Noname manuscript No.
(will be inserted by the editor)

An Investigation of the Fault-proneness of Clone Evolutionary Patterns

Liliane Barbour, Le An, Foutse Khomh, Ying Zou, Shaohua Wang

Received: date / Accepted: date

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 in order to properly propagate any changes between clones. A clone pair may experience many changes during the creation and maintenance of a software system. 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. When 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 paper, we examine clone genealogies to identify fault-prone "patterns" of states and changes. We explore the use of clone genealogy information in fault prediction. We conduct a quasi-experiment with four long-lived software systems (*i.e.*, Apache Ant,

Liliane Barbour
Department of Electrical and Computer Engineering
Queen's University, ON, Canada
E-mail: l.barbour@queensu.ca

Le An
SWAT, École Polytechnique de Montréal, QC, Canada
E-mail: le.an@polymtl.ca

Foutse Khomh
SWAT, École Polytechnique de Montréal, QC, Canada
E-mail: foutse.khomh@polymtl.ca

Ying Zou
Department of Electrical and Computer Engineering
Queen's University, ON
E-mail: ying.zou@queensu.ca

Shaohua Wang
School of Computing
Queen's University, ON, Canada
E-mail: shaohua@cs.queensu.ca

ArgoUML, JEdit, Maven) and identify clones using the NiCad and iClones clone detection tools. Overall, we find that the size of the clone can impact the fault-proneness of a clone pair. However, there is no clear impact of the time interval between changes to a clone pair on the fault-proneness of the clone pair. We also discover that adding clone genealogy information can increase the explanatory power of fault prediction models.

Keywords Clone genealogies; Fault-proneness; Metrics.

1 Introduction

Cloning occurs when two code segments are highly similar or identical to each other. Each code segment is known as a clone, and the two clones form a clone pair. A clone group is a set of code segments, where any two of them form a clone pair. Cloning is a common practice in software development. Some clones are introduced intentionally through the copy and paste actions of developers, while others are introduced accidentally [1].

Once created, clones evolve as they are modified during both the development and maintenance phases of software systems. A clone pair is in a consistent state if the clones are identical or similar. A clone pair is in an inconsistent state if they are no longer similar. Over time, a clone pair is either in a consistent state or an inconsistent state. Clones that become inconsistent can be later re-synchronized, and consistent clones can diverge. 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 a specific "clone evolutionary pattern", which defines a specific ordering of states and changes that occur frequently in clone genealogies over the lifetime of a software system. For example, a consistent clone pair that transitions to an inconsistent state, and then re-synchronizes to a consistent state is known as late propagation [2]. Clone pairs that transition to inconsistent states during their evolution are difficult to monitor using clone detection tools. Hence, they are more at risk of faults due to a lack of propagation of changes.

Previous studies [2,3,4,5] on clone genealogies have defined specific clone evolutionary patterns and studied their relationship with faults. Specific clone evolutionary patterns have been identified as fault-prone without detailed information. More specifically, a genealogy only provides details about the past, but cannot inform a developer about whether the current state or the next change will be risky. Moreover, the history of the clone groups has also not been considered when predicting faults in clones. In our work, we examine clone evolutionary patterns and changes within clone genealogies and their relationship with faults and strive to provide insights on the characteristics of fault-prone clones. For each clone group, we analyze all the clone pairs within the clone group. We chose to study clone pairs instead of clone groups since clone pairs within the same clone group are not equally risky. Additionally, we investigate if metrics collected from clone pair genealogies can improve the performance of prediction models when identifying clone pairs that are at a higher risk of faults.

We investigate the clone genealogies of four open source software systems (*i.e.*, Apache Ant, ArgoUML, JEdit, Maven). 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 previous 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: Can we predict faults in software clones using clone genealogy information? One snapshot of a software system provides limited information that can be used to predict faults in a clone pair. However, genealogy information about a clone pair takes more effort to collect and track. We propose metrics to capture information about the genealogy of a clone pair and we use statistical models to establish and inspect dependencies between the metrics and faults in clone pairs.

We provide three contributions in this paper:

- We give a formal definition of clone pair genealogies and clone evolutionary patterns.
- We identify characteristics of fault-prone clone pair states and changes in clone pair genealogies that can be used to locate the most fault-prone clones.
- We show that clone genealogy information can increase the explanatory power of fault prediction models. In particular, the number of previous faults in a clone pair can help predict future faults in the clone fragments.

Organization – Section 2 summarizes related studies on clone genealogies and prediction of faults. Section 3 discusses the building blocks of genealogies and clone evolutionary patterns. Section 4 outlines our study approach. Section 5 summarizes the study results. Section 6 reports on a qualitative evaluation of the genealogies and discusses the results of the study. Section 7 discusses the threats to the validity of our study. Section 8 concludes the paper and outlines avenues for future work.

2 Related Work

2.1 Clone Genealogies

Kim *et al.* [6] performed the first study of clone evolution. They analyzed groups of clone snippets, known as clone classes, and described the types of changes that can be experienced by a clone class. In our work 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. Kim *et al.* also performed a case study using CCFinder and found that clones are very volatile. Half the clones became inconsistent within eight check-ins. In our work, we continue to examine clones after they become inconsistent to examine their fault-proneness.

2.2 Bug-proneness of code Clone

Rahman et al. [7] explored the relationship between defect-proneness and code clones by analyzing four open-source C projects. They did not observe a strong correlation between bugs and code clones, nor a correlation between bug-proneness and cloned code size. Their findings challenge the Fowler et al's [8] claim that code clones are "bad code smells". However, this study did not take the evolution of code clones nor a more popular programming language, Java, into account. Juergens et al. [9] conducted a case study on open-source and commercial systems to investigate whether code clone's inconsistent changes can lead to defects. They observed that nearly half of the unintentionally inconsistent changes caused defects. Although this work only studies one type of clone evolution, it leads us to further discover the relationship between bug-proneness and other clone evolution types.

2.3 Analysis of Clone Genealogies

Krinke [10] examined inconsistent and consistent changes to clones in 200 weekly snapshots of five open source systems. The study examined identical clones. About half of the changes to identical clones were consistent. The 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. In our work, we examine the relationship between delay since the last change and faults.

Krinke performed a second study on the stability of cloned code, which was repeated and extended by Göde *et al.* [11]. In Göde *et al.*'s work, they performed clone detection using a token-based clone detection tool, but used a time interval between snapshots of one commit. Overall, their findings agreed with Krinke that cloned code is much more stable than non-cloned code. They also experimented with the parameters of their clone detection tool and showed that the results are impacted by the choice of parameters. Because of this result, we use two different clone detection tools in our study, with each one implementing a different clone detection technique. These two clone detection tools (*i.e.*, NiCad and iClones) were found to achieve higher precision and recall by Svajlenko et al. [12]. For each clone detection tool, we use the same parameters as in the study of Svajlenko et al. [12], that compared 11 different clone detection tools.

Göde *et al.* [13] performed a different study on code clones and found that over half of the clones in three systems were stagnant. In other words, once they were formed, they were never modified. About 12% of the clones experienced a change. In a similar study [13], Göde *et al.* found that 87.8% of clones are never changed or only changed once. They suggest that these clones are irrelevant to developers. In our study, we consider all the changes that occurred during the evolutionary history of the clones and identify the most fault-prone ones. We also consider metrics that contain historical information taken from clone genealogies to identify fault-prone clone pairs.

Göde *et al.* [14] performed a study examining consecutive change pairs within clone genealogies. They defined four different types of consecutive changes. They examined the relationship between the change pair type, the delay between changes, the change author, and the location of the clones in the project structure and whether an inconsistent change was intentional. Overall, they found that two consecutive changes are the most common change pair, and that few inconsistent changes were accidental.

Thummalapenta *et al.* [2] performed a study that looked at four different types of clone evolutionary patterns within clone classes. They classified their clone classes into consistent evolution, independent, delayed propagation, and late 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 our work, we examine clones in more detail and define further clone evolutionary patterns.

Barbour *et al.* [4,5] investigated faults in 8 different types of late propagation and found that late propagation clones are more fault-prone when: (i) clones in the pair undergo a diverging modification followed by a reconciling change that modifies both clones in the clone pair; or (ii) clones in pair experience diverging changes, followed by a reconciling change that modifies only the diverging clone in the clone pair. They also reported that the size of the clones experiencing late propagation has an effect on the fault-proneness of specific types of late propagation genealogies. Recently, Mondal et *et al.* [15] investigated the frequency of late propagation for different types of clones (*i.e.*, type 1, type 2, and type 3) using the NiCad clone detection tool. They found that late propagation occurs more frequently in type 3 clones. They also observed that late propagation of type 3 clones are more fault-prone than late propagations of either type 1 or type 2 clones. In this paper, we build on these previous works to analyze the fault-proneness of all types of clone evolutionary patterns (*i.e.*, not only late propagation).

Xie *et al.* [16] investigated two evolutionary phenomena on clones: the mutation of the type of a clone during the evolution of a system, and the migration of clone segments across the repositories of a software system. They observed that clone migration and clone mutation occur frequently in clone genealogies, and that increasing the distance between code segments in a clone group during the evolution of the system increases the risk for faults. They also found that mutating clones to type 2 or type 3 increases the risk for faults. In a follow-up study [17], they examined the fault-proneness of clone migration in clone genealogies and found that migrated clone segments, clone groups, and clone genealogies are not equally fault-prone. They also found that when a clone mutation occurs during a clone migration, the risk for faults in the migrated clone is increased. The migration of a clone that was not changed for a long period of time is also reported to be risky.

2.4 Statistical Explanatory Models

Several studies have investigated the use of process and product metrics to build fault prediction and explanatory models.

Khoshgoftaar *et al.* [18] analyzed two consecutive releases of a large software system used in telecommunications and showed that the number of past added/removed lines of code is a good predictor of future faults at the module level. Bernstein *et al.* [19] used the number of revisions and corrections on a file, recorded in a given amount of time, to predict the location of faults. Graves *et al.* [20] investigated different predictors of faults using statistical models and found that the sum of contributions from all changes to a module is the best predictor of faults in a module. Nagappan and Ball [21] analyzed the relation between code churn (*i.e.*, the amount of lines added, modified or deleted to a file) and fault density in Windows Server 2003 and concluded that relative code churns are better predictors of fault density than absolute code churns. Hassan [22] introduced the notion of entropy of changes to capture the complexity of a source code change process. He performed a case study using six open-source software systems and found that the entropy of changes is a better predictor of faults than traditional predictors like the amount of changes or the number of previous faults.

El Emam *et al.* [23] combined Chidamber & Kemerer metrics [24] with Briand *et al.*'s coupling metrics [25] to predict faults in a large commercial Java system. Nagappan *et al.* [26] investigated the use of source code metrics to predict post-release faults at the module level using five Microsoft software systems. They found that complexity metrics can successfully predict post-release faults, but that the set of best predictors was system-dependant. Zimmermann *et al.* [27] also used source code metrics to predict faults in Eclipse. Arisholm and Briand [28] proposed the use of code quality, class structure, changes in class structure, and the history of class-level changes and faults to predict faulty classes.

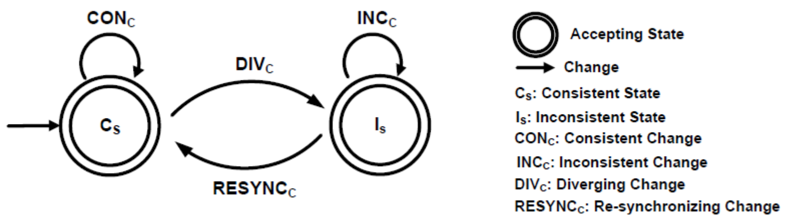


Fig. 1: Clone Pair States and Changes

Moser *et al.* [29] performed a comparative analysis of the predictive power of process and source code metrics for fault prediction and found that process metrics are better predictors of faults than product metrics. In this work, we examine whether clone genealogy metrics can be used to increase the performance of fault prediction models built using product and process metrics.

Kononenko *et al.* [30] investigated the relationships between the quality of code review and technical, personal, and participation factors in the code review process. They found that both personal and participation factors can influence the quality of code review. McIntosh *et al.* [31] built explanatory models to explore the impact of the code review process on software quality. They found a significant correlation between code review quality and the factors on code review coverage, participation, and reviewers' expertise. In this paper, we built explanatory models to investigate the relationship between clone genealogies metrics and fault-proneness.

3 Clone Evolutionary Patterns

3.1 States and Transitions of Clone Pairs

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 1. A clone pair is in a consistent state if the code segments in the pair are identical or similar (*i.e.*, have a cloned-relationship). A clone pair is in an inconsistent state if the code segments in the pair are no longer similar (*i.e.*, the cloned-relationship has been removed). 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.

A change is an input action that modifies the content of one or both of the code segments in a clone pair. A change can transition the clone pair between states, or maintain the clone pair's current state. For example, if a clone pair is in a consistent state and experiences a change that removes the cloned-relationship between the code segments in the pair, the clone pair transitions into an inconsistent state. If the change preserves the cloned-relationship between the code segments, the clone pair remains in a consistent state.

There are four possible changes:

- Consistent Change (CON_c): A change modifies one or both code segments of a clone pair in a consistent state. Such change keeps the code segments in the clone pair in a cloned-relationship (*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 code segments of a clone pair in an inconsistent state. The code segments continue to be dissimilar, 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 code segments of a clone pair in an inconsistent state. The change causes the code segments to have a cloned-relationship. 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 code segments in a clone pair in a consistent state. The change removes the cloned-relationship between the code segments (*i.e.*, diverging change DIV_c is the transition from consistent state C_s to inconsistent state I_s).

A clone genealogy describes the evolutionary history of a clone pair. We define a clone genealogy as a finite transition system, $G = \{S, Act, Trans, I_0, A\}$, 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 $A = \{C_s, I_s\}$

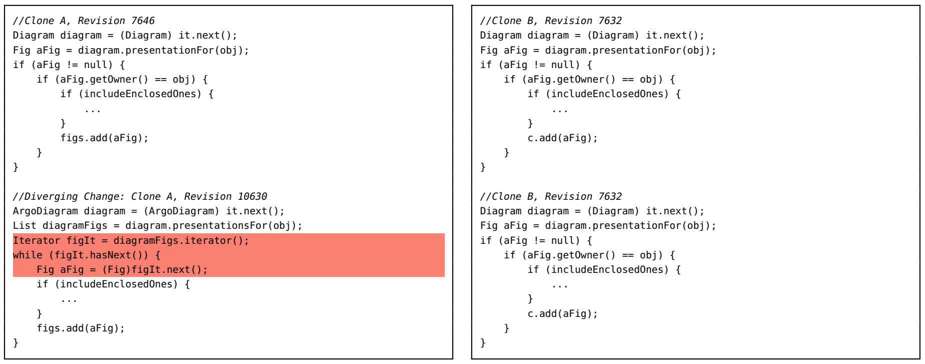
Figure 1 is a pictorial representation of the clone genealogy transition system. A genealogy is a finite model, and grows as changes are applied to a clone pair, terminating in either a consistent or an inconsistent state. A clone pair starts from a consistent state when the clone pair can be detected. Therefore, a clone genealogy is always initiated in a consistent state.

3.2 Six Clone Evolutionary Patterns

A "clone pair evolutionary pattern" is a path in a graph G. The graph G represents the history of states and changes for a clone pair. It is a finite sequence of states $P = s_0 s_1 s_2 \dots s_n$, where $s_0, s_1, s_2, \dots, s_n \in S = \{C_s, I_s\}$. The following six evolutionary patterns define all possible paths in graph G, where n is an integer ≥ 1:

– Unchanged Pattern (UNC_p): The clone pair is formed, but never experiences any changes (*i.e.*, UNC_p is defined as the path C_s 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 $C_sC_s^n$ 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 $C_sI_s^n$ 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 $C_sC_s^nI_s^n$ 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 $(C_s^nI_s^n)^nC_s^n$ 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 $(C_s^nI_s^n)^nC_s^nI_s^n$ in graph G).

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



```
//Clone A, Revision 7646
Diagram diagram = (Diagram) it.next();
Fig aFig = diagram.presentationFor(obj);
if (aFig != null) {
    if (aFig.getOwner() == obj) {
        if (includeEnclosedOnes) {
            ...
        }
        figs.add(aFig);
    }
}

//Diverging Change: Clone A, Revision 10630
ArgoDiagram diagram = (ArgoDiagram) it.next();
List diagramFigs = diagram.presentationsFor(obj);
Iterator figIt = diagramFigs.iterator();
while (figIt.hasNext()) {
    Fig aFig = (Fig)figIt.next();
    if (includeEnclosedOnes) {
        ...
    }
    figs.add(aFig);
}
```

```
//Clone B, Revision 7632
Diagram diagram = (Diagram) it.next();
Fig aFig = diagram.presentationFor(obj);
if (aFig != null) {
    if (aFig.getOwner() == obj) {
        if (includeEnclosedOnes) {
            ...
        }
        c.add(aFig);
    }
}

//Clone B, Revision 7632
Diagram diagram = (Diagram) it.next();
Fig aFig = diagram.presentationFor(obj);
if (aFig != null) {
    if (aFig.getOwner() == obj) {
        if (includeEnclosedOnes) {
            ...
        }
        c.add(aFig);
    }
}
```

Fig. 2: An Example of an Inconsistent Genealogy from ArgoUML using NiCad (inconsistent lines are highlighted)

Figure 2 shows an example of inconsistent clone genealogy. The example is a code segment from a clone containing 18 lines of code and is taken from ArgoUML using the clone detection tool NiCad. When the clone pair is created in revision 7646 it is in a consistent state (C_s). Its genealogy is described by the graph G and it exhibits an unchanged pattern (UNC_p). Clone A then experiences a diverging change (DIV_c) that modifies several lines of code. The clone pair is now in an inconsistent state (I_s). This gives it the path C_sI_s in graph G, which belongs to the inconsistent evolutionary pattern (INC_p).

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 [32]. 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 [2]. However, previous studies [2] have shown that the late propagation is risky and fault-prone. For example, the diverging change in a late propagation may be accidental, given that the clone pair is later re-synchronized. However, accidental changes to clones are considered risky. Therefore, late propagation is considered risky [2]. 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.

4 Study Design

This section describes the setup of our quasi-experiment that aims to identify fault-prone states and changes in clone genealogies. Figure 3 shows an overview of the steps we use to extract clone information from a source code repository and build clone genealogies. We describe our steps in more detail in the following subsections. We share our analytic scripts and data at: `https://github.com/swatlab/clone_genealogies`.

4.1 Subject Systems

We select four open-source Java systems as the subjects systems. All of the subject systems possess a long development history, which is suitable for our clone genealogy study.

- Apache Ant is an open-source build-tool with an extensive Java library. We study its revision history from January 2000 to July 2016.
- ArgoUML is a UML-modelling software system. We study its commit history from January 1998 to January 2015 (*i.e.*, until the most recent version of the project).

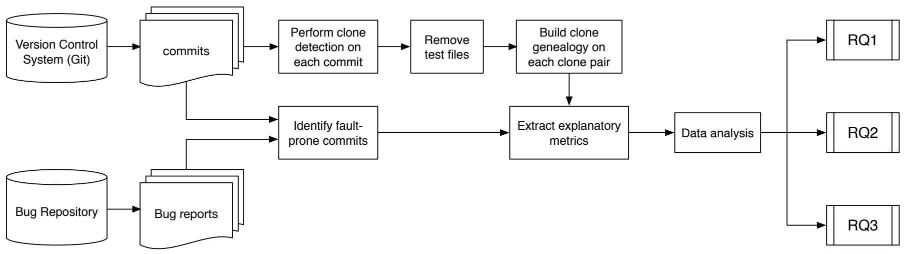

Version Control System (Git)
commits
Perform clone detection on each commit
Remove test files
Build clone genealogy on each clone pair
Identify fault-prone commits
Extract explanatory metrics
Data analysis
RQ1
RQ2
RQ3
Bug Repository
Bug reports

Fig. 3: Overview of the Analysis Process.

Table 1: Characteristics of the Systems

System	# LOC	% Java code	# Commits	# Clones		# Genealogies	
				NiCad	iClones	NiCad	iClones
ArgoUML	253.1k	70.1	17.8k	96.7M	17.3M	16.6k	7.6k
Ant	172.3k	80.0	13.4k	7.8M	7.2M	5.5k	7.1k
JEdit	215.8k	55.9	7.7k	3.7M	8.8M	6.8k	6.5k
Maven	88.2k	79.8	10.3k	1.6M	3.3M	0.7k	0.4k

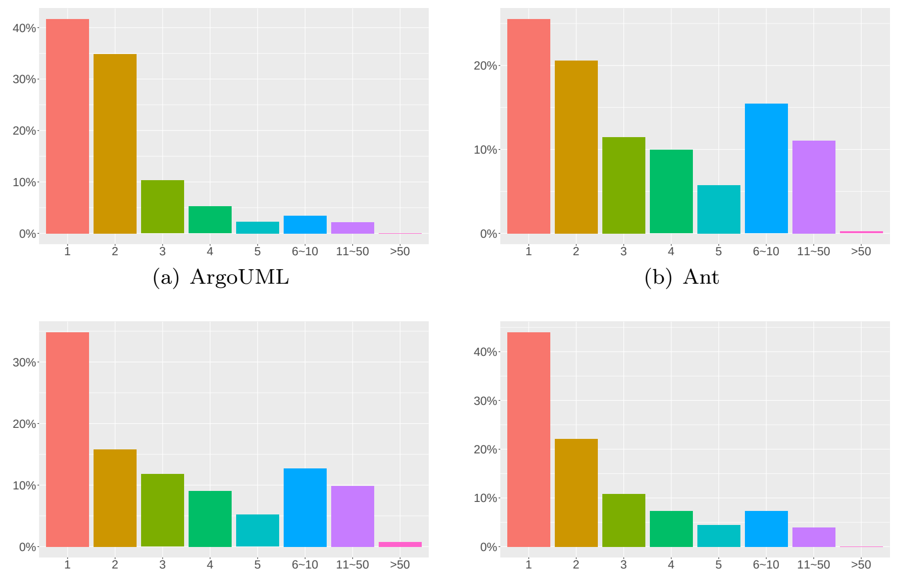
– JEdit is an open-source text editor built for programmers. It is written in Java, and provides support for editing more than 200 programming languages. Many plug-ins have been written for JEdit. In this study, we only examine the editor. The project started in 1998 and is still under development. We examine its revision history from September 2001 to July 2015.
– Maven is a build automation tool used primarily for Java projects. We study its commit history from September 2003 to July 2016.

Table 1 summarizes the characteristics of each system. We use the SLOC-Count tool [33] to count the total number of lines of code (LOC) and the percentage of Java code in each project. For each project, we provide LOC for the last studied revision. Table 2 shows the numbers of faulty changes and the numbers of clean changes for each subject system.

We examined the length of the clone genealogies contained in the selected software systems and observed that more than 50% of the genealogies only experienced 1-2 changes. Figures 4 and 5 show the frequency of the number changes in each of the clone genealogies. Overall, although the studied systems contain high numbers of genealogies, the genealogies tend to be short. In this paper, we do not consider the unchanged clone pattern (UNC_p), hence, all the studied clone genealogies experienced at least one change. Figure 6 depicts the number of clone genealogies deriving from a specific commit. In this figure, we eliminated outliers. The median value for each project is less than 3; implying that there are only few clone genealogies starting from each commit.

Table 2: Number of faulty and clean clone changes in each system.

	NiCad				iClones			
	ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
Faulty	3,246	643	49	300	2,967	363	48	256
Clean	2,629	3,440	319	436	1,876	2,292	273	1,168



(a) ArgoUML (b) Ant

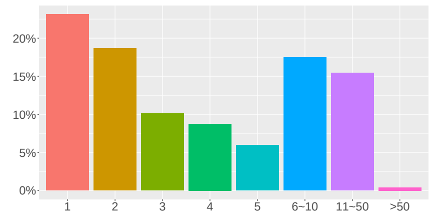
(c) JEdit (d) Maven

Fig. 4: Percentage of the frequency of the number of changes in a studied clone genealogy detected by NiCad

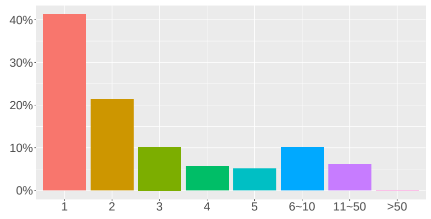
4.2 Data Preprocessing

To analyze a repository's history, Git provides high-performance functions to extract changed files, renamed files, and blame faulty files. Since the source code of ArgoUML and JEdit is managed by SVN, we use Git's `git-svn` command to convert the two systems' repositories to Git. Then, we use the following command to extract each commit's commit ID, committer email, commit date, and commit message:

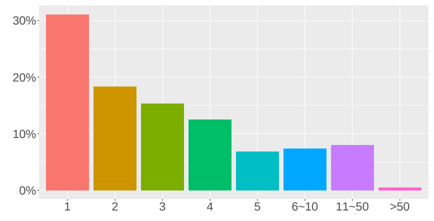
```
git log --pretty=format:"%H,%ae,%ai,%s"
```



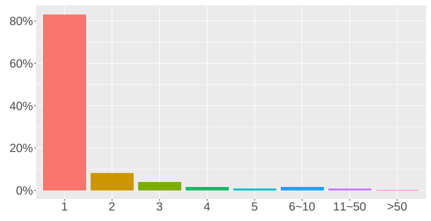
(a) ArgoUML



(b) Ant

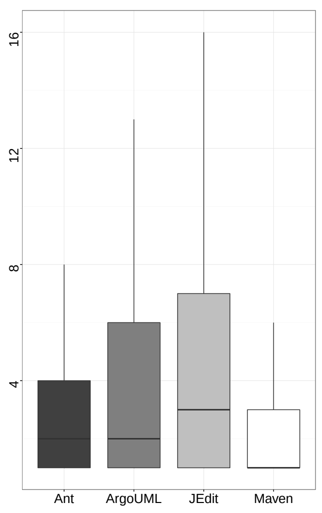


(c) JEdit

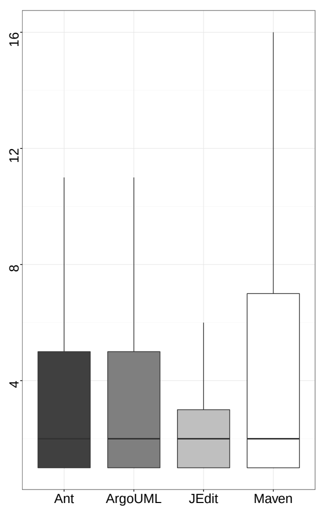


(d) Maven

Fig. 5: Percentage of the frequency of the number of changes in a studied clone genealogy detected by iClones



(a) NiCad



(b) iClones

Fig. 6: Number of clone genealogies starting from a specific commit

4.3 Detecting Faulty Changes

We leverage the SZZ algorithm [34] to detect changes that introduced faults. We first apply Fischer et al.'s heuristic [35] to identity fault-fixing commits by using regular expressions to detect bug IDs from the studied commit messages. We then mine the subject systems' bug tracking systems (*issuezilla* for ArgoUML, *Jira* for Ant and Maven, and *SourceForge* for JEdit) to extract their bugs' creation date. Next, we extract the modified files of each fault-fixing commit through the following Git command:

```
git log [commit-id] -n 1 --name-status
```

In this paper, we only take modified Java files into account. Given each file F in a commit C, we extract C's parent commit C'. For Ant and Maven, we use the `[commit-id]^` command to obtain C'; while for ArgoUML and JEdit, since their repositories were converted from SVN, we find the C's precedent commit C' by time, *i.e.*, C' is the nearest commit prior to C. Then, we use Git's `diff` command to extract F's deleted lines. We apply Git's `blame` command to identify commits that introduced these deleted lines, noted as the "candidate faulty changes". We eliminate the commits that only changed blank and comment lines. Finally, we filter the commits that were submitted after their corresponding bugs' creation date.

4.4 Extracting Clone Genealogies

Extracting clone genealogies from each subject system requires three steps: removing test files, detecting clones, and building clone genealogies.

Removing Test Files. Test files are frequently copied and then modified to create multiple test cases, so they often contain clones. These files are used for development purposes and not used during the normal execution of the system. They may also contain syntactically incorrect code. For all these reasons, we believe that clones in test code should be studied separately from clones in production code. Therefore, we exclude test files from our study. In future work, we plan to examine the evolution of clones in test code which are nevertheless clones and need to be maintained. To remove the test files, we perform a search on each system for files and folders with a filename containing the word "test". We then manually verify each file before removing it from the study to prevent the automatic removal of a non-test file, such as a file with the name "updateState.java". At the end of this semi-automatic process, we also manually verify all the remaining files in our data set, to ensure that no semantically test-related files remain in the data set of our study.

Detecting Clones. We use two existing clone detection tools to detect clones in the four systems: NiCad[36] and iClones [37]. We use the most recent versions of both tools: NiCad-4 and iClones-0.2. We select these two clone detection tools because they are recommended by Svajlenko et al. [12] who compared the performance of 11 clone detection tools from the literature. Today, NiCad and iClones are considered as state-of-the-art tools by the clone community [12].

Both NiCad and iClones use a hybrid approach to detect clones. We use the default settings of Nicad to detect clones greater than 10 lines of code, while using the default setting of iClons to detect clones with minimum 100 tokens. We detect identical clones and clones where the variable names are different (*i.e.*, "blindrename"). The same settings were used by Svajlenko et al. [12] in their comparison of 11 clone detection tools. With these settings, NiCad and iClones were found to achieve higher precision and recall in comparison to the other 9 clone detection tools that were studied.

We use the Git `checkout` command to retrieve a system's snapshot for a specific commit. Then we perform clone detection on each of the snapshots of the studied systems. Table 1 summarizes the number of clone pairs and clone genealogies found in each subject system using both clone detection tools. For ArgoUML and Ant, Nicad detected more clone pairs than iClones; while for JEdit and Maven, iClones detected more clone pairs. This difference in the number of clones found by the two detection tools is likely due to the lack of agreement on the definition of code clones [38] and to the implementation of the tools.

Building Clone Genealogies. Each clone detection tool outputs a list of clones within each source code repository. To create a set of clone pair genealogies, we link the clone pairs between each commit. A change to a clone can affect its size. A change to the file containing the clone, even if it does not affect the clone itself, can shift the clone's line numbers. To account for these changes when mapping clones, we use the Git *diff* command to query for a list of changes to each Java file. We limit our genealogies to describe only changes that modify the clone contents, not the clone line numbers. This is because a shift in the line numbers cannot cause the clone pair to transition to a different state.

We build a clone genealogy for each clone pair detected by the clone detection tool. We first extract a system's commit sequence list. For ArgoUML and JEdit, which were originally managed by SVN, we sort their commits by time in ascending order. For Ant and Maven, which are managed by Git, we make a list and put a system's last commit as the first element. Then we recursively look for the list's last element's parent commit until the system's first commit is met. We reverse the lists to obtain Ant and Maven's commit sequence lists.

For each clone pair, we track its modification in every commit along the commit sequence list. If a commit, C_{new}, changed a file that contains code in the clone pair, we use the `diff` command to compare the commit with its previous commit, C_{old}, in order to check whether the clone snippets are modified and to map the start line and end line numbers from C_{old} to C_{new}. We use Python's third-party patch parsing library *whatthepatch* [39] to extract the line mapping on a clone file between C_{old} and C_{new}. In case that the first lines, $L_1 \sim L_n$, of a clone snippets are deleted in C_{old} and no corresponding line added in C_{new} to replace these deleted lines, we map L_{n+1} from C_{old} to C_{new} as the start line. Similarly, in case that the last lines, $L_x \sim L_{x+n}$ are deleted in C_{old} and no corresponding line added in C_{new} to replace them, we map L_{x-1} from C_{old} to C_{new} as the end line.

We decide whether a clone is changed when there is any deleted or added lines performed in the clone's boundaries. If a clone is modified, we determine whether the new state of the clone pair is inconsistent (I_s) or consistent (C_s). We verify this by searching the clone pair list generated by a clone detection tool. We query the list for a matching clone pair in the new commit, C_{new}, that contains the start and end line numbers of the clone pair. If no clone pair is found, then the state of the clone is inconsistent and an inconsistent state (I_s) is added to the genealogy. If a clone is found, then a consistent state (C_s) is added to the genealogy. This process is repeated for each commit in the commit sequence list or until one or both of the clones is deleted. We use the following command to extract renamed files in a new commit:

```
git diff [old-commit] [new-commit] --name-status -M
```

This command can extract file pair, where a file is deleted and another file is added in the new commit and the two files have a code similarity greater than 50%. In this paper, we only consider the file pairs with more than 99% of code similarity as renamed files. When searching the clone sequence list, we allow a matching clone to be bigger than the clone pair, and contain the smaller clone. For example, if one of the clones in a clone pair is from lines 1 to 10, a matching clone in the clone pair list could be from lines 1 to 20. Although we add the bigger clone from the clone pair list to our genealogy, we continue to monitor only the smaller clone to generate the genealogy. The bigger clone (*i.e.*, lines 1 to 20 in our example) might disappear in a future revision, but the smaller clone (*i.e.*, lines 1 to 10 in our example) persists after the bigger clone is removed.

5 Study Results

This section reports and discusses the results of our study.

5.1 RQ1: Which clone evolutionary patterns and clone changes are most at risk of faults?

Motivation Developers are interested in identifying areas of a software applications that have a higher likelihood of faults. Previous studies [40] have identified clones as more fault-prone than non-cloned code. Clones occur frequently, with as much as one fifth of a software system containing duplicate code [1]. However, not all the clones lead to faults. It can be resource consuming to monitor all clone pairs for faults. It is beneficial if we can identify characteristics of fault-prone clone pairs, risky clone pairs can be highlighted for monitoring. In this research question, we examine whether the evolutionary pattern exhibited by the clone pair can be used to locate fault-prone clone pairs. Additionally, we study the different changes described in Section 3 to determine whether some types of changes are more likely to induce faults than others. This will make developers more aware of the potential risk of performing a specific type of change to a system.

Approach We examine this research question using the odds ratio (OR) and validate the statistical significance of the results using the Fisher's exact test. The Fisher's exact test [41] determines whether there are non random associations between two categorical variables (*e.g.*, a clone evolutionary pattern and the occurrence of faults). In this paper, we use a 95% confidence level (*i.e.*, $\alpha = 0.05$) as the cutoff to decide whether there exists statistically significant difference between a clone evolutionary pattern and the occurrence of faults. Since we will perform more than one comparison, we use Bonferroni correction [42] to control the familywise error rate. Concretely, we use the adjusted p-value, which is multiplied by the number of comparisons. The odds ratio compares the odds of an event occurring in two different groups, the "control" group and the "experimental" group. An $OR = 1$ implies that the event is equally likely in both the control and experimental group, an $OR > 1$ implies that the event is more likely in the experimental group, and an $OR < 1$ implies that it is more likely in the control group. An OR value close to zero or infinity means that the difference between the *ratios of the odds* of experiencing a fault by clone evolutionary patterns from the two groups is very large.

After building the set of clone genealogies for a subject system, we identify all clone evolutionary patterns within the genealogies. For each genealogy graph G, we visit each state in G and identify the clone evolutionary pattern (*i.e.*, the path P). Using the SZZ algorithm described in Section 4.3, we identify faulty states. We also check each change within the genealogy graph G to determine the type of the change, and verify whether the change is fault-inducing.

For the result of each clone detection tool, we perform the following three tests:

Faults vs. Clone Evolutionary Patterns: 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 experienced a fault-inducing change.*
We chose the $SYNC_p$ evolutionary pattern as our control group because we expect that clones that are maintained consistently (all the changes are propagated on time consistently) throughout their evolution history would be less prone to faults than others.

Faults vs. Changes: 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 experienced a fault fix as a consequence of the change.*
We chose CON_c changes as our control group because we expect a change that

Table 3: Contingency Tables for Clone Evolutionary Patterns

Pattern	Faulty	NiCad				iClones			
		ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
$SYNC_p$	Yes	514	191	5	19	709	178	3	15
	No	10,052	2,104	134	146	916	1,467	115	54
INC_p	Yes	717	191	3,438	87	2,054	564	2,283	82
	No	2,929	896	2,481	253	1,439	2,809	3,755	177
DIV_p	Yes	836	409	187	44	1,508	464	80	38
	No	802	1,521	183	94	538	1,443	196	64
LP_p	Yes	19	21	1	2	62	28	4	6
	No	564	37	1	3	74	40	5	2
$LPDIV_p$	Yes	70	30	6	4	273	26	2	6
	No	49	58	2	3	59	67	9	5

keeps two clone fragments in a consistent state to be less risky (*i.e.*, to have a low probability of introducing a fault in the system).

Faults vs. Evolutionary Patterns and Changes: We examine evolutionary patterns and changes together to determine the most fault-prone changes when a clone pair exhibits a specific clone evolutionary pattern (*e.g.*, late propagation followed by a consistent change). Using the inconsistent (INC_p) evolutionary pattern followed by a diverging change (INC_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. The final state of a clone evolutionary pattern is always consistent for the pattern. For example, a synchronous pattern ($SYNC_p$) will always end in a consistent state (C_s). Therefore, a clone pair can only be in one of two states at any time (*i.e.*, consistent or inconsistent). Each state only has two possible transitions, with each transition representing a change to a clone pair. 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 experienced a fault fix as a consequence of the change.*
We chose the inconsistent (INC_p) evolutionary pattern followed by a diverging change (INC_c) as our control group because we expect this combination of pattern and operation to be the riskiest. Clones that experience these operations cannot be tracked with a clone detection tool, hence, developers can easily fail to propagate changes to clone fragments. The combination of INC_p and INC_c is therefore a good reference upon which we can compare the odds of faults occurring in the other combinations of genealogies and change operations.

Results We now discuss the results of the aforementioned three tests. Each of the following subsections summarizes the results for one of the three tests. For each evolutionary pattern, change, and combination of genealogy and change,

Table 4: Statistical Analyses for Clone Evolutionary Patterns

Pattern	Test	NiCad				iClones			
		ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
$SYNC_p$	OR	1	1	1	1	1	1	1	1
INC_p	OR	4.79	2.35	41.57	2.64	1.84	1.65	23.31	1.67
	p-value	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	0.55
DIV_p	OR	20.39	2.96	27.39	3.60	3.62	2.65	15.65	2.14
	p-value	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	0.17
LP_p	OR	0.66	6.25	26.80	5.12	1.08	5.77	30.67	10.80
	p-value	0.36	**<0.01**	0.33	0.46	1	**<0.01**	**<0.01**	**0.02**
$LPDIV_p$	OR	27.94	5.70	80.40	10.25	5.98	3.20	8.52	4.32
	p-value	**<0.01**	**<0.01**	**<0.01**	**0.03**	**<0.01**	**<0.01**	0.23	0.13

Table 5: Contingency Tables for Clone Pair Changes

Change	Faulty	NiCad				iClones			
		ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
CON_c	Yes	8,263	1,080	43	317	4,942	763	43	164
	No	17,164	9,627	584	339	2,902	6,495	450	139
DIV_c	Yes	2,922	950	2,383	206	3,979	1,009	3,240	181
	No	3,188	2,303	4,334	191	2,384	4,525	3,106	213
INC_c	Yes	4,300	5,874	6,955	310	8,828	12,456	6,326	302
	No	2,833	7,758	20,921	360	3,994	18,816	14,613	321
$RESYNC_c$	Yes	115	20	5	7	309	19	4	13
	No	595	129	6	5	186	146	18	10

Table 6: Statistical Analyses for Clone Pair Changes

Change	Test	NiCad				iClones			
		ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
CON_c	OR	1	1	1	1	1	1	1	1
DIV_c	OR	1.90	3.68	7.47	0.76	0.98	1.90	10.92	0.72
	p-value	**<0.01**	**<0.01**	**<0.01**	0.07	1	**<0.01**	**<0.01**	0.10
INC_c	OR	3.15	6.75	4.52	0.92	1.30	5.64	4.53	0.80
	p-value	**<0.01**	**<0.01**	**<0.01**	1	**<0.01**	**<0.01**	**<0.01**	0.32
$RESYNC_c$	OR	0.40	1.38	11.32	1.50	0.98	1.11	2.33	1.10
	p-value	**<0.01**	0.52	**<0.01**	1	1	1	0.39	1

we provide the number of faulty and clean occurrences in Table 3, Table 5, and Table 7.

Faults vs. Clone Evolutionary Patterns: Table 4 summarizes the results of the odds ratio and Fisher's exact test. For each clone evolutionary pattern we show the obtained odds ratios and p-values. If an adjusted p-value of the Fisher's exact test is less than 0.05, it is marked in bold.

For all studied system, when the p-value is less than 0.05 (*i.e.*, the difference is statistically significant), the OR values of INC_p, DIV_p, LP_p, and $LPDIV_p$ are greater than 1; meaning that the risk for faults is higher when clones follow other patterns in comparison to the $SYNC_p$ pattern. In JEdit and Maven, we could not find enough occurrences of LP_p or $LPDIV_p$, which may lead to some insignificant p-values. Since all the obtained OR values are $\neq 1$, we reject H_{01}.

Table 7: Contingency Tables for Evolutionary Patterns and Changes

Pattern	Change	Faulty	NiCad				iClones			
			ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
INC_p	INC_c	Yes	2,002	1,343	6,491	219	4,471	7,976	6,051	189
		No	1,585	1,926	19,638	231	2,137	12,297	13,926	210
INC_p	$RESYNC_c$	Yes	44	7	4	4	93	15	3	8
		No	400	54	4	5	85	88	12	2
$SYNC_p$	DIV_c	Yes	1,071	642	239	66	1,701	430	180	56
		No	825	1,373	133	75	635	1,535	101	55
$SYNC_p$	CON_c	Yes	4,292	628	13	181	2,434	381	12	70
		No	7,217	5,657	100	159	991	3,138	72	46
DIV_p	INC_c	Yes	2,137	4,261	459	84	3,899	4,210	272	99
		No	1,138	5,409	1,259	122	1,621	6,241	667	105
DIV_p	$RESYNC_c$	Yes	66	13	1	3	199	4	1	2
		No	192	72	1	0	91	54	4	7
$LPDIV_p$	INC_c	Yes	166	270	5	7	475	270	3	17
		No	110	425	24	7	243	278	21	6
$LPDIV_p$	$RESYNC_c$	Yes	21	1	0	0	32	0	1	7
		No	13	6	1	0	23	12	2	2
LP_p	DIV_c	Yes	91	29	1	2	242	36	2	7
		No	28	59	7	5	90	57	9	4
LP_p	CON_c	Yes	669	12	0	3	320	21	2	1
		No	261	96	3	7	110	100	7	1

Table 8: Statistical Analyses for Evolutionary Patterns and Changes

Pattern	Change	Test	NiCad				iClones			
			ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
INC_p	INC_c	O-R	1	1	1	1	1	1	1	1
INC_p	$RESYNC_c$	O-R	0.09	0.19	3.03	0.84	0.52	0.26	0.58	4.44
		p-value	**<0.01**	**<0.01**	1	1	**<0.01**	**<0.01**	1	0.49
$SYNC_p$	DIV_c	O-R	1.03	0.67	5.44	0.93	1.28	0.43	4.10	1.13
		p-value	1	**<0.01**	**<0.01**	1	**<0.01**	**<0.01**	**<0.01**	1
$SYNC_p$	CON_c	O-R	0.44	0.16	0.39	1.20	1.17	0.19	0.38	1.69
		p-value	**<0.01**	**<0.01**	**<0.01**	1	**<0.01**	**<0.01**	**0.01**	0.14
DIV_p	INC_c	O-R	1.49	1.13	1.10	0.73	1.15	1.04	0.94	1.05
		p-value	**<0.01**	**0.03**	0.76	0.57	**<0.01**	1	1	1
DIV_p	$RESYNC_c$	O-R	0.27	0.26	3.03	inf	1.05	0.11	0.58	0.32
		p-value	**<0.01**	**<0.01**	1	1	1	**<0.01**	1	1
$LPDIV_p$	INC_c	O-R	1.19	0.91	0.63	1.05	0.93	1.50	0.33	3.15
		p-value	1	1	1	1	1	**<0.01**	0.66	0.15
$LPDIV_p$	$RESYNC_c$	O-R	1.28	0.24	0	na	0.67	0	1.15	3.89
		p-value	1	1	1	1	1	**0.04**	1	0.85
LP_p	DIV_c	O-R	2.57	0.70	0.43	0.42	1.29	0.97	0.51	1.94
		p-value	**<0.01**	1	1	1	0.42	1	1	1
LP_p	CON_c	O-R	2.03	0.18	0	0.45	1.39	0.32	0.66	1.11
		p-value	**<0.01**	**<0.01**	1	1	**0.03**	**<0.01**	1	1

Clone pairs exhibiting inconsistent, divergent, late propagation, and late propagation with diversion patterns are more likely to experience a fault than clone pairs that are maintained consistently (i.e., all the changes are propagated on time consistently) throughout their evolution history.

Faults vs. Changes: The results of the odds ratio and Fisher's exact test are summarized in Table 6. For each type of change, we show the obtained odds ratios and p-values.

For all studied system (with the exception of $RESYNC_c$ detected by NiCad for ArgoUML), when the p-value is less than 0.05 (*i.e.*, the difference is statistically significant), the OR values of DIV_c, INC_c, and $RESYNC_c$ are greater than 1; meaning that all of the changes are more fault-prone than consistent changes (CON_c). These results are expected, because clone pairs experiencing inconsistent changes are difficult to monitor using clone detection tools and are more likely to cause bugs. For Maven, none of the results is statistically significant (all adjusted p-values are > 0.05). Hence, we cannot reject H_{02}. We explain this outcome by the low number of DIV_c, INC_c, $RESYNC_c$ changes performed in Maven, compared to other systems.

We conclude that in general, any other changes are more fault-prone than CON_c changes. Developers should be careful when breaking a clone-relationship between a pair of code fragments.

Faults vs. Evolutionary Patterns and Changes: The results of the odds ratio and Fisher's exact test are summarized in Table 8. For each combination of clone evolutionary pattern and type of change, we show the obtained odds ratios and p-values.

Using the NiCad clone detection tool, we obtained the following results:
In ArgoUML, Ant, and JEdit, a consistent change on a clone pair that follows the $SYNC_p$ pattern is less likely to introduce a fault than an inconsistent change on a clone pair that follows the INC_p pattern. This result is statistically significant (adjusted p-value < 0.01).
In ArgoUML and Ant, a re-synchronizing change on a clone pair that follows the INC_p pattern or follows the DIV_p pattern is less likely to introduce a fault than an inconsistent change on a clone pair that follows the INC_p pattern. This result is statistically significant (adjusted p-value < 0.01).
In Ant, an inconsistent change on a clone pair that follows the DIV_p pattern is more likely to introduce a fault than an inconsistent change on a clone pair that follows the INC_p pattern. This result is statistically significant (adjusted p-value < 0.01).
In ArgoUML, an inconsistent change on a clone pair that follows the DIV_p pattern, a re-synchronizing change that follows the late propagation patten, as well as a consistent change that follows the late propagation pattern are more likely to introduce a fault than an inconsistent change on a clone pair that follows the INC_p pattern. This result is statistically significant (adjusted p-value < 0.01).

Using the iClones clone detection tool, we obtained the following results:
In Ant and JEdit, a consistent change on a clone pair that follows the $SYNC_p$ pattern is less likely to introduce a fault than an inconsistent change on a clone pair that follows the INC_p pattern. This result is statistically significant (adjusted p-value < 0.01).
In ArgoUML and Ant, a re-synchronizing change on a clone pair that follows the INC_p pattern is less likely to introduce a fault than an inconsistent change on a clone pair that follows the INC_p pattern. This result is statistically significant (adjusted p-value < 0.01).
In ArgoUML and JEdit, a diverging change on a clone pair that follows the $SYNC_p$ pattern is more likely to introduce a fault than an inconsistent change on a clone pair that follows the INC_p pattern. This result is statistically significant (adjusted p-value < 0.01).

In ArgoUML, a consistent change following the $SYNC_p$ pattern, an inconsistent change following the DIV_p, as well as a consistent change following the LP_p pattern are more likely to introduce a fault than an inconsistent change on a clone pair that follows the INC_p pattern. This result is statistically significant (adjusted p-value < 0.05).
In Ant, an inconsistent change on a clone pair that follows the $LPDIV_p$ pattern is more likely to introduce a fault than an inconsistent change on a clone pair that follows the INC_p pattern. In addition, a diverging change following the $SYNC_p$, a re-synchronizing change following the DIV_p or the $LPDIV_p$, as well as a consistent change following the LP_p pattern are less likely to introduce a fault than an inconsistent change following the INC_p pattern. This result is statistically significant (adjusted p-value < 0.05).
In the case of Maven, there is no statistically significant result. Hence, we cannot reject H_{03}.

Overall, these results suggest that developers should be careful when performing an inconsistent change on a clone pair that experienced a DIV_p pattern. Also, a diverging change on a clone pair that consistently followed the $SYNC_p$ pattern in the past can be fault-prone.

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

Motivation In this question we examine the effect of two metrics on fault proneness: the time interval since the last change and the size of the clone. We examine the time interval because 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. It is also expected that a smaller clone will be less prone to faults, as it is less complex and may require less effort to modify. Using our set of clone pair genealogies, we examine whether the time interval between changes or the size of the clone relates to faults. 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, we can determine whether fault-proneness is affected by either of these two metrics.

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. We performed this discretization because the Fisher test requires categorical variables. A change is flagged if it is fault-inducing. Using “One Day” as the control group, we calculate the odds ratios between the control group and each of the other time period and perform the Fisher's exact test. We test the following null hypothesis H_{04}: *The time interval between modifications to a clone pair has no relationship with faults.*

Table 9: Contingency Tables for Evolutionary Patterns Considering the Time Interval between Changes

Interval	Faulty	NiCad				iClones			
		ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
One day	Yes	1,140	700	4,900	93	1,559	1,244	5,240	91
	No	1,177	6,481	9,785	140	825	8,192	5,659	136
One week	Yes	1,353	975	679	118	2,054	1,978	412	118
	No	1,689	3,258	2,474	119	1,292	5,497	1,426	151
One month	Yes	1,887	1,233	564	131	3,148	2,396	378	121
	No	2,423	4,357	3,143	158	1,675	7,265	1,734	126
One year	Yes	9,665	3,698	1,921	314	9,208	6,492	2,270	206
	No	15,057	3,377	8,208	343	4,385	5,764	7,070	210
More than one year	Yes	1,555	1,318	1,322	184	2,089	2,137	1,313	124
	No	3,434	2,344	2,235	235	1,289	3,264	2,298	60

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 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 or equal to the median lines of clone of a studied system detected by the tool, or "small" if it is smaller than the median lines of clone of the system detected by the tool. For each state, we use the SZZ algorithm to determine whether it is faulty or not. We calculated the odds ratios and the p-value of the Fisher's exact test, and test the following null hypothesis H_{05}: *The size of the clone has no relationship with faults*. When calculating the odds ratio, we select the synchronous evolutionary pattern with a small clone size as our control group. Since a large size is known to be correlated with a high risk of fault, we expect the synchronous evolutionary pattern with a small clone size to be less fault-prone than the other patterns, hence our choice of this pattern as our control group.

To better understand the correlational relationship between bug-proneness and time interval (respectively clone size), we build a linear regression model for each studied system. The linear regression models have the following form:

$$Faulty = \alpha Interval + \beta Size + \gamma \tag{1}$$

We leverage `R` to create the GLM models, in which, time interval and clone size are independent variables, and whether a clone is faulty is the dependent variable. We will compare the explanatory power of time interval and clone size with other metrics in RQ3.

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. For each time interval and evolutionary pattern considering cloned code size, we provide the number of faulty and clean occurrences in Table 9 and Table 11.

Table 10: Statistical Analyses for Evolutionary Patterns Considering the Time Interval between Changes

Interval	Test	NiCad				iClones			
		ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
One day	O-R	1	1	1	1	1	1	1	1
One week	O-R	0.83	2.77	0.55	1.49	0.84	2.37	0.31	1.17
	p-value	**<0.01**	**<0.01**	**<0.01**	0.13	**<0.01**	**<0.01**	**<0.01**	1
One month	O-R	0.80	2.62	0.36	1.25	0.99	2.17	0.24	1.44
	p-value	**<0.01**	**<0.01**	**<0.01**	0.99	1	**<0.01**	**<0.01**	0.21
One year	O-R	0.66	10.14	0.47	1.38	1.11	7.42	0.35	1.47
	p-value	**<0.01**	**<0.01**	**<0.01**	0.16	0.10	**<0.01**	**<0.01**	0.10
More than one year	O-R	0.47	5.21	1.18	1.18	0.86	4.31	0.62	3.09
	p-value	**<0.01**	**<0.01**	**<0.01**	1	**0.02**	**<0.01**	**<0.01**	**<0.01**

Table 11: Contingency Tables for Evolutionary Patterns Considering the Cloned Code Sizes

	Faulty	NiCad				iClones			
		ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
$SYNC_p$ Small	Yes	40	82	2	3	428	107	1	5
	No	4,582	955	37	65	391	711	41	23
DIV_p Small	Yes	86	174	95	11	852	281	14	11
	No	152	606	28	40	213	659	69	38
INC_p Small	Yes	53	114	1,630	35	1,033	309	770	39
	No	1,369	582	1,296	120	545	1,371	1,994	95
$LPDIV_p$ Small	Yes	11	13	3	2	171	13	1	4
	No	4	11	0	0	15	32	3	4
LP_p Small	Yes	0	7	0	0	31	19	2	1
	No	209	23	0	0	20	25	0	2
$SYNC_p$ Big	Yes	474	109	3	16	281	71	2	10
	No	5,470	1,149	97	81	525	756	74	31
DIV_p Big	Yes	750	235	92	33	656	183	66	27
	No	650	915	155	54	325	784	127	26
INC_p Big	Yes	664	77	2,218	52	1,021	255	1,513	43
	No	1,560	314	1,185	33	894	1,438	1,761	82
$LPDIV_p$ Big	Yes	59	17	3	2	102	13	1	2
	No	45	47	2	3	44	35	6	1
LP_p Big	Yes	19	14	1	2	31	9	2	5
	No	355	14	1	3	54	15	5	0

Faults and Time Interval Between Changes: Table 10 summarizes the results of the Odds ratio and Fisher tests. We obtained the following result with NiCad: For ArgoUML, changes occurring after one week are always less fault-prone than changes performed within a day. For Ant, on the contrary, any changes occurring after one week are always more fault-prone than changes performed within a day. For JEdit, changes occurring after one week and less than one year are less fault-prone than changes performed within a day; while changes occurring after a year become more fault-prone than changes performed within a day. Regarding Maven, none of the results is statistically significant.

But when we look at the results obtained with iClones, we see that for ArgoUML, changes occurring after one week but within one month, as well as changes occurring after more than one year are less fault-prone than changes performed within a day. For Ant, we obtained the same results as those of

Table 12: Statistical Analyses for Evolutionary Patterns Considering the Cloned Code Sizes

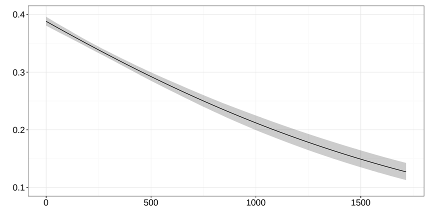
	Test	NiCad				iClones			
		ArgoUML	Ant	JEdit	Maven	ArgoUML	Ant	JEdit	Maven
$SYNC_p$ Small	O-R	1	1	1	1	1	1	1	1
DIV_p Small	O-R	64.81	3.34	62.77	5.96	3.65	2.83	8.32	1.33
	p-value	**<0.01**	**<0.01**	**<0.01**	0.07	**<0.01**	**<0.01**	0.17	1
INC_p Small	O-R	4.43	2.28	23.27	6.32	1.73	1.50	15.83	1.89
	p-value	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	1
$LPDIV_p$ Small	O-R	315.01	13.76	inf	inf	10.41	2.70	13.76	4.60
	p-value	**<0.01**	**<0.01**	**<0.01**	**0.04**	**<0.01**	0.06	1	0.78
LP_p Small	O-R	0	3.54	*na*	*na*	1.42	5.05	inf	2.30
	p-value	1	0.08	1	1	1	**<0.01**	**0.03**	1
$SYNC_p$ Big	O-R	9.93	1.10	0.57	4.28	0.49	0.62	1.11	1.48
	p-value	**<0.01**	1	1	0.21	**<0.01**	**0.04**	1	1
DIV_p Big	O-R	132.17	2.99	10.98	13.24	1.84	1.55	21.31	4.78
	p-value	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**0.04**
INC_p Big	O-R	48.76	2.86	34.63	8.47	1.04	1.18	35.23	2.41
	p-value	**<0.01**	**<0.01**	**<0.01**	**<0.01**	1	1	**<0.01**	1
$LPDIV_p$ Big	O-R	150.19	4.21	27.75	14.44	2.12	2.47	6.83	9.20
	p-value	**<0.01**	**<0.01**	0.06	0.31	**<0.01**	0.14	1	1
LP_p Big	O-R	6.13	11.65	18.50	14.44	0.52	3.99	16.40	inf
	p-value	**<0.01**	**<0.01**	1	0.31	0.06	**0.03**	0.45	**<0.01**

Table 13: Coefficients and p-values of the linear regression model on the relationship between fault-proneness and time interval of changes as well as cloned code size.

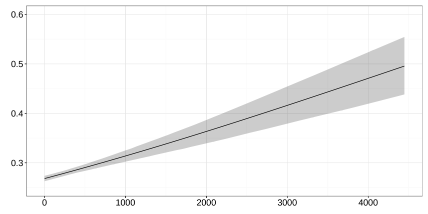
	System	Interval	p-value	Size	p-value
NiCad	ArgoUML	-0.0009	<0.0001	0.022	<0.0001
	Ant	0.0002	<0.0001	0.0026	<0.0001
	JEdit	0.0004	<0.0001	0.0026	<0.0001
	Maven	-0.0001	0.43	-0.0018	0.26
iClones	ArgoUML	-0.0004	<0.0001	-0.010	<0.0001
	Ant	0.0002	<0.0001	0.0024	<0.0001
	JEdit	-0.0002	<0.0001	-0.049	<0.0001
	Maven	0.0009	<0.0001	-0.0051	0.047

NiCad. For JEdit, changes occurring after one week are always less fault-prone than changes performed within a day. For Maven, changes occurring after one year are more fault-prone than changes performed within a day.

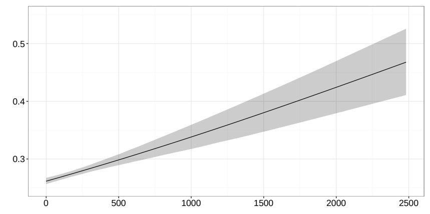
Table 13 shows the coefficients and p-values in the linear regression model that investigates how time interval of changes and size impact the fault-proneness. Figures 7 and 8 depict the trends of the fault-proneness probability changes, with respect to the time interval. Based on the results of both clone detection tools, the probability of fault-proneness increases with the increase of the time interval for Ant; while the probability decreases with the increase of the time interval for ArgoUML. For JEdit and Maven, the trends diverge depending on different clone detection tools. Some of the results seem inconsistent with the those from Table 10 because time interval and size can interfere with each other in the regression model. In the future, we plan to build non-linear regression models [43] to explore whether the models contain



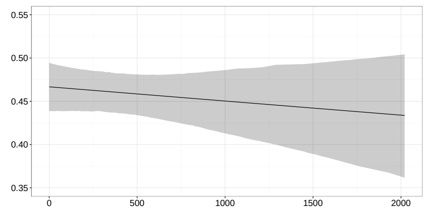
(a) ArgoUML



(b) Ant

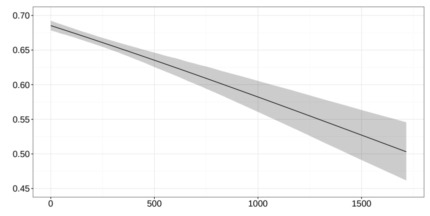


(c) JEdit

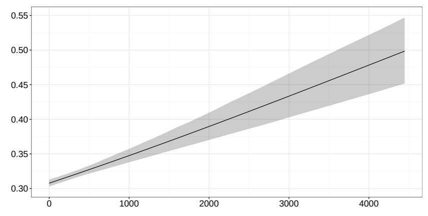


(d) Maven

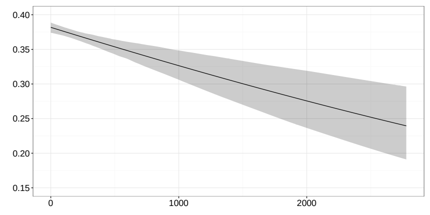
Fig. 7: Estimated fault-proneness probability for various time interval sizes (in days) based on NiCad's detection



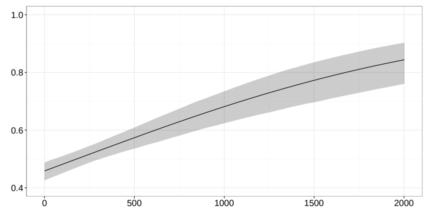
(a) ArgoUML



(b) Ant



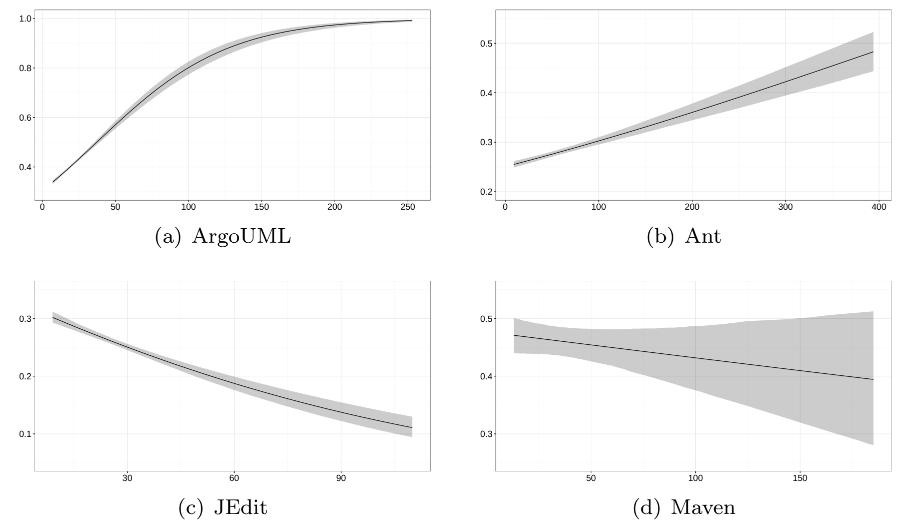
(c) JEdit



(d) Maven

Fig. 8: Estimated fault-proneness probability for various time interval sizes (in days) based on iClones' detection

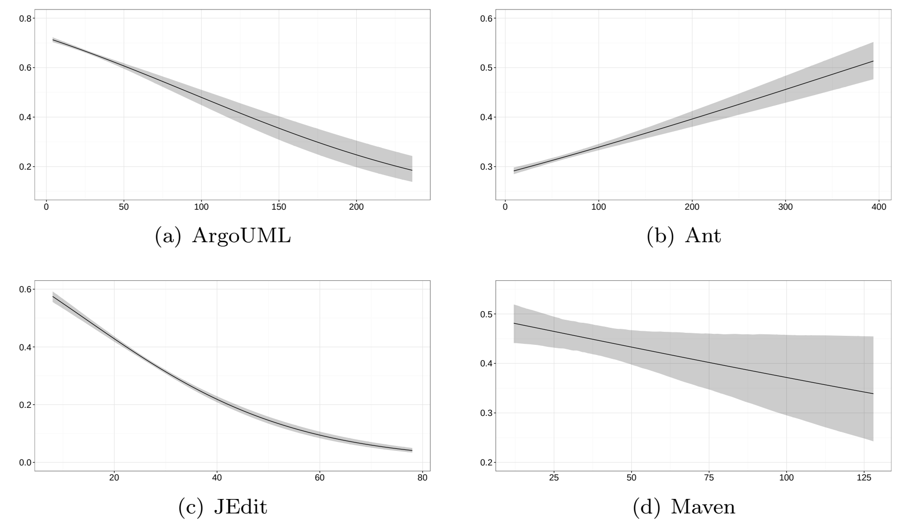
any knot that changes the direction of the trends. This kind of model can better reflect trends obtained for JEdit results based on NiCad's detection in Table 10. In summary, the results are system dependent, so we cannot reject H_{04} in general.



(a) ArgoUML (b) Ant

(c) JEdit (d) Maven

Fig. 9: Estimated fault-proneness probability for various clone sizes based on NiCad's detection



(a) ArgoUML (b) Ant

(c) JEdit (d) Maven

Fig. 10: Estimated fault-proneness probability for various clone sizes based on iClones' detection

Faults and Size of Clone: The odds ratios of the evolutionary patterns classified by the size of the clone are summarized in Table 12.

Consistent with our findings from RQ1, most patterns involved with inconsistent change are more fault-prone than $SYNC_p$ small, except $SYNC_p$ big for ArgoUML and Ant based on iClones' detection. This exception may be due to the relatively small time interval observed between the big changes, which can also explain the trend of ArgoUML in Figure 8. Once again, the time interval and size factors may interfere with each other. From the results of Table 12, we can reject H_{05}.

Table 13 shows the coefficients and p-values in the linear regression model that investigates how cloned code sizes affect fault-proneness. Figures 9 and 10 depict the trends of the fault-proneness probability changes with respect to the size of the clones. Based on the results of both clone detection tools, the probability of fault-proneness increases with the increase of the cloned size for Ant; while the probability decreases with the increase of the cloned size for JEdit. For ArgoUML and Maven, the trends diverge depending on different clone detection tools.

Overall, we conclude that in general, the time interval between changes to a clone pair does not seem to affect the fault-proneness of the clone pair; while bigger clone size tend to increase the probability of fault-proneness. We did not observed a uniform trend of changes in fault-proneness among the systems, as the time interval or cloned code sizes changed.

5.3 RQ3: Can we predict faults in software clones using clone genealogy information?

Motivation Tracking the genealogy of all clone pairs in an entire system is resource intensive. When building prediction models to identify faulty code clones, developers face a tradeoff between relying on only traditional fault prediction metrics or collecting additional genealogy metrics which provide richer information on the history of a clone pair. Knowing the gain achieved by adding genealogy metrics to fault explanatory models is important to help developers decide whether the added effort justifies the results.

Approach In this question, we propose metrics to capture the genealogy information of a clone pair. We combine these metrics with traditional product and process metrics and investigate their statistical relationships with future faults in cloned code. Table 14 presents the description of all the metrics used in this study. The metrics are divided into three categories: product metrics, process metrics and genealogy metrics. Product metrics 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. Process metrics are

Table 14: Clone Pair Metrics

Metrics	**Description**
	Product metrics
$CLOC$	The number of cloned lines of code.
$CFltFix$	The current commit 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).
$CommitterExp$	The experience of a committer (*i.e.*, the number previous commits submitted before a specific commit).
	Process metrics
$EFltDens$	The number of fault fix modifications to the clone pair since it was created divided by the total number of commits that modified the clone pair.
$TChurn$	The sum of the added and changed lines of code in the history of a clone.
TPC	The total number of changes in the history of a clone.
$NumOfBursts$	The number of change bursts on a clone. A change burst is a sequence of consecutive changes with a maximum distance of one day between the changes.
$SLBurst$	The number of consecutive changes in the last change burst on a clone.
$CFltRate$	The number of fault-prone modifications to the clone pair divided by the total number of commits that modified the clone pair.
	Genealogy 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.
$EConStChg$	The number of consistent change of state within the clone pair genealogy.
$EIncStChg$	The number of inconsistent change 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 in days since the previous change to the clone pair.

collected using the history of changes on clone pairs. For example, "TPC" measures the total number of changes in the history of a clone. Genealogy metrics capture state changes in the history of clone pairs. For example, "EConStChg" measures the number of consistent changes of states within a clone pair genealogy.

For each state in a clone genealogy instance, we collect all the metrics from Table 14. Since each clone in the clone pair will have its own set of metrics (*e.g.*, $MLOC$), we compute the maximum value of each metric across the two clones. To reduce the skewness observed on metric values, we apply a standard log transformation to each metric. From the measurements obtained, we create linear regression models that set the number of reported faults in relation to our three groups of metrics. The linear regression models have the following form:

$$Faults = \sum_i \alpha_i ProductM_i + \sum_j \beta_j ProcessM_j$$

$$+ \sum_{k} \gamma_k GenealogyM_k + \delta \qquad (2)$$

With this model, we investigate the statistical relationships between product, process and genealogy metrics, which are represented by the regression variables ($ProductM_i$, $ProcessM_j$, and $GenealogyM_k$), and the number of reported faults, represented by the dependent variable of the model ($Faults$). We follow the same methodology as in the work of Cataldo *et al.* [44]. First, we compute the variance inflation factors (VIF) [45] of each metric to examine multi-collinearity between the variables of our regression model. Next, we construct *Generalized Linear Models* to investigate the relative impact of each of our three groups of metrics on future faults. We remove from the models all variables with VIF > 5, as recommended by [46].

We create the models following a hierarchical modelling approach: we start out with a baseline model that contains only product metrics as regression variables. We then build subsequent models by adding step by step, respectively, process metrics and clone genealogy metrics. We chose to follow a hierarchical modelling approach because contrary to a step-wise modelling approach, the hierarchical approach has the advantage of minimizing the artificial inflation of errors and therefore the overfitting [44].

We report for each statistical model the explanatory power, deviance, of the model and the percentage of deviance explained. The deviance of a model M is defined as $D(M) = -2.LL(M)$, where $LL(M)$ is the log-likelihood of the model M. The deviance explained is the ratio between $D(Faults \sim Intercept)$ and $D(M)$. For each subsequent model M_{S+E} derived from a model M_S, we also test the statistical significance of the difference between M_{S+E} and M_S. For each explantory metric, we report its corresponding p-values. We use the `varImp` package in R to calculate the importance of the metrics, and report the top 3 metrics, which have the strongest explanatory power.

Results In this subsection we describe the results for RQ3. Table 15 and Table 16 presents the results of our hierarchical analysis. In these tables, M_S represents a model built using product metrics only (*i.e.*, the basic model). M_{S+E} is a model built using product and process metrics, while M_{S+E+G} is a model containing product, process and genealogy metrics. The results of Table 15 and Table 16 show that genealogy metrics only slightly contribute to the explanatory power of the fault-proneness models. The biggest improvement is obtained on Maven (*i.e.*, 2.1%) thanks to the EchgTimeInt metric.

On average, the explanatory power of a fault prediction model built using both product and process metrics (*i.e.*, M_{S+E}) is increased by 4.3% when genealogy information is added to the model. This increase is statistically significant. The increase is the highest for Ant when the iClones detection tool is used (*i.e.*, 8.5%).

Table 15: Hierarchical Analysis of Linear Regression Models for NiCad

Metrics	ArgoUML			Ant			JEdit			Maven		
	M_S	M_{S+E}	M_{S+E+G}	M_S	M_{S+E}	M_{S+E+G}	M_S	M_{S+E}	M_{S+E+G}	M_S	M_{S+E}	M_{S+E+G}
CLOC	0.81***	0.35***	0.38***	0.09***	0.09***	0.13***	-0.13***	-0.02***	-0.07*	-0.03	-0.08	-0.06
CFltFix	-0.07	-0.99***	-1.03***	-0.56***	-0.50***	-0.68***	-0.74*	-3.70***	-3.56***	0.34*	-0.27	-0.26
CPathDepth	0.02	0.059*	0.06*	-0.15***	-0.21***	-0.23***	-1.47***	-1.40***	-1.40***	0.07	0.07	0.08
CCurSt	0.66***	0.30***	0.77***	1.77***	1.44***	1.47***	1.48***	0.71***	1.37***	-0.12	-0.15	-0.41
Experience	-0.17***	-0.17***	-0.17***	-0.12***	-0.13***	-0.13***	0.64***	7.38***	0.70***	0.06	0.06	0.07*
EfltDens		1.92***	2.11***		-3.82***	-3.14***		7.38***	7.18***		1.48***	1.43***
TChurn		0.21***	0.20***		-0.03*	-0.02		0.34***	0.36***		0.11	0.11*
TPC		0.08***	–		0.03	–		-0.09***	–		-0.24*	–
NumOfBursts		–	–		–	–		–	-0.05**		–	–
SLBurst		-0.11*	-0.10*		-0.62***	-0.59***		0.08	-0.12**		0.12	0.18
CFltRate		1.79***	1.77***		2.14***	2.04***		-0.83***	-0.75***		0.58**	0.59**
EEvPattern			–			–			–			–
EConChg			–			0.23***			–			–
EIncChg			–			–			–			–
EConStChg			0.26***			–			–			-0.36*
EIncStChg			-0.08**			-0.13***			–			-0.14
EFltsConStChg			1.15***			-0.21			-0.28			-0.31
EFltIncStChg			-0.58***			0.07			0.20**			0.22
EChgTimeInt			-0.006			0.13***			-0.05***			0.03
Deviance	48544	46272	46057	29646	28305	27730	36894	35773	35736	2513	2486	2482
Dev. Explainained	8.2%	12.5%	12.9%	10.7%	14.7%	16.2%	9.7%	12.4%	12.5%	0.7%	1.8%	1.9%
Delta deviance		2272	215		1341	485		1121	37		27	4
Top variables			CFltRate Experience CLOC			CFltRate CCurSt EchgTimeInt						EFltDens CFltRate EconStChg

* p-value < 0.05, ** p-value < 0.01, *** p-value < 0.001; otherwise, p-value ≥ 0.05
– metric removed by the VIF analysis

Table 16: Hierarchical Analysis of Linear Regression Models for iClones

Metrics	ArgoUML			Ant			JEdit			Maven		
	M_S	M_{S+E}	M_{S+E+G}	M_S	M_{S+E}	M_{S+E+G}	M_S	M_{S+E}	M_{S+E+G}	M_S	M_{S+E}	M_{S+E+G}
CLOC	-0.32***	-0.43***	-0.43***	-0.005	-0.10***	-0.04***	0.06	0.08	0.14*	-0.32**	-0.37**	-0.23
CFltFix	-1.10***	-1.13***	-1.1***	-0.43***	-0.62***	-0.76***	0.18	-0.74*	-0.70*	-0.56**	-0.97***	-0.91**
CPathDepth	0.36***	0.19***	0.20***	-0.59***	-0.52***	-0.50***	-1.96***	-2.49***	-2.47***	0.09	0.07	0.07
CCurSt	0.21***	-0.18***	-0.15**	1.29***	0.72***	0.45***	1.86***	2.16***	–	-0.29*	-0.36**	–
Experience	-0.004	-0.007	-0.008	-0.14***	-0.17***	-0.17***	0.40***	0.37***	0.31***	-0.04	-0.04	0.02
EfltDens		-0.03	-0.04		-4.32***	-4.19***		4.95***	5.09***		0.85	0.66
TChurn		0.42***	0.42***		0.12***	0.12***		0.10***	0.08**		0.08	0.08
TPC		0.01	–		0.42***	–		–	–		-0.14	-0.50***
NumOfBursts		–	–		-0.44***	0.28***		-0.04	-0.08**		–	–
SLBurst		-0.20***	-0.22***		-0.25***	-0.22***		0.58***	0.02		0.10	0.62**
CFltRate		1.70***	1.71***		2.70***	2.60***		-1.89***	-1.51***		0.86***	0.86***
EEvPattern			–			–			–			–
EConChg			–			–			–			–
EIncChg			–			–			–			–
EConStChg			0.04			–			-2.46***			0.24
EIncStChg			-0.003			0.24***			–			–
EFltsConStChg			-0.07			-0.19			-1.57**			1.03**
EFltIncStChg			-0.04			0.27***			2.61***			-0.38
EChgTimeInt			-0.02*			0.10***			-0.17***			0.19***
Deviance	34341	32585	32576	52139	47706	47428	30760	29891	39350	1836	1816	1776
Dev. Explainained	3.1%	8.0%	8.1%	6.2%	14.2%	14.7%	14.2%	16.6%	18.1%	1.4%	2.5%	4.6%
Delta deviance		1756	9		4433	278		869	541		20	40
Top variables			TChurn CFltRate CFltFix			CFltRate CpathDepth experience			CPathDepth EchgTimeInt Experience			EchgTimeInt CFltRate TPC

* p-value < 0.05, ** p-value < 0.01, *** p-value < 0.001; otherwise, p-value ≥ 0.05
– metric removed by the VIF analysis

Results from Table 15 and Table 16 suggest that CFltRate would a be a good predictor; meaning that the number of previous fault-prone modifications can help predict future faults. This result is consistent with the findings of our previous work [47]. We recommend that practitioners use this metric in combination with traditional product and process metrics when predicting faults in software clones.

6 Discussion

Our identification of clone genealogies are based on a line mapping algorithm, which may not be 100% accurate. To examine the accuracy of our results, we manually examined 50 commits that generated more than 10 new clone genealogies. We found that all of these commits involved with a large amount of new classes or reconstructions. Examining these genealogies helped us to better understand why there are a large number of clone genealogies detected in some systems, such as ArgoUML, and helped us validate our clone detection scripts. For example, in ArgoUML, based on iClones' detection, 230 different clone genealogies started from the commit `559aca3` (SVN revision `122992`), because there are 1,818 new files created in this commit. Another example is, in Ant, based on NiCad's detection, 3,257 different clone genealogies start from the commit `d1064de`, because there are 1,903 new Java classes created in this commit. In our manual validation, we also examined whether a clone genealogy was introduced from the first commit in the genealogy, and whether it disappeared after the last commit in the genealogy. For example, the `zipFile` method was introduced in respectively two classes (`proposal/myrmidon/src/main/org/apache/tools/ant/taskdefs/Ear.java` and `proposal/sandbox/antlib/src/main/org/apache/tools/ant/taskdefs/Antjar.java`) in Ant commit `d1064de`. The two methods were very similar at the beginning. They experienced three consistent changes (`b8c5034`, `7c0bc50C`, and `669a7ea`). However, at the commit `0a07be8`, the second file changed the algorithm of the method `zipFile`, *i.e.*, the clone pair became inconsistent. Finally, the second file was removed from the system at the commit `99cdb67`. Another example is, in ArgoUML, the `buildConnection` method which was introduced at commit `a6a72d7` (SVN revision `11634`) in respectively two new files `src/model-euml/src/org/argouml/model/euml/UmlFactoryEUMLImpl.java` and `src/model-mdr/src/org/argouml/model/mdr/UmlFactoryMDRImpl.java`. The two methods were identical at the beginning. They experienced consistent changes at commits `1eb1d05`(revision `11993`) and `964f121` (revision `11994`). But since the commit `4e6285c` (revision `12105`), the first file changed its exception handling statements. The clone pair became dissimilar until the last studied commit.

We expected that INC pattern would be the most fault-prone However, according to the results, DIV pattern is highly fault-prone, because a fault could be fixed by propagating changes performed on one clone segment to the other segments. Here are two examples that we manually examined in

Ant. In Bug 41353, running tasks in parallel generated a problem. The solution to the fault was to clone the properties in data which was accessed in parallel, which resulted in an inconsistent change on the clone contained in files `src/main/org/apache/tools/ant/PropertyHelper.java @ 472:480` and `proposal/embed/src/java/org/apache/tools/ant/PropertyHelper.java @ 501:513`. As a result of this, the clone pair evolved into a DIV_p genealogy pattern. In the case of Bug 42736, in order to encapsulate the reference to a method inside the delegate object, the clone has created an interface with add method `add(...)` and `getDelegates` and `getDelegateInterfaces` invoked methods to retrieve a collection of delegates of the specified type. This modification resulted into the two clone segments diverging, resulting into an INC_p genealogy pattern.

An interesting phenomenon is the migration of clones across repositories. Among the genealogies that were analyzed, we observe that faults occur more frequently among clones from files located in different directories. And to fix these faults, developers often propagate changes from one clone segment to the other. This was the case for example for Ant's bugs 19897, 22326, and 7552. A closer look at the files involved in these clones reveal that developers duplicated code to experiment on new changes. However instead of doing this in separate branches, they performed it in the main code base and committed their experimentations in the trunk, whenever they were satisfied with their experimentations, the modifications are propagated to the main files of Ant that would be released to the public. We found that this phenomenon explains a large proportion of the fault fix observed on the DIV genealogies.

7 Threats to Validity

In this section we discuss the threats to validity of our study.

Construct validity threats involve the relationship between theory and observation. The source of threats in this study are due to measurement errors experienced by the clone detection tools. To reduce the number of false positive clone detection results, we repeat the study using two clone detection tools that use different clone detection techniques and that have both been used in previous studies and reported to achieve good precision and recall (see [12]).

In this study, we have chosen to analyze clone pairs instead of clone groups since clone pairs within the same clone group are not equally risky. However, all analysis presented in this paper can be replicated on clone groups easily.

The SZZ heuristic used to identify fault-inducing changes is not 100% accurate. However, it has been successfully used in multiple previous studies from the literature, with satisfying results. In our implementation, we remove all fault-inducing commit candidates that only changed blank lines or comment lines.

Threats to *internal validity* do not affect this study, as it is an exploratory study [48]. We cannot claim causation, we simply report observations and correlations, although our discussion tries to explain these observations.

Threats to *conclusion validity* address the relationship between the treatment and the outcome. We are careful to acknowledge the assumptions of each statistical test. We used non-parametric tests that do not require making assumptions about the data set distribution. To exclude test files from our study, we manually examined all files in our data set.

Threats *External validity* address the generalizability of our results. We examine four different sized systems from four different domains. Nevertheless, more studies on more systems should be done to further validate our results. All of our subject systems are written in Java. Our results may not be able to generalized to systems with other programming languages. However, Java, C, and C++ all belong to the "C-family programming languages" [49], *i.e.*, they share some common features in syntax. We believe that our approach can yield similar results on C/C++ systems. In the future, we plan to extend this study on more programming language, such as C and C++. We also welcome software practitioners and researchers to replicate and validate our work on other programming languages.

Threats to *reliability validity* take into account the possibly of replicating our study. In this paper, we provide all the details needed to replicate our study. All our four subject systems are publicly available for study. The data and scripts used in this study is also publicly available and can be downloaded here[1].

8 Conclusion

In this paper, we examine the states within clone genealogies and changes to clone pairs to determine their relationship with 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 clone pairs exhibiting inconsistent and divergent patterns are more likely to experience a fault than clone pairs that are maintained consistently. We also show that the size of the cloned region of a clone pair can impact the fault-proneness of the clone pair. But, there is no clear relationship between the cloned code changed time and the fault-proneness of a clone pair. Next, we investigate the statistical relationships between product, process, genealogy metrics, and the number of future faults in cloned code. Our results show that adding genealogy information to a fault prediction model built using product and process metrics can increase the explanatory power of the model. We found that clone pairs causing faults in the past can help indicate future faults in the clone fragments. In the future, we intend to explore more factors that can be correlated with fault-proneness of code clones, such as the number of different maintainers, and the domain of the system. We also plan to replicate our study on more systems using different clone detection tools. Moreover, we will use the results of our study to build recommendation systems to assist

[1] `https://github.com/swatlab/clone_genealogies`

maintenance teams in the management of software clones. The data used in this study is publicly available and can be found at: `https://github.com/swatlab/clone_genealogies`.

Acknowledgements The authors would like to thank the anonymous reviewers for their detailed feedback and useful suggestions that greatly contributed to improving this paper. This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. School of Computing TR 2007-541, Queen's University **115** (2007)
2. Thummalapenta, S., Cerulo, L., Aversano, L., Di Penta, M.: An empirical study on the maintenance of source code clones. Empirical Software Engineering **15** (2010) 1–34
3. Aversano, L., Cerulo, L., Di Penta, M.: How clones are maintained: An empirical study. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering. (2007) 81 –90
4. Barbour, L., Khomh, F., Zou, Y.: Late propagation in software clones. In: Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM). (2011) 273 –282
5. Barbour, L., Khomh, F., Zou, Y.: An empirical study of faults in late propagation clone genealogies. Journal of Software: Evolution and Process **25** (2013) 1139–1165
6. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. ESEC/FSE-13, New York, NY, USA, ACM (2005) 187–196
7. Rahman, F., Bird, C., Devanbu, P.: Clones: What is that smell? Empirical Software Engineering **17** (2012) 503–530
8. Fowler, M.: Refactoring: improving the design of existing code. Pearson Education India (2009)
9. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society (2009) 485–495
10. Krinke, J.: A study of consistent and inconsistent changes to code clones. Proceedings of the 14th Working Conference on Reverse Engineering **0** (2007) 170–178
11. Göde, N., Harder, J.: Clone stability. In: Proceedings of the 15th European Conference on Software Maintenance and Reengineering. (2011)
12. Svajlenko, J., Roy, C.K.: Evaluating modern clone detection tools. In: Proceedings 30th IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE (2014) 321–330
13. Göde, N., Koschke, R.: Frequency and risks of changes to clones. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE), ACM (2011) 311–320
14. Göde, N., Harder, J.: Oops!... I changed it again. In: Proceedings of the 5th International Workshop on Software Clones, ACM (2011) 14–20
15. Mondal, M., Roy, C.K., Schneider, K.A.: A comparative study on the intensity and harmfulness of late propagation in near-miss code clones. Software Quality Journal (2016) 1–33
16. Xie, S., Khomh, F., Zou, Y.: An empirical study of the fault-proneness of clone mutation and clone migration. In: Proceedings of the 10th Working Conference on Mining Software Repositories. MSR '13, Piscataway, NJ, USA, IEEE Press (2013) 149–158
17. Xie, S., Khomh, F., Zou, Y., Keivanloo, I.: An empirical study on the fault-proneness of clone migration in clone genealogies. In: Proceedings of 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE). (2014) 94–103

18. Khoshgoftaar, T.M., Allen, E.B., Goel, N., Nandi, A., McMullan, J.: Detection of software modules with high debug code churn in a very large legacy system. In: Proceedings of the The Seventh International Symposium on Software Reliability Engineering. ISSRE '96, Washington, DC, USA, IEEE Computer Society (1996) 364–371
19. Bernstein, A., Ekanayake, J., Pinzger, M.: Improving defect prediction using temporal features and non linear models. In: 9th international workshop on Principles of software evolution (IWPSE), NY, USA, ACM (2007) 11–18
20. Graves, T.L., Karr, A.F., Marron, J.S., Siy, H.: Predicting fault incidence using software change history. IEEE Transactions on Software Engineering **26** (2000) 653–661
21. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on Software engineering (ICSE), NY, USA, ACM (2005) 284–292
22. Hassan, A.E.: Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering (ICSE). (2009)
23. El Emam, K., Melo, W., Machado, J.C.: The prediction of faulty classes using object-oriented design metrics. Journal of Systems and Software **56** (2001) 63–75
24. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans Software Eng. **20** (1994) 476–493
25. Briand, L.C., Daly, J.W., Wüst, J.K.: A unified framework for coupling measurement in object-oriented systems. IEEE Trans. Softw. Eng. **25** (1999) 91–121
26. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering (ICSE), New York, NY, USA, ACM (2006) 452–461
27. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: Third International Workshop on Predictor Models in Software Engineering. (2007)
28. Arisholm, E., Briand, L.C.: Predicting fault-prone components in a java legacy system. In: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE), NY, USA, ACM (2006) 8–17
29. Moser, R., Pedrycz, W., Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the International Conference on Software Engineering, New York, NY, USA, ACM (2008) 181–190
30. Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., Godfrey, M.W.: Investigating code review quality: Do people and participation matter? In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, IEEE (2015) 111–120
31. McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E.: An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering (2015) To appear
32. Kapser, C., Godfrey, M.W.: "cloning considered harmful" considered harmful. In: Proceedings of the 13th Working Conference on Reverse Engineering, Washington, DC, USA, IEEE Computer Society (2006) 19–28
33. Wheeler, D.A.: SLOCCount. http://www.dwheeler.com/sloccount/ (2016) Online; Accessed March 31st, 2016.
34. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: ACM sigsoft software engineering notes. Volume 30., ACM (2005) 1–5
35. Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, IEEE (2003) 23–32
36. Roy, C., Cordy, J.: Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on. (2008) 172 –181
37. Gode, N., Koschke, R.: Incremental clone detection. In: Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on, IEEE (2009) 219–228
38. Lakhotia, A., Li, J., Walenstein, A., Yang, Y.: Towards a clone detection benchmark suite and results archive. In: Program Comprehension, 2003. 11th IEEE International Workshop on. (2003) 285–286
39. Corley, C.S.: whatthepatch - Python's third party patch parsing library. https://pypi.python.org/pypi/whatthepatch (2016) Online; Accessed August 29th, 2016.

40. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. Software Engineering, IEEE Transactions on **28** (2002) 654 – 670
41. Sheskin, D.: Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition). Chapman & All (2007)
42. Dmitrienko, A., Molenberghs, G., Chuang-Stein, C., Offen, W.: Analysis of Clinical Trials Using SAS: A Practical Guide. SAS Institute (2005)
43. Harrell, F.E.: Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis. Springer Science & Business Media (2013)
44. Cataldo, M., Mockus, A., Roberts, J.A., Herbsleb, J.D.: Software dependencies, work dependencies, and their impact on failures. IEEE Trans. Softw. Eng. **35** (2009) 864–878
45. Kutner, M., Nachtsheim, C., Neter, J.: Applied Linear Regression Models. 4^{th} International Edition, McGraw-Hill/Irwin. (2004)
46. Rogerson, P.A.: Statistical methods for geography: a student's guide. Sage Publications (2010)
47. An, L., Khomh, F.: An empirical study of crash-inducing commits in mozilla firefox. In: Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering, ACM (2015) 5
48. Yin, R.K.: Case Study Research: Design and Methods - Third Edition. 3 edn. SAGE Publications (2002)
49. Wikipedia: C-family programming languages. `https://en.wikipedia.org/wiki/List_of_C-family_programming_languages` (2017) Online; Accessed January 24th, 2017.