent patterns both describe the scenario where two consistent clones diverge. We chose to investigate these individually, since a clone pair that does not experience any consistent changes before diverging may be a false positive clone. Lastly, we investigate two types of late propagation patterns. Previous studies [9, 10] have investigated late propagation. However, clones that experience late propagation and then diverge again are not described by the late propagation pattern. We define the late propagation with diversion pattern to describe these clones. Clones that experience this pattern may also be more unstable than those that experience a late propagation pattern.

4.2 Challenges

In this study, we address the following three challenges in locating faults in code clones:

- **Identifying Risky Clones.** Cloning is a common practice in software systems. It is unreasonable to inspect and test every clone in a software system for faults. The clones need to be filtered to determine the most risky clones and to better allocate testing efforts. We identify clone evolutionary patterns, changes, and clone characteristics that can be used to highlight fault-prone clone pairs.
- **Tracking Clones Requires Additional Effort.** Prediction models can be used to filter clones and select clones for additional testing. Tracking the evolutionary history of all the clones in an entire software system is resource intensive. We determine the increase in precision and recall by adding clone genealogy information to fault prediction models.
- **Prediction Metrics.** Building and executing prediction models to predict risky clones is resource intensive. Previous studies **[11, 58-65]** have identified metrics that are good

predictors of faults in software systems. However, none consider clone evolutionary metrics that can be used as predictors to identify risky clones. We examine prediction models to measure any gain in the precision and recall of the models from using clone evolutionary metrics as predictors.

4.3 Research Questions

We investigate the clone genealogies of three open source software systems. Using the cloning information from each system, we address the following research questions:

- *RQ1: Which clone evolutionary patterns and clone changes are most at risk of faults?* We examine if a specific evolutionary pattern or change is found to be more prone to faults. Clone pairs exhibiting a fault-prone pattern or experiencing a fault-prone change should be flagged for future monitoring.
- *RQ2: Does the size of a clone or the time interval between changes affect the fault-proneness of a clone pair?* We expand on the first question to determine if the size of the clone (in LOC) or the time interval between consecutive changes to a clone pair can be used to highlight fault-prone clone pairs. We suggest that these characteristics may influence the fault-proneness of clone evolutionary patterns and changes. Our results can be used to refine the identification of clone pairs at risk of faults. This helps determine where testing and review efforts should be focused.
- *RQ3: Do clone evolutionary metrics improve the prediction of future faults in software clones?* One snapshot of a software system provides limited information that can be used to predict future faults in a clone pair. However, evolutionary information about a clone pair takes more effort to collect and track. We quantify the gain in precision and recall from adding evolutionary clone pair metrics to fault prediction models.

4.4 Subject Systems

The context of this study consists of the change history of three open-source Java systems: ArgoUML, Apache Ant, and JBoss. They are selected because of their varying sizes and so that we can examine systems from a variety of domains. A summary of the characteristics of each subject system is shown in Table 4-1. The total number of genealogies extracted from each system using both the CCFinder and Simian clone detection tools is also summarized in the table.

Table 4-1: Characteristics of the Subject Systems

System	# LOC	# Revisions	# Genealogies CCFinder	# Genealogies Simian
ArgoUML	3.1M	18k	14k	111
Ant	2.3M	1.0M	30k	461
JBoss	1.6M	109k	59k	771

ArgoUML is a UML-modeling application. 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 [9]. 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), that is, the first 2576 commits.

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.

JBoss is a Java-based application server. The project was created in 1999 and is still under development as a division of Red Hat, a Linux distribution vendor and service provider. JBoss has over 110k revisions and 1.7M LOC in its software repository. We study its revision history from April 2000 to December 2010.

Table 4-1 shows that there is a large discrepancy (in orders of magnitude) between the number of clone genealogies 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 about 10 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.

4.5 RQ1 – What type of clone genealogies and clone changes are most at risk of faults?

Motivation. Developers are interested in identifying areas of a software system that have a higher likelihood of faults. Previous studies [12] have identified clones as more fault-prone than non-cloned code. However, clones occur frequently, with as much as one fifth of a software system containing duplicate code [8]. Therefore, it is unreasonable to monitor all clone pairs for

faults. However, if we can identify characteristics of fault-prone clone pairs, risky clone pairs can be highlighted for monitoring. In this research question, we examine if the evolutionary pattern exhibited by the clone pair can be used to locate fault-prone clone pairs. Additionally, we study changes to determine if some types of changes are more likely to be followed by a fault-fixing change than others. This will make developers more aware of the potential risk of performing a specific type of change in a system.

Approach. After building the set of clone genealogies for a subject system, we identify all clone evolutionary patterns within the genealogies. Thus, for each genealogy graph G, we visit each state in G and identify the clone evolutionary pattern (*i.e.*, the path P). We determine if the next state is a fault-fix. We also examine each change within the genealogy graph G, determine the type of change, and check if the next change is a fault-fixing change.

We calculate three sets of odds ratios and check the results using the Chi-Square Test. Each test is performed twice on each subject system - once for each clone detection tool. First, using the synchronous ($SYNC_p$) evolutionary pattern as the control group, we calculate the odds ratios between the control group and each of the different evolutionary patterns (the "experimental" groups). We test the following null hypothesis H_{01}: *Each type of clone evolutionary pattern has the same proportion of clone pairs that experience a fault fix in the next state*.

Second, using consistent changes (CON_c) as our control group, we calculate the odds ratios between the consistent changes and each of the different types of changes. We test the following null hypothesis H_{02}: *Each change type has the same proportion of clone pairs that experience a fault fix in the next change.*

Third, we examine evolutionary patterns and changes together, to determine the most fault-prone change when a clone pair is exhibiting a specific clone evolutionary pattern (*e.g.*, late

propagation followed by a consistent change). Using the synchronous ($SYNC_p$) evolutionary pattern followed by a diverging change (DIV_c) as the control group, we calculate the odds ratio between the control group and each of the different combinations of evolutionary patterns and changes. Each evolutionary pattern can be followed by only two of the four types of changes. For example, since a late propagation (LP_p) ends in a consistent state (C_s), it can only be followed by a consistent change (CON_c) or a diverging change (DIV_c). We test the following null hypothesis: H_{03}: *Each combination of evolutionary pattern and change type has the same proportion of clone pairs that experience a fault fix in the next change.*

Results. We now discuss the results of our three tests examining the relationship between faults and evolutionary patterns, changes, and evolutionary patterns and changes in combination. Each subsection summarizes the results for one of the three tests.

Clone Evolutionary Patterns

Table 4-2 summarizes the results of the odds ratio and Chi-square test. The first five rows of data contain the odds ratio values. We select the first row of data, the synchronous pattern ($SYNC_p$), as our control group for calculating the odds ratios. The final row gives the *p-value* obtained using the Chi-square test. In all cases using the Simian clone detection tool, the *p-value* is greater than *0.01*, failing to pass the Chi-square test. Therefore, we do not discuss the results of the Chi-Square test for Simian. Simian identifies fewer clones than CCFinder in all our subject systems. Therefore, there are fewer clone genealogies, which has an impact on the results of the Chi-Square test.

Overall, the synchronous ($SYNC_p$) evolutionary pattern is less fault-prone than the other evolutionary patterns. Therefore, based on the analysis of the three systems, we reject H_{01} in general. However, in Ant using CCFinder, the late propagation (LP_p) and late propagation with

diversion ($LPDIV_p$) patterns are less prone to faults. In this case study, we are careful to consider the late propagation evolutionary pattern (LP_p) and the late propagation with diversion ($LPDIV_p$) individually. The result in the table show that for ArgoUML and JBoss, both using CCFinder, late propagation (LP_p) genealogies are more fault-prone than late propagation with diversion ($LPDIV_p$) genealogies. Thus, although a clone pair experiences further diverging and inconsistent changes, those changes do not necessarily lead to faults in those subject systems.

Table 4-2: Contingency Tables for Clone Evolutionary Patterns

	Ant CCFinder	**Ant Simian**	**ArgoUML CCFinder**	**ArgoUML Simian**	**JBoss CCFinder**	**JBoss Simian**
$SYNC_p$	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
DIV_p	1.06	0.99	1.31	1.23	1.91	0
INC_p	1.01	1.04	1.38	2.37	2.56	0.55
$LPDIV_p$	0.90	0.97	1.43	1.19	1.70	0
LP_p	0.69	2.04	1.64	1.46	4.39	0
p-value	**<0.01**	0.29	**<0.01**	0.08	**<0.01**	0.43

Changes

The results of the odds ratio and Chi-square test are summarized in Table 4-3. In the third column of Table 4-3 the *p-value* is greater than *0.01*. Therefore, Ant using Simian fails the Chi-square test and is removed from our discussion. Like for the evolutionary patterns, Ant using Simian has very few data points, which causes it to fail the Chi-Square test.

We select the first row of data, consistent changes (CON_c), as our control group for calculating the odds ratios. Overall, most of the changes are either more or less fault-prone than consistent changes (CON_c), so for the three subject systems, we reject H_{02} in general. The results do not generalize across systems or clone detection tools. Therefore, they are system dependent.

Table 4-3: Contingency Tables for Clone Pair Changes

	Ant CCFinder	Ant Simian	ArgoUML CCFinder	ArgoUML Simian	JBoss CCFinder	JBoss Simian
CON_c	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
DIV_c	1.06	1.06	1.63	1.86	2.18	0.28
INC_c	1.02	1.04	1.25	2.55	2.68	0.81
$RESYNC_c$	0.74	2.15	1.72	1.46	5.43	0
p-value	**<0.01**	0.13	**<0.01**	**<0.01**	**<0.01**	**<0.01**

Evolutionary Patterns and Changes

The results of the odds ratio and Chi-square test are summarized in Table 4-4. The first row of data in the table contains the control group, a synchronous evolutionary pattern ($SYNC_p$) followed by a diverging change (DIV_c). For all systems using Simian, the *p-value* is greater than *0.01*. Therefore, Simian fails the Chi-square test and is removed from our discussion. All three systems using Simian have very few data points, which cause them to fail the Chi-Square test.

In general, the odds ratio of each combination of evolutionary pattern and change was greater or smaller than the synchronous pattern with a diverging change. Thus, in an examination of the three subject systems, we reject H_{03}.

In all cases except Ant using CCFinder, a divergent evolutionary pattern (DIV_p) followed by a re-synchronizing change ($RESYNC_c$) has a much higher odds ratio than the control combination. This indicates that re-synchronizing changes on clone pairs are more highly correlated to faults than a synchronous evolutionary pattern ($SYNC_p$) followed by a diverging change (DIV_c). Therefore, developers should be more careful when they re-synchronize a clone pair since the risk of introducing a fault is higher.

Overall, the results are system-dependent, so we cannot conclude that specific combinations of evolutionary patterns and changes should be monitored for faults.

Table 4-4: Contingency Tables for Evolutionary Patterns and Changes

Evolutionary Pattern	Change	Ant CCFinder	Ant Simian	ArgoUML CCFinder	ArgoUML Simian	JBoss CCFinder	JBoss Simian
$SYNC_p$	**DIV_c**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
$SYNC_p$	CON_c	0.98	0.00	0.65	-	0.38	0.00
DIV_p	INC_c	0.91	0.51	0.89	4.24	0.90	0.00
DIV_p	$RESYNC_c$	0.82	5.47	2.88	Inf	2.80	-
INC_p	INC_c	0.89	0.74	0.88	6.87	1.47	-
INC_p	$RESYNC_c$	0.64	1.31	0.93	1.93	2.82	0.00
LP_p	DIV_c	0.77	0.91	1.42	0.00	0.67	0.00
LP_p	CON_c	0.46	0.00	0.84	-	0.30	-
$LPDIV_p$	INC_c	0.83	0.61	0.87	3.94	1.08	-
$LPDIV_p$	$RESYNC_c$	0.37	-	0.32	-	0.00	-
	p-value	**<0.01**	0.08	**<0.01**	0.03	**<0.01**	0.30

4.6 RQ2 – Does the size of a clone or the time interval between changes affect the fault-proneness of a clone pair?

Motivation. 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. Using our set of clone pair genealogies, we examine the effect of the time interval between changes on faults.

It is expected that a smaller clone will be less prone to faults, as it is less complex and may require less effort to modify. An evolutionary history of a clone pair tracks the types and frequency of changes to clone pairs. By examining the evolutionary history of clone pairs in conjunction with the size of the clone, we can determine if the fault-proneness is affected by the size of a clone.

Approach. In this question, we classify each change by the time interval since the last change. We divide the changes into five time periods: one day, one week, one month, one year, and more than one year. A change is flagged if it is a fault fix. Using "One Day" as the control group, we calculate the odds ratios between the control group and each of the other time periods and perform the Chi-square test. We test the following null hypothesis H_{04}: *The time interval between modifications to a clone pair has no relationship with fault fixes*.

When examining the effect of clone size on faults, we examine each state from each genealogy graph G. For each state, we identify the evolutionary pattern of the clone pair up to and including that state and measure the number of lines of cloned code in a clone pair. The size of the clone is then labeled as either "big" if it is greater than 10 lines of code or "small" if it is smaller than 10 lines of code. The cut-off of 10 lines of code was determined by examining the distribution of clone sizes. For each state, we determine if the next state is a fault fix. We calculated the odds ratios and the Chi-square test, and test the following null hypothesis H_{05}: *The size of the clone has no relationship with fault fixes*. When calculating the odds ratio, we select the synchronous evolutionary pattern with a small clone size as our control group.

Results. In this subsection we summarize our results when investigating the relationship between the time interval between changes or the size of the clone and faults.

Time Interval Between Changes

Table 4-5 summarizes the results of the odds ratio tests. The first row of data in the table contains the control group, changes that occur within one day. All three subject systems pass the Chi-square test using both clone detection tools. The odds ratios for each time period are different, thus when examining our three subject systems, we reject H_{04}. In all cases of a change occurring within a month, the change is always less prone to faults than changes within a day. In half the cases, changes that occur within one year are more prone to faults than those that occur

within the first day. However, if a clone pair is not modified for more than a year, the change is almost equally prone to faults as changes within the first day, except for the JBoss subject system. It is surprising that changes within one day are among the most fault-prone. It is expected that a longer time interval would make developers less aware of clones, leading to faults [32]. More investigations are needed to determine the reason for this result.

Size of the Clone

The odds ratios of the evolutionary patterns classified by the size of the clone are summarized in Table 4-6. We select small instances of the synchronous evolutionary pattern ($SYNC_p$) as our control group. The odds ratios for all the significant cases from Table 4-6 are plotted as a bar chart in Figure 4-3. A solid black line in the figure represents the control group. For all five significant cases, the small instances of late propagation (LP_p) have a very high odds ratio. In order to show the smaller odds ratios in the plot, we set a limit of 9 for the odds ratios in the plot. The values for the small late propagation instances (LP_p) are found in Table 4-6.

Table 4-5: Contingency Tables for Evolutionary Patterns Considering the Time Interval Between Changes

	Ant CCFinder	**Ant Simian**	**ArgoUML CCFinder**	**ArgoUML Simian**	**JBoss CCFinder**	**JBoss Simian**
One Day	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
One Week	0.93	0.81	0.21	1.64	1.83	0.16
One Month	0.81	0.70	0.38	0.71	0.41	0.17
One Year	1.20	1.89	0.55	1.25	0.18	0.24
More than a Year	0.96	1.52	0.91	0.91	0.06	0
p-value	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**

Table 4-6: Contingency Tables for Evolutionary Patterns Considering the Cloned Code Size

	Ant CCFinder	Ant Simian	ArgoUML CCFinder	ArgoUML Simian	JBoss CCFinder	JBoss Simian
$SYNC_p$ Small	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
DIV_p Small	0.99	0.60	1.61	2.80	4.11	-
INC_p Small	0.94	0.68	1.65	5.50	2.76	-
$LPDIV_p$ Small	0.62	0.00	2.81	0.00	2.09	-
LP_p Small	34.61	24.58	36.70	84.00	22.09	-
$SYNC_p$ Big	0.84	0.92	2.13	8.40	1.47	-
DIV_p Big	0.92	0.99	1.94	4.45	2.34	-
INC_p Big	0.88	1.02	2.08	7.53	3.53	-
$LPDIV_p$ Big	0.82	0.99	1.76	3.65	2.23	-
LP_p Big	0.51	1.84	2.86	0.00	3.59	-
p-value	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	NA

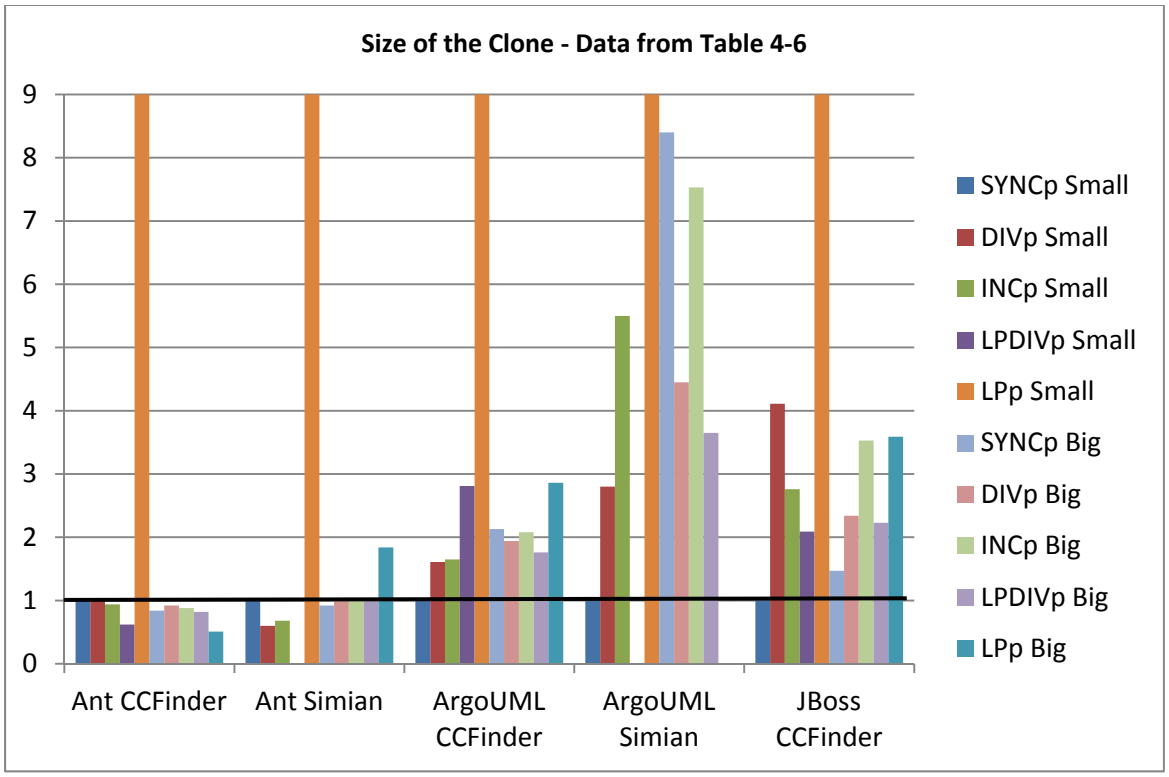

Figure 4-3: Odds Ratios for the Size of the Clone

In most cases, the odds ratio of both the big and small instances of the same evolutionary pattern are consistently greater or less than 1, so they have an odds ratio that is consistently greater than or less than the odds ratio of the small synchronous evolutionary pattern ($SYNC_p$). The magnitude of the odds ratio can vary depending on the evolutionary pattern, there is a relationship between the size of clones and fault fixes. The case of late propagation in small clones is striking. In all cases the odds ratio is at least an order of magnitude greater than for late propagation in big clones, and even as high as 84 for ArgoUML using Simian. It would be expected that a smaller clone would be easier to modify, leading to fewer faults. However, our results contradict this expectation. As with the time interval between changes results, we suggest that further investigations be done to determine the reason for a large number of faults in late propagation (LP_p) evolutionary patterns within smaller clones. Thus, for the three subject systems, we reject H_{05}.

4.7 RQ3 – Do clone evolutionary metrics improve the prediction of future faults in software clones?

Motivation. Tracking the evolutionary history of all clone pairs in an entire system is resource intensive. However, the history can provide more information than is available in a single system snapshot. Knowing the gain in precision and recall achieved by adding evolutionary metrics to fault prediction models helps a developer decide if the added effort justifies the results.

The metrics in Table 4-7 are divided into two categories: system snapshot and evolutionary metrics. The metrics in the first category, system snapshot, describe metrics that can be collected using the snapshot of the system that contains the clone pair. For example, $CPathDepth$ describes the number of folders that the clones in a clone pair have in common within the system directory structure. The system snapshot metrics also include information about

the methods containing each of the clones in the clone pair. For example, the $MCyclo$ measures the cyclomatic complexities of the methods containing the two clones in a clone pair. The metrics in the last category describe the evolutionary metrics. There are two sets of metrics within this category: states changes and clone changes. This is because the clone pair can experience multiple changes without changing states.

Table 4-7: Clone Pair Metrics

Metrics	Description
System Snapshot Metrics	
CLOC	The number of cloned lines of code
CFltFix	The current revision was a fault fix (true or false)
CPathDepth	The number of common folders within the project directory structure
CCurSt	The current state of the clone pair (consistent or inconsistent)
Csibs	The number of clone siblings of the clone pair
CFltSibsNorm	The number of buggy clone siblings of the clone pair divided by the total number of siblings
MLOC	The number of lines of code of the method containing the clone
Mnest	The maximum nesting depth of the method containing the clone
Mcyclo	The cyclomatic complexity of the method containing the clone
Msibs	The number of method clone siblings of the method containing the clone
MFltSibsNorm	The number buggy clone siblings of the method containing the clone divided by the total number of clone siblings
Evolutionary Metrics	
EEvPattern	One of $SYNC_p$, DIV_p, INC_p, LP_p, or $LPDIV_p$
EConChg	The number of consistent changes experienced by the clone pair
EIncChg	The number of inconsistent changes experienced by the clone pair
EFltDens	The number of fault fix modifications to the clone pair since it was created divided by the total number of revisions that modified the clone pair
EConStChg	The number of consistent changes of state within the clone pair genealogy
EIncStChg	The number of inconsistent changes of state within the clone pair genealogy
EFltsConStChg	The number of re-synchronizing changes (*i.e.*, $RESYNC_c$) that were a fault fix
EFltIncStChg	The number of diverging changes (*i.e.*, DIV_c) that were a fault fix
EChgTimeInt	The time interval since the previous change to the clone pair. One of “One Day”, “One Week”, “One Month”, “One Year”, or “More than One Year”

Approach. In this question we use the Random Forest models described in Section 2.6.5 to predict future fault fixes. We build two sets of models using two different sets of metrics: snapshot metrics, and snapshot metrics in conjunction with evolutionary metrics.

For the second set of models, we extend each model to include the evolutionary metrics described in Table 4-7. The evolutionary metrics are added to the 8 predictors selected for each snapshot model. We compare the two sets of models to determine if the evolutionary metrics improve the prediction of faults in clone pairs.

Results. In this subsection we describe the results for RQ3. First, we examine the precision, recall, and F-Measure for snapshot and evolutionary models. Next, we discuss the most important predictors of faults in clone pair.

Precision, Recall, and F-Measure

The precision, recall, and F-Measure for both sets of models are summarized in Table 4-8. For each of the three *IR* metrics, we list the value for the model containing only snapshot metrics, the value for the model that includes the evolutionary metrics, and the percentage difference between the two models. The data in Table 4-8 is plotted in Figure 4-4. Each bar in Figure 4-4 represents the percentage difference of the precision, recall, and F-Measure between the two models.

Overall, there is an increase in the precision, recall, and F-Measure across all three subject systems using both clone detection tools. Ant and JBoss had the greatest increase, with a minimum 10% increase across all measures. ArgoUML have the most modest increase, with the F-Measures for both CCFinder and Simian being less than 10%. We conclude that adding evolutionary metrics increases the precision, recall, and F-Measure of our models for all three subject systems, but the size of the gain is system-dependent.

Importance of Predictors

Table 4-9 summarizes the top predictors based on their importance for both the snapshot and evolutionary models. The first column lists the subject systems and clone detection tools. The second column indicates if the model is based on snapshot metrics or evolutionary metrics. The third column lists the most important predictors in the order of their importance. Therefore, the number of clone siblings of the methods containing the clones ($MSibs$) is one of the most important predictors.

Metrics related to the size of the methods and clones, and the cyclomatic complexity of the method (*MCyclo*) containing the clone, the size of the method containing the clone (*MLOC*), and the size of the clone (*CLOC*) are also some of the most important predictors for models based on data from our three subject systems. This is similar to the finding of the previous research question where we show that there is a relationship between size of the clone and fault-fixes.

Table 4-8: Comparison of Precision, Recall, and F-Measure between Snapshot Models and Evolutionary Metrics

	Precision			**Recall**			**F-Measure**		
	Snapshot	**Evolutionary**	**Difference (%)**	**Snapshot**	**Evolutionary**	**Difference (%)**	**Snapshot**	**Evolutionary**	**Difference (%)**
Ant–CCFinder	0.72	0.80	10.79	0.59	0.74	26.41	0.65	0.77	18.83
Ant–Simian	0.61	0.64	4.55	0.40	0.58	46.95	0.48	0.61	26.88
ArgoUML–CCFinder	0.80	0.81	1.25	0.65	0.76	16.97	0.72	0.79	9.31
ArgoUML–Simian	0.72	0.76	5.28	0.60	0.65	7.64	0.66	0.70	6.55
JBoss–CCFinder	0.73	0.83	13.00	0.57	0.77	34.49	0.64	0.80	24.11
JBoss–Simian	0.60	0.72	19.67	0.60	0.73	21.50	0.60	0.72	20.50

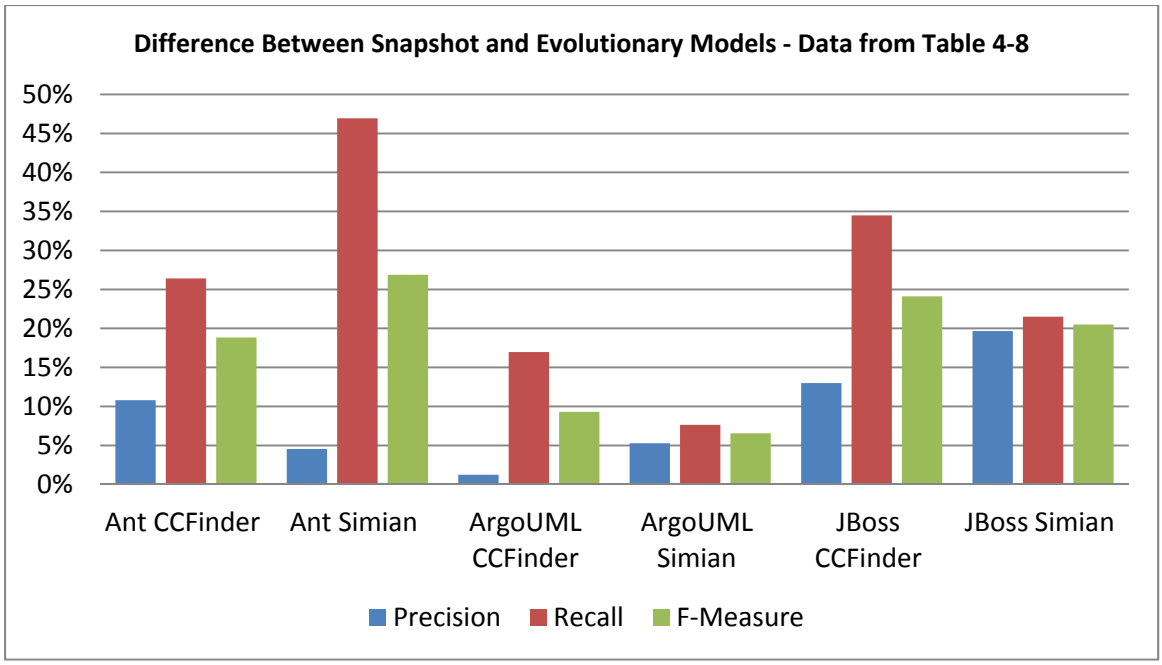

Figure 4-4: Difference Between Snapshot and Evolutionary Models

Table 4-9: Important Predictors

System	Snapshot or Evolutionary Metrics	Metrics
Ant CCFinder	Snapshot	MSibs, MCyclo, CLOC, MLOC, MFltSibsNorm
	Evolutionary	EChgTimeInt, MSibs, MCyclo, CLOC, MLOC, MFltSibsNorm
Ant Simian	Snapshot	MCyclo, MSibs, CLOC
	Evolutionary	MSibs, Mcyclo, CLOC, Csibs, EchgTimeInt
ArgoUML CCFinder	Snapshot	MLOC, MFltSibsNorm, CLOC, Mnest
	Evolutionary	EChgTimeInt, MLOC, Mnest, CLOC
ArgoUML Simian	Snapshot	CFltFix, MLOC
	Evolutionary	CFltFix, MLOC
JBoss CCFinder	Snapshot	MLOC, MCyclo, Msibs
	Evolutionary	MCyclo, MLOC, MSibs, EChgTimeInt, CPathDepth
JBoss Simian	Snapshot	MSibs, MLOC
	Evolutionary	MSibs, MCyclo, MLOC

The most important evolutionary metric in the three evolutionary models is the time interval since the last change (*EChgTimeInt*). This relates to the previous research question, where we show that there is a relationship between the time interval between changes and the fault-proneness of a change.

4.8 Summary

In this case study, we examine the states within clone genealogies and changes to clone pairs at a finer-grained level to determine their relation to faults in software systems. We formally define six different clone evolutionary patterns and four types of changes experienced by a clone pair. Using these definitions, we show that each clone evolutionary pattern has a different level of fault-proneness. We then extend this result and determine that the size of the clone has an impact on the fault-proneness of the clone evolutionary patterns, and the time interval between changes to a clone pair has an impact on the fault-proneness of changes. Next, we build Random Forest prediction models using the code clone metrics that can be collected from one snapshot of a software system. We then extend the models, adding metrics collected from the genealogy of the clone pair. We suggest that genealogy information can improve the prediction of faults in clones. In this case study, we show that the addition of the evolutionary metrics increases the F-Measure of the models by as much as 26%. Overall, we conclude that the use of historical clone pair metrics may justify the added effort needed to track and collect the added evolutionary metrics. For both sets of models, we also identify the most important predictors. The size of the method and the clones, as well as the time between changes was among the most important predictors, which is a similar result to the first part of our case study. All of these findings are based on an analysis of three subject systems. Further systems should be studied to determined the generality of our results.

Chapter 5

An Empirical Study of Late Propagation in Software Clones

In this chapter, we present the second case study in this thesis. We extend the late propagation evolutionary pattern definition by classifying each instance of late propagation into one of eight different types of late propagation. We examine each of the eight types and their relationship with faults in three subject systems.

5.1 Classification of Late Propagation Genealogies

Definition. In the current state of the art, the late propagation evolutionary pattern is defined as a clone pair that experiences a diverging change that is later followed by a re-synchronizing change [9]. While the clone pair is in an inconsistent state, it may experience further inconsistent changes. 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.

Each instance of late propagation can be broken down into three distinct 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.

Late Propagation Types. Using all combinations of the three phases, we identify eight possible types of late propagation evolutionary patterns. Since they are not extracted from existing systems, the types may not appear in all the systems in this case study. The characteristics of the individual types are described in Table 5-1. It describes the possible sets of modifications to two clones, Clone A and Clone B, that make up a clone pair.

Table 5-1: 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 and B	B
	LP3	<A,B>	A	A	A and B
Propagation May or May Not Occur	LP4	<A,B>	A	A and B	A
	LP5	<A,B>	A	A and B	A and B
	LP6	<A,B>	A and B	A and B	A or B
	LP7	<A,B>	A and B	A and B	A and B
Propagation Never Occurs	LP8	<A,B>	A	A	A

We divide the types of late propagation into three categories:

- A propagation always occurs
- A propagation may or may not occur
- A propagation never occurs

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 5-1, 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. Table 5-2 is an example of a genealogy experiencing an LP1 evolutionary pattern taken from ArgoUML using CCFinder as the clone detection tool. The code that is modified in each revision is bolded.

Table 5-2: An Example of an LP1 Genealogy in ArgoUML

Revision Number	Clone A	Clone B
595	`addField(new UMLComboBox(typeModel),1,0,0);`	`addField(new UMLComboBox(classifierModel),2,0,0);`
602	`addField(new` **`UMLComboBoxNavigator(this,"NavClass", new UMLComboBox(typeModel))`**`,1,0,0);`	`addField(new UMLComboBox(classifierModel),2,0,0);`
604	`addField(new UMLComboBoxNavigator(this,"NavClass", new UMLComboBox(typeModel)),1,0,0);`	`addField(new` **`UMLComboBoxNavigator(this,"NavClass", new UMLComboBox(classifierModel))`**`,2,0,0);`

As shown in this genealogy, 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 late propagation type in the third category, as shown in Table 5-1. In LP8, Clone A is modified, diverging the clone pair. The change is later reverted, re-synchronizing the clone pair.

Table 5-3 is an example of a genealogy experiencing an LP8 pattern taken from Ant using CCFinder as the clone detection tool.

Table 5-3: An Example of an LP8 Genealogy in Ant

Revision Number	Clone A	Clone B

270250	`if( destFile == null )`
	`{`
	`  destFile = new File(`
	`  destDir, file.getName()`
	`  );`
	`}`
		`if (destFile == null) {`
		`  destFile = new`
		`  File(destDir,`
		`  file.getName());`
		`}`

270264	`if( m_destFile == null )`
	`{`
	`  m_destFile = new File(`
	`  m_destDir,`
	`  m_file.getName() );`
	`}`
		`if (destFile == null) {`
		`  destFile = new`
		`  File(destDir,`
		`  file.getName());`
		`}`

271109	`if (destFile == null) {`
	`  destFile = new`
	`  File(destDir,file.getName`
	`  ());`
	`}`
		`if (destFile == null) {`
		`  destFile = new`
		`  File(destDir,`
		`  file.getName());`
		`}`

As shown in this genealogy, 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 genealogy examples discussed in this section contain only the interesting lines of code extracted from bigger clones.

5.2 Research Questions

In this case study, we study the characteristics of genealogies experiencing late propagation and estimate the likelihood of faults. We address the following three research questions:

- *RQ1: Are there different types of late propagation?* We perform an exploratory study to examine several late propagation evolutionary 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?* Late propagation genealogies have been identified by previous researchers [10] as fault-prone. We classify each genealogy exhibiting the late propagation pattern into the eight types described in Section 5.1, and determine if the each type is consistently fault-prone.
- *RQ3: Which type of late propagation experiences the highest proportion of faults?* In this question, we examine if the quantity of faults experienced is different for each type of late propagation.

5.3 Subject Systems

Table 5-4: Frequency of Late Propagation

	CCFinder			Simian		
System	**# Gen**	**# LP Gen**	**% LP Gen**	**# Gen**	**# LP Gen**	**% LP Gen**
ArgoUML	14k	2k	14 %	111	23	21 %
Ant	30k	6k	20 %	461	103	22 %
JBoss	59k	2k	3 %	771	12	2 %

We examine the same three subject systems as in the study in Chapter 4. Both studies use the same clone genealogies extracted from each subject system. In this study, we are interested in genealogies that exhibit a late propagation clone evolutionary pattern. The frequency of late propagation in each of our three subject systems is summarized in Table 5-4.

5.4 RQ1 – Are there different types of Late Propagation?

Motivation. Late propagation has been defined by several researchers [10, 18] as an inconsistent change followed by a re-synchronizing change. This question is preliminary to questions RQ2 and RQ3. It provides the quantitative data about the how often the different types of late propagation occur in our studied systems.

Approach. We address this question by classifying all instances of late propagation using the three characteristics described in Section 5.1. For each type of late propagation, we report the number of occurrences in the systems. For each system we examine the instances of late propagation using both clone detection tools.

Results. Table 5-5 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. Each column (*e.g.*, ArgoUML - Simian) in Table 5-5 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 5-5, four types of late propagation are dominant across all three 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 5-5, the instances of LP2 and LP3 are low. Therefore, the ‘propagation always occurs’ category (*i.e.*, LP1, LP2, and LP3) accounts for a minority of instances of an inconsistent change followed by a re-synchronization. For all cases, the ‘propagation never occurs’ category (*i.e.*, LP8) contributes more instances of late propagation than the ‘propagation always occurs’ category. As shown in Table 5-5, LP7 occurs in an average of 45% 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 genealogies experiencing late propagation.

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.

5.5 RQ2 – Are some types of Late Propagation more fault-prone than others?

Motivation. Previous researchers have determined that late propagation is more prone to faults than other clone genealogy evolutionary patterns [10]. Using the classification of late propagation clone genealogies described in Section 5.1, we evaluate late propagation in greater depth, and examine if the risk of faults remains consistent across all types of late propagation.

Approach. We compute the number of fault-containing and fault-free genealogies in each late propagation category. We compute the same values for clone genealogies that experience at least one change and do not experience a late propagation genealogy pattern (*i.e.*, non-late propagation genealogies). We test the following null hypothesis: H_{01}: *Each type of late propagation genealogy has the same proportion of clone pairs that experience a fault fix.*

We use the Chi-square test [67] and compute the odds ratio (OR) [67] as described in Section 2.6. Specifically, we compute two sets of odds ratios. As described in Section 2.6, to compute an odds ratio we must select an experimental and a control group. First, we select the clone pairs that underwent a late propagation as experimental group. Second, we form one experimental group for each type of late propagation and re-compute the odds ratios. In both cases, we select the non-LP genealogies as the control group.

Table 5-5: Number of Clone Pairs that Underwent a Late Propagation

		Ant				ArgoUML				JBoss			
Propagation Category	**LP Type**	**CCFinder**		**Simian**		**CCFinder**		**Simian**		**CCFinder**		**Simian**	
		number	%	number	%	number	%	number	%	number	%	number	%
Propagation Always Occurs	LP1	521	8.07	21	20.39	57	2.65	0	0.00	45	2.28	0	0
	LP2	46	0.71	0	0	13	0.60	0	0.00	1	0.05	0	0
	LP3	121	1.87	0	0	226	10.50	1	4.35	55	2.79	0	0
Propagation May or May Not Occur	LP4	33	0.51	0	0	18	0.84	2	8.70	4	0.20	0	0
	LP5	70	1.08	0	0	73	3.39	0	0	44	2.23	1	8.33
	LP6	166	2.57	1	0.97	129	5.99	5	21.74	70	3.55	0	0.00
	LP7	2482	38.45	29	28.16	1242	57.71	6	26.09	995	50.46	8	66.67
Propagation Never Occurs	LP8	3016	46.72	52	50.49	394	18.31	9	39.13	758	38.44	3	25.00
	TOTAL	**6455**	**100.00**	**103**	**100.00**	**2152**	**100.00**	**23**	**100.00**	**1972**	**100.00**	**12**	**100.00**

Results. Table 5-6 summarizes the results of the Chi-square test and the odds ratios for instances of late propagation compared to non-late propagation genealogies (non-LP).

Fault-proneness of Late Propagation

The first and second columns in Table 5-6 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 cases where ArgoUML and JBoss are analyzed using Simian, 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 [67]. In this case 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 for all three subject systems, genealogies experiencing late propagation are more fault-prone than non-LP genealogies. However, for ArgoUML and JBoss using Simian, the results of the Chi Square test are 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 [9] that found that late propagation is more at risk of faults.

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 5-8 and Table 5-9 summarize the results obtained from CCFinder and Simian, respectively. 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 genealogies with no late propagation as the control group.

Table 5-6: Contingency Table and Chi Square Test Results for Clone Genealogies With and Without Late Propagation

System	No LP Faults	No LP No Faults	LP Faults	LP No Faults	p-values	Odds Ratio
CCFinder						
Ant	10064	18706	3437	3018	**<0.01**	2.12
ArgoUML	8755	9869	1398	754	**<0.01**	2.09
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

The Chi-Square test results for ArgoUML and JBoss using Simian in Table 5-9 are greater than 0.01, so they are insignificant. Therefore, they are excluded from consideration.

Figure 5-1 plots the data from Table 5-8 and Table 5-9 as a bar chart. The plot only includes the significant cases from Table 5-8 and Table 5-9. A solid black line represents the control group, the non-LP genealogies. In order to describe the details in the plot, we set an upper limit of 10 for the odds ratios. The values for odds ratios greater than 10 can be found in Table 5-8 and Table 5-9.

An examination of the significant cases in Table 5-8 and Table 5-9 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 four exceptions to this observation:

- LP6 in Ant using Simian (Table 5-9)
- LP5 and LP7 in Ant using CCFinder (Table 5-8)
- LP7 in JBoss using CCFinder (Table 5-8)

The exceptions are shown in Figure 5-1 as bars that are below the black solid line. Thus, in an examination of our three subject systems, the late propagation types are not more fault-prone than non-LP genealogies. All three exceptions belong to the 'propagation may or may not occur' category.

Comparing Table 5-8 and Table 5-9 to Table 5-5, 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, LP2 has a high odds ratio (*e.g.*, 26.64 in Ant in Table 5-8), but accounts for less than 1% of all late propagation instances in Table 5-5.

The most common late propagation type in the previous research question, LP7, in general has low odds ratios in Table 5-8 and Table 5-9. This indicates that although it occurs frequently, it is less fault-prone than other less common late propagation types (*e.g.*, LP2).

Overall, each type of late propagation has a different level of fault-proneness. Thus, in this case study, we reject H_{01} in general.

5.6 RQ3 – Which types of late propagation experience the highest proportion of faults?

Motivation. 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 contributes the most faults in each system. In other words, we examine if, when faults occur, do they occur in large numbers?

Approach. 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 each type of late propagation is identical. Therefore, we test the following null hypothesis H_{02}: *Different types of late propagation have the same proportion of clone pairs that experience a fault fix*.

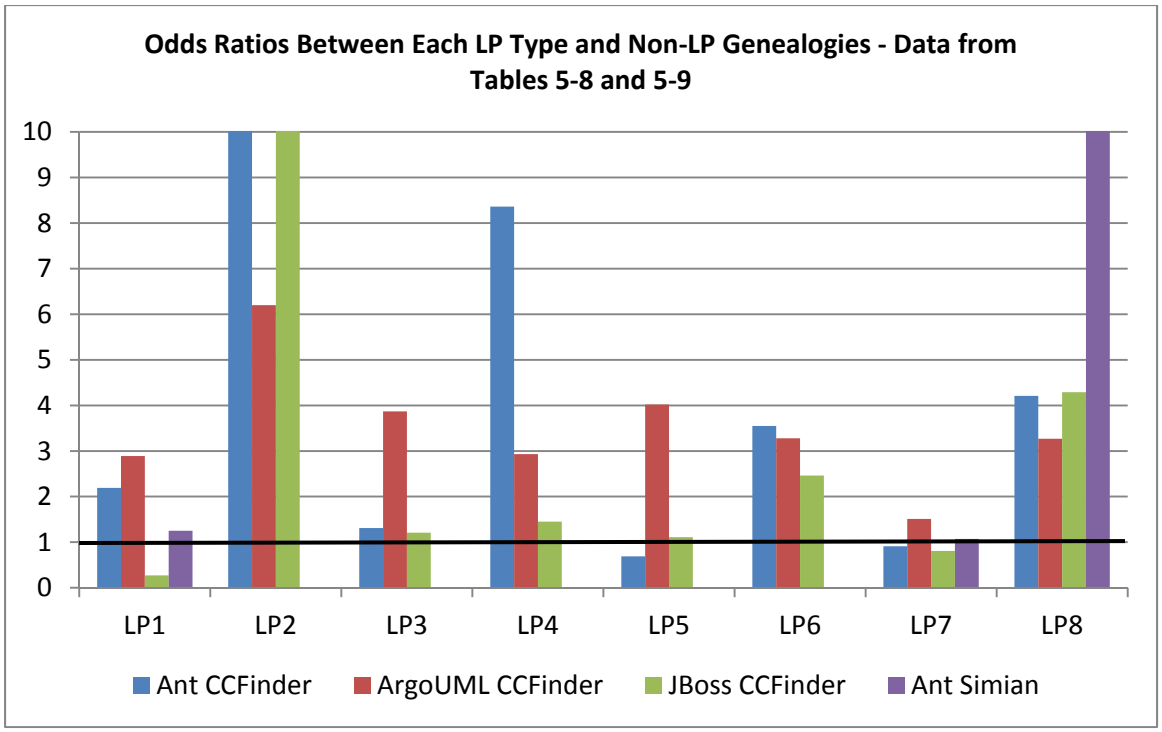

Figure 5-1: Odds Ratios Between Each LP Type and Non-LP Genealogies

Results. Table 5-10 presents the distribution of faults for different types of late propagation. The 'Total' row represents the total numbers of faults over all genealogies that experience late propagation. For example, for Ant using CCFinder, there are 5566 fault fixes across all genealogies, as shown in the final row in Table 5-10. 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.

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 5-7 summarizes the results of the Kruskal Wallis test. We observe a statistically significant difference between the distributions of faults across all the groups of late propagation types across

all three subject systems. From Table 5-7, only ArgoUML and JBoss using Simian are not statistically significant.

Figure 5-2 plots the data from Table 5-10 as a bar chart. Only the significant cases are included in the plot.

Examining the results in Table 5-10 for the significant cases, we see that LP7 and LP8 contribute to a large proportion of the faults. In the previous question, LP8 generally had a high odds ratio. In Ant, LP8 contributes over half of all fault fixes. 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. In Table 5-8 and Table 5-9, LP7 had a low odds ratio. In some cases it was less prone to faults than genealogies without late propagation. However, it contributes a large proportion of the faults. In the case of JBoss using Simian, LP7 instances are responsible for over 50% of all faults. Therefore, when faults occur, they are likely to occur in large numbers.

The remaining results are system-dependent. For example, in the case of ArgoUML using CCFinder in Table 5-10, the category where propagation always occurs contributes almost the same amount of faults than the category where propagation never occurs. This trend does not hold across all systems in this case study.

Overall, we can conclude that across our three subject systems, types LP7 and LP8 are the most dangerous, with the other types being system-dependent in their fault-proneness. Each type of late propagation is responsible for a different proportion of faults. Therefore, we reject H_{02}.

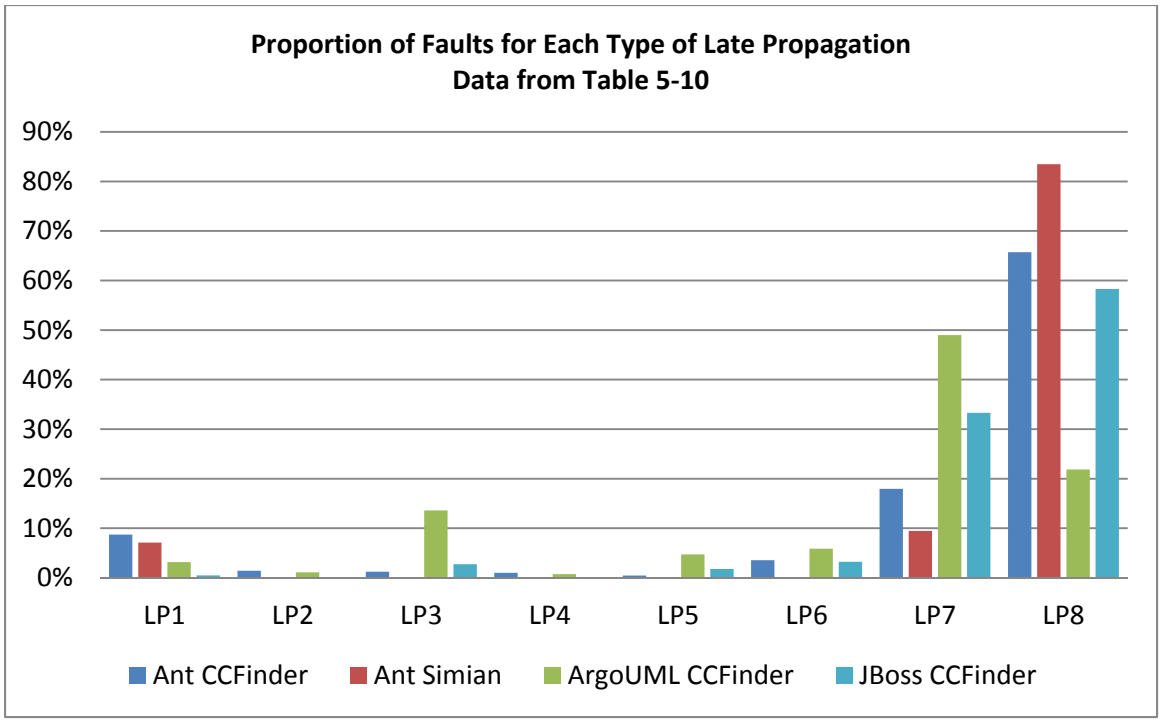

Figure 5-2: Proportion of Faults for Each Type of Late Propagation

Table 5-7: Results of the Kruskal Wallis Tests

System	Kruskal Wallis p-values
Ant – CCFinder	< 0.01
Ant – Simian	< 0.01
ArgoUML – CCFinder	< 0.01
ArgoUML – Simian	0.25
JBoss – CCFinder	< 0.01
JBoss – Simian	0.47

Table 5-8: CCFinder – Contingency Tables with the Chi-Square Test for Different Late Propagation Types

		Ant				ArgoUML				JBoss			
Propagation Category	**Type**	**Faults**	**No Faults**	**Odds Ratio**	**p-value**	**Faults**	**No Faults**	**Odds Ratio**	**p-value**	**Faults**	**No Faults**	**Odds Ratio**	**p-value**
	No LP	10064	18706	1	**< 0.01**	8755	9869	1	**< 0.01**	5684	8268	1	**< 0.01**
Propagation Always Occurs	LP1	282	239	2.19	**< 0.01**	41	16	2.89	**< 0.01**	7	38	0.27	**< 0.01**
	LP2	43	3	26.64	**< 0.01**	11	2	6.20	**< 0.01**	1	0	Inf	**< 0.01**
	LP3	50	71	1.31	**< 0.01**	175	51	3.87	**< 0.01**	25	30	1.21	**< 0.01**
Propagation May or May Not Occur	LP4	27	6	8.36	**< 0.01**	13	5	2.93	**< 0.01**	2	2	1.45	**< 0.01**
	LP5	19	51	0.69	**< 0.01**	57	16	4.02	**< 0.01**	19	25	1.11	**< 0.01**
	LP6	109	57	3.55	**< 0.01**	96	33	3.28	**< 0.01**	44	26	2.46	**< 0.01**
	LP7	814	1668	0.91	**< 0.01**	712	530	1.51	**< 0.01**	356	639	0.81	**< 0.01**
Propagation Never Occurs	LP8	2093	923	4.21	**< 0.01**	293	101	3.27	**< 0.01**	566	192	4.29	**< 0.01**

Table 5-9: Simian - Contingency Tables with the Chi Square Test for Different Late Propagation Types

		Ant				ArgoUML				JBoss			
Propagation Category	**Type**	**Faults**	**No Faults**	**Odds Ratio**	**p-value**	**Faults**	**No Faults**	**Odds Ratio**	**p-value**	**Faults**	**No Faults**	**Odds Ratio**	**p-value**
	No LP	124	251	1	**< 0.01**	44	45	1	0.65	10064	18706	1	0.52
Propagation Always Occurs	LP1	8	13	1.25	**< 0.01**	0	0	NA	NA	0	0	NA	NA
	LP2	0	0	NA	**< 0.01**	0	0	NA	NA	0	0	NA	NA
	LP3	0	0	NA	**< 0.01**	0	1	0	0.65	0	0	NA	NA
Propagation May or May Not Occur	LP4	0	0	NA	**< 0.01**	2	0	Inf	0.65	0	0	NA	NA
	LP5	0	0	NA	**< 0.01**	0	0	NA	NA	5	8	0	0.52
	LP6	0	1	0	**< 0.01**	2	3	0.68	0.65	0	0	NA	NA
	LP7	10	19	1.07	**< 0.01**	3	3	1.02	0.65	2	6	0.53	0.52
Propagation Never Occurs	LP8	50	2	50.60	**< 0.01**	5	4	1.28	0.65	2	1	3.20	0.52

Table 5-10: Proportion of Faults for Each Type of Late Propagation

		Ant				ArgoUML				JBoss			
Propagation Category	**LP Type**	**CCFinder**		**Simian**		**CCFinder**		**Simian**		**CCFinder**		**Simian**	
		#	**%**	**#**	**%**	**#**	**%**	**#**	**%**	**#**	**%**	**#**	**%**
Propagation Always Occurs	LP1	484	8.70	9	7.09	72	3.15	0	0.00	7	0.48	0	0
	LP2	79	1.42	0	0	25	1.09	0	0.00	1	0.07	0	0
	LP3	68	1.22	0	0	311	13.59	0	0.00	40	2.74	0	0
Propagation May or May Not Occur	LP4	56	1.01	0	0	17	0.74	4	22.22	2	0.14	0	0
	LP5	25	0.45	0	0	108	4.72	0	0.00	26	1.78	0	0
	LP6	197	3.54	0	0	134	5.86	2	11.11	47	3.22	0	0
	LP7	999	17.95	12	9.45	1121	48.99	6	33.33	486	33.29	4	57.14
Propagation Never Occurs	LP8	3658	65.72	106	83.46	500	21.85	6	33.33	851	58.29	3	42.86
TOTAL		**5566**	**100.00**	**127**	**100.00**	**2288**	**100.00**	**18**	**100.00**	**1460**	**100.00**	**7**	**100.00**

5.7 Summary

In this case study, we extend previous studies to examine late propagation in more detail. We first confirm the conclusion from previous studies that genealogies that experience late propagation are more risky than other clone genealogies. We then identify eight types of late propagation and study them in detail to identify which contribute the most to faults in late propagation. Overall, in a study of three software systems, we find that one type of late propagation (LP8) is riskier than the others, in terms of its fault-proneness and the magnitude of its 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. Another type of late propagation (LP7), is generally less fault-prone than other late propagation types, but when faults occur, they occur in large numbers. 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 genealogies exhibiting a late propagation evolutionary pattern require monitoring.

Chapter 6

Conclusion

Cloning has been identified by previous researchers as a risky practice in software development and maintenance. However, software projects have limited resources for reviewing and testing code. Identifying the clones most at risk of faults can help allocate the limited resources. In this chapter, we outline the contributions of this thesis in identifying fault-prone clones, recommendations based on this thesis, the threats to validity of both case studies, and discuss future work.

6.1 Contributions

This thesis makes the following contributions:

- *A formal definition of clone pair genealogies and clone evolutionary patterns.* We provide a formal definition for clone pair changes, states, and genealogies. Using these definitions, we identify six different clone pair evolutionary patterns.
- *We identify risky clone pair states and changes in clone pair genealogies.* We examine the six different clone evolutionary patterns, the states, and the changes within code clone genealogies. We show that each clone evolutionary pattern exhibits a different level of fault-proneness. We then build on this result and examine the size of code clones. We determine that size has an effect on the fault-proneness of the different clone evolutionary patterns.
- *We show that adding clone evolutionary metrics to prediction models increases the F-Measure of clone prediction models by up to 26%.* We build a series of Random Forest fault-prediction models to predict faults in code clones. We compare models consisting of metrics that can be collected from one snapshot of a software system to models that also include metrics from a clone pair's genealogy history. Overall, we find that adding historical

information increases the F-Measure of our prediction models in the three subject systems by up to 26%. We also determine that the size of the methods containing the clones, and the time between changes to a clone pair are the most important predictors of faults in clones.

- *We examine late propagation in more detail and determine specific characteristics of fault-prone genealogies experiencing late propagation.* We divide the late propagation evolutionary pattern into eight different types. Using these types, we show that some types are less fault-prone than others. We also show that propagation does not always occur in genealogies exhibiting a late propagation pattern. Lastly, we show that late propagation without any propagation is the most risky late propagation type in our three subject systems, in terms of both its fault frequency and quantity of faults.

6.2 Recommendations

This thesis makes the following recommendations and observations:

- *All clones are not equally risky.* Certain characteristics of clones, like their evolutionary patterns or the size of the clones, can be used to locate the most fault-prone clones and prioritize them for testing.

- *Clone evolutionary metrics increase the precision and recall of fault-prediction models.* We recommend that clone evolutionary metrics be tracked and used to locate the most fault-prone clones. However, the increase in precision and recall is system-dependent. The system should be modeled to determine if the increase in precision and recall merits the additional resources needed to track the evolutionary history of clones.

- *Late propagation does not always involve any propagation of changes.* The combination of a diverging change later followed by a re-synchronizing change does not necessarily involve propagating changes between two clones. In addition, clones that do not experience any

propagation are among the most fault-prone clones. Developers should exercise caution when reverting changes that modify clones.

6.3 Threats to Validity

This section discusses the threats to validity of our thesis, following the guidelines for case study research [77].

Construct validity threats concern the relationship between theory and observation. In this thesis the threats are mainly due to measurement errors possibly introduced by our chosen clone detection tools. The tools are not ideal. They identify false positive clones and fail to detect some clones within a software system. To reduce the possibility of misclassification of a code fragment as a clone, we chose two clone detection tools that use two different clone detection approaches and repeat each case study using both tools.

Both of the clone detection tools in this thesis can detect identical (*i.e.*, type 1) and near-identical clones (*i.e.*, type 2). CCFinder can identify type 1 and 2 clones with some gaps. Additional lines of code within one clone in 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 could be inaccurate.

J-REX's use of heuristics to identify fault-fixing changes is a threat to construct validity. 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*. [78] and Mockus *et al*. [74]. Additionally, we 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 our case studies, as they are exploratory studies [77]. Although we cannot claim causation, we do identify in our late propagation case study a relationship 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. In our case study on clone genealogies, we simply report observations and correlations, although our discussion tries to explain these observations.

Conclusion validity threats concern the relationship 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 three Java systems that are different sizes and belong to different domains, further validation on more systems should be performed. Many of our results are system-dependent. For example, a different set of predictors is used for each subject system and clone detection tool when building our prediction models. Therefore, studies on more systems should be done to further validate our results.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the details needed to replicate our case studies. All three of our selected subject systems are publicly available for study.

6.4 Future Work

- **Extending the Study to Include More Systems**

Both case studies examined Java systems. Two of the systems in our case studies are built using a plug-in architecture. In the future we will extend our studies to include systems that use a variety of programming languages.

- **Examining Type 3 Clones**

We limit our study of genealogies to type 1 and 2 clones. In the future, we will repeat the study using clone detection tools that can identify type 3 clones. We will outline a series of assumptions to handle the situation where lines of code are added from one clone and not another in a clone pair. Currently this type of change could continue to be detected by a clone detection tool as a type 3 clone, even if the change should have been propagated.

- **Examining the Effect of Size on Instances of Late Propagation**

In our thesis, we showed that the size of a clone pair has an effect on the fault-proneness of different clone evolutionary patterns. This effect was especially pronounced for the late propagation evolutionary pattern. In the future, we will extend our examination of late propagation to consider the size of the clone when determining the fault-proneness of different types of late propagation.

Bibliography

[1] L. Barbour, F. Khomh, and Z. Ying, "Late Propagation in Software Clones," in *Proc. 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 273-282.

[2] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An Empirical Study of Code Clone Genealogies," in *Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Lisbon, Portugal, 2005, pp. 187-196.

[3] R. Geiger, B. Fluri, H. Gall, and M. Pinzger, "Relation of Code Clones and Change Couplings," in *Proc. 9th International Conference of Fundamental Approaches to Software Engineering (FASE)*, 2006, pp. 411-425.

[4] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley, 1999.

[5] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the Harmfulness of Cloning: A Change Based Experiment," in *Proc. Fourth International Workshop on Mining Software Repositories (MSR)*, 2007, pp. 18-18.

[6] C. Kapser and M. W. Godfrey, ""Cloning Considered Harmful" Considered Harmful," in *Proc. 13th Working Conference on Reverse Engineering (WCRE)*, Washington, DC, USA, 2006, pp. 19-28.

[7] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," in *Proc. International Symposium on Empirical Software Engineering (ISESE)*, 2004, pp. 83-92.

[8] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," *School of Computing TR 2007-541, Queen's University*, vol. 115, 2007.

[9] 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.

[10] L. Aversano, L. Cerulo, and M. Di Penta, "How Clones are Maintained: An Empirical Study," in *Proc. 11th European Conference on Software Maintenance and Reengineering (CSMR)*, 2007, pp. 81-90.

[11] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting Common Bug Prediction Findings using Effort-Aware Models," in *Proc. IEEE International Conference on Software Maintenance (ICSM)*, Washington, DC, USA, 2010, pp. 1-10.

[12] 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.

[13] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone Smells in Software Evolution," in *Proc. IEEE International Conference on Software Maintenance (ICSM)*, 2007, pp. 24 -33.

[14] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting Code Clones for Refactoring Using Combinations of Clone Metrics," in *Proc. 5th International Workshop on Software Clones (IWSC)*, Waikiki, Honolulu, HI, USA, 2011, pp. 7-13.

[15] S. Schulze, M. Kuhlemann, and M. Rosenmuller, "Towards a Refactoring Guideline using Code Clone Classification," in *Proc. 2nd Workshop on Refactoring Tools (WRT)*, Nashville, Tennessee, 2008, pp. 1-4.

[16] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating Code Clone Genealogies at Release Level: An Empirical Study," in *Proc. 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2010, pp. 87-96.

[17] N. Göde and R. Koschke, "Frequency and Risks of Changes to Clones," in *Proc. 33rd International Conference on Software Engineering (ICSE)*, 2011.

[18] N. Göde, "Evolution of Type-1 Clones," in *Proc. 9th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2009, pp. 77-86.

[19] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?," in *Proc. 7th IEEE Working Conference on Mining Software Repositories (MSR)*, 2010, pp. 72-81.

[20] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl, "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," in *Proc. International Conference on Software Maintenance (ICSM)*, 1997, pp. 314-321.

[21] G. Antoniol, "Analyzing Cloning Evolution in the Linux Kernel," *Information and Software Technology*, vol. 44, pp. 755-765, 2002.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Transactions on Software Engineering*, vol. 32, pp. 176-192, 2006.

[23] N. Göde, "Clone Removal: Fact or Fiction?," in *Proc. 4th International Workshop on Software Clones (IWSC)*, Cape Town, South Africa, 2010, pp. 33-40.

[24] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy, "Analyzing and Forecasting Near-Miss Clones in Evolving Software: An Empirical Study," in *Proc. 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2011, pp. 295-304.

[25] G. Jin and Z. Ying, "Detecting Clones in Business Applications," in *Proc. 15th Working Conference on Reverse Engineering (WCRE)*, 2008, pp. 91-100.

[26] D. Martin and J. R. Cordy, "Analyzing Web Service Similarity Using Contextual Clones," in *Proc. 5th International Workshop on Software Clones (IWSC)*, Waikiki, Honolulu, HI, USA, 2011, pp. 41-46.

[27] J. R. Cordy, T. R. Dean, and N. Synytskyy, "Practical Language-Independent Detection of Near-Miss Clones," in *Proc. Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, Markham, Ontario, Canada, 2004, pp. 1-12.

[28] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding Software License Violations through Binary Code Clone Detection," in *Proc. 8th Working Conference on Mining Software Repositories (MSR)*, Waikiki, Honolulu, HI, USA, 2011, pp. 63-72.

[29] I. J. Davis and M. W. Godfrey, "From Whence It Came: Detecting Source Code Clones by Analyzing Assembler," in *Proc.17th Working Conference on Reverse Engineering (WCRE)*, 2010, pp. 242-246.

[30] A. Santone, "Clone Detection Through Process Algebras and Java Bytecode," in *Proc. 5th International Workshop on Software Clones (IWSC)*, Waikiki, Honolulu, HI, USA, 2011, pp. 73-74.

[31] N. Göde and J. Harder, "Clone Stability," in *Proc. 15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011.

[32] N. Göde and J. Harder, "Oops!... I Changed It Again," in *Proc. 5th International Workshop on Software Clones (IWSC)*, 2011, pp. 14-20.

[33] J. Krinke, "Is Cloned Code More Stable than Non-cloned Code?," in *Proc. 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Los Alamitos, CA, USA, 2008, pp. 57-66.

[34] J. Krinke, "A Study of Consistent and Inconsistent Changes to Code Clones," in *Proc. 14th Working Conference on Reverse Engineering (WCRE)*, Los Alamitos, CA, USA, 2007, pp. 170-178.

[35] C. K. Roy and J. R. Cordy, "Near-Miss Function Clones in Open Source Software: An Empirical Study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, pp. 165-189, 2010.

[36] C. K. Roy and J. R. Cordy, "An Empirical Study of Function Clones in Open Source Software," in *Proc. 15th Working Conference on Reverse Engineering (WCRE)*, 2008, pp. 81-90.

[37] J. Ossher, H. Sajnani, and C. Lopes, "File Cloning in Open Source Java projects: The Good, the Bad, and the Ugly," in *Proc. 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 283-292.

[38] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the Effectiveness of Clone Detection by String Matching," *Journal of Software Maintenance*, vol. 18, pp. 37-58, 2006.

[39] J. H. Johnson, "Substring Matching for Clone Detection and Change Tracking," in *Proc. International Conference on Software Maintenance (ICSM)*, 1994, pp. 120--126.

[40] A. Marcus and J. I. Maletic, "Identification of High-Level Concept Clones in Source Code," in *Proc. 16th Annual International Conference on Automated Software Engineering (ASE)*, 2001, pp. 107--114.

[41] B. S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," in *Proc. 2nd Working Conference on Reverse Engineering (WCRE)*, 1995, pp. 86-95.

[42] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code," *IEEE Trans. Software Eng.*, vol. 32, pp. 176-192, 2006.

[43] W. S. Evans, C. W. Fraser, and F. Ma, "Clone Detection via Structural Abstraction," *Software Quality Journal*, vol. 17, pp. 309-330, 2009.

[44] V. Wahler, D. Seipel, J. von Gudenberg, and G. Fischer, "Clone Detection in Source Code by Frequent Itemset Techniques," in *Proc. 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, 2004, pp. 128--135.

[45] W. Yang, "Identifying Syntactic Differences between Two Programs," *Software - Practice and Experience*, vol. 21, pp. 739-755, 1991.

[46] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Proc. 8th International Symposium on Static Analysis (SAS)*, 2001, pp. 40--56.

[47] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Proc. 8th Working Conference on Reverse Engineering (WCRE)*, 2001, pp. 301--309.

[48] C. Liu, C. Chen, J. Han, and P. Yu, "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis," in *Proc. 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2006, pp. 872-881.

[49] Y. Xue, Z. Xing, and S. Jarzabek, "CloneDiff: Semantic Differencing of Clones," in *Proc. 5th International Workshop on Software Clones (IWSC)*, Waikiki, Honolulu, HI, USA, 2011, pp. 83-84.

[50] K. Kontogiannis, R. D. Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern Matching for clone and Concept Detection," *Automated Software Engineering*, vol. 3, pp. 79-108, 1996.

[51] F. Lanubile and T. Mallardo, "Finding Function Clones in Web Applications," in *Proc. 7th European Conference on Software Maintenance and Reengineering (CSMR)*, 2003, pp. 379--386.

[52] G. A. D. Lucca, M. D. Penta, and A. R. Fasolino, "An Approach to Identify Duplicated Web Pages," in *Proc. 26th Annual International Computer Software and Applications Conference (COMPSAC)*, 2002, pp. 481--486.

[53] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," in *Proc. International Conference on Software Maintenance (ICSM)*, 1996, pp. 244--253.

[54] R. K. Saha, C. K. Roy, and K. A. Schneider, "An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies," in *Proc. 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 293-302.

[55] L. Jiang, Z. Su, and E. Chiu, "Context-Based Detection of Clone-Related Bugs," in *Proc. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE)*, Dubrovnik, Croatia, 2007, pp. 55-64.

[56] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An Empirical Study on Inconsistent Changes to Code Clones at Release Level," in *Proc. 16th Working Conference on Reverse Engineering (WCRE)*, 2009, pp. 85-94.

[57] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do Code Clones Matter?," in *Proc. IEEE 31st International Conference on Software Engineering (ICSE)*, 2009, pp. 485-495.

[58] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, pp. 653-661, 2000.

[59] A. E. Hassan, "Predicting Faults using the Complexity of Code Changes," in *Proc. 31st International Conference on Software Engineering (ICSE)*, 2009, pp. 78-88.

[60] R. Moser, W. Pedrycz, and G. Succi, "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction," in *Proc. 30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany, 2008, pp. 181-190.

[61] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," in *Proc. 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, USA, 2005, pp. 284-292.

[62] N. Ohlsson and H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches," *IEEE Transactions on Software Engineering*, vol. 22, pp. 886-894, 1996.

[63] S. Qinbao, M. Shepperd, M. Cartwright, and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction," *IEEE Transactions on Software Engineering*, vol. 32, pp. 69-82, 2006.

[64] K. Sunghun, E. J. Whitehead, and Z. Yi, "Classifying Software Changes: Clean or Buggy?," *IEEE Transactions on Software Engineering*, vol. 34, pp. 181-196, 2008.

[65] T. Zimmermann and N. Nagappan, "Predicting Defects using Network Analysis on Dependency Graphs," in *Proc. ACM/IEEE 30th International Conference on Software Engineering (ICSE)*, 2008, pp. 531-540.

[66] T. Mende and R. Koschke, "Effort-Aware Defect Prediction Models," in *Proc. 14th European Conference on Software Maintenance and Reengineering (CSMR)*, 2010, pp. 107-116.

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

[68] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting Failures with Developer Networks and Social Network Analysis," in *Proc. 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE)*, Atlanta, Georgia, 2008, pp. 13-23.

[69] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*: Addison-Wesley, 1999.

[70] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, pp. 5-32, 2001.

[71] M. Robnik-Sikonja, "Improving Random Forests," in *ECML*, 2004, pp. 359-370.

[72] T. Hothorn, K. Hornik, and A. Zeileis, "party: A Laboratory for Recursive Part(y)itioning," in *R package version 0.9-0*, 2006.

[73] W. Shang, Z. M. Jiang, B. Adams, and A. E. Hassan, "MapReduce as a General Framework to Support Research in Mining Software Repositories (MSR)," in *Proc. 6th IEEE International Working Conference on Mining Software Repositories (MSR)*, 2009, pp. 21 -30.

[74] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes using Historic Databases " in *Proc. International Conference on Software Maintenance (ICSM)*, 2000, p. 120.

[75] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. Matsumoto, "Predicting Re-opened Bugs: A Case Study on the Eclipse Project," in *Proc. 17th Working Conference on Reverse Engineering (WCRE)*, 2010, pp. 249 -258.

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

[77] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*. London: SAGE Publications, 2002.

[78] A. E. Hassan and R. C. Holt, "Studying The Evolution of Software Systems Using Evolutionary Code Extractors," in *Proc. 7th International Workshop on Principles of Software Evolution (IWPSE)*, Washington, DC, USA, 2004, pp. 76-81.