AN ABSTRACT OF THE DISSERTATION OF

Caius Brindescu for the degree of Doctor of Philosophy in Computer Science presented on July 7, 2020.

Title: An Investigation of the Effects of Merge Conflicts on Collaborative Software Development

Abstract approved: _____

Anita Sarma Carlos Jensen

Merge conflicts have long plagued software development. With larger and more dispersed teams comes greater risk of developers working on the same code at the same time. While merge conflicts are known to be painful, their exact impact on software is still largely unknown. Are merge conflicts an isolated problem, or are they linked to bigger issues in the code base? This thesis tries to answer this. The first 2 parts of this thesis look at the correlation between code smells, bugs, and merge conflicts. Our results show that code that is involved in merge conflicts most frequently is more likely to have bugs, or exhibit code smells.

To better understand where tool support is lacking, in the 3rd part, we investigated how developers resolve merge conflicts. We used a sensemaking approach to analyze qualitative data collected in situ, where participants were working on production software. We found different patterns developers use when resolving merge conflicts, and different areas where they struggled. Our results show that merge conflicts have a real impact on software quality and also show potential avenues for tool improvements.

Finally, we propose a way to predict the difficulty of a merge conflict. This can allow developers to plan ahead and set aside time and resources to handle the merge conflict resolution.

An Investigation of the Effects of Merge Conflicts on Collaborative Software Development

by

Caius Brindescu

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented July 7, 2020
Commencement June 2021

Doctor of Philosophy dissertation of Caius Brindescu presented on July 7, 2020

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Caius Brindescu, Author

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support of many. I am thankful for all your support and guidance over the years.

First of all, I would like to thank my advisors, Dr. Anita Sarma and Dr. Carlos Jensen for their guidance and mentorship. Your patience and care for my professional development has helped me reach where I am today.

I'd also like to thank the faculty at Oregon State University, Dr. Alex Groce, Dr. Margaret Burnett, and Dr. Danny Dig for their guidance and advice over the years. I would also like to thank the team at Oregon State CASS for allowing me to conduct my studies, for having patience and granting me access without which this dissertation would not have been possible.

I would also like to thank my colleagues and peers at Oregon State and University of Illinois at Urbana-Champaign with which I have worked. I would like to thank Iftekhar Ahmed, Umme Ayda Mannan, Nick Nelson, Souti Chattopadhyay, Lara Letaw, Mihai Codoban, Rahul Gopinath, Amin Alipour and Cosmin Rădoi. Our conversations over coffee and beer have been memorable.

I would like to give a shout out to Radu Marinescu for giving the opportunity to do research in undergrad and for sparking the passion for knowledge in me.

I am also thankful to my family for their immense support over these years. I'd like to thank my mother for her patience over this long journey. Her drive has inspired me to never give up, and keep pushing forward.

I would also like to thank the team at Etleap for the support they have given over the past 2 years.

Last, but not least, I would like to thank my partner, Heather Hill. Thank you for being there for me all these years, during long terms and nights. This PhD was possible only through your support.

CONTRIBUTION OF AUTHORS

This dissertation consists of four main chapters, each being a research paper submitted to either a conference or a journal. The following authors contributed to the manuscript: Caius Brindescu (CB), Iftekhar Ahmed (IA), Umme Ayda Mannan (UM), Yenifer Ramirez (YR), Rafael Leano (RL), Anita Sarma (AS), Carlos Jensen (CJ). Their individual contribution is summarized in Table 0.1, below.

TABLE 0.1: Contribution of authors

Task	Conception	Data Collection	Analysis	Final Draft
Chapter 2	IA, CB	IA, CB	CB, IA	IA, CB, AS, CJ
Chapter 3	CB	CB, IA	CB, IA	CB, AS, CJ
Chapter 4	CB	CB	CB, YR	CB, AS, CJ
Chapter 5	CB	CB, RL	CB, IA, RL	CB, IA, AS

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

AN INVESTIGATION OF THE EFFECTS OF MERGE CONFLICTS ON COLLABORATIVE SOFTWARE DEVELOPMENT

1 INTRODUCTION

Software development is a team effort. Modern version control systems, such as Git, allow teams to efficiently work together while keeping track of the changes they make. This allows organizations to build reliable systems by pooling the talents of many developers. However, collaborative development is not always a smooth experience. With large teams, it's not uncommon for two developers to make changes to the same piece of code at the same time [137, 117, 108]. This results in merge conflicts, which are only apparent after developers have finished all their changes. Once a merge conflict occurs, developers need to interrupt whatever they are doing, understand the other set of changes being made, and finally resolve the conflict before they can move on with their work.

However, despite the prevalence of merge conflicts, their impact on collaborative software development is not fully understood. In this dissertation, we investigate the effect of merge conflicts on collaborative software development from three angles. First, we looked at the impact on software quality. Second, we looked at the human perspective, and try to understand what makes a merge conflict resolution difficult. Third, we looked at we can do to improve the merge conflict resolution proces and make developers lives easier.

Our first research goal is to *understand whether and in what ways merge conflicts are related to software quality.* Are they just a nuisance, or are they an indicator of a deeper problem? To understand this, we conducted 2 large scale empirical studies on over

100 software projects. We analyzed almost 7,000 merge conflicts and 32,000 merges to understand the code where merge conflicts occur.

This investigation included 2 threads. First we looked at the design of the software itself. Problems in the design of software are made evident by the presence of code smells. Code smells indicate an area in the code where the design is lacking [65]. They can adversely impact the maintainability of the code [96]. Code smells can be automatically detected [103] and have been used extensively by researchers as a proxy for software quality [103, 12, 143, 122, 121, 75, 105]. The presence of code smells makes the code more difficult to change, as the changes will impact more of the code. This, in turn, can increase the likelihood of merge conflicts occurring.

In the second thread we looked at software correctness. We investigated whether code became lower quality after being modified after a merge conflict. Since a merge conflict requires developers to "cobble" together a solution from 2 distinct sets of changes, it's likely that they could introduce bugs in the process. We investigated if merge conflicts are associated with future bug fixes.

The two threads of the investigation showed a strong correlation between merge conflicts and code smells. Similarly, we also observed a strong correlation between merge conflicts and bug proneness. We also noticed that lines involved in semantic merge conflicts (which occurred because of a conflict in code logic) were a lot more likely to have bug fixes applied to them in the future. All this indicates that merge conflicts are more than a nuisance: They are a severe problem.

Given this strong correlation between merge conflicts and software quality, it is important to help developers during merge conflict resolution. However, before we can design a solution, we need to understand where they struggle. Therefore, our second research goal is to *understand what makes conflict resolution difficult.*

We can't understand this by analyzing source code, so we conducted a field study

of developers in the wild. We conducted an observation study of professional developers working on production code. This gave us a rich dataset, which we analyzed from a qualitative perspective.

Our results showed that, when resolving a merge conflict, developers need to gather information from many disparate information sources. In some cases, our participants had to collect information from over 20 different artifacts, spread across 6 different categories (e.g., code, history, documentation, etc). Each category contains a different type of data presented in a different way and it's up to the developer to decide which information is relevant and how pieces of different information are related. Our results showed that participants struggled find the right information, and also to contextualize it. This is the heart of why merge conflict resolution is difficult.

Helping developers gauge this difficulty ahead of time was our third research goal. Specifically, we investigated *ways of giving developers advanced notice about how difficult a merge conflict resolution might be,* so they'd be less inclined to "kick the can" further down the road by finding a way to postpone the resolution [117]. Postponing merge conflict resolution is attractive to developers because they don't have to step "out of the flow" of what they're doing.. In fact, a study found that at least 56.18% of developers have deferred a merge conflict resolution at least once [117].

If developers know what type of merge conflict resolution experience is coming, they can make an informed decision about what to do instead of procrastinating because there are too many unknowns. Given enough information, they could, for example, feel justified in consulting a team member or setting aside part of their work day to address the problem. Out results showed that it's possible to accurately predict the difficulty of a merge conflict.

1.1 Research goals

To summarize, this dissertation has 3 research goals:

1. Understand the link between merge conflicts, code smells and future bug fixes;
2. Understand what makes merge conflict resolution difficult, and;
3. Help developers prepare for a resolution, by giving them an advance indication of the difficulty of the resolution.

1.2 Thesis structure

This thesis is structured around 4 papers:

- Chapter 2 presents the paper *"An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts"* co-written with Iftekhar Ahmed, Umma Ayda Mannan, Anita Sarma and Carlos Jensen, and presented at ESEM 2017.
- Chapter 3 presents the paper *"An Empirical Investigation into Merge Conflicts and their Effect on Software Quality,"* written with Iftekhar Ahmed, Anita Sarma and Carlos Jensen. Both the papers answer the first research goal.
- Chapter 4 presents the paper *"Struggles in Sensemaking: A Field Study of Merge Conflict Resolution Behavior,"* written with Yenifer Ramirez, Anita Sarma and Carlos Jensen, which is currently under review at ICSME 2020. This answers the second research goal.
- Chapter 5 presents the paper *"Planning for Untangling: Predicting the Difficulty of Merge Conflict"* written with Iftekhar Ahmed, Rafael Leano and Anita Sarma, and presented at ICSE 2020. This papers addresses our third and final research goal.

Finally, in Chapter 6 we present conclusions and future directions for the work.

2 AN EMPIRICAL EXAMINATION OF THE RELATIONSHIP BETWEEN CODE SMELLS AND MERGE CONFLICTS

Iftekhar Ahmed, Caius Brindescu[2], Umme Ayda Mannan, Carlos Jensen, Anita Sarma

2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)

2.1 Introduction

Modern software systems are becoming more and more complex and requires a large development team to develop and maintain. Modern Version Control Systems (VCS) have made parallel development easier by streamlining and coordinating code management, branching, and merging. This enables large teams to work together efficiently. But it has been shown that this process is sometimes halted when isolated private development lines are synchronized and the developer runs into merge conflicts. Conflicts distract the developers as they have to interrupt their workflow to resolve them. Developers have to reason about the conflicting changes and find an acceptable merging solution. This process of conflict resolution can itself introduce bugs. Prior work has found that in complex merges, developers may not have the expertise or knowledge to make the right decisions [50, 118] which might degrade the quality of the merged code.

Researchers have looked at many ways of preventing merge conflicts, and make developer's lives easier when they do occur. Researchers have proposed workspace awareness tools [24, 48, 77, 135, 139] that help prevent merge conflicts by making the developers aware of each other's changes. Also, new merge techniques [15, 16, 90] have been proposed that would reduce the number of merge conflicts. However, little research has been

[2]Both authors contributed equaly

devoted to the causes of merge conflicts. Are there any endemic issues that arise from the design itself? We are interested in knowing whether the design of the codebase has an effect on the merge conflicts and what is its impact on the overall quality.

Just like merge conflicts, bad design can inflict pain on developers. Bad design makes maintenance and future changes difficult and error prone. Code smells, an indication of bad design, imply that the structure of the code is badly organized. This can lead to developers stepping on each other's toes as they make their changes. This, in turn, can lead to merge conflicts.

If there are "fundamental flaws" in the design itself, as the project grows, and the codebase grows in size and complexity, understanding and working around these "rough spots" becomes more challenging. Thus, the chances of creating a conflict increases because of the need to generate workarounds. This means that as projects grow, merge conflicts should be more likely to occur, especially around the smelly parts of the code. We aim to examine whether there is a correlation between the two, to examine whether such a link is credible.

In order to evaluate the design we look at the code smells [97]. We investigate if there is a connection between entities that contain code smells, the code smells they contain, and the merge conflicts that surround the smelly entities.

It is important to note that not all smells are created equal. Some might be more associated with a merge conflict than others. For example, a class is considered a *God Class* if it contains an oversized part of the entire functionality of the final product. Therefore, any changes have a high likelihood of involving changes in the *God Class*. When multiple developers are working, they all have a high likelihood of touching the *God class*. This can easily lead to merge conflicts down the road. If the changes involved are not trivial then the task of merging them will be not trivial as well.

In this paper, we investigate the following questions:

RQ1: Do program elements that are involved in merge conflicts contain more code smells?

RQ2: Which code smells are more associated with merge conflicts?

RQ3: Do code smells associated with merge conflicts affect the quality of the resulting code?

To answer these questions, we investigated 143 projects. Across them, we had 36,122 merge commits, out of which 6,979 were conflicting. We identified 7,467 code smells instances across our whole corpus. We found that merge conflicts involved more "smelly" program elements than merges that did not conflict. Our results also show that not all code smells are created equal. Some are more likely to cause problems than others. When we looked at the difficulty of merge conflicts, we found that some of the smells are more likely to be involved in *semantic* merge conflicts than others. Finally, we found that code smells have a negative impact on code quality.

2.2 Related Work

2.2.1 Code smells and their impact

Various measures of software quality have been proposed. Boehm et al. [**?**], and Gorton et al. [70], to mention a few, have explored measures including completeness, usability, testability, maintainability, reliability, efficiency etc. Some of these metrics are difficult to measure, especially in the absence of requirement documents or other supporting information. Researchers have also used code smells as a measurement of software quality [104, 103], though smells are often focused on future maintainability issues. The concept of code smells was first introduced by Fowler [65]. Code smells are symptoms of poor design and implementation choices [65] in code base which eventually affect the maintainability of a software system [96]. Studies also showed that there is an association

between code smells and bugs [100, 120] and code maintainability problems [65]. Code smells also leads to design debt. Zazworka et al. [155] found that the *God Class* smell is related to technical debt. Ahmed et al. [14] found how software gets worse over time in terms of design degradation. They analyzed 220 open source projects in their study and confirmed that ignoring the smells leads to "software decay".

Researchers have proposed many different approaches for detecting code smell, such as metric based [53, 54, 97, 100, 103] and meta-model based [115]. Researchers used different techniques for identifying code smells. Fontana et al. [63] used machine learning techniques for detecting code smells. Researchers also used both static analysis [53, 54, 100] and techniques that rely on the evaluation of successive versions of a software system [88, 97, 122].

2.2.2 Work related to code smells and bugs

Researchers have also considered the relationship between the presence of code smells and bug appearance in the code base. Khomh et al. [91] showed that classes affected by design problems ("code smells") are more likely to contain bugs in the future. Hall et al. [75] also found relationships between code smells and fault-proneness. According to their study some code smells indicate fault-proneness in the code base but the effect size is small (under 10%). Zazworka et al. [155] found that *God Classes* are fault-prone in some cases. Li et al. [103] also studied the relationship between code smells and the probability of faults in industrial systems, and found that the *Shotgun Surgery* smell was correlated with a higher probability of faults. To the best of our knowledge no work has tried to research on the relationship between code smells and how it impacts collaborative work flow, specifically merging individual works.

2.2.3 Merge conflicts

Several studies have been done on identification of conflicts and developers' awareness about potential conflicts. Awareness is frequently defined as an understanding of the activities of others to give a context for one's activities [58], which is a very important issue in Global Software Engineering (GSE) [134]. Researchers have looked at different techniques of avoiding merge conflicts by increasing the developer's awareness of the changes others made to the source code. Biehl et al. [24] proposed FastDash, which sends notifications about potential conflicts when two or more developers are modifying the same file. Another awareness tool called Syde by Hatori et al. [77] consider the source code changes at Abstract Syntax Tree (AST) level operations to detect conflicts by comparing tree operations. Da Silva et al. [48] introduced Lighthouse, which is another tool for increasing awareness among developers about the conflict. Palantír by Sarma et al. [135] detects the changes made by other developers and show them in a graphical, non-intrusive manner. Servant et al. [139] also presented a tool and visualization that can be used to understand the impact of developers' changes to prevent indirect conflicts.

Guimaraes et al. [74] introduce WeCode which continuously merges uncommitted and committed changes in the IDE to detect merge conflicts as soon as possible. Brun et al. [31] used the similar approach in Crystal, to detect both direct and indirect conflicts. A software development model presented by Dewan et al. [55] aims to reduce conflicts by notifying developers who are working on the same file.

2.2.4 Work related to merge conflict resolution

Researchers have also studied different ways of managing the merge of developers' changes to efficiently resolve conflicts. This resolution could be either in an automated way or by preserving and presenting a useful context for the developer trying to resolve the conflict. A comprehensive survey of merge approaches was done by Mens [110]. Apel et al. [16, 15] presented a merging technique called semistructured merge. This considers

the structure of the code which is being merged. Operation based merging by Lippe et al. [101] considers all the changes performed during development, in addition to the result, when merging.

Kasi and Sarma [90] present a technique of avoiding merge conflicts by scheduling tasks in a way that the probability of a conflict is minimized. SafeCommit by Wloka et al. [152] uses a static analysis approach to identify changes in a commit with no test failure. They proposed to use this approach when detecting indirect conflicts.

2.2.5 Conflict categorization

Researchers have come up with different ways of categorizing conflicts. Sarma et al. [135] grouped conflicts into two categories. One is direct conflicts, where the changes conflict directly. The other is indirect conflicts, where the files don't conflict directly, but integrating the changes cause build or test failures. Similarly, Brun et al. [31], categorized conflicts as first level (textual) conflicts and second level (build and test failure) conflicts. Buckley et al. [33] proposed a taxonomy of changes based on properties like time of change, change history, artifact granularity etc. Their taxonomy deals with software changes in general or conflicts at a coarser level.

2.2.6 Tracking code changes and conflicts

Researchers have proposed various algorithms for tracking individual lines of code across versions of software. Canfora et al. [35] proposed an algorithm that uses Levenstein edit distance to compute similarity of lines, matching "chunks" of changed code. Zimmerman et al. [159] proposed annotation graphs which works at the region level for tracking lines. Godfrey et al. [69] described "origin analysis", a technique for tracking entities across multiple revisions of a code base by storing inexpensively computed and easily comparable "fingerprints" of interesting software entities in each revision of a file. These fingerprints can then be used to identify areas of the code that are likely to match

before applying more expensive techniques to track code entities. Finally, Kim et al. [93] propose an algorithm, SZZ, for tracking the origin of lines across changes.

2.3 Methodology

Our goal was to identify the effect of design issues on merge conflicts and the quality of the resulting code (whether these changes are associated with bug fixes or other improvements.)

Here we discuss the various steps of collecting data: (1) selecting the sample of projects for the study, (2) identifying which merge commits lead to merge conflicts, (3) tracking the lines of code through different versions and merges to investigate how the code evolved and which lines were associated with conflicts, (4) identifying code smells at the time of the conflicting merge commit. Next, we determine the nature of the code updates (e.g. was the commit a result of a bug fix or a new feature etc.) taking place on those lines. In order to do this, we manually classify a subset of the commits as bug-fix related or other. We train a machine-learning classifier to classify the rest. Finally, we build a model to predict the total number of bug fixes that would occur on a conflicting line that also contained a code smell. The following subsections describe each of these steps in detail.

2.3.1 Project Selection Criteria

We wanted to make sure that our findings would be representative of the code developed in real world, thus we selected active, open source projects hosted in GitHub. We decided to use Java as the language of focus. This decision was influenced by 2 factors: First, Java is one of the most popular languages (according to the number of projects hosted on Github and the Tiobe index [3]). The second was the availability of code smell detection tools for Java, as compared to other programming languages. Further, for ease

of building and analyzing the code, we select projects using the Maven [4] build system.

We started by randomly selecting 900 projects, the first to show up when using the GitHub search mechanism. From these, we eliminated aggregate projects (which could skew our results), leaving 500 projects. After eliminating projects that did not compile (for reasons such as unavailable dependencies, or compilation errors due to syntax or bad configurations), 312 projects remained. Finally, we eliminated projects our AST walker, implemented using the GumTree algorithm [61], could not handle. This left us with a total of 200 projects.

Next, we removed projects that were too small, that is, having fewer than 10 files, or fewer than 500 lines of code. We also removed projects that had no merge conflicts. These selection criteria were used, since we are interested in the effect of design issues and merge conflicts in moderately large, collaborative projects. Our final data set contained 143 projects. Table 2.1 provides a summary of features and other descriptive information of the projects in our study.

We also manually categorized the domain of the projects by looking at the project description and using the categories used by Souza et al. [49]. Table 2.2 has the summary of the domains of the projects.

TABLE 2.1: Project Statistics

Dimension	**Max**	**Min**	**Average**	**Std. dev.**
Line count	542,571	751	75,795.00	105,280.10
Duration (Days)	6,386	42	1,674.54	1,112.11
# Developers	105	4	72.76	83.19
Total Commits	30,519	16	3,894.48	5,070.73
Total Merges	4,916	1	252.60	522.73
Total Conflicts	227	1	25.86	39.49

2.3.2 Code smell detection tool selection

We chose to use InFusion [2] to identify code smells because it has been found to identify the broadest set of smells [17]. Researchers have found that the metric-based

TABLE 2.2: Distribution of Project by Domain

Domain	Percentage
Development	61.98%
System Administration	12.66%
Communications	6.42%
Business & Enterprise	8.10%
Home & Education	3.11%
Security & Utilities	2.61%
Games	3.08%
Audio & Video	2.04%

approach identified by Marinescu [104] has the highest recall and precision (precision: 0.71, recall: 1.00) for finding most code smells [138]. InFusion uses this same principle and set of thresholds for identifying code smell, which was another reason for using InFusion. Researchers [14] have evaluated the smell detection performance of InFusion where they found it to have precision of 0.84, recall of 1.00 and an F-measure of 0.91.

2.3.3 Conflict Identification

Since Git does not record information about merge conflicts, we had to recreate each merge in the corpus in order to determine if a conflict had occurred. We used Git's default algorithm, the recursive merge strategy, as this is the most likely to used by the average Git project. From our sample of 143 projects we extracted 556,911 commits. This included 36,122 merge commits. The average number of merge commits was 253. Out of all the merges, 6,979 (19.32%) were identified as leading to a conflict. The distribution of merge conflicts is shown in Figure 2.1. We see that projects experience an average of 25 merge conflicts, or 19.32% of all merges. Merge conflicts, therefore, are a common part of the developer experience.

We then collected statistics regarding each file involved in a conflict. We tracked the size of the changes being merged, the difference between the two branches (in terms of LOC, AST difference, and the number of methods and classes involved). To determine the AST difference, we used the Gumtree algorithm [**?**]. We also tracked the number of

authors involved in the merge.

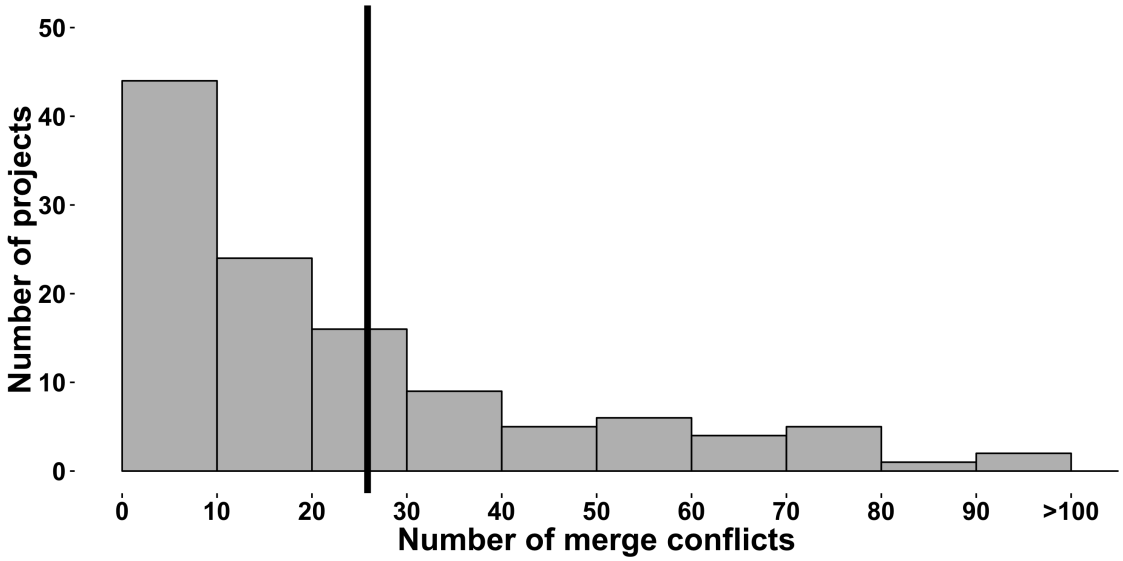

FIGURE 2.1: Distribution of merge conflicts. The vertical line represents the mean (25.86)

2.3.4 Conflict Type Classification

To answer our second research question, we needed to categorize the conflicts based on the type of changes (e.g., whitespace or comment added vs. variable name changed).

We identified two categories of conflicts. The first one being *semantic* conflicts which requires understanding the program logic of the changes in order to successfully resolve the conflict. The other type of conflict is *non-semantic* which easier and less risky to resolve since they do not affect the programs' functionality. We manually classify 606 randomly sampled commits. We classify each conflict based on the type of changes causing the merge conflict (e.g., whitespace or comment added vs. variable name changed). Two of the authors coded 300 of these commits using qualitative thematic coding [47]. They achieved an interrater agreement of over 80% on 20% of the data: we obtained a Cohen's Kappa of 0.84. Having reached an agreement, one of the authors classified the remaining 306 commits. The codes and their definitions are given in Table 2.3.

TABLE 2.3: Conflict Categories

Category	Definition	Example
Semantic	Conflicts involving semantic changes	A refactoring and a bug fix involving the same lines.
Non-Semantic	Conflicting changes in formatting/comments	One of the branches contains only formatting changes (whitespace).

To train the classifier (to differentiate between *semantic* and *non-semantic* commits) we use a set of 24 features, including: the total size of the versions (LOC) involved in a conflict, the number of statements, methods and classes involved in the conflict. Details of the features are in the accompanying website [5]. We use the set of 606 (10%) commits as training data for a machine learning classifier. We used Adaptive Boost (AdaBoost) ensemble classifier that can only be used for binary classes. We categorized the 6,979 conflicting commits. We use 10 fold cross-validation to test the performance of our classier. The precision of predicting the *semantic* conflicts is high at 0.75.

2.3.5 Measuring Code Smells and Tracking Lines

For each of the 6,979 conflicting commits we collected the code smells that were associated with a conflict. We needed to track them to measure the effect of having the smell and being involved in a conflict on the quality of the resulting code.

We use GumTree [61] for our analysis, as it allows us to track elements at an AST level. This way we can track only the elements that we are interested in (statements), and ignore other changes that do not actually change the code. The GumTree algorithm works by determining if any AST node was changed, or had any children added, deleted or modified. The algorithm maps the correspondence between nodes in two different trees, which allows it to accurately track the history of the program elements. This algorithm has unique advantages over other line tracking algorithms, such as SZZ [93]. These advantages include: ignoring whitespace changes, tracking a node even if its position in the file changes

(e.g. because lines have been added or deleted before the node of interest), and tracking nodes across refactorings, as long as the node stays within the same file. Using this technique, we can track a node even when it has been moved, for example, because of an extract method refactoring.

For each node (in the AST) involved in a conflict and having a smell, we identify all future commits that touched the file containing said node and tracked the AST node forward in time. For Java, it is possible for multiple statements to be expressed in the same line (e.g., a local variable declaration inside an if statement). In this case, we considered the innermost statement, as this gives the most precise results.

2.3.6 Commit Classification

In order to answer our third research question, we needed to categorize the type of change for a code commit. For our purpose, code commits can be broadly grouped into one of two categories: (1) bug-fixes and improvements (modifying existing code), and (2) Other — commits that introduced new features or functionality (adding new code) or commits that were related to documentation, test code, or other concerns. Two key problems with this classification are: (1) it is not always trivial to determine which category a commit falls under, and (2) larger projects see a huge amount of activity. Manual classification of all commits was not an option, and we decided to use machine learning techniques for this purpose, rather than limiting the statistical power of our study (especially as arbitrarily dropping the most active subjects would clearly potentially introduce a large bias into our results.)

In order to build a classifier, we randomly selected and manually labeled a set of 1,500 commits. The first two authors worked independently to classify the commits. Their datasets had a 33% overlap, which we used to calculate the inter-rater reliability. This gave us a Cohen's Kappa of 0.90. In our training dataset, the portion of bug-fixes was 46.30%, with 53.70% of the commits assigned to the Other category. Some keywords

indicating bug-fixes or improvements were Fix, Bug, Resolves, Cleanup, Optimize, and Simplify, and their derivatives. Anything that did not fit into this pattern was marked as Other.

Not all bug-fixing commits include these keywords or direct reference to the issue-id; commit messages are written by the initial contributor, and there are few guidelines. A similar observation was made by Bird et al. [**?**], who performed an empirical study showing that bias could be introduced due to missing linkages between commits and bugs. This means that we are conservative in identifying commits as bug-fixes.

We trained a Naive-Bayes (NB) classifier and a Support Vector Machine (SVM) by using the SciKit toolset [124]. We used 10% of the data to train the classifier. We applied the classifiers to the training data using a 10-fold cross-validation. As before, we used the F1-score to measure and compare the performance of the models. The NB classifier outperformed the SVM. Therefore, we used the NB classifier to classify our full corpus.

Table 2.4 has the quality indicator characteristics of the NB classifier. Tian et al. [147], suggest that for keyword-based classification the F1 score is usually around 0.55, which also occurs in our case. While our classifier is far from perfect, it is comparable to "good" classifiers in the literature, and we believe it is unlikely for the biases to have a confounding effect on our analysis. Since our analysis only relies on relative counts of bug-fixes for statements, so long as we do not systematically undercount bug-fixes for only some statements, our results should be valid.

TABLE 2.4: Naive Bayes classifier details

	Precision	**Recall**	**F1-measure**
Bug-fix	0.63	0.43	0.51
Other	0.74	0.86	0.80

For each line of code resulting from a merge conflict, we count the number of (future) commits in which it appears, as long as those commits are identified as bug-fixes. We stop

the tracking when we encounter a commit that is classified as Other. Our reasoning is that once an element has seen a change that is not a bug-fix, it is no longer fair to assume that subsequent bug fixes are associated with the original merge conflict.

2.3.7 Regression analysis

In order to answer our third research question that is related to the effect of code smells on quality of the resulting code, we needed to build a regression model to identify the impact of code smell on the number of bug-fixes that occur on lines of code that are associated with a code smell *and* a merge conflict. We use Generalized Linear Regression [41]. The dependent variable (count of bug fixes occurring on smelly and conflicting lines) follows a Poisson distribution. Therefore, we use a Poisson regression model with a log linking function.

In order to build our model, we collect information about the smells and the conflicts. We use Understand [8] to count the number of references to, and from other files to the files that are involved in a conflict. We collect this information as a proxy for the importance of the file. We assume that the more a file is referenced by other files, the more central that file is, and hence more important. Any change in these central files can increase the chance of a change being required in other files, and therefore lead to multiple developers making changes to these files, which can in turn lead to conflicting changes.

We also collect the following factors for each commit such as the difference between the two merged branches in terms of LOC, AST difference, and the number of methods and classes being affected. Our intuition is that larger "chunks" of changes should have a higher chance of causing a conflict. We also calculate the number of authors who made commits to the branches that were merged, since there is a higher likelihood of conflicts if multiple developers are involved.

We also determine the experience level of each developer by splitting them into two categories: *core* and *non-core*. To calculate the category for each developer, we split the

development history into quarters. For each commit a developer is classified as *core* if he is in the top 20% of the developers in that quarter (calculated by the number of commits). Otherwise he is *noncore*. We use this process because, in open source projects, authors come and go. Also, an author can be classified as *core* and *non-core* in different quarters, depending on his contribution to the project.

After collecting these metrics, we checked for multi-collinearity using the Variance Inflation Factor (VIF) of each predictor in our model [41]. VIF describes the level of multicollinearity (correlation between predictors). A VIF score between 1 and 5 indicates moderate correlation with other factors, so we selected the predictors with VIF score threshold of 5. This step was necessary since the presence of highly correlated factors forces the estimated regression coefficient of one variable to depend on other predictor variables that are included in the model.

2.4 Results

2.4.1 RQ1: Do program elements that are invovled in merge conflicts contain more code smells?

As a first step, we collect the total number of code smells for each of the 6,979 conflicting commits in our dataset. Table 2.5 contains the percentage of each smell and the percentage of projects that have a particular smell. We find that *external* and *internal duplication* have a much higher instance than others when considering the percentage of smells in the dataset. However, about 50% of projects have *Data Class* and *SAP Breakers* smells.

We next compare the mean number of code smells associated with each merge commit, for cases when they conflict and for cases when they do not conflict. Note that a commit can involve multiple files, and a file can contain multiple smells. We calculate the total number of smells for each file. For example, a conflicting merge commit in the

TABLE 2.5: Percentage of code smells

Smell	**% of smells in the full dataset**	**% of projects w/ smell**
External Duplication	42.79	22.53
Internal Duplication	**34.05**	**23.80**
Feature Envy	4.04	28.42
Data Clumps	3.71	20.36
Intensive Coupling	3.50	14.30
Data Class	3.18	48.05
Blob Operation	2.58	30.05
Sibling Duplication	2.35	10.86
SAP Breakers	1.52	52.76
God Class	**0.89**	**19.10**
Schizophrenic Class	0.58	20.00
Message Chains	0.33	5.34
Tradition Breaker	0.17	6.33
Refused Parent Bequest	**0.19**	**5.25**
Shotgun Surgery	0.01	1.72
Distorted Hierarchy	**0.003**	**0.36**

`commandhelper` project (with the SHA1 of `a91faa`) contains one conflicting file, and that conflicting files contains a total of 8 smells.

The mean number of smells in conflicting program elements is **6.54**, whereas the mean for non-conflicting program elements is **1.92**. The results are statistically significant (Mann-Whitney test, U=6.24e6, p<4.77e-10.); we use the non-parametric Mann-Whitney test since our population is not normally distributed. Therefore, we find that *program elements that are involved in merge conflicts are, on average, more smelly than entities that are not involved in a merge conflict.*

2.4.2 RQ2: Which code smells are more associated with merge conflicts?

Next, we compare the occurrence of each individual smell across conflicting and non-conflicting commits. Since we are performing multiple tests, we have to adjust the significance value accordingly to account for multiple hypothesis correction. We use the Bonferroni correction, which gives us an adjusted p-value of 0.0031.

For 12 out of 16 total smells, we find significant differences (Mann-Whitney test,

α<0.0031) between the means of conflicting and non-conflicting commits. The conflicting commits have a higher incidence of smells. Table 2.6 presents the results for code smells where the difference was significant along with the p-values of individual comparisons..

TABLE 2.6: Mean number of Smells in Conflicts vs. Non-conflict Commits Calculated per Commit

Smell	**Smells in conflicts**	**Smells in non conflict**	**p-value**
God Class	1.23	0.25	0.0001
Data Clump	0.65	0.27	0.0001
Sibling Duplication	0.58	0.10	0.000001
Data Class	0.47	0.12	0.000001
Distorted Hierarchy	0.45	0.05	0.000001
Unnecessary Coupling	0.33	0.10	0.0001
Internal Duplication	0.24	0.08	0.000001
SAP Breaker	0.12	0.07	0.000001
Tradition Breaker	0.10	0.05	0.00007
Blob Operation	0.07	0.06	0.0001
Message Chain	0.04	0.03	0.00062
Shotgun Surgery	0.01	0.00769	0.00021

The following are the top 5 smells in terms of their (mean) numbers per conflict: *God Class, Data Clump, Sibling Duplication, Data Class* and *Distorted Hierarchy*. It is worth noting that the distribution of smells per conflict (Table 2.6) is different from Table 2.5. This is because in Table 2.6 we are looking only at the smells that affect the entities involved in merge conflicts, whereas Table 2.5 shows all the smells in the project. This discrepancy is an effect of the fact that merge conflicts exhibit a different smell pattern compared to the overall project.

Next, we perform two steps. First, we investigate the correlation between each smell and the merge conflicts to identify which of the above smells are more strongly associated with conflicts. Then, we categorize merge conflicts into *semantic* and *non-semantic* conflicts to further explore the associations of smells to these types of conflicts.

Code smells and conflicts. We perform a correlation analysis between the count of smells and merge conflicts to distill which of the smells from Table 2.6 are more closely

associated with conflicts, and should be attended to. We use the Kendall correlation test because it is a non-parametric test and it is more accurate with a smaller sample size. As we perform the tests for each smell, we are splitting out data into smaller chunks. Therefore, the Kendall correlation test is more appropriate.

We find that, except for *External Duplication, Schizophrenic Class*, *SAP Breaker* and *Data Class* all smells are correlated with merge conflicts (Kendall correlation test, α<0.0031). We report the statistically significant results in Table 2.7.

The three strongest correlation to conflicts are with the following smells: *God Class*, *Internal Duplication* and *Distorted Hierarchy*. These smells all relate to cases where object-oriented design principles of encapsulation and structuring is not well used, leading to problems with developers making conflicting parallel changes. We discuss these reasons further in Section 2.5.

TABLE 2.7: Correlation between conflict and smell count

Smell	**Correlation**	**p-value**
God class	0.18	<0.0001
Internal Duplication	0.17	<0.0001
Distorted Hierarchy	0.13	<0.0001
Refused Parent Bequest	0.10	<0.0001
Message Chain	0.10	<0.0001
Data clump	0.09	<0.0001
Feature Envy	0.09	<0.0001
Tradition Breaker	0.09	<0.0001
Blob Operation	0.08	<0.0001
Shotgun Surgery	0.07	<0.0001
Unnecessary Coupling	0.05	0.00007
Sibling Duplication	0.04	0.00021

Types of conflicts and their classification. Not all conflicts are the same, some involve changes to the actual code structure and require the developer to understand the logic behind the changes before they can be integrated (*semantic* conflicts), whereas others can be formatting or cosmetic changes (*non-semantic*). *Semantic* conflicts are inherently harder to resolve. Therefore, we investigate whether specific types of code smells are more

likely to occur with *semantic* conflicts. We use the conflict classification methodology in Section 2.3.4.

Recall, we manually labeled 606 conflicts to classify them into *semantic* or *non-semantic*, which we then use for the automated classification of 6,979 commits. We present the distribution between the manual and automatic classification in Table 2.8. The distributions of *semantic* and *non-semantic* conflicts in the automatically classified data match the distribution of our manual labeling (training data), which shows the efficacy of the automated classifier.

TABLE 2.8: Conflict types based on their frequency of occurrence

Category	**# of Conflicts**	**% of total (classifier)**	**% of total (training)**
Semantic	5,250	75.23%	76.12%
Non-Semantic	1,729	24.77%	23.88%

Semantic conflicts are more common (76.12% in the manually labeled data and 75.23% in the automated classified data), as compared to the *non-semantic* conflicts (23.88% in manually labeled and 24.77% in automated classified data).

Semantic conflicts and code smells: To understand if there is any correlation between *semantic* conflicts and the types of code smells we perform the Kendall correlation test for each smell in the presence of *semantic* merge conflicts (in our total dataset). We use the Kendall correlation test and found significant correlation (α<0.0031) only for *Internal Duplication* and *Blob Operation*. Table 2.9 contains all correlations, where the cells marked with ** are significant.

Since the correlation for both *Blob Operation* and *Internal Duplication* are small, we perform an odds-ratio test to understand which of these smells are more likely to be involved in a *Semantic* merge conflict, as compared to entities that do not have these smells, but were involved in a conflict. Since we are performing two comparisons, we have to adjust the significance value to adjust for multiple hypothesis testing. Like in the

previous sections, we performed a Bonferroni correction, which gives us significance value of α=0.0025 to test at.

TABLE 2.9: Smell Categories for Semantic Conflicts (significance level α=0.0031)

Smell	**Correlation**	**p-value**
Blob Operation **	0.05	0.0030
Internal Duplication **	0.07	0.0002
Message Chain	0.01	0.4970
Refused Parent Bequest	0.03	0.0492
SAP Breaker	-0.02	0.2652
Schizophrenic Class	-0.03	0.0832
Shotgun Surgery	-0.008	0.6597
Sibling Duplication	0.031	0.1029
Tradition Breaker	0.018	0.3291
Unnecessary Coupling	-0.01	0.5681
Data Class	-0.02	0.1524
Data Clumps	0.030	0.1103
Distorted Hierarchy	0.034	0.0670
Feature Envy	0.009	0.6206
God Class	0.050	0.0072

We performed an odds ratio test (Fisher's exact test) for the *Blob Operations* and find that they are 1.7 times more likely to be involved in a *Semantic* merge conflict (odds ratio: 1.77, p=0.0024). *Blob Operations* are methods that are very complex and have many responsibilities. Therefore, any change to the method will likely impact multiple lines, which may intersect with logical changes made by another developer to the same method. This explains the high likelihood of their involvement in *Non-Semantic* conflicts.

For *Internal Duplication*, we found that they are 1.55 times more likely to be involved in merge conflict (odds ratio: 1.55, p=0.0001.) We attribute this to the fact that, because of duplication, a change has to be repeated in multiple locations. This increases the chances of developers making overlapping changes.

2.4.3 RQ3: Do code smells associated with merge conflicts affect the quality of the resulting code?

We aim to model the effects of code smells on the bugginess of a line of code involved in a merge conflict. As defect prediction literature has already identified several factors

(e.g., the size of the module under investigation [59], number of committers [**?**], centrality of files [37]) that affect bugginess, we include them in our model also. Table 2.10 lists the final set of factors that we use, which include metrics that are code-based (F1, F2), change-related (F5-F8), author-related (F3, F4), and code-smells. We compute whether a developer is core or non-core based on our methodology in Section 2.3.7.

To answer our third research question, we build two Generalized Linear Models (GLM). The first contains the number of code smells as a factor, and the second does not. The first (Poisson regression) model is built with a log linking function as explained in Section 2.3.7. After filtering the factors with VIF ≤ 5, we had a set of 8 factors out of 43 factors. All eight factors were statistically significant (see Table 2.10). The predicted value is the total number of bug fixes occurring on a line of code that was involved in a merge conflict. Note that smell count was a significant factor in the model ($p<0.05$), with an estimate of 0.427.

The McFadden Adjusted R^2 [79] of this model is 0.47. We calculated McFadden's Adjusted R^2 as a quality indicator of the model because there is no direct equivalent of R^2 metric for Poisson regression. The ordinary least square (OLS) regression approach to goodness-of-fit does not apply for Poisson regression. Moreover, adjusted R^2 values like McFadden's cannot be interpreted as one would interpret OLS R^2 values. McFadden's Adjusted R^2 values tend to be considerably lower than those of the R^2. Values of 0.2 to 0.4 represent an excellent fit [79]

To understand the impact that code smells have, we built the *same* model by removing the total number of smells as a factor. This decreased the adjusted-R^2 from 0.47 to 0.44. We can therefore conclude that code smells have a significant impact on the final quality of the code. Since McFadden's adjusted R^2 penalizes a model for including too many predictors, had the code smells not mattered, removing it could have increased the adjusted-R^2 instead of reducing it.

TABLE 2.10: Poisson regression model predicting bug-fix occurrence on Lines of Code involved in a merge conflict

Factor#	**Factor**	**Estimate**	**p-value**
F1	In Deps	3.195	<0.0001
F2	Out Deps	-0.053	<0.0001
F3	Noncore author	-3.799	<0.0001
F4	No. Authors	0.129	<0.0001
F5	No. Classes	-0.373	<0.0001
F6	No. Methods	0.244	<0.0001
F7	AST diff	0.001	<0.0001
F8	LOC diff	0.00002571	<0.0001
F9	Number of Smells	0.427	<0.0001

2.5 Discussion

To the best of our knowledge, we are the first to investigate the association of code smells with that of merge conflicts, and their impact on the bugginess of the merged results (line of code). We find that program elements that are involved in merge conflicts contain, on average, 3 times more code smells than program elements that are not involved in a merge conflict.

Not all code smells are equally correlated to merge conflicts. 12 out of the 16 code smells that co-occur with conflicts are significant associated with merge conflicts. The top five code smells from this list are: *God class, Message Chain, Internal Duplication, Distorted Hierarchy* and *Refused Parent Bequest.* Interestingly, the only (significant) code smells associated with *Semantic* conflicts are *Blob Operation* and *Internal Duplication.*

All the above code smells arise when developers do not fully exploit the advantages of object-oriented design, leading to high coupling, duplication, or large containers. These factors lay the groundwork for parallel conflicting efforts, where developers step on each other's toes. For example, the *Blob Operation* is a large and complex method that grows over time becoming hard to maintain. In such a situation, multiple developers may need to make changes to the same method and, therefore, collide when merging. Similarly,

Internal Duplication arises when code is duplicated, which bloats methods and makes it hard to ensure all clones evolve in the same way. In such a situation, developers might have to "touch" multiple parts of the method to ensure all clones are being updated, causing situations of parallel, conflicting edits.

It is interesting to observe that *Semantic* merge conflicts are associated with smells at the method level. For example, the *Blob Operation and Internal Duplication* smells are 1.77 times and 1.55 times, respectively, more likely to be present in a *semantic* conflict as compared to a *non-semantic* conflict. This indicates that bloated methods or duplicated code in methods increase the spread of the change a developer is likely to make, which in turn increases the likelihood of two or more changes conflicting during a merge. Prior work has associated code duplication with negative consequences such as increased maintenance cost [131, 94] and faults [18, 87]. Our findings indicate that duplication also negatively impacts the collaborative workflow by making it difficult to merge changes.

It is worth noting that while smells, such as *God Class* have a significant correlation with overall merge conflicts, they do not have a significant correlation with *semantic* merge conflicts. We posit that a large container (class) with cohesive logical units (methods) can lead to multiple developers making parallel changes that are localized to specific areas (methods) and do not intersect. In these cases, when changes are merged conflicts can arise because of the movement of code or formatting changes (*non-semantic* conflicts). The same reasoning is also applicable for *Distorted Hierarchy, Refused Parent Bequest* and *Message Chain*. In contrast, as discussed earlier method-level smells seem are correlated with *semantic* conflicts.

To the best of our knowledge, ours is the first empirical study to investigate the effects of merge conflicts and code smells on the bugginess of code. We found that the presence of code smells on the lines of code involved in a merge conflict has a significant impact on its bugginess (see Table 2.10). Including code smells as a factor increases the

McFadden's adjusted R^2 value from 0.44 to 0.47. Since McFadden's adjusted R^2 penalizes a model for including too many predictors, an increase in the value signifies that adding code smells as a factor was valuable. We find that factors such as incoming-dependencies and the number of code smells have the highest correlation estimate, indicating their importance to the model.

We find that some factors, such as non-core author, number of classes, and outward dependencies have a negative effect on bugginess. This is counter intuitive. We had assumed that changes from multiple non-core authors are more likely to be buggy. We believe that the following reasons lead to this surprising outcome. It might be the case that non-core contributors are more thorough and put more effort towards submitting code that is less bug prone. Or it might be the process via which newcomers' contributions are accepted. For example, core developers might pay more attention to changes coming from non-core contributors. Further empirical studies on the differences in review processes for core vs. non-core developers will be interesting. We also found that the number of classes involved in a conflict has a negative correlation to its bugginess. This might be because changes that involve multiple classes are more likely to be refactoring or licensing changes, and therefore, less likely to introduce bugs.

Implications: Our findings have a number of implications for software practitioners, tool builders and researchers.

Code smells have been historically associated with maintenance issues, which are known to be a problem in the long term. However, developers are often unaware of code smells. Yamashita et al. found that a considerable portion (32%) of developers did not know about code smells [154]. Our findings shed a different light on the impact of code smells and on the importance of addressing them. Our results show that code smells are an immediate concern for day-to-day activity such as merging changes.

Merge conflicts delay the project by requiring an examination of the conflict, and

disrupting the developers' workflow. Anecdotal evidence shows that developers hate resolving conflicts. Developers are known to follow informal processes (e.g., check in partial code, email the team about impending changes etc.) or rush to commit their work in an effort to avoid having to resolve conflicts [50]. A developer may also choose to delay the incorporation of others' work, fearing that a conflict may be hard to resolve [50]. Such processes can have a detrimental effect on team productivity and morale. This situation can only become worse as the project evolves on two fronts. First, the number of code smells is likely to increase as the project ages [14]. Second, there is a likelihood of increase in merge conflicts as more developers start to contribute. Our results indicate that practitioners should pay more attention to code smells, as it will not only make the code quality better, but will also help them minimize the number of merge conflicts they need to resolve.

Practitioners, when investigating the root cause of a merge conflict can start by looking for smelly program elements in the code. Moreover, since changes that involve entities containing code smells are more likely to lead to semantic merge conflicts, integrators (or code reviewers) should pay particular attention to and attempt to remove code smells when reviewing commits. Practitioners should also pay attention to "good" software engineering processes when they deal with smelly program elements. For example, when changes are being made to smelly parts of the code base developers should merge more frequently and perform more thorough code reviews.

Our results show that code smells are a good predictor of merge conflict and the level of difficulty of that conflict. Therefore, tool builders can use the information of incidence of code smells to support distributed work – either in predicting likelihood of conflicts or their difficulty. Code smells can also be used as a factor to schedule tasks (e.g., program elements that have code smells should not be edited in parallel) or assign tasks (e.g., developers with higher experience should work on smelly program elements).

Our results have implications for researchers. Since code smells together with merge conflicts can predict bugginess, researchers can use this information in bug prediction models to increase their effectiveness. To the best of our knowledge, no merge conflict prediction tool exists. Our results show that code smells have a strong association with merge conflicts, therefore, researchers can use this information to predict impending merge conflicts. Our results also have implications in testing. For example, increasing the test coverage of smelly lines that were involved in a merge conflict can be used as an objective/fitness function in the field of search based software engineering.

2.6 Threats to Validity

Our research findings may be subject to the concerns that we list below. We have taken all possible steps to neutralize the impacts of these possible threats, but some couldn't be mitigated and it's possible that our mitigation strategies may not have been effective.

Bias due to sampling: Our samples have been from a single source - Github. This may be a source of bias, and our findings may be limited to open source programs from Github and not generalizable to commercial programs. However, the threat is minimal since we analyze a large number of projects spanning eight different domains.

Bias due to tools used: The smell detection tool we used uses static code analysis to identify smells and research shows that code smells that are "intrinsically historical" such as *Divergent Change*, *Shotgun Surgery* and *Parallel Inheritance* are difficult to detect by just exploiting static source code analysis [122]. So the number occurrence of such "intrinsically historical" smells should be different when historical information based smell detection technique is used.

Secondly, we used the Gumtree algorithm [**?**] for tracking program elements across

commits. However, the algorithm used is unable to track program elements across renames or movement to another folder. Further, refactoring that involves modification of scope, such as moving the code out of the current compilation unit also causes the algorithm to lose track of the program element after refactoring.

Bias Due to using classifiers: We use machine learning to group conflicts into the two categories, and to determine whether a commit was a bug-fix. As with any classifier, we have some mislabeling. While our results do not require those results to be anywhere near perfect, this threat is low as our classifiers have good F1-measure and high precision.

Regarding the bug-fix classifier, our recall and precision measures are on par with past work [**?**]. Since our analysis relies on relative count of bug fixes, as long as we do not systematically undercount bug fixes, our results are valid.

Finally, we have assumed that all bugs were found and fixed by developers when we use it as a metric of bugginess of merged lines of code. This may not always be true, and hence our results are conservative.

2.7 Conclusions

In this paper, we study the history of 143 open source projects, from which we extract 6,979 merge conflicts to see if there is any correlation between code smells and merge conflicts. We found that entities involved in merge conflicts contain almost 3 times more code smells than non-conflicting entities.

To have a better understanding of the effect of code smells on merge conflicts, we categorized conflicts into *semantic* conflicts – changes to the AST and hard to resolve – and *non-semantic* – changes that are cosmetic. We found two method-level code smells (*Blob Operation* and *Internal Duplication*) to be significantly correlated with *semantic* conflicts. More specifically, methods that contained the *Blob Operation* and *Internal Duplication*

smells were more likely to be involved in a *semantic* merge conflict, by 1.77 times and 1.55 times respectively. We also found that code smells have a significant impact on the final quality of the code. Count of code smells was a significant factor when we modeled the bugginess of lines of code involved in a merge conflict.

Our results show that code smells, thought to be a maintenance issue and often neglected by practitioners, have an immediate impact in how distributed development is managed. Their presence is not only associated with difficult merge conflicts (*semantic*), but also with the likely-hood of bugs getting introduced in the code base.

Acknowledgments

This work was funded in part by NSF IIS-1559657,CCF-1560526. This work was also funded in part by IBM. We would also like to thank the Oregon State University HCI group for their input and feedback on the research.

3 AN EMPIRICAL INVESTIGATION INTO MERGE CONFLICTS AND THEIR EFFECT ON SOFTWARE QUALITY

Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, Anita Sarma

Empirical Software Engineering, Volume 25, Issue 1, pages 562-590

3.1 Introduction

Modern software development effort is, more often than not, a team effort. The complexity of software projects, such as the Linux Kernel has grown exponentially over the past years, and the Kernel now stands at over 21 million lines of code [44]. Software of such complexity requires a large development team to develop and maintain; last year alone more than 1,500 people made over 70,000 individual code contributions to the Linux Kernel [44]. While most software systems are simpler than the Linux Kernel, the kind of distributed and collaborative work model which powers this project is broadly representative of how much of today's software development is done, especially in open source projects.

Modern Version Control Systems (VCS) have made parallel development easier by streamlining and coordinating code management and merging. For example, in Git, creating parallel branches or cloning an entire project can be done through a single command. This ability to create private development lines makes it easy for developers to experiment without impacting or being impacted by others' work. However, as is well known, isolation of private development lines can cause problems when changes are synchronized [31, 135, 50].

Merge conflicts can occur when developers make concurrent changes to the same code artifacts. The changes are generally authored by two different developers, but merge

conflicts can also happen between the edits of one developer. While automatic merge tools help, manual intervention is required when changes overlap. Resolving merge conflicts, even easy ones, can disrupt the flow of programing, forcing developers to shift their focus on the resolution process. Other times, a conflict resolution requires a deeper understanding of the program's structure and goals. For example, when a function's signature or parameters are changed, a developer first needs to understand the rationale behind the change, and any dependencies, before she can resolve the conflict. Prior work has found that in complex merges, developers may not have the expertise or knowledge to make the right decisions [118, 46], which might degrade the quality of the merged code.

Another problem associated with merge conflicts is that they often take the developer outside of their development process. For instance, a developer may follow an established process of peer review of code submissions, but the merged code maybe "cobbled" together. This could lead to an increased likelihood of bugs slipping through, as could an incomplete understanding of the nature of changes and dependencies.

Prior research has shown that merging long lived branches can lead to merge conflicts (when the VCS fails to merge the two branches due to current edits to the same lines, also called *direct* conflicts), or *indirect* conflicts (integration failures where the merged code does not compile, or tests fail) [125, 26, 142]. While others have shown that merge conflict occurrence is frequent [125], what is not known is whether the code emerging from the resolution of merge conflicts is sub-par, whether different types of merge conflicts affect the resultant code quality differently, and whether there are any differences in the merged code created by experienced developers' as compared to novices.

Currently, therefore, there is an important gap in our understanding of the nature of merge conflicts and the results of their resolution. Closing these gaps can help us not just build better tools and processes to help developers, but also develop a more nuanced and in depth understanding of the risks and problems associated with merge conflicts.

The goal of this paper is to *characterize the different types of merge conflicts and identify any effects they have on the quality of the resulting code.* We ask the following research questions:

- RQ1: What are the most common types of merge conflicts?
- RQ2: How likely is code resulting from a merge conflict to contain bugs?
- RQ3: What are the factors that affect the quality of the code resulting from a merge conflict resolution?
- RQ4: What are the resolution strategies used by developers when resolving merge conflicts?

To answer these questions, we analyzed a broad sample of 143 open source projects. From these projects, we collected 556,911 commits, 36,122 of which were merge commits. 19.32% of these merges, in turn, resulted in conflicts.

Our research identified six types of direct merge conflicts. Some types of conflicts require resolution that is trivial (e.g. formatting differences), while others are challenging (e.g. those that make changes to the underlying Abstract Syntax Tree). About 60% of conflict resolutions in our dataset involve changes to the Abstract Syntax Tree (AST) that are entangled (henceforth called SEMANTIC merge conflicts). We also found that code associated with a merge conflict is twice as likely to have a bug, and code associated with SEMANTIC merge conflicts are 26 times more likely to have a bug compared to code associated with other conflicts.

In summary, our contributions are:

- A taxonomy of merge conflicts;
- Empirical results that show that direct merge conflicts are indicative of changes that are bug-prone;

- Identification of factors that are associated with direct merge conflicts that are bug-prone;
- Identification of most commonly used resolution strategies.

3.2 Related Work

3.2.1 Merge conflicts

Researchers have looked at strategies for merging code, problems associated with them, and how to proactively avoid them in order to support collaborative development efforts. Mens et al. [110] present a survey on the state of the art of software merging. Their survey goes into depth regarding merge strategies and ways to reduce conflicts. However, it does not provide evidence of the kinds of problems merge conflicts present.

Empircal Studies: Merge conflicts are known to be costly [72, 125, 50]. They delay the project, requiring an examination of the conflict, and developing a consensus solution. Several empirical studies have detailed how merge conflicts are a problem, and the strategies that developers follow to evade having to resolve conflicts. For example, Perry et al. [125] found that increased parallel work, in addition to causing conflicts, can also lead to an increase in software defects. Developers are known to follow informal processes (e.g., check in partial code, email the team about impending changes) to avoid having to resolve conflicts when committing changes [25], or rush to commit their work in an effort to avoid being the developer who has to resolve the conflicts [50]. A developer may also choose to delay the incorporation of others' work, fearing that a conflict may be hard to resolve [50]. Such processes can have a detrimental effect on team productivity and morale. Our work is the first to investigate the root cause of a merge conflict, and to try and quantify how problematic merge conflicts are for developers.

Proactive conflict detection: There have been a number of papers that focus on

developing tools to proactively detect or avoid merge conflicts. Sarma et al. [135] and Ripley et al. [130] presented Palantír, a tool that helps make developers aware of changes to each other's workspaces. Similarly, Biehl et al. [24] presented FASTDash, and de Silva et al. [48] introduced Lighthouse, which also try to foster awareness among developers. Kasi et al. [90] presented Cassandra, a tool that schedules tasks in order to minimize the chance of conflicts occurring. Brun et al. [31, 32] developed Crystal, which identifies two different types of merges conflicts: *"textual,"* which are detected by the version control system's merge tool, and *"higher level conflicts,"* which are not detected until a build or test fails. The tool merges changes in a shadow repository as they are committed in order to catch these types of conflict as early as possible. Similarly, Guimarães et al. [74] introduced a technique to continuously merge changes in the IDE in order to detect merge conflicts as soon as possible. Servant et al. [139] presented a tool and visualization that enable developers to understand the impact of their changes, which can then prevent indirect conflicts. Dewan et al. [55] presented a software development model aimed at reducing conflicts by notifying developers when they work on the same file, and allowing them to collaborate when resolving conflicts. While proactive detection tools help developers prevent merge conflicts, it is not always possible (e.g. important security fixes cannot be delayed to avoid such a conflict). Our work aims to bring understanding of the merge conflicts that do occur.

Merge help: Researchers have investigated techniques to manage merging of changes, in order to more efficiently resolve conflicts, either in an automated way, or by preserving and presenting useful context for the developer trying to resolve the conflict. Apel et al. [16, 15] and Cavalcanti et al. [39] presented a new merging technique, semistructured merge, which considers the structure of the code being merged. Lippe et al. [101] presented Operation Based Merging, which, when merging, considers the changes performed, not just the end result. However, McKee et al. [108] have shown that developers do not

trust tools if they don't understand how they work, and they prefer to resolve merge conflicts manually. Our work can help researches focus on the merge conflicts that are more "painful" for developers.

Conflict categorization: Finally, researchers have looked at ways of categorizing conflicts. Sarma et al. [135] categorized conflicts as *direct conflicts*, when parallel files have been changed and *indirect conflicts*, when parallel changes to dependent files cause a conflict. Similarly, Brun et al. [31], distinguish between first level (textual) conflicts from second level (build and test failure) conflicts. Buckley et al. [33] proposed a taxonomy of changes based on properties like time of change, change history, artifact granularity etc. Their taxonomy deals with software changes in general or conflicts at a coarser level. We are intersted in creating a taxonomy that provides finer details about merge conflicts and the impact of their resolution. Finally, Accioly et al. [10] and Menezes [109] present a classification that considers the types of changes that generate the conflict. Our approach is different, as we are using a human-centered approach at identifying the root cause of a conflict. We are the first to propose and validate a categorization that looks at the root cause of a merge conflict.

3.2.2 Measuring software quality

One goal of our research is measuring the impact that conflicts have on software quality. Various measures of software quality have been proposed. Boehm et al. [28], and Gorton et al. [70], to mention a few, have explored measures including completeness, usability, testability, maintainability, reliability, efficiency, etc. Some of these metrics are difficult to measure, especially in the absence of requirement documents or other supporting information. Researchers have also used code smells as a measurement of software quality [103, 104], though smells are often focused on future maintainability issues.

3.2.3 Tracking code changes and conflicts

Our analysis requires mining the history of each line of code to determine if and when it was involved in a merge conflict, a software patch/upgrade, or a bug fix. Researchers have proposed various algorithms for tracking individual lines of code across versions of software. Canfora et al. [35] proposed an algorithm that uses Levenstein edit distance to compute similarity of lines, matching "chunks" of changed code. Zimmerman et al. [159] proposed annotation graphs which work at the region level for tracking lines. Godfrey et al. [69] described "origin analysis," a technique for tracking entities across multiple revisions of a code base by storing inexpensively computed and easily comparable "fingerprints" of interesting software entities in each revision of a file. These fingerprints can then be used to identify areas of the code that are likely to match before applying more expensive techniques to track code entities. Kim et al. [93] propose a seminal algorithm, SZZ, for tracking the origin of lines across changes. Finally, Falleri et al. [61] propose the GumTree algorithm for tracking changes at an AST level. This is the approach that we chose for our experiments.

3.3 Methodology

Our goal here is to quantify and categorize merge conflicts across a wide set of representative open source projects, and the fault-proneness of the resulting code (whether these changes were associated with bug fixes or other improvements). Additionally, we want to collect relevant metrics about the projects themselves, and the contributors (such as whether these contributors were part of the core development team or intermittent/new developers).

To achieve these goals, we first select a project sampling strategy that allows us to gather data from projects that are representative of the kind of development practice

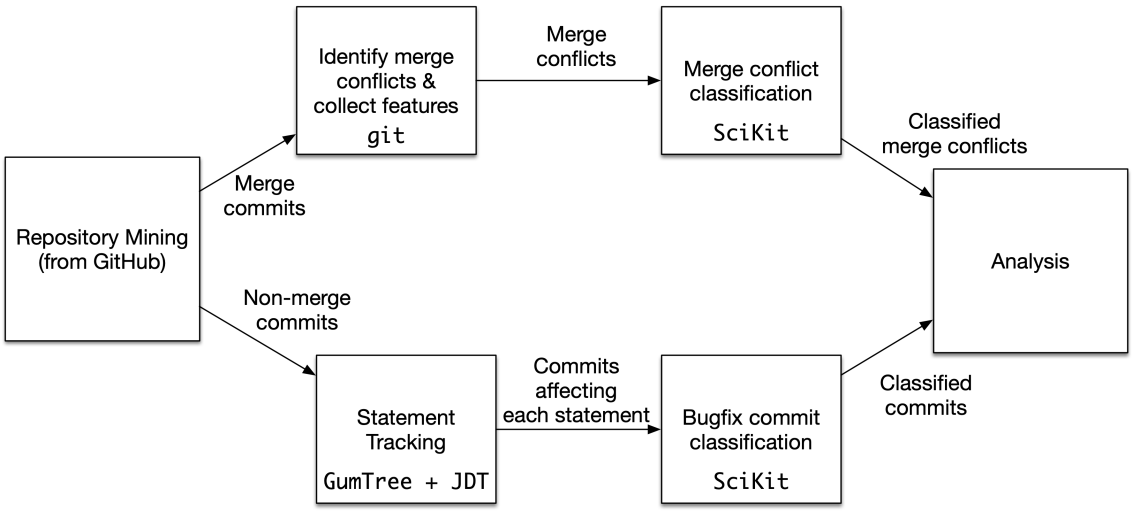

FIGURE 3.1: An overview of our data collection and analysis process. Tools used are listed using a `monospaced typeface` in the lower part of the boxes. JDT stands for Java Development Toolkit[3]

we are interested in. We then track the lines of code through versions and code merges in order to study how the code evolved, and which lines were associated with conflicts, updates, and bug fixes. Next, we determine the nature of code updates (e.g. was this a bug fix, or a new feature, etc.). In order to do this, we manually classify a subset of the commits and trained an automated classifier to classify the rest. Finally, we use the data to build a model to predict the total number of bug fixes that would occur on a conflicting line. The following subsections describe each of these steps in detail, and the overall picture is presented in Figure 3.1.

We also released the tooling we used for analyzing the projects. The tool for identifying merge conflicts is available here: `https://github.com/caiusb/conflict-detector`. The tool for collecting the metrics is available here: `https://github.com/caiusb/MergeConflictAnalysis`. Finally, our statement tracker is available here: `https://github.com/caiusb/statement-history`.

[3] `https://www.eclipse.org/jdt/`

3.3.1 Project Sampling

We collected projects from GitHub [6] for our empirical evaluation. Our goal was to ensure that the projects chosen offered a reasonably unbiased representation of modern software practices. We also tried to reduce the number of variables that could contribute to random noise during evaluation. With these goals in mind, we decided to focus on Java projects using the popular Maven build system [4]. This decision was influenced by the fact that Java is one of the most popular languages (according to the number of projects hosted on GitHub [6]), and the availability of analysis tools.

We started by randomly selecting 900 projects, the first to show up when using the GitHub search mechanism. From these, we eliminated aggregate projects (which could skew our results), leaving 500 projects. After eliminating projects in which we could not compile more than half of the merge commits (for reasons such as unavailable dependencies, or compilation errors due to syntax or bad configurations), 312 projects remained. Finally, we eliminated projects that our AST difference tool [61] could not handle. This left us with a total of 200 projects.

We followed the guidelines presented by Kalliamvakou et al. [89] for mining Git repositories. We removed projects that were too small, that is, having fewer than 10 files, or fewer than 500 lines of code, or those projects that were not active in the past 6 months. We also removed projects that had no merge conflicts. This was essential because there is a long tail of small and short-lived projects on GitHub, which include trial projects, projects with single author or no parallel development. Since these projects do not represent the kinds of development efforts we are interested in, we remove them from consideration. The thresholds chosen are based on similar studies [12].

Our final data set contained 143 projects across different domains. As a check on our sampling, we manually categorized the domains of our projects by looking at their project description, and using the categories used by Souza et al. [52]. Table 3.1 presents

TABLE 3.1: Distribution of projects by domain

Domain	Percentage
Development	61.98%
System Administration	12.66%
Communications	6.42%
Business & Enterprise	8.10%
Home & Education	3.11%
Security & Utilities	2.61%
Games	3.08%
Audio & Video	2.04%

the summary of the domains of the projects.

Next, we discuss the individual project characteristics. Table 3.2 provides a summary of features and other descriptive information of the projects in our study.

TABLE 3.2: Project characteristics

Dimension	Max	Min	Average	Std. dev.
LOC	542,571	751	75,795.04	105,280.1
Duration (Days)	6,386	42	1,674.54	1,112.11
# of Developers	105	4	72.76	83.19
Total Commits	30,519	16	3,894.48	5,070.73
Total Merges	4,916	1	252.60	522.73
Total Conflicts	227	1	25.86	39.49

The line counts (lines of code in the project) were taken from the version of the code that was in the repository on March 1st, 2016, and the duration is the number of days from the date of the first commits to March 1st, 2016[4]. The number of developers is the number of unique individuals that contributed at least once over the life of the project. Individuals were identified by the name in the "Author" field in each Git commit. The standard deviations are high, which suggest that our sample contains a diverse set of projects. This acts in favor of making our findings more generalizable.

From our sample of 143 projects we extracted 556,911 commits. This included 36,122 merge commits. Our data shows a high standard deviation in the total number

[4]Some projects may have migrated to GitHub from other platforms, so this is a lower-bound figure

of commits as well as merges. Therefore, we further investigate the distribution of merge commits (see Figure 3.2). From the figure we can see that merge commits are not scarce, as projects have an average of 252.6 merge commits. Out of all the merges, we identified 6,979 (19.32%) conflicts, as described in the next section.

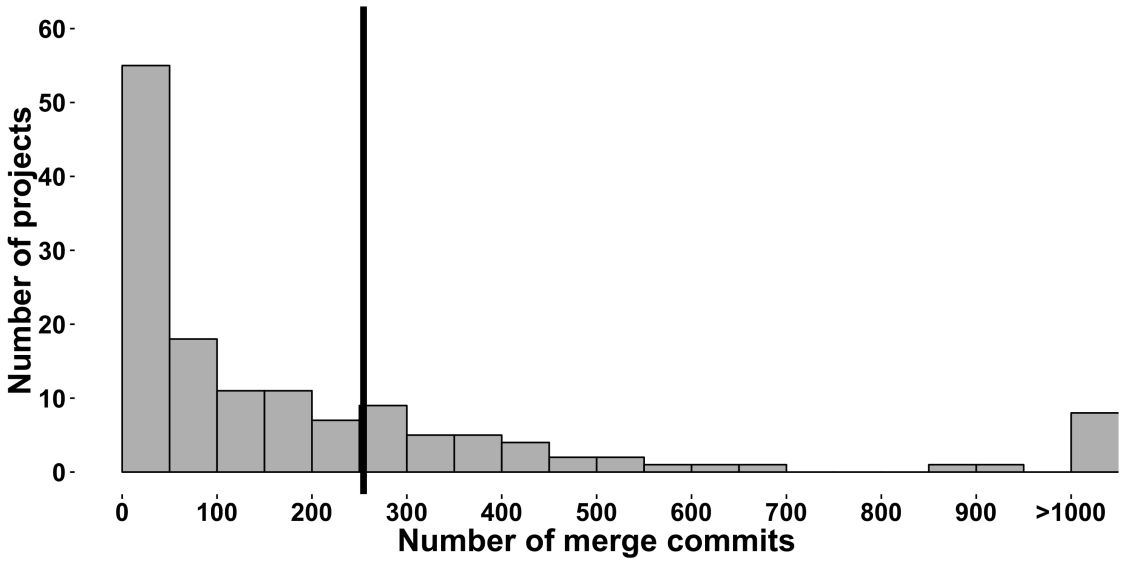

FIGURE 3.2: Distribution of merge commits. The vertical line represents the mean (252.60)

3.3.2 Conflict Identification

Since Git does not explicitly record information about merge conflicts, we recreate each merge in the corpus to determine if a conflict had occurred. We use Git's default algorithm, the *recursive merge strategy*, as this is likely to be most commonly used by the average Git project.

This also allows us to identify each conflicting commit and the affected file. More specifically, we used the `git merge` command that automatically merges the commits, and flags merges with overlapping changes as merge conflicts. The distribution of merge conflicts is shown in Figure 3.3. We see that projects experienced an average of 25 merge conflicts, or 19.32% of all merges. Merge conflicts, therefore, are a common part of the

developer experience (in our dataset).

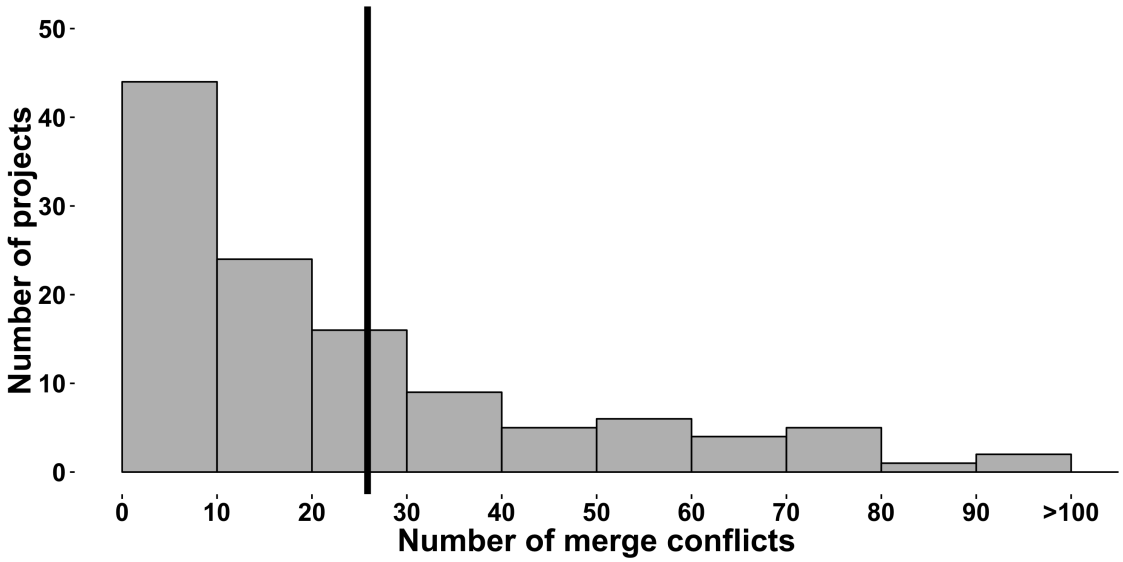

FIGURE 3.3: Distribution of merge conflicts. The vertical line represents the mean (25.86)

We then collect statistics regarding each file involved in a conflict, including files that have conflict and those that merged cleanly. We track the size of the changes being merged, the difference between the two branches (in terms of LOC, AST difference, and the number of methods and classes involved). We track the types of AST nodes involved (e.g., `BinaryExpression`, `MethodInvocation`, `ExpressionStatement` etc.) There were 81 node types in total. We use the Gumtree algorithm [61] to determine the AST differences. We also collect meta information about the merge, for example, if the change was merged into the master branch or not. Finally, we track the number of authors involved in the merge.

In this the paper, we only analyze merge conflicts, that are detected by Git, and we do not consider indirect conflicts, that are not detected by Git, but are noticeable because of build or test failures.

TABLE 3.3: Conflict categories. A semantic change is a change that affects the program logic.

Category	Definition	C.[i]	Example
SEMANTIC	Two conflicting semantic changes, where two different changes in the program logic overlap	NT[ii]	`https://github.com/zanata/zanata-server/commit/49fda3`
DISJOINT	Semantically unrelated changes that overlap textually	NT[ii]	`https://github.com/jdktomcat/cat/commit/0cbbd0`
DELETE	A conflict in which one of the branches deletes code modified on the other branch	NT[ii]	`https://github.com/osmandapp/Osmand/commit/defe2e`
FORMATTING	Conflicting changes due to formatting (whitespace changes)	T[iii]	`https://github.com/scudderfish/MSLoggerBase/commit/b495d3`
COMMENTS	Conflicting changes are limited to comments only	T[iii]	`https://github.com/scudderfish/MSLoggerBase/commit/b495d3`
OTHER	Not belonging to any of the above	T[iii]	

[i] Complexity [ii] Non-trivial [iii] Trivial

3.3.3 Conflict Types

To understand the root cause for each conflict we manually investigated and classified 606 randomly sampled commits. We classify each conflict based on the type of changes causing the merge conflict (e.g., whitespace or comment added vs. variable name changed). When classifying a conflict into a category, we chose the most "severe" category. As an example, if a merge contained conflicts in both comments (FORMATTING) and program logic (SEMANTIC), we classify it as SEMANTIC. We do so since we want to identify those conflicts that require the most developer reasoning (at least for some part of the conflict). The first two authors independently coded 300 of these commits using qualitative thematic coding [47]. Using Cohen's Kappa, they achieved an inter-rater agreement of 0.84 on 20% of the data. The first author then classified the remaining

306 commits. The codes and their definitions are given in Table 3.3. Detailed examples for each category can be found in the companion website [1]. Next, we will present an example of each category of NON-TRIVIAL merge conflicts.

Listing 1: An example of a SEMANTIC merge conflict. Taken from the Catdroid project, commit `c03c15`[5]

```
<<<<<<< HEAD
File projectXMLFile = new File(Utils.buildPath(Utils.
    buildProjectPath(projectName), Consts.PROJECTCODE_NAME));
SimpleDateFormat dateFormat = new SimpleDateFormat("dd.MM.
    yyyy␣HH:mm");
Date projectLastModificationDate = new Date(projectXMLFile.
    lastModified());
holder.dateChanged.setText(dateFormat.format(
    projectLastModificationDate));
=======
//set last changed:
SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yy␣HH:mm")
    ;
Date resultDate = new Date(projectData.lastChanged);
holder.dateChanged.setText(sdf.format(resultDate));
>>>>>>> 20b7c5e
```

Listing 1 presents a SEMANTIC conflict from our manually classified corpus. In this example, we have two refactorings (renaming `resultDate` to `projectLastModificationDate` and `sdf` to `dateFormat`) as well as introducing and using a new variable (`projectXMLFile`). The developer resolving this merge conflict would have to untangle the changes, in order to keep the correct refactorings and make sure that the new variable is used where required.

Listing 2: Example of a DISJOINT merge conflict. Taken from the Catdroid project, commit `3ba518`[6]

```
<<<<<<< HEAD
R.string.formula_editor_function_round,
R.string.formula_editor_function_true,
R.string.formula_editor_function_false };
=======
R.string.formula_editor_function_round,
R.string.formula_editor_function_mod };
>>>>>>> f665d49
```

[5] https://github.com/Catrobat/Catroid/commit/c03c15

[6] https://github.com/Catrobat/Catroid/commit/3ba518

Listing 2 presents an example of a DISJOINT merge conflict. In this example, two developers added different values to an existing enum. In this case, the resolution is straight forward, as the correct solution is having all the enum values added on both branches.

Listing 3: Git's output when encountering a DELETE merge conflict. Taken from the WordPress-Android project, commit `5fe68c`[7]

```
CONFLICT (modify/delete): src/org/wordpress/android/ui/
   notifications/BigBadgeFragment.java deleted in HEAD and
   modified in fff496c. Version fff496c of src/org/wordpress/
   android/ui/notifications/BigBadge
```

Finally, Listing 3 shows the Git's output when a file is deleted on one branch, and modified in the other. In this case, the developers is left with the modified file. It's up to them to find out why was the file deleted. It is also up to them to understand the changes that were made, and decide if to keep the file, or if the changes need to be reimplemented elsewhere, if the code was reorganized as part of a larger refactoring.

3.3.4 Conflict Type Classification

We use this set of 606 (10%) commits as training data for a machine learning classifier to use on the full set of merge commits. Each class is classified using an Adaptive Boost (AdaBoost) ensemble classifier, where we use 100 Decision Trees as weak classifiers. We choose AdaBoost as it had the highest performance (precision and recall) when compared with Support Vector machine (SVM) for our dataset. This was not surprising as ensemble of classifiers have shown significantly improved performance in other domains such as prediction [144]. We categorize all of the 6,979 conflicting commits using AdaBoost.

Performance of any prediction is dependent on the features used. Therefore, we wanted to use a comprehensive set of features. We performed a literature search, and we used factors from these 2 papers [110, 16] to decide on the final set. We also included

[7] `https://github.com/wordpress-mobile/WordPress-Android/commit/5fe68c`

TABLE 3.4: Features used to train the classifier

Feature	Description
AST Size[i]	The total number of AST Nodes involved in a merge conflict
LOC Size[i]	The sum of the LOC of the files involved in a conflict
AST diff size between the branches	The difference in number of AST nodes involved in the conflict
AST diff branch-solution[i]	The difference in AST nodes between a branch and the merge conflict resolution solution
LOC diff size between the branches	The total number of LOC involved in the conflict
LOC diff branch-solution[i]	The difference in AST nodes between a branch and the merge conflict resolution solution
AST Size of the solution	The total number of nodes for the solved merge conflict (only the files affected by a conflict)
LOC Size of the solution	The LOC size of the solved merge conflict (only the files affected by a conflict)
# authors	The number of authors whose changes are involved in a merge conflict
Merged in master	*"True"* if the branch was merged in the master branch, *"False"* otherwise (was merged in a different branch)
Branch time[i]	The timestamp of the last commit on each branch
Solution time	The timestamp of the merge conflict resolution commit
# methods	The total number of methods involved in the merge conflict
# classes	The total number of classes involved in a merge conflict
# statements	The total number of statements involved in a merge conflict.
AST Nodes in conflict	The total number of AST nodes involved in the merge conflict.
Is AST Conflict	*"True"* if the merge conflict is at the AST level.

[i] Collected for both branches

features that are known to influence the comprehension of a program, such as changes size and the spread of the change (number of affected program elements).

We gathered these factors from either the Git repository, or we derived them by analyzing the source code, when the factors are related to the process and code metrics (characterized in numerical form)

To train the classifier we use a set of 24 features, including: the total size of the versions involved in a conflict, the size of the conflicting area, the number of statements, methods and classes involved in the conflict. The complete list can be found in Table 3.4 and in our companion website [1]. The features were chosen based on the existing literature. We also considered factors that are known to influence the comprehension of a program, as well as the authors' experience.

Our goal was to achieve high precision and recall. We use 10 fold cross-validation to test the performance of our classifier, measured using the F1-score. The F1-score considers precision and recall by taking their harmonic mean. The average F1-score of the 10 rounds for the conflict type classifier is 0.64, the precision is high at 0.75. The F1-score is defined as the harmonic mean between the precision and recall scores.

We present the confusion matrix in Figure 3.4. The diagonal elements represent the number of points for which the predicted label is equal to the true label, while off-diagonal elements are those that are mislabeled by the classifier. While our classifier has decent results, it's worth noting that there are some mislabeled commits. This is happening because for some categories (e.g. COMMENTS) less training data was available. We discuss this further in the Threats to Validity (Section 4.5).

3.3.5 Tracking statements

We needed to track statements that were involved in merge conflicts in order to track the eventual outcome of merge conflicts. We decided to use GumTree [61] for our analysis, as it allows us to track elements at an AST level. This way we can track only

Predicted classes \ Actual classes	Comments	Delete	Formatting	Other	Semantic	Disjoint
Comments	10	0	1	0	2	0
Delete	4	4	0	1	0	3
Formatting	13	0	3	2	30	2
Other	27	4	4	0	17	18
Semantic	69	0	9	5	219	23
Disjoint	41	2	5	7	68	14

FIGURE 3.4: The confusion matrix for our classifier

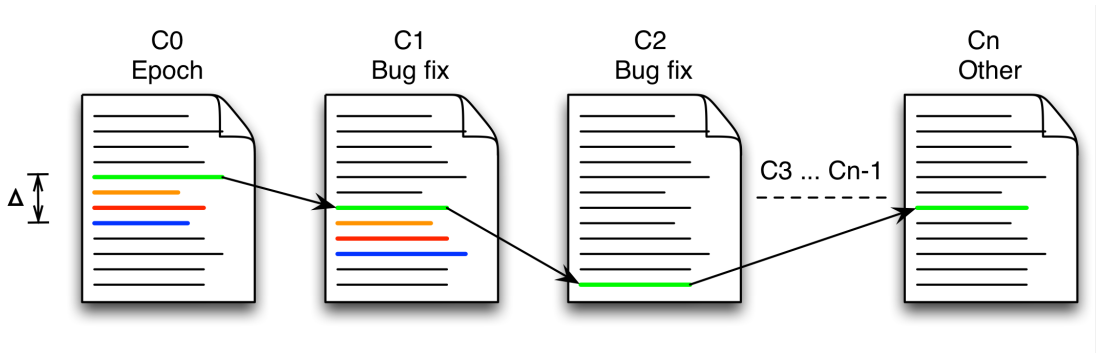

FIGURE 3.5: An overview of the tracking algorithm. The lines marked with Δ represent the conflicting region of a merge. For each line we identify the AST nodes, and track all the modifications (at AST level) forward in time. We stop when we hit a commit that was classified as *Other*.

the elements that we are interested in (statements), and ignore other changes that do not effectively change the code. The GumTree algorithm works by determining if any AST node was changed, or had any children added, deleted or modified. The algorithm maps the correspondence between nodes in two different trees, which allows it to accurately track the history of the program elements. This algorithm has unique advantages over other line tracking algorithms, such as SZZ [93]. These advantages include: ignoring whitespace changes, tracking a node even if its position in the file changes (e.g. because lines have been added or deleted before the node of interest), and tracking nodes across refactorings, as long as the node stays within the same file. Using this technique, we can track a node

even when it has been moved, for example, because of an extract method refactoring.

For each statement of interest, we use the version of the source code at the point preceding the merge conflict as the starting point. We use it to identify the AST nodes corresponding to the line of interest. We consider the code as changed if the AST node was changed, or had children that were added, deleted or modified. An example of a change to AST node is: changing `int x = 0;` to `int y = 0;`. This modifies the AST node (`SimpleName`: `x` in the first snippet) by changing its `name` property (x to y). Similarly, the change `x = x +1;` to `x = 1;` modifies the node because it modifies it's subtree. The algorithm maps the correspondence between nodes in two different trees, which allows us to track the history of any statement.

AST differencing has three advantages over simple line based differencing. The first is that it ignores whitespace changes. Second, we are able to track a node even if its position in the file changes (e.g. lines are added or deleted before the node of interest). Third, we are able to track nodes across refactorings, as long as the node stays within the same file.

For each node involved in a conflict, we identified all future commits that touched the file containing that node. To do so, we track the Java statement that corresponds to the (changed) AST node. Note that in Java, there might be multiple statements in the same line (e.g., a large `if-else` statement block), therefore, it is important to track only the statements that corresponds to the changed AST node.

We then repeated the same analysis for statements that were not involved in a conflict to determine if there was any difference between lines associated with a merge conflict and those which were not.

3.3.6 Commit Classification

In order to answer research questions 2 and 3 we needed to group commits into one of two categories: (1) bug-fixes and improvements (modifying existing code), and (2) commits

that introduced new functionality (adding new code) or were related to documentation, test code etc.

We investigate bug-fixes as it gives an objective measure for the definition of quality for open source projects, which often lack detailed requirements, roadmaps, or even test harnesses. The more bug fixes and changes a line of code faces over a period of time, the more one can argue that that line of code was incomplete or poorly implemented [13]. As this measure works at the individual line level, just like merge conflicts, we decided to use it as our measure of code quality.

It is not always trivial to determine which category a commit falls under, especially when larger projects see a large amount of activity. Manual classification of commits was therefore not an option, and we decided to use machine learning techniques.

In order to build a classifier, we randomly selected and manually labeled a set of 1,500 commits. Two evaluators worked independently to classify the commits. Their datasets had a 33% overlap, which we used to calculate the inter-rater reliability. This gave us a Cohen's Kappa of 0.90. In our training dataset, the portion of bug-fixes was 46.30%, with 53.70% of the commits assigned to the "Other" category. Some keywords indicating bug-fixes or improvements were "Fix," "Bug," "Resolves," "Cleanup," "Optimize," and, "Simplify," together with their derivatives. Anything that did not fit into this pattern was marked as "Other."

Not all bug-fixing commits include these keywords or a direct reference to an issue-id; commit messages are written by the initial contributor, and there are few guidelines. A similar observation was made by Bird et al. [26], who performed an empirical study showing that bias could be introduced due to missing linkages between commits and bugs.

We trained a Naive-Bayes (NB) classifier and a Support Vector Machine (SVM) using the SciKit toolset [124]. The frequencies of the words in the commit message were used as the predictors. We used 10% of the data to train the classifier. We applied

the classifiers to the training data with 10-fold cross-validation. As before, we used the F1-score to measure and compare the performance of the models. The NB classifier outperformed the SVM. We used the NB classifier to classify the full set of 16,571 commits.

Table 3.5 has the quality indicator characteristics of the NB classifier. Tian et al. [147], suggest that for keyword-based classification the F1-score is usually around 0.55, which happened in our case. While our classifier is far from perfect, it is comparable to "good" classifiers in the literature. Further, we believe it is unlikely for the biases to have a confounding effect on our analysis. Since our analysis only relies on relative counts of bug-fixes for statements, as long as we do not systematically undercount bug-fixes for only some statements, our results should be valid. A manual inspection of the classification results did not show any evidence of systemic over- or under-counting.

TABLE 3.5: Details of the bugfix (Naive Bayes) classifier.

	Precision	Recall	F1-score
Bug-fix	0.63	0.43	0.51
Other	0.74	0.86	0.80

For each line of code resulting from a merge conflict, we count the number of future commits in which it appears, as long as those commits are identified as bug-fixes. We stop tracking when we encounter a commit that is classified as "Other" (see Figure 3.5, where we count commits C_1 and C_2, but not C_n). Our reasoning is that once an element has seen a change that is not a bug-fix, it is no longer fair to assume that subsequent bug fixes are associated with the original merge conflict.

3.3.7 Core authors identification

We categorize developers as core or non-core based on the amount of their contributions. This is because, typically, a small core team is responsible for more than 80% of contributions to open source projects [114]. Therefore, those developers who are in the core are likely those with higher experience. We calculate this by first splitting the project

history into quarters. We then identify those developers who had the most contributions in a quarter. We do so because in open source there is high developer turnover, or developers become inactive for periods of time, therefore, it is better to gauge experience in shorter time periods. We identify developers as core contributors by evaluating if they are in the top 20% (in terms of number of commits) of the developers in that quarter.

We note that some developers started as non-core and transitioned to core, whereas some went from active to inactive. Therefore, an author can switch between core and non-core across quarters, based on their levels of contribution, and vice-versa.

3.3.8 Regression analysis

In order to answer our third research question, we needed to build a regression model to identify the factors that impact the number of bug fixes occurring on lines of code resulting from merge conflicts. We use Generalized Linear Regression [41]. In our data, the dependent variable (count of bug fixes occurring on conflict lines) follows a Poisson distribution. Therefore, we use a Poisson regression model with a log linking function.

We include the following factors in our regression model: file dependencies, source code and change metrics, and author related metrics. For calculating the dependencies of the files that are involved in a conflict, we use *Understand* [8] to count the number of references to- and from other files. We collect this information as a proxy for the importance of the file. We assume that the more a file is referenced by other files, the more central that file is, and hence more important. Any change in these central files can increase the chance of a change being required in other files.

For each conflict commit, we record the information about the size of the change – the difference between the two merged branches, in terms of LOC, AST difference, and the number of methods and classes being affected. Our intuition is that larger changes should have a higher chance of causing a conflict. We also calculate the number of authors

who made commits to the branches that were merged.

After collecting these metrics, we checked for multi-collinearity using the Variance Inflation Factor (VIF) of each predictor in our model [41]. VIF describes the level of multicollinearity (correlation between predictors). A VIF score between 1 and 5 indicates moderate correlation with other factors, so we selected the predictors with VIF score threshold of 5. This step was necessary since the presence of highly correlated factors forces the estimated regression coefficient of one variable to depend on other predictor variables that are included in the model.

3.3.9 Identifying resolution strategies

Not all conflicts are created equal, and their resolution is dependent on the nature of the changes being merged. A developer can choose different strategies for resolving conflicts. The easiest strategy is to simply *select one* of the branches that is being merged. While this might work for trivial conflicts (such as FORMATTING), it might not work for more complex conflicts. In some cases, it might be possible to mix and match the changes by *interleaving* existing code from the two branches. However, in more complicated cases, a developer might need to *adapt* the code from both branches to successfully merge the changes.

In order to determine the strategy that developers had used, we look at the existing solutions to merge conflicts. For each conflict we perform a line difference between the solution and the tips of the two branches being merged. Lines that were different from one branch, but not the other are considered to have been selected by the developer for integration. Lines that are different from both branches implies that they have been changed in order to be successfully integrated. Based on these observations we identified the following three resolution strategies:

1. SELECT ONE: The solution is the same as one of the branches (the difference to one of the branches is 0);

2. INTERLEAVE: The solutions contains lines from both the branches; none of the lines were changed and no new lines were added (we can match each line in the solution to a line in one of the branches);

3. ADAPTED: Existing lines were changed or/and new lines were added.

3.4 Results

In the following section, we present our results structured around our four research questions.

3.4.1 Merge Conflict Characteristics (RQ1)

In our dataset, 19.32% of merges resulted in merge conflicts (6,979 merge conflicts in 36,122 merge commits). This means that *almost 1 in 5 merges resulted in conflicts that required human intervention*. Our results are similar to those found by Brun at al. [**?**] and Sarma et al. [135]. A merge conflict not only means that developers have to stop their work, reason about the conflicting changes, and figure out the best way to integrate the changes. It also creates a situation where it is possible for bugs to slip through if developers do not refactor and run test cases, or have their proposed (adapted) code peer reviewed.

Types of merge conflicts

To further understand how different types of merge conflicts can impact the source code, we examine the type of changes leading to a merge conflict (see Section 3.2 for discussion of categories and methodology). Our results are presented in Table 3.6, where we present both the results of the automated classifier, as well as the results from our manual classification of the 606 merge conflicts. The distributions of the automatically classified merge conflict types match the distributions of our manual labeling (training

data), showing the efficacy of the automated classifier.

TABLE 3.6: Merge conflict types and their frequency of occurrence

Category	# of conflicts	% of total (classifier)	% of total (training)	Δ
Semantic	4,150	59.46%	50.86%	+8.6%
Disjoint	1,014	14.53%	22.74%	-8.21%
Delete	86	1.24%	2.52%	-1.28%
Formatting	1,620	23.21%	11.84%	+11.35%
Comments	42	0.60%	2.21%	-1.61%
Other	67	0.96%	3.31%	-2.40%

The resolution of the first two types of merge conflicts (Semantic and Disjoint) requires understanding the program logic of the changes in order to successfully resolve the merge conflict. We find that the most common type of merge conflict is Semantic (59.46% of all merge conflicts) where changes are entangled. 14.53% of merge conflicts emerge from concurrent changes to program logic that do not interact with each other and can co-exist (Disjoint). This means that in the vast majority (at least 75.23% of cases, for the Semantic, Disjoint and Delete categories) of merge conflicts, a developer needs to reflect on the program logic when integrating changes, and that for the majority of merge conflicts (at least 59.46% of cases, which represents the Semantic category), new code has to be written.

Merge conflicts due to lines of code being deleted are easier to merge, but still require the developer to reason about the deletion. Delete conflicts constituted a small percentage (1.24%) of all merge conflicts.

Although resolving merge conflicts due to Formatting changes (23.21% of all merge conflicts) or Comments (0.60% of all merge conflicts) require human intervention, they are easier and less risky to resolve since they do not affect the programs' functionality. It is interesting to note that merge conflicts caused by comments are the rarest of all — alluding to the fact that inline comments are rarely changed (if added at all), and most likely only updated when the code is changed.

Change types

We first examine the kinds of changes that are associated with merge conflicts to understand the types of tasks that are most associated with merge conflicts. We were curious to see if merge conflicts arose from behaviors like multiple developers fixing the same bug. We found this not to be the case. The majority of the changes involved in merge conflicts were non-bug fix changes (see Figure 3.6).

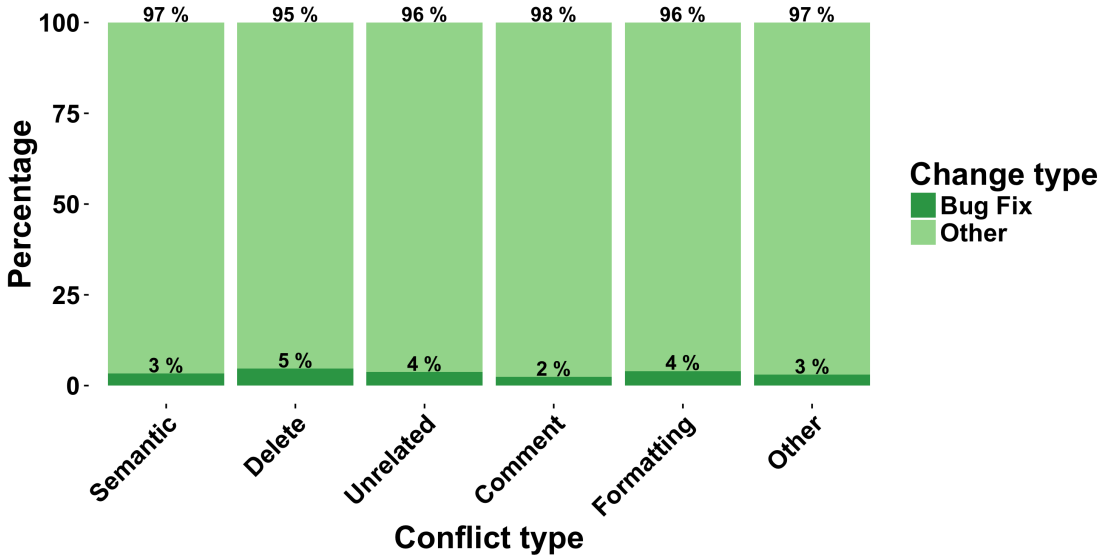

FIGURE 3.6: Frequency of change type by merge conflict type

Merge Conflict characteristics

Next we characterize merge conflicts based on the type and size of changes, the number and type of developers making the changes (Table 3.7). For the former, we determine whether the merger conflicts are non-trivial, that is, if they were associated with changes to the AST. The AST difference captures the impact of changes and is a good estimator of the amount of information a developer must process when resolving merge conflicts. The average size of a Java file is 4,845 AST Nodes, and the maximum is 168,790 AST Nodes. We find that DELETE merge conflicts are associated with the highest

AST difference (median of 1,662 nodes, which is 34% of the average file). We believe that a majority of these changes are associated with refactoring, a result of "chunks of code" that are moved, sometimes across files. Since these (cross-file) moves get counted twice (once for the removal, and once for the addition), it inflates the AST node numbers and LOC differences.

TABLE 3.7: Characteristics of merges conflicts

Conflict Category	Median AST diff (% of median)	Median LOC diff	Median # authors	Overall core developers %
Semantic	387 (8%)	2,351.0	4	89.08%
Disjoint	88 (2%)	1,037.0	3	89.65%
Delete	1,662 (34%)	16,102.5	2	86.01%
Formatting	0 (0%)	62.0	2	91.67%
Comments	19 (< 1%)	427.5	3	80.00%
Other	6 (< 1%)	238.0	4	86.49%

Semantic merge conflicts also led to large AST differences (median 387 nodes). This is intuitive because larger changes have a higher chance of having semantic interactions. In our data set, independent (Disjoint) changes (affecting a median of 88 AST nodes) are not as complex as those in the prior two categories. As expected Formatting merge conflicts have a median AST difference of 0. Comments merge conflicts also have AST node differences (median of 19). This is because Eclipse's JDT has AST nodes for comments. The LOC differences mirror the AST differences.

The median number of authors involved in conflicting commits ranged from 2 to 4. Note that Semantic merge conflicts included changes from a median of four developers, which means that the developer resolving the merge conflict has to reason about changes made by multiple developers — not an easy task. Even Disjoint merge conflicts included changes from a median of three developers.

Next, we look at the experience level of developers resolving merge conflicts. We classify developers as core or non-core based on their level of contributions as described in Section 3.3.8. We find that core members are involved in the majority of merge conflict

resolutions across all merge conflict types. This is intuitive, as core developers have more knowledge of the system and are therefore the best suited for solving merge conflicts. However, a number of merge conflicts are solved by non-core developers. This can still present problems, as they have less experience and insight into the code, which may make it difficult for them to correctly resolve merge conflicts.

In summary, we identified 6 types of merge conflicts, and the vast majority (75.20%) of merge conflicts impact the program logic, and therefore, require reasoning about the goals of the changes and the best way to integrate them.

3.4.2 Merge Conflicts and Code Quality (RQ2)

To answer RQ2 we examine whether the changes involved in merge conflicts are more likely to contain bugs than non-conflicting merges. We perform this analysis at the statement level; for every line of code involved in a merge we examine whether it was involved in a future bug fix based on the approach described in Sections 3.3.6 and 3.3.8. We use Fisher's Exact Test to compare the number of bug-fix commit between conflicting and non-conflicting merges. The results show that commits that are involved in a *merge conflict are 2.38 times more likely to contain a (future) bug fix* (Fisher's Exact Test, odds ratio = 2.38, $p < 0.05$).

Next we determine whether there are differences between different types of merge conflicts, as not all merge conflicts take the same effort to resolve. For our analysis we cluster merge conflicts into two groups:

- Non-trivial merge conflicts include the Semantic, Disjoint and Delete categories. Developers have to determine how to resolve the logical changes in these merge conflicts.

- Trivial merge conflicts include the other categories: Formatting, Comments and Other. Since these merge conflicts do not involve semantic changes, a trivial

merge resolution, such as choosing one version over the other, is feasible.

As before, we use Fisher's Exact Test to examine the difference between TRIVIAL and NON-TRIVIAL merge conflicts and future bug fixes. We find that NON-TRIVIAL *merge conflicts are 26.81 times more likely to need a bug fixing commit compared to lines involved in* TRIVIAL *merge conflicts* (Fisher's Exact Test, odds ratio = 26.81, $p < 0.05$.) This confirms that merge conflicts in the NON-TRIVIAL category are more challenging for developers to resolve, and might either introduce bugs or are associated with changes that themselves are likely to cause bugs.

We also ran Fisher's Exact Test and found a statistical difference between SEMANTIC and DISJOINT merge conflicts (odds ratio = 0.47, $p < 0.05$.) This means that merge conflicts arising from disjoint (independent) changes are half as likely to be buggy compared to SEMANTIC merge conflicts. There was no statistical difference between DELETE and SEMANTIC or DELETE and DISJOINT merge conflicts. There was no significant difference between the merge conflicts types in the TRIVIAL group. As we perform repeated tests, we use the Bonferroni adjustment for the α values.

In summary, we find that code that was involved in a merge conflict has a higher likelihood of being involved with a future bug. While some bugs are injected in the merge resolution, others were likely already there (e.g., changes in DISJOINT merge conflicts). In either case, a closer scrutiny of code involved in a merge conflict is warranted.

3.4.3 Factors Correlated with Bugs (RQ3)

As reported in the defect prediction literature, there are several factors that correlate with the bugginess of code. A critical factor is the size of the module under investigation [59]. Therefore, we posit that the size of a change in a merge should be a predictor of bug-proneness. Another factor that has been associated with defects is the number of committers — the "too many cooks" [151] phenomena. Finally, it has been noted that changes made to central files have a higher likelihood of reducing software quality [37].

Therefore, we model the total number of bug fixes to conflicting commits by the size of the change, file dependencies, number of authors, and their experience level.

We build a Poisson regression model with a log linking function. After filtering the factors with $VIF \leq 5$, we had a set of eight factors (Table 3.8) out of 43 total; all eight factors were significant at $p < 0.05$. These factors are: number of references to other files, number of references to the file involved in the merge, number of non-core contributor authors involved in the merge, number of authors involved in the merge, number of AST nodes changed, number of classes involved in the merge, number of methods involved in the merge and the number of LOCs changed.

TABLE 3.8: Poisson regression model predicting bug-fix occurrence

Factor	Coefficients
# of References to other files	0.08408
# of References to the file involved in merge	-0.03501
# of Non-core contributor	-1.898
# of authors w/ changes involved in the merge	-0.5634
# of classes involved in the conflict[8]	-0.1636
# of methods involved in the conflict[9]	0.3756
# of AST nodes changed	0.0007278
# of LOC changes	0.00003705

The McFadden Pseudo R-squared [79] of our model is 0.36. We calculated McFadden's Pseudo R-squared as a quality indicator of the model because there is no direct equivalent of R-squared for Poisson regression. The ordinary least square (OLS) regression approach to goodness-of-fit does not apply for Poisson regression. Moreover, pseudo R-squared values like McFadden's cannot be interpreted as one would interpret OLS R-squared values. McFadden's Pseudo R-squared values tend to be considerably lower than those of the R-squared. Values of 0.2 to 0.4 represent an excellent fit [79].

The effect of outward dependencies (number of external references from the file involved in a merge conflict) is positive, therefore changes that refer to external file (class) elements are more likely to contain bugs. We found a negative effect of inward depen-

dencies (number of references to the file involved in merge). We also found a negative coefficient for the size of the change (number of classes involved in the merge conflict), the number of authors involved in the change and the number of non-core contributor. The correlation between the number of authors with changes involved in a merge conflict and the number of future bugfixes is also negative.

Finally, in line with previous research [113, 92, 151] we find a positive effect of number of methods involved in the merge conflict, number of LOC changes and number of AST nodes changed on future bug fixing commit counts.

3.4.4 Resolution Strategies (RQ4)

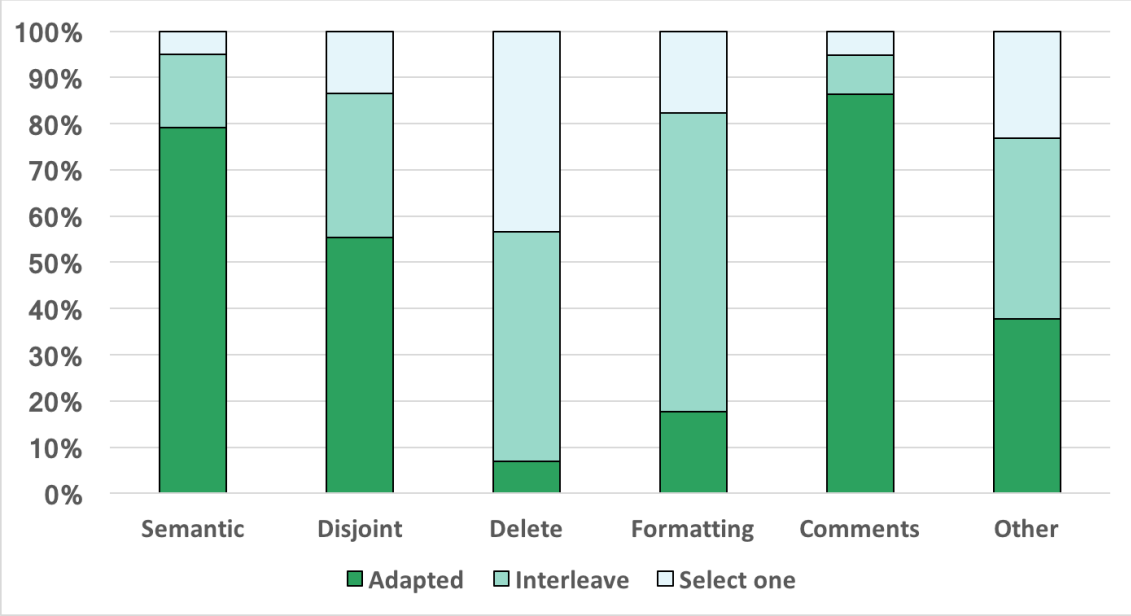

FIGURE 3.7: Resolution strategy based on commit type.

In this section, we analyze the strategies developers use when resolving merge conflicts. Overall, we find that ADAPTED was the most commonly used resolution strategy (60.82%), compared to INTERLEAVE (26.38%) and SELECT ONE (12.80%). Figure 3.7 presents the results detailed by merge conflict type. In the case of SEMANTIC merge conflicts, we observe that about 80% of the resolutions modify the existing code (ADAPTED)

in order to successfully resolve the merge conflict. As expected, merge conflicts that have a semantic nature require a bit of "coercion" for them to work together correctly.

Another category where a high percentage of resolutions use the ADAPTED strategy is COMMENTS. We hypothesize that this is because merge conflicts in comments are actually conflicting edits in two English texts, which require changing the text to make the sentence structure coherent when integrating the changes.

We see that in FORMATTING merge conflicts, the vast majority (over 80%) involve a resolution by either SELECT ONE or by INTERLEAVE strategies. The small amount of ADAPTED strategy is probably because developers were reformatting the code to integrate the changes.

An interesting result is that more than half of the merge conflicts that are generated by DISJOINT changes still require the developers to use ADAPTED in order to resolve it. Our hypothesis that while the changes themselves might be disjoint, simply interleaving them might not be sufficient.

We now investigate who resolves merge conflicts. We found that the vast majority of merge conflicts are resolved by one of the authors that contributed to one of the branches, as shown in Figure 3.8. Core developers solve most of them (89.47%), while non-core developers solve only a small fraction (10.53%). We also saw that about 2% of the merge conflicts were solved by THIRD PARTY developers (who have not contributed to any of the branches). We also analyze whether these third party contributors were core or non-core developers. We present the data in Table 3.9. For the SEMANTIC category, over 80% of the third party developers resolving a merge conflict were core contributors. We postulate that these developers are "integrators" for the project. Only a very small fraction of the merge conflicts are resolved by a third party developer who is a non-core developer. The same trend can be seen for the other merge categories as well.

In summary, we find that ADAPTED is the most used strategy for resolving merge

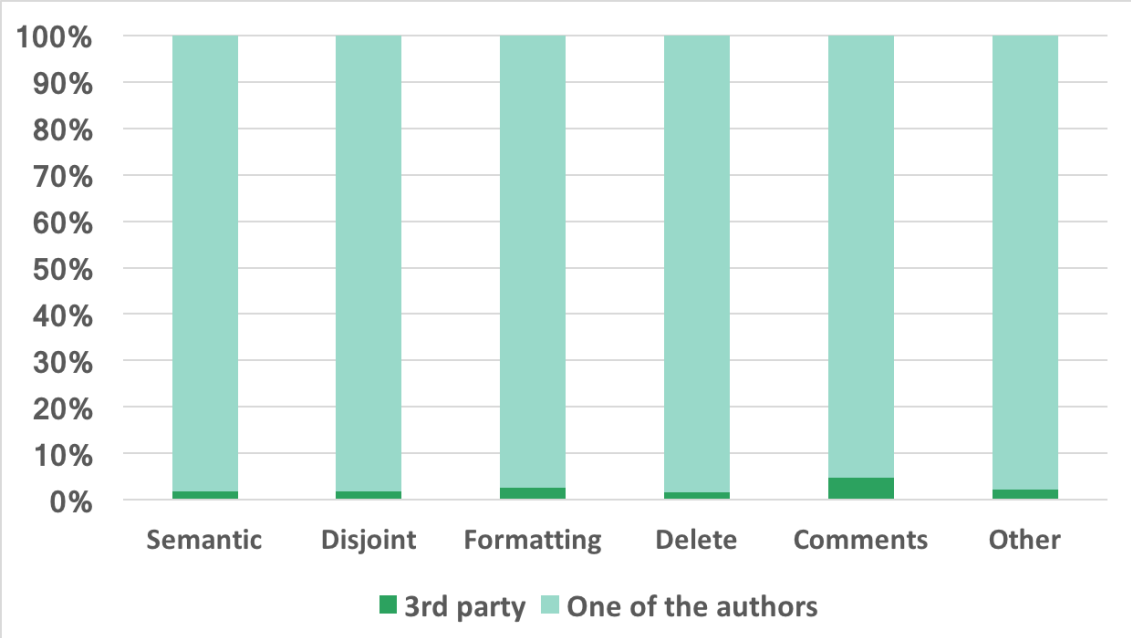

FIGURE 3.8: Developer category for each merge conflict category

TABLE 3.9: Distribution of developer types resolving merge conflicts

Category	Third party		One of the authors	
	Core	Non-core	Core	Non-core
Semantic	80.97%	19.03%	90.19%	9.80%
Disjoint	75.00%	25.00%	91.38%	8.62%
Delete	91.67%	8.33%	91.57	8.33 %
Formatting	81.25 %	18.75%	86.61%	13.39%
Comments	85.71%	14.29%	76.92%	23.08%
Other	76.06 %	23.94 %	88.47%	11.53%

conflicts. When dealing with Non-trivial merge conflicts the percentage increases significantly, possibly indicating the higher difficulty these merge conflicts pose.

3.5 Discussion

Merge conflicts are far from a solved problem. Despite advances in VCS tools and prescribed development practices, such as frequent commits and review workflows, merge

conflicts still occur frequently. We found that 1 out of every 5 merge commits in our dataset resulted in a merge conflict that required human intervention. Additionally, lines of code involved in a merge conflicts were 2x more likely to be buggy. Our taxonomy of merge conflicts shows that in 60% of cases, merge conflicts arise because of interacting, semantic changes. Furthermore, lines of code involved in these types of merge conflicts are 26x more likely to be buggy than in other types of merge conflicts.

Our results have implications for developers, tool builders, and researchers. Developers should focus their testing and code reviews on code resulting from merge conflicts, as they are more likely to be buggy. This is even more critical when the merge involves more changes, or when the conflicting changes span multiple methods.

Tool builders can use our merge conflict taxonomy and conflict (commit) classifier to automatically identify and flag lines of code resulting from merge conflicts, so that these receive higher test coverage or increased code review. Similarly, tools can use the information of the files, methods, lines of code involved in a merge conflict to prioritize testing, especially in the context of swarm testing [73, 80].

Our results indicate that 24% of merge conflicts (Formatting, Comments) are trivial to resolve. However, they still require human intervention, which interrupts developers' workflow. Tool builders should aim to better support automated code integration for such cases, so that it doesn't require human intervention. For example, Git, has the option of performing a merge that ignores whitespace changes. However, this is not a default option. Research has shown [85, 86, 145, 133] that users, and developers [123], are affected by default and anchoring biases. This indicates a preference of using the default options, instead of changing them. Perhaps the description of the option to ignore whitespace while merging should be more prominent, or enabled by default. As we see that a fairly large number of merge conflicts (over 20%) are caused by formatting changes, enabling the option by default could significantly reduce the number of merge conflicts

developers face.

Ours is the first empirical study to investigate the effects of merge conflicts on the bugginess of code. Some of our results are counterintuitive and present opportunity for new lines of research. For example, we observe that an increased number of non-core developers is correlated to fewer bug-fixes in the future. We posit that this happens because when non-core developers introduce changes, core developers review them when merging, "eliminating" some of the bugs in the process. We have observed this behavior while analyzing the merge conflict resolution strategies in Section 3.4.4 where we see that most merges are solved by core developers.

As shown in Section 3.4.3, we found a negative effect of inward dependencies (number of references to the file involved in merge.) We posit that changes to central files might have to pass more tests or that bugs are identified earlier because more people depend on them.

In the case of the number of authors with changes involved in the merge conflict, we speculate that we are seeing the effect of Linus' law, which states that *"many eyes make all bugs shallow"* [129]. We plan to perform further research to investigate how many of the bug fixes that were found after the merge conflict existed before the merge (were dormant), and how many were introduced as part of the merge conflict resolution. The negative correlation between number of classes and bugginess can be attributed to the fact that changes that span multiple classes are more likely to be refactorings, compared to other changes.

When looking at the resolution strategies, we find that developers primarily use ADAPTED as the resolution strategy when dealing with SEMANTIC merge conflicts. Lines involved in SEMANTIC merge conflicts are also most likely to be involved in future bug fixes.

We hypothesize that changes introduced by ADAPTED are not subject to the same

level of review as regular commits. Therefore, there is a chance that the ADAPTED merge resolution strategy itself could introduce new bugs. Also, they might contain preexisting bugs. Further studies are needed to answer these questions.

We also saw that non-core developers who were not involved in the development of either of the branches (3rd party) resolve merge conflicts. This is counterintuitive, as non-core developers are less likely to have an in depth understanding of the code base and they are more likely to introduce bugs. On top of that, not being involved in the development of either branch makes their ADAPTED resolution strategy bug prone. It might also be the case that a developer is classified as non-core, but she might have localized experience. Further investigation is needed to explore this.

We posit that because resolving merge conflicts take developers outside of the established workflow, they may not adhere to the strict testing or reviewing process that they would otherwise follow when making their original changes. Developers may pay more attention to changes that include multiple authors, because of which lines of code resulting from such merge resolution are negatively associated with bugginess. Further investigation of the review and testing processes that developers follow for merge resolution is needed to validate this idea.

Our study has included a broad spectrum of projects from different domains. However, they are all open source. Commercial projects have different constraints, development priorities, and practices. Future research comparing the differences between these two types of development with respect to the types of merge conflicts, merge resolution processes, and bugginess of resulting code needs to be performed.

3.6 Threats to Validity

Our empirical study, like any, has threats to validity. Where possible, we have taken steps to ameliorate their impact.

Identifying merges: Not all projects use `git merge` to integrate their change. Some use `rebase` or `squash` for that purpose. It is currently impossible to tell, from the public history alone, when rebasing or squashing occurs, as all we see is a clean, linear history. This presents the threat that we might under-sample the total number of merge conflicts that occur in practice. However, we believe this thread has minimal impacts on our results. This just reduces the number of merge conflicts available for analysis, which is compensated by the increased corpus size.

Sampling Bias: All our projects are sampled from a single source — GitHub [6], so our findings may be limited to open source programs. To ensure representativeness of our samples, we used search results from the Github repository of Java projects that use the Maven build system. So, our sample of programs could be biased by skew in the projects returned by Github. Github's selection mechanisms favoring projects based on some unknown criteria may be another source of error. While this makes our results less generalizable, the threat is minimal since we analyze a large number of projects: 500 projects were extracted, from which 143 were selected spanning eight different domains.

Fitness of our approach: We use Gumtree algorithm [**?**] to track program elements across commits when calculating AST differences. Gumtree, however, does not track program elements across renames or moves to other folders. Despite this drawback, Gumtree is a robust algorithm used to track refactoring and moves within the same file, and is better than line differencing tools.

Machine learning classifiers: We use machine learning to group merger conflicts into the six categories, and to determine whether a commit was a bug-fix. As with any

classifier, we may have some mislabeling. This threat is low as our classifiers have good F1-measure and high precision. The confusion matrix (Figure 3) shows that the highest misclassification is likely to happen when SEMANTIC merge conflicts are identified as DISJOINT merge conflicts. This does not affect our findings for RQ2 (effect of merge conflict on code quality) since both (SEMANTIC and DISJOINT) merge conflict types are in the same NON-TRIVIAL category. Also, our misclassification is asymmetrical because we are classifying more TRIVIAL (COMMENTS) merge conflicts as NON-TRIVIAL (SEMANTIC, DISJOINT) than vice versa. So this makes our results conservative, because any misclassified NON-TRIVIAL commits reduces the future bug-fix count used to answer RQ2. Regarding the bug-fix classifier, our recall and precision measures are on par with past work [**?**]. Since our analysis relies on relative count of bug fixes, as long as we do not systematically undercount bug fixes, our results are valid.

Bug-fix commit categorization: It's not always possible to untangle bug fixing changes from other kinds (refactorings, formatting, even adding new features). The same is true of *other* commit types, where bug fixes might have "crept in." As mention before, as long as we don's systematically over- or under-count the number of bug-fixing commits, our results should still be valid.

Finally, we have assumed that all bugs were found and fixed by developers when we use it as a metric of bugginess of merged lines of code. This may not always be true, and hence our results are conservative.

3.7 Conclusions

Our empirical study spanning 143 open source projects found that merge conflicts occur frequently. Moreover, code involved in a conflict had a 2x higher chance of being buggy. To create a better understanding of the merge conflicts, their frequency, and

impact, we create a taxonomy of conflicts, which includes six categories of conflicts depending on the type of resolution effort needed. We found about 74% of merge conflicts include interacting semantic changes (to the underlying AST). Moreover, code resulting from these conflicts was 26x times more likely to be buggy as compared to that from other conflicts. They are also more likely to be solved using an ADAPTED strategy, compared to other types of conflicts.

Our analysis of the factors associated with merge conflicts show that conflicts that involved files with high (outward) dependencies were correlated with more bugs. However, inward dependencies, number of non-core contributors, and the number of authors were associated with fewer bugs.

We conjecture that the merge resolution process disrupts the normal review and testing workflow, which might be the reason why lines involved in a merge conflict have a higher likelihood of being buggy. We plan to perform more research to investigate the review and testing workflow around a merge resolution. We also plan to study the frequency and distribution of bugs that are actually introduced as part of the resolution process as opposed to bugs that were dormant and remaining in the original code.

3.8 Acknowledgements

We would like to thank the Software Engineering and HCI research groups at Oregon State University for their input and feedback on this project. This work was in part made possible through support from the National Science Foundation (Grants IIS:1559657, CCF:1253786), and IBM.

4 STRUGGLES IN SENSEMAKING: A FIELD STUDY OF MERGE CONFLICT RESOLUTION BEHAVIOR

Caius Brindescu, Yenifer Ramirez, Anita Sarma, Carlos Jensen

Under review

4.1 Introduction

Version control systems are an essential component of collaborative software development. In Git and other version control systems, developers work on their changes in private workspaces, periodically synchronizing their changes with others by merging into the main development line. While many commits merge cleanly, parallel changes can overlap, resulting in merge conflicts. Prior work has found merge conflicts to regularly occur in development projects from 8% to 47% of the time [90, 12, 157, 31].

When merge conflicts occur, they disrupt the development process, especially when changes diverge significantly. This is because developers have to pause what they were doing to resolve merge conflicts, which (1) displaces them from their workflow; (2) saddles them with the potentially high-complexity cognitive task of understanding another developer's changes; and, (3) forces them into making code changes that may introduce bugs, which can be especially true for novice developers [50, 118]. Poorly-performed merge conflict resolutions have been known to cause integration errors [27], workflow disruptions, and jeopardize project efficiency and timelines [60].

Resolving merge conflicts is nontrivial. Practitioners are aware of the resolution "pains" and have developed workarounds to avoid having to resolve conflicts; e.g. sending out emails to the rest of the team, performing partial commits, or racing to finish changes [50, 36]. Unfortunately, these practices can cause changes to diverge even more,

further complicating merge conflict resolutions [31].

Existing tool support for resolving merge conflicts is still in its infancy. While support exists for visualizing the code under conflict, tools fail to communicate the causes, side-effects, and implications of conflicting code. This becomes critical when trying to merge large, tangled change sets. In such situations, developers are tasked with making sense of and seeing the connections between fragments of information pulled from a variety of different information sources such as parts of the code base, commit messages, issue trackers, etc.

Previous work has examined awareness and prediction of merge conflicts [135, 31, 74], proposed tools for resolving merge conflicts [119, 110], and has examined the difficulties inherent in merge conflict resolution processes [108, 117]. However, there is no body of knowledge about the in-situ steps developers take when resolving merge conflicts. We do not have an understanding of what kinds of information developers use, in what order, how they use the information from different sources, or where they struggle. Without answers to these questions, tools builders might be working under wrong assumptions, and researchers might miss opportunities for improving the state of the art.

To answer the above key questions, we conducted an in-situ observation of professional software developers resolving merge conflicts in their production code. The lead researcher colocated at the team's workplace during working hours, and when a developer faced a merge conflict they called the researcher to observe the resolution. We asked participants to use the think aloud protocol to capture their thoughts and reasoning [132]. We audio and screen recorded the resolution process (when participants consented) of 10 conflict resolutions resulting in 96 minutes of recordings.

In this paper, we take a first step towards understanding the merge conflict resolution process through a qualitative, empirical study using the lens of sensemaking. While this study only reports on the resolution of 10 merge conflicts, these observations are rich

in detail and provide us some initial data about patterns and challenges. Further research will be needed to address the generalizability question.

We structured our study around the following research questions:

- **RQ1**: *What types of information do software developers seek when resolving merge conflicts, and how?*
- **RQ2**: *How do they synthesize information to resolve a merge conflict?*

4.2 Background

In prior work, we broadly identified five stages developers go through while managing and resolving merge conflicts: (1) *development*: writing code, (2) *awareness*: becoming aware that a merge conflict has occurred, (3) *planning*: deciding how to resolve the conflict, (4) *resolution* resolving the conflict, and (5) *evaluation*: checking whether the code is still correct after the resolution [117]. The context of our present work is within the third through fifth stages of this model: after the developer has detected the merge conflict.

Because our objective with this work was to examine developer merge conflict resolution processes in detail, we selected a more complex model as the starting point for our analyses: Pirolli and Card's sensemaking model [126], presented in Figure 4.1. Sensemaking has been used to understand a number of software engineering tasks. For example, Grigoreanu et al. [71] successfully applied sensemaking to investigating end users' successes and failures at debugging spreadsheets. We focus our results on two internal loops of the sensemaking model: the *foraging loop* (the External Data Sources to the Evidence File stages) and the *sensemaking loop* (the Hypothesis Testing to the Reevaluation) stages. Within our context, the foraging loop focuses on how developers search, collect, filter and store information during merge conflict resolutions and the sensemaking loop focuses on how developers organize information they found and how they use it to

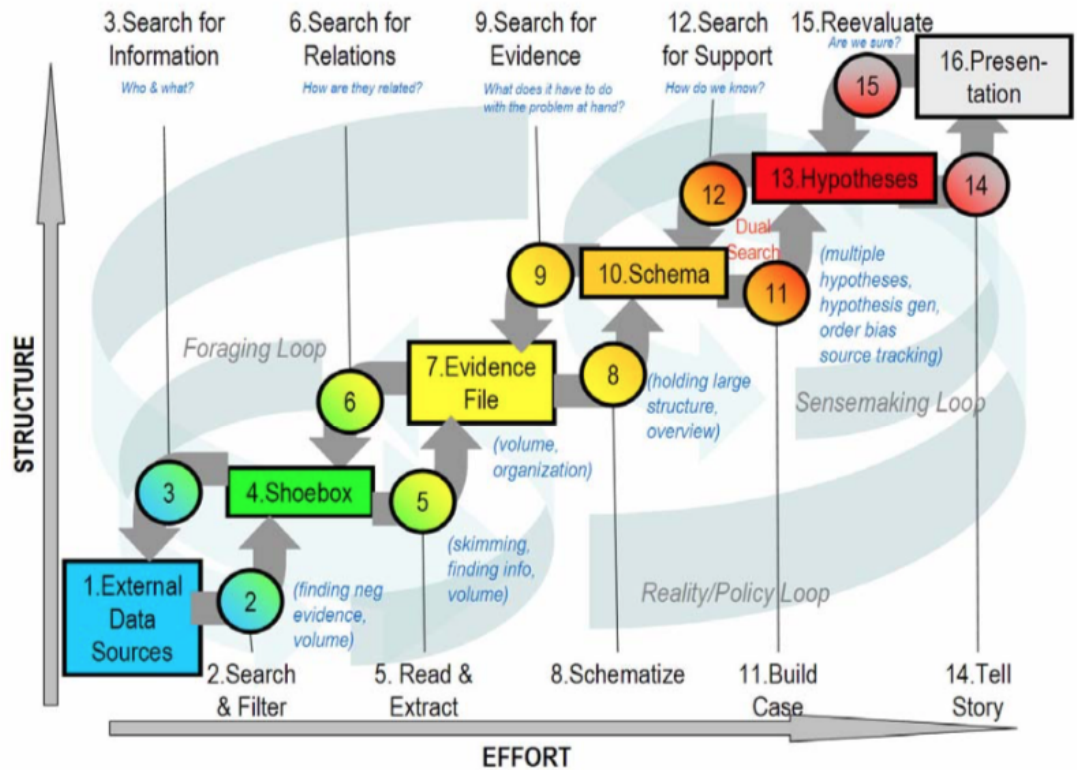

FIGURE 4.1: The sensemaking loop, as introduced by Pirolli and Card [126]

build a final solution.

4.3 Methodology

4.3.1 Participants and Data Collection

We performed an observational study of developers while they were resolving merge conflicts during their regular daily workflow. We chose this form of study because it allowed us to gather ground data about activities developers perform during the merge conflict resolution, on code they were familiar with (or had at least worked with). Because of the exploratory nature of this work, we wanted to gather large amounts of qualitative data to help us understand the "why" behind the actions we would observe. To that end, we asked participants to follow the think-aloud protocol by verbalizing their thoughts and

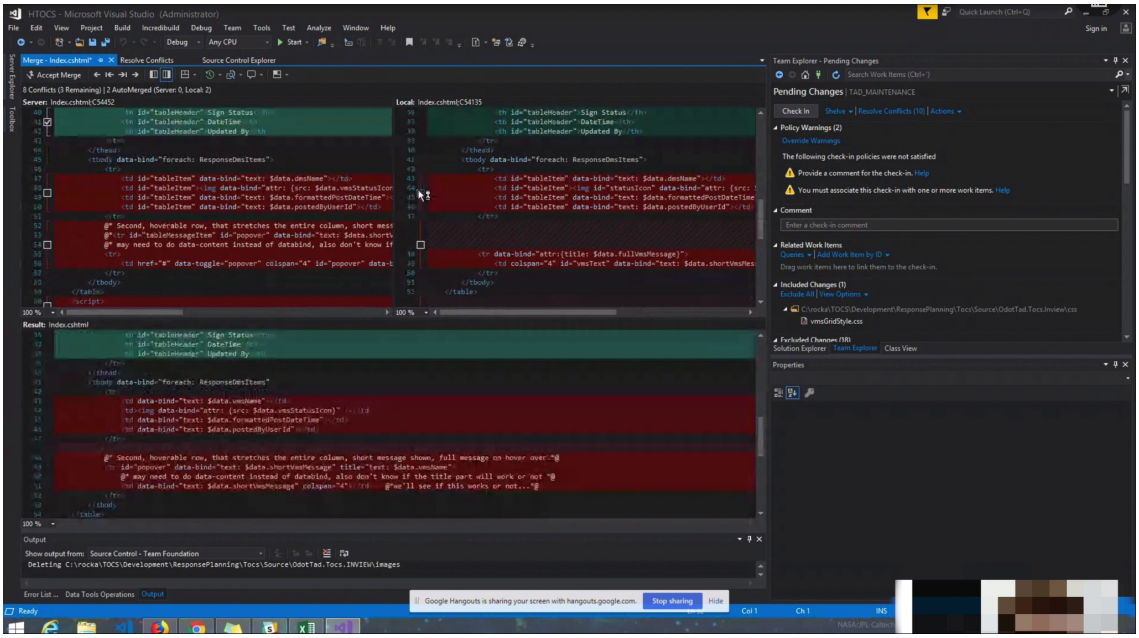

FIGURE 4.2: Example of a merge conflict (P5-C1) in the Visual Studio IDE. The IDE presents a *text only* view of the code, and it's the developers job to find why the changes conflict, as well as how to resolve it.

actions [82].

We collected the data through a university outsourcing development lab that worked on both commercial and Open-Source Projects. The team we studied consisted of approximately 20 part- and full-time developers, 12 of which signed up for our study. Part-time developers were generally students in the junior year, or higher, while the full-time developers were professionals, some with over 10 years of experience. The teams worked in an open-office layout on 3 development projects. The study focused on one of the projects, which was being developed for a state agency. We collected data across 17 days, from April 24, 2018 to May 11, 2018.

The study was designed to minimize disruption to the developers, and to maximize the external validity of the data we gathered. Before the study started, the researchers introduced themselves to the team and enrolled any developers who were willing to participate in the study. Overall, we observed 10 merge conflicts resolved by 7 participants.

The details of the participants can be seen in Table 4.1.

The first author was present in the office during working hours (9AM to 5PM, Monday through Friday). Although some developers worked outside those hours, the majority were in the office during this time frame. When a merge conflict occurred, the participant called the researcher to their workstation. The researcher then started a screen recording session via Google Hangouts and used a GoPro HERO5 Session to record the participant, the screen, and participant's verbalizations.

TABLE 4.1: Participant Demographics

Ptc.[i]	**Gender**	**Exp.**[ii]	**Language(s)**[iii]	**Proj. Exp.**[iv]
P1	M	7y 0m	C, C++, C#, JavaScript, Python	0y 6m
P2	M	2y 6m	C#, Python	0y 2m
P3	M	4y 0m	C++, C#, JavaScript	0y 1m
P4	M	10y 0m	C#	10y 0m
P5	W	4y 0m	C#, Python, SQL	0y 3m
P6	M	15y 7m	C#, .NET	5y 2m
P7	M	15y 0m	C#, .NET	10y 0m

[i] Ptc. = Participant [ii] Exp. = Years (y)/months (m) of software development experience [iii] Preferred programming language(s) [iv] Proj. Exp. = Years (y)/months (m) of experience contributing to the current project

During the study, we collected over 96 minutes of screen recording and verbalizations. We transcribed the audio of each merge conflict resolution and split each transcription into 10 second segments to prepare for qualitative coding.

4.3.2 Analysis

We defined a codebook of 6 *sensemaking steps*, which we adapted from Grigoreanu et al. [71]. As discussed in Section 4.2, they in turn adapted Pirolli and Card's Sensemaking framework [126] for end-user debugging. We first started with the 7 codes proposed by Pirolli and Card [126], but we iteratively refined the codebook. Our final set of sensemaking steps is presented in Table 4.2.

To help keep our data collection and analyses reliable and consistent, we omitted the *Hypothesis* and *Schema* steps and replaced them with a proxy: the HYPOTHESIS TESTING step. The HYPOTHESIS TESTING step is defined in terms of specific actions we could observe participants performing. The reason we adapted the sensemaking model in this way is because the Pirolli and Card steps are about what is going on inside the person's head, which can be difficult to detect. Even with a think-aloud protocol, we felt unable to reliably extract the details of person's conceptual *schema*, especially since individuals may not themselves understand what internal frameworks they're using to process and organize information. Similarly, participants also do not always state the *hypotheses* they're making.

Once our codebook was established, we randomly sampled 20% of the data from the 10 merge conflict resolutions and two authors independently coded the sample, achieving strong agreement in measured inter-rater reliability (IRR) (Cohen's K=0.82) [42]. For each segment, it was possible to assign multiple sensemaking steps if the Participant transitioned between steps during the time window. Once reaching agreement for IRR, the two authors divided the remaining data among themselves and individually coded the rest of the dataset. In total, we assigned 868 codes from our codebook to the merge conflict resolution screen recordings and verbalizations we gathered.

The first author also coded, by information source, the artifacts developers used during merge conflict resolutions. Our information source definitions are listed in Table 4.3.

TABLE 4.2: Codebook of Sensemaking Steps for Merge Conflict Resolution, adapted from Grigoreanu et al. [71]

Step	**Definition**	**Example**
External Data Sources	Participant looks at all the information sources available, including the source code, files under conflict, and version history	The participant investigates the list of conflicting files, or the participant looks at the files changed in a commit.
Shoebox	Participant opens a particular file to work on. Or they open the history view, or any other artifact they consider useful.	When the participant switches views, they are in the Shoebox step, as each view can be viewed as a Shoebox.
Evidence File	Participant reads or scrolls though a part of the source code (conflicting or not).	The participant reads the changes made on a particular file.
Hypothesis Testing	Participant edits code, or selects/deselects the same or parallel chunks (as viewed in the VS editor).	The Participant is editing the code, to resolve the conflict.
Presentation	Participant reads a chunk on either side, then moves on to the next chunk, either by scrolling, or by selecting a different chunk.	The Participant selects one version as correct when resolving a merge conflict.
Reevaluation	Participant runs the build, or performs testing (manually or automated). Similarly, if a participant comes back to a a resolved chunk, and edits/deselects it, etc. is also a reevaluation.	The Participant runs the application, or test suite, to check their resolution.

As this was an objective step, we did not need to calculate the IRR.

TABLE 4.3: Information sources developers use.

Info. Source	Description
Diff	The *diff* view presenting the difference between two versions of the same file (code artifact)
History	The commit history of the project, or of an individual file
Code	The source code itself.
Run	The output from the running the application/project
Build/Tests	Build and test output
Documentation	Reading external documentation, in issue tracking system, wikis etc.
Colleagues	Asking colleagues for specific information regarding the changes made, or the codebase

4.3.3 Pattern identification

We identified three types of patterns participants exhibited while resolving a merge conflict: (1) Information source packings (i.e., number of different information sources packed into a merge conflict resolution); (2) information usage patterns (i.e., path through the information sources during the merge conflict resolution); (3) sensemaking patterns (i.e., path through sensemaking steps during the merge conflict resolution).

To detect distinct information source packings, we used the k-means clustering algorithm included as part of the standard unsupervised learning package in R [81]. We found that a two-cluster partition resulted in the lowest variability within our dataset (within cluster sum of squares (SS) by cluster: $\frac{\text{between SS}}{\text{total SS}} = \frac{32.26667}{38.10000} = 84.7\%$). We identified 2 information source packings: Sparse and Dense (described in Section 4.4.2).

We used the same k–means clustering algorithm to detect patterns of information usage (within cluster sum of squares (SS) by cluster: $\frac{\text{between SS}}{\text{total SS}} = \frac{19.91523}{26.78481} = 74.4\%$). We identified 2 information usage patterns: Sequential and Interleaved (described in Section 4.4.2).

To detect patterns of sensemaking, we used a sliding time window to group steps

(transitions from one sensemaking step to the next) together into five-item transactions. We then looked for the most common transactions, and through this approach identified four sensemaking patterns (two novel): StuckForaging, HuntingForData, SkippingTheHypothesis, and QuickResolution (described in Section 4.4.3). To be considered a pattern, the same behavior had to appear five or more times, across at least two participants.

Some participants had periods of stalling, when they did not seem to be progressing through or transitioning between any of the sensemaking steps. While in some cases this was caused by our participants reading or inspecting an information source, it was hard to infer exactly which sensemaking processes were going on inside the participants' minds. To reduce noise in our data, we chose to eliminate any periods of "inactivity" from our pattern analysis. A period of "inactivity" is defined as a participant staying in the same sensemaking step for two or more consecutive steps.

4.4 Results

Here we first describe how participants' conflict resolution behavior can be mapped onto the sensemaking steps (Section 4.4.1). Using the sensemaking lens we then describe how participants sought information (RQ1, Section 4.4.2), and how they synthesized it (RQ2, Section 4.4.3).

4.4.1 Sensemaking steps in conflict resolution

We explain sensemaking terms that we use throughout the results section by using P5's conflict (P5-C1 in Figure 4.3) resolution behavior. The X-axis presents time (seconds) and the Y-axis the sensemaking steps through which participants progressed. Line color encodes the information sources they used (explained later in Table 4.3).

The first step in sensemaking is External Data Sources. In our context, this

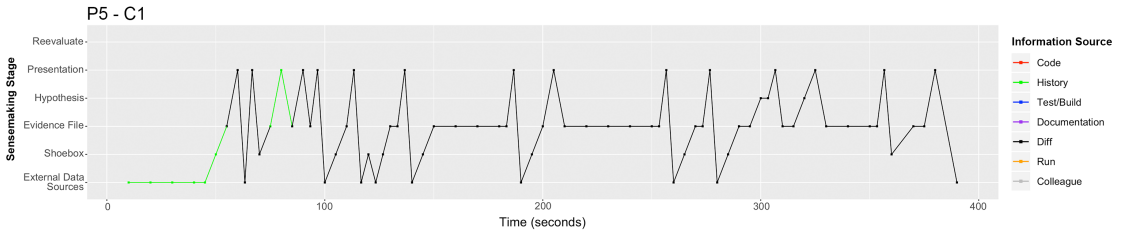

FIGURE 4.3: Activity Graph of Conflict P5-C1, showcasing a SPARSE information packing with a SEQUENTIAL access pattern. The red arrows indicate where the participant was having trouble finding the right information.

involves participants looking at a summary or overview of information sources to identify one which can help them understand why the conflict occurred. P5 started by investigating a summary of the commit *history* (green line) related to the merge conflict.

Next, they identified a commit from the conflict branch that was of interest. This reflects the second sensemaking step (SHOEBOX), which involves deciding on which information source (artifact) to examine first.

The next step is EVIDENCE FILE, which involves delving deeper into an artifact of interest (e.g., a file, commit message, diff region, etc.). In our example, P5 identified the relevant commit then delved deeper into the changes introduced by that commit by double-clicking the commit details in the IDE (moving to the EVIDENCE FILE step). Once the commit details were open, they examined two versions of the source code using the code differences view (black dot in Figure 4.3).

The fourth sensemaking step is HYPOTHESIS TESTING. This step entails forming a hypothesis about how to resolve the conflict, and then taking actions that show evidence of that hypothesis (requiring evidence through action was a modification we made to the Pirolli and Card sensemaking model [126], as thoughts were not always verbalized making them difficult to observe). However, P5 skipped the HYPOTHESIS TESTING step altogether, instead selecting a conflicting chunk to keep immediately after the EVIDENCE FILE step. They neither verbalized a specific hypothesis or resolution strategy, nor showed evidence of forming a hypothesis. They largely selected their changes over the conflicting

code as theirs was more recent. Therefore, we say that P5 moved directly from Evidence File to Presentation, skipping Hypothesis Testing.

The Presentation step reflects that the participant had finished the resolution (can be an entire conflict or a conflicting chunk) and the Reevaluation step includes activities to verify the resolution by either building the code, executing test cases, running the application or inspecting the output. Figure 4.3 reflects P5 considered the (chunk of conflict) resolved when they accepted their change and moved to the next conflicting piece of code. They did not perform any verification of their merged code in the entire resolution time period. This was likely since P5 largely kept their changes, they might not have felt it necessary to verify the merged code before committing it.

Using the sensemaking steps to represent conflict resolution behavior, allows us to find common patterns and steps where participants struggled. For example, while P5's overall progression was smooth, there were couple times when their progress stalled. At around 150 seconds, and after 200 seconds (as annotated by the red arrows in Figure 4.3), P5 "stalled" in the Evidence File step. This was because they were having trouble understanding the difference between the two versions of the code. It turned out to be a subtle change of one parameter in a method signature that was difficult to spot because the editor highlighted the whole line (instead of only the part that was different).

The next two subsections discusses in depth participants' conflict resolution behaviors.

4.4.2 Seeking Information (RQ1)

A merge conflict resolution begins with understanding where the conflict is and why it occurred. This requires some information foraging—needing to sift through mounds of code and its history, while mentally keeping track of the conflicting pieces of code and how they interact with each other as well as the rest of the codebase. To understand how developers seek information at this stage, we analyzed the different information sources

participants accessed and the patterns they used to access them.

How many information sources did participants use?

The number and type of information sources used in a merge conflict resolution is important because different information sources have different structures and exist in different places, requiring different approaches for accessing the information within. For example, understanding the "history" of changes is different than understanding why a "test" failed, which is very different from talking to a "colleague." Table 4.3 provides a list of information sources participants used. In addition to reviewing the conflicting changes to better understand the context of the changes, participants accessed the code, its history and outputs, external documentation, as well as talked with colleagues.

Consuming information from multiple sources requires increased cognitive effort, as developers need to piece together the different pieces of information. Here, we only distinguish between different types of information sources (i.e., even if a participant accessed 12 different code files, they are all combined to represent a single information source type—"code").

The number of information sources alone does not paint the full picture. What also matters is how these information sources are accessed. For example, frequent switching between multiple sources is challenging because every time developers switch between information sources they have to cognitively re-orient themselves, absorb information that's presented in a different way, perform window management, etc. Also, in complex conflict resolutions there is a higher chance of making mistakes, increasing the likelihood of errors creeping into the code.

Participants in our study used between 1 and 6 information sources with a maximum of 25 artifacts (see Table 4.4). The k-means clustering analysis (see Section 4.3) classified the access to information sources into two distinct categories—Sparse (up to 3 information sources) and Dense (between 4 and 6 sources). Table 4.4 column 5 shows the

two types of "information source packings" and column 6 shows the "information usage patterns."

TABLE 4.4: Information access strategies, including information source packing and usage (ordered on the number of information sources.)

Conflict (Partic.# - Conflict#)	# of Info. Sources	# of Artifacts	Conflict Resolution Duration (s)	Info. Source Packing	Info. Usage Pattern
P2-C1	1	1	40	SPARSE	—
P3-C1	1	2	80	SPARSE	—
P4-C1	2	2	190	SPARSE	INTER.
P5-C1	2	13	390	SPARSE	SEQ.
P1-C1	2	2	100	SPARSE	INTER.
P7-C2	3	3	130	SPARSE	SEQ.
P7-C3	4	4	440	DENSE	SEQ.
P7-C4	6	6	820	DENSE	SEQ.
P6-C1	6	24	1,430	DENSE	INTER.
P7-C1	6	25	2,190	DENSE	SEQ.

Sparse Packing Five participants used the SPARSE packing (see Table 4.4). These participants used an average of 1.8 information sources, with an average resolution time of 7 minutes and 18 seconds.

The P5-C1 conflict resolution is an example of a SPARSE information packing (see Figure 4.3), where P5 consulted only two information sources (the commit "history" in green, the "diff" in black). P5-C1 spanned 12 files, which P5 resolved by using the strategy: (1) view the list of conflicting files; (2) select a file to go into the code difference view; (3) read and scroll through the code to understand the conflict; and, (4) accept one of the (conflicting) code chunks. They were able to complete their resolution session in 6.5 minutes, processing each file in about 30 seconds on average.

A likely reason P5 was able to rapidly move through each file was because they were already familiar with the conflicting changes: *"This is me pretty much for all of it ... We changed how we decided to do things ... That's really common, ... It's probably just an automatic change ... "*. Moreover, the conflicting code "chunks" were straightforward, as

the changes were not semantically related.

In general, when participants used few information sources, they tended to mostly only refer to the source code that was under conflict and the relevant version history and commit messages.

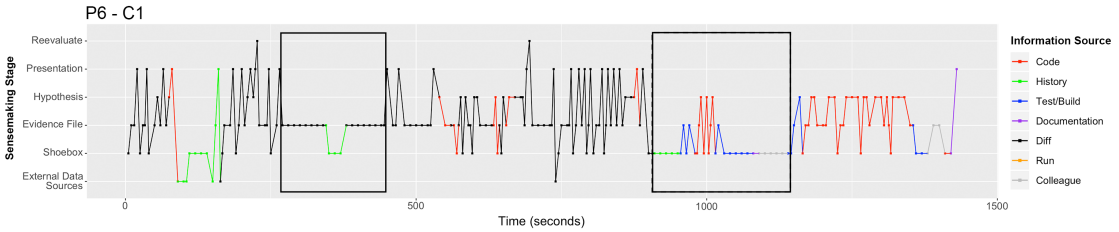

FIGURE 4.4: Activity Graph of Conflict P6-C1, showcasing a DENSE information packing with a INTERLEAVED access pattern. The black boxes indicate where participants were stuck in the STUCKFORAGING pattern

Dense Information Source Packing Resolution of conflicts with DENSE information source packing took longer to complete than SPARSE packing: the average was 12 minutes and 4 seconds vs. 7 minutes and 18 seconds.

As an example, during P6-C1 (shown in Figure 4.4), P6 used 24 artifacts from six different information sources to resolve the conflict. After scrolling through the "diff view" (in black), "source code files" (in red), and "commit history" (in green) for more than 15 minutes, P6 began committing fixes for *indirect* conflicts they had found. Here, we define indirect conflicts as instances where the merge succeeds, but the merged code fails to compile, run, or produce passing test results. The first indirect conflict P6 faced in P6-C1 was in configuration files for the runtime environment. To address this problem, P6 decided to re-implement the changes: *"Rather than having to go through all the compare, I'm just gonna go ahead and re-add our service, because I don't know what they did on their branch"*. P6 resorted to rewriting the new code entirely to avoid using the diff functionality provided by Visual Studio.

Next, P6 encountered a potential language version incompatibility problem, and

consulted a colleague: *"Do you know which version of TypeScript we're on?"* The colleague told them that, because of an ill-timed update, some machines ended up on an earlier TypeScript version—information P6 could not have gathered through the version control software tools they were using. P6 then asked their colleague several additional questions, which seemed to help P6 move fairly quickly and efficiently through the remainder of the resolution. In total, it took P6 about 24 minutes to resolve the conflict, the second-longest merge conflict resolution session we observed.

Indirect conflicts featured in multiple DENSE merge conflict resolution sessions we observed. Figuring out the context of these changes and how they impacted other parts of the code was a primary reason why resolving DENSE conflicts took longer than SPARSE conflicts (see Table 4.4).

How frequently do participants switch between artifacts?

Participants switched between information sources using two patterns: the INTERLEAVED pattern involves frequently switching back-and-forth and between information sources and artifacts; the SEQUENTIAL pattern involves less frequent switching and is more linear. The two patterns are derived from the k–means clustering previously described in Section 4.3.

Table 4.4 shows how each of the conflicts we observed was classified. There was no correlation between information source usage and information source packing. This is interesting because it indicates that (1) it's possible to move through many information sources fairly linearly; and, (2) even a small number of information sources can burden developers during merge conflict resolutions. Cases like the first could provide clues for tool developers about how to make merge conflict resolution less complex and more linear. Cases like the second could indicate there is something unhelpful about how some information sources are presented (e.g., they need to be broken apart, restructured, or related to other information sources in a better way).

Two of the merge conflict resolution sessions (P2-C1 and P3-C1) did not fit into our classification because they involved only one information source.

Sequential (Infrequent Switching) Four merge conflict resolutions used this pattern. For these resolutions, the participants' paths through different information sources and artifacts was relatively linear. On average, participants using this pattern switched information sources 3.73 times per minute, and used a single artifact for a median of about 2 minutes and 17 seconds. Within this classification, two of the merge conflict resolutions were Sparse (P5-C1 and P7-C2) and three were Dense (P7-C1, P7-C3, and P7-C4).

The conflict resolution of P5-C1 was an instance of Sequential Sparse. P5 switched between information sources infrequently (see Figure 4.3), and used each information source for a longer period of time than participants using the Interleaved pattern (e.g., P6-C1, shown in Figure 4.4). In this particular case, P5 had enough knowledge about the changes such that just seeing the code differences between the versions was sufficient for them to successfully resolve the conflict. P5-C1 was also relatively quick, taking only 6.5 minutes. The other Sequential Sparse merge conflict, P7-C2, was resolved even more quickly, in 2 minutes and 10 seconds.

P7-C1 is an example of a Sequential Dense merge conflict resolution illustrated in Figure 4.5. P7 had the most project experience (over 10 years) out of all our participants and 15 years of software development experience. Despite that, P7-C1 took the longest out of all merge conflict resolutions we observed: 36.5 minutes. Even though the resolution took a long time, P7 moved accessed the different information sources (and its artifacts) in a relatively linear fashion (roughly following this sequence: code, run, history, diff, text/build, documentation). P7 encountered multiple issues, including indirect conflicts and breaking changes that had to be left for a front-end developer to fix. P7 also encountered a database conflict that had to be resolved by switching to an entirely

new Visual Studio solution and a different place in version control: *"Now you've got two branches to have to keep in sync and you have to just know ... and that really sucks."* Our overall impression of P7-C1 was that P7 knew what to do (including when to leave the fix to others), allowing them to move from one information source to the next to resolve problems. Also, P7 had institutional knowledge about the project (and its past decisions) that newer developers to the project might not be aware of—and that information would not be available through any of the software tools the team was using.

Interleaved (Frequent Switching) Four instances of merge conflict resolutions used this pattern. On average, participants using this pattern switched information sources 6.55 times per minute, and used artifacts for a median of about 1 minute and 39 seconds. Within this classification, two of the merge conflict resolutions were Sparse (P1-C1 and P4-C1) and one was Dense (P6-C1).

In both the Interleaved Sparse merge conflict resolutions (P1-C1 and P4-C1), the participants used only the diff and test/build information sources, switched between information source artifacts frequently, and took less than 3.5 minutes to resolve. This gives us reason to conclude that these were very simple merge conflicts.

In contrast, the Interleaved Dense merge conflict, P6-C1, took almost 24 minutes to resolve (Figure 4.4). As mentioned previously, P6's conflict involved a change to configuration files. P6 had to piece together information from different sources and combine and contextualize that information within their own mind; gathering that information required referring back to the (information source) artifacts multiple times. The combination of having to look through many kinds of information sources, and having to look back and forth between information source artifacts, suggests that the merge conflict resolution tools available to P6 were particularly unhelpful at resolving that configuration-related merge conflict.

Implications

Once participants understood how to resolve their conflicts, implementing those fixes was relatively quick—it was sifting through information that took time. This was in part because the information sources participants had to go through were lengthy and detailed, but the information itself wasn't helpful: It was irrelevant to the fix but participants still had go through it, which wasted their time and didn't lead to a resolution. For example, we saw this with P5 who, even though they were familiar with the conflicts, had to spend several minutes confirming that lines of code marked as "conflicted" were actually identical or only differing in a variable name or value. One possible way tools could help with situations like these is by helping developers pay attention to information that is directly relevant to the conflict and potential resolutions (e.g., by hiding irrelevant code, commit messages, history, etc.)

Another way participants spent their time was trying to understand the connections between information sources. For example, during indirect merge conflicts, participants struggled to understand the implications of test failures and compile/runtime errors messages in terms of how they related to the conflict. P6 encountered this, and spent substantial time trying to track down what turned out to be a TypeScript version incompatibility—in the end, it wasn't the tool that led to resolution, it was a colleague who happened to be within shouting distance. Tools could help with situation like these by more intelligently interpreting errors, better understanding developers' machine state (e.g., configuration), and by keeping track of differences in machine state *between* machines of developers on the same project.

4.4.3 Synthesizing Information (RQ2)

Information seeking and the process of sensemaking are often entwined (see Figure 4.1). Once developers find information they need about a conflict during the *foraging loop*, they switch to the *sensemaking loop* to synthesize and contextualize the changes from

TABLE 4.5: Observed Sensemaking Patterns

Pattern Name	Description	# of instances
StuckForaging	The participant is stuck in the foraging loop, and does not progress past the Evidence File step for more than 5 steps.	13
HuntingForData	The participant is constantly switching between the Evidence File and Hypothesis Testing steps.	59
QuickResolution	The participant reaches the Presentation from the beginning of the resolution, or since the previous Presentation step, without backtracking through the sensemaking hierarchy.	27
SkippingTheHypothesis	When a participant skips the Hypothesis Testing step and moves straight to the Presentation or Reevaluation steps.	25

each branch with respect to the overall codebase and its history. This synthesis drives the potential resolution solution.

Four common sensemaking patterns emerged from our results: Two relate to when participants were "stuck" finding the right information (StuckForaging) or synthesizing it HuntingForData) and two that relate to the resolution of the conflict itself (QuickResolution, SkippingTheHypothesis). Table 4.5 defines these patterns and lists their number of occurrences. Recall, these patterns were identified based on the sensemaking steps participants followed and the sequence of those steps (Section 4.3). Note also that different instances of these sensemaking patterns can co-occur in a single conflict resolution instance. For example, P7 interleaved HuntingForData (Figure 4.5 (a)) with QuickResolution (Figure 4.5 (b)) when resolving P7-C1.

I am stuck finding the right information

Participants were in the StuckForaging pattern when they had difficulty gathering the information they needed—they were *stuck* in the foraging loop. Specifically, StuckForaging is when participants did not progress past the Evidence File step for

more than five steps. There were 13 instances of this pattern across five participants.

Participants were either stuck in the SHOEBOX step or the EVIDENCE FILE step (a majority were in the latter group). For example, P6 (when resolving P6-C1), was stuck in the SHOEBOX step trying to find the configuration problems generated by the merge conflict. Figure 4.4 shows the two instances where they were stuck (in black boxes). Earlier in the conflict (Figure 4.4 (a)), they were stuck in the EVIDENCE FILE step for 4 minutes, trying to understand what caused a large conflict (over 50 lines). The extensive code differences made it difficult to understand the changes and what was causing the conflict from the "diff" view (black dots). Even after re-consulting the version history (green lines), it still took P6 some time to figure out a solution.

In the second instance, highlighted with a box in Figure 4.4 (b), the participant was having trouble compiling because of a TypeScript version incompatibility. The team had upgraded the project version, but the participant was still using the older version and was not aware of the upgrade. They found some hints but were still stuck: *"I'm just trying to figure out... there's some TypeScript errors after the merge."* They checked the history, the build output and the source code multiple times but to no avail. Finally, they consulted a colleague who mentioned the version upgrade. Having this information allowed P6 to move forward in their resolution. At the end of the conflict resolution, P6 made sure to reflect the version upgrade in the project's documentation (final purple line).

I am stuck making sense of the information

Once participants thought they had found the right information, they had to synthesize it to see if it helped with the conflict resolution. Oftentimes, they realized that the data they had gathered was insufficient or incorrect and had to go back to the *foraging loop* to collect more data. In our analysis, we refer to this as the HUNTINGFORDATA pattern, where participants switched between the HYPOTHESIS TESTING and EVIDENCE FILE steps when they collected additional information.

Figure 4.5 shows an example of this pattern, with instances of HuntingForData in solid-line boxes. P7's merge was initially successful but then they encountered an indirect merge conflict—the version control system did not detect any conflicts, but the code failed to run after the merge was complete (the blue line at the start of Figure 4.5). P7 commented about this: *"If I hadn't run the application, I would have not got this."*. During the conflict resolution, P7 showed the HuntingForData pattern three times at the beginning of the resolution process, where they constantly went between looking for information (in the Evidence File step) and trying to understand why the build was failing by investigating the code (red lines in Figure 4.5). P7 was in this pattern for 4 minutes before making progress on the resolution.

During the resolution, they were also unsure why their attempts at a solution were not working and they struggled to understand the reason behind the conflicting changes: *"So what are we trying to do here?"* P7 would edit the code, then the IDE would report a problem with their edit—this happened about 14 times in a row until P7 was finally able to get to the bottom of it: *"Now I think I understand why they had to cast it,"* although the IDE warning messages were not pointing to the root cause of the issue: *"Alright, what does it not like?"*

P7 ultimately found that the failure was caused by a change to a function's return type, and modified the code so that the compiler would detect similar conflicts in the future: *"Now the compiler would catch if this happens again."*. Finally, he made a note of the issue in the project documentation (purple line at the end of Figure 4.3).

Another factor that made merge conflict resolution difficult for multiple participants was when changes had conflicting requirements, which meant the parallel changes were inherently incompatible and an acceptable resolution required finding a middle-ground solution. For example, P7 ran into a particularly difficult case of this during P7-C4, when they had to merge two sets of other people's changes. Unfortunately, P7 wasn't familiar

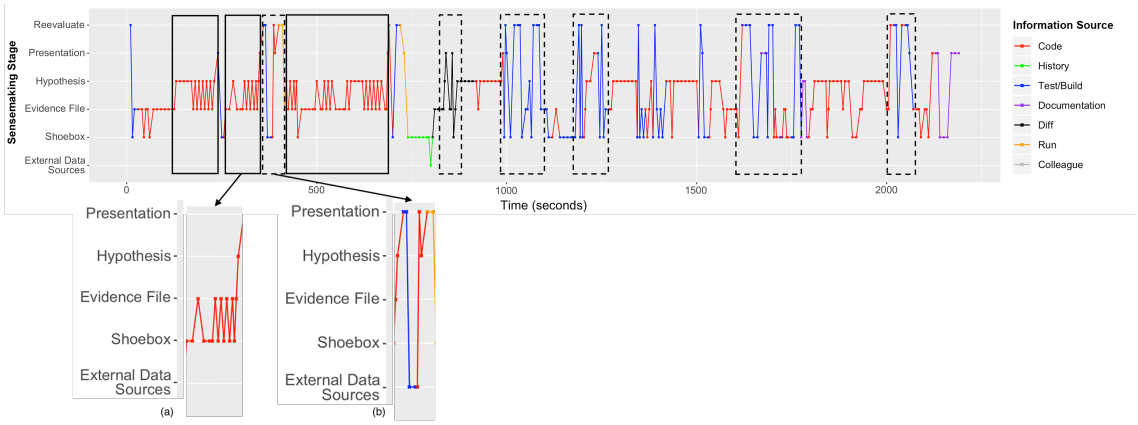

FIGURE 4.5: Resolution pattern of P7-C1. The solid-line boxes are time spans when instances of the HuntingForData pattern occurred; an instance is magnified in subfigure (a). Dashed-line boxes are time spans when instances of the QuickResolution pattern occurred; an instance is magnified in subfigure (b).

with changes on *either* side of the conflict: *"I'm gonna have to go back . . . and see if I can see when they were changed, why they were changed, and see if I can decide which one to pick."* This conflict took P7 13 minutes and 40 seconds to resolve.

Another reason participants constantly switched between loops might have been because the information sources were complex, creating memory and comprehension challenges. For example, P7 repeatedly switched between test results (which explained why the test failed) and the code: *"What did it not like? . . . What did it not like?"*

I now know what to do

Once participants knew what caused the conflict, they were able to quickly resolve it—reflecting the QuickResolution and SkippingTheHypothesis sensemaking patterns.

QuickResolution is defined as when a participant begins trying to resolve part of a merge conflict and reaches the Presentation step without backtracking through the sensemaking hierarchy. This pattern indicates the developer is in the "fast lane", quickly zipping through the sensemaking steps towards a resolution. QuickResolution

occurred most often after the participant emerged from HuntingForData. After they found the right information, they could *"power through"* the remainder of the conflict. There were 27 instances of this.

In some cases, participants already knew why the conflict occurred or the solution was obvious from the outset. In other cases (e.g., P7-C1, Figure 4.5), instances of QuickResolution patterns occurred interspersed with other patterns. For example, once P7 understood why there was a conflict, they were able to quickly resolve the conflicts, in many cases skipping over the Hypothesis Testing step as they had a good understanding of the problem.

The P7-C1 resolution included six instances of QuickResolution.

SkippingTheHypothesis is when a participant bypassed the Hypothesis Testing step altogether, meaning they were so certain about which version of the conflicted code to commit that they jumped to the end of the resolution immediately after understanding what the conflict was about. When participants were stuck in this pattern, they resolved the merge conflict without examining or modifying code or (seemingly) taking time to consult any information sources at all. Participants used this pattern when the solution to the conflict was obvious. We had 25 instances of the SkippingTheHypothesis pattern.

Other strategies for making sense

A few participants used different strategies for becoming unstuck, or for getting help with making sense of the merge conflict. For example, P6 asked a colleague for help. Even though they were one of the most experienced participants on the team, they asked the author of the changes clarifying questions about the problems they were facing. In other cases (e.g., P7-C1), the more experienced participants knew information about the project structure that was crucial for correctly resolving a build error generated by a merge conflict resolution—they were aware of information that novice participants or

participants new to the project were less likely to be aware of. New tools and processes could make this type of information easier to attain, or it could point developers to team members likely to have the necessary knowledge.

Implications

Finding the right information is important for successful merge conflict resolution. However, finding the right information can be difficult; developers can get StuckForaging. Knowing where to look, or who to ask, can make this task a lot easier for the developer, especially if they are novice to the project. A recommendation system, which could point developers to the right information source, or the right person, could help developers be more effective when resolving merge conflicts.

We also observed that evaluating a potential solution to a merge conflict resolution can be challenging, resulting in participants getting stuck in the Hypothesis Testing step. A developer can only run/test their solution after all the conflicts have been resolved, as most modern version control systems will render the code uncompilable once a merge conflict has occurred. This makes it very difficult for developers to evaluate the code while the merge conflict resolution is in progress. Providing ways to evaluate the code when the resolution is partially complete could alleviate some of the pains developers face.

Finally, if tools are able to anticipate runtime conflicts such as the one P7 experienced, the problem could have been resolved by the developer who wrote the code, before they committed it and broke the build for others.

4.5 Threats to Validity

As with any empirical research, our investigation has threats to validity. In this section, we explain threats related to our investigation and ways we guarded against them.

Generalizability threats

Our goal was to qualitatively investigate the conflict resolution behavior of professional developers. Therefore, we opted to get a small, but rich data set (through in-situ observation), which makes it difficult to generalize our results. Our results serve as a first step toward a deeper understanding of why and where developers struggle when resolving conflicts. Additional studies are needed to generalize the results across different companies or domains.

Construct threats

As with any observational study, our data is susceptible to the Hawthorne effect [11], whereby participants may have presented an ideal behavior. However, we believe this effect is minimized because merge conflict resolution can be an inherently difficult task. While developers might want to perform better, they will still face the same barriers, whether they were being observed or not. Second, because participants had to prompt the researcher to observe them, they may have chosen only "straightforward" or "noteworthy" conflicts to bring to the researcher's attention. If the former is the case, then we may have missed some challenges that developers face with conflict resolution. If the latter occurred then we missed conflicts that were trivial and would not have added any further insights to the struggles with resolution.

Internal threats

The definitions we chose for mapping the sensemaking steps to the merge conflict activity could be subjective. However, the codebook of sensemaking steps was validated by the first and second author, by independently coding 20% of the data. We achieved an IRR of 0.82, measured using Cohen's kappa, which suggested the two authors were in near perfect agreement [107].

4.6 Related Work

4.6.1 Conflict Avoidance

Biehl et al. [24] propose *FastDASH*, a tool that enables awareness between team members by providing a dashboard that shows check outs, modifications, and staging of files among members of a team. Servant et al. [139] propose *CASI* as a conflict avoidance tool that relies on visualizations of changing elements within a program to allow coordination among developers. Estler [60] describes a collaboration framework that integrates fine-grained changes and real-time awareness capabilities to prevent conflicting changes from being submitted to shared code repositories.

4.6.2 Workspace Awareness

da Silva et al. [48] propose *Lighthouse* to show design level changes (in the form of UML diagrams) among all developers on a project. While this approach–and all those in the previous subsection–provide awareness of potential conflicts, they require the developer to actively monitor and discern if a conflict will (or has already) occurred.

Sarma et al. [137, 136] go a step further and propose *Palantír*, which monitors other developer's workspaces and non-obtrusively notifies the developer if a conflict has happened. Whereas, Brun et al. [31] propose *Crystal*, which monitors selected branches in a repository and preemptively merges in order to detect conflicts earlier. Crystal detects both *direct* conflicts (changes to the same line of code), and *indirect* conflicts (changes to a different line that cause build or test failures) and provides notification to developers.

Similarly, Hattori and Lanza [77] propose *Syde* that monitors for changes at the Abstract Syntax Tree (AST) level to provide more precise information to the developer. Finally, Guimarães and Silva [74] propose *WeCode*, which also merges in uncommitted code in order to reduce the time between introduction and detection of merge conflicts. However, these tools and techniques assume that developers consistently react and resolve

merge conflicts in a unified manner.

4.6.3 Merging Algorithms

Westfechtel [150] propose a merging technique that uses the lexical information of a language when performing a merge. Apel et al. propose *JDime*, which provides support for semi-structured [16] and structured merges [15]. However, these techniques only use structural information when the unstructured (i.e., text only) merge has failed. Binkley et al. [25] propose using call graph information to correctly merge different versions of the program.

Accioly et al. [9] used a semi-structured approach to understand the types of merge conflicts. These types characterized the frequency and variety of merge conflicts, but not the patterns that developers employ when resolving such conflicts. Lippe and van Oosterom [101] propose an operation-based merging technique that would replay the changes from two branches, in the order in which they were performed. Dig et al. [57] use this technique and empirically show that additional types of merge conflicts could be resolved by tools that understand the semantics of changed code.

4.7 Conclusions

Our observations of seven real-world software development professionals resolving 10 merge conflicts yielded the following key results:

- Participants sought out many information sources when attempting to resolve merge conflicts, and tried sorting through and making sense of the information using two different *information usage patterns* and four different *sensemaking patterns*.
- Two of the sensemaking patterns revealed pain points in the merge conflict resolution process, in (1) attempting to locate information necessary for the resolution, and in

(2) attempting to implement different potential solutions, without success.

- Despite the number of information sources accessed, participants often needed to switch between information sources frequently, potentially incurring high cognitive costs.

Now having gained a sense of how software developers resolve merge conflicts, we find there's a disconenct between human problem-solving processes and the technological systems meant to support these human processes. This indicates limits—perhaps unncessary ones—on the rate of growth for successful and efficient computer-human systems with low cognitive costs: For humans to be kept "in-the-loop", we must adapt our technological support systems to *how humans think*—not the other way around.

5 PLANNING FOR UNTANGLING: PREDICTING THE DIFFICULTY OF MERGE CONFLICTS

Caius Brindescu, Iftekhar Ahmed, Rafael Leano, Anita Sarma

The 42nd International Conference on Software Engineering, 2020

5.1 Introduction

Version Control Systems (VCS) have made large teams possible, enabling thousands of developers to contribute together in building Open Source Software (OSS), and proprietary software and technologies. However, despite the introduction of new, sophisticated, distributed version control systems, the basic protocol of using VCS still remains the same: code in private workspaces and synchronize periodically.

One challenge with this coordination protocol is merge conflicts. Merge conflicts occur when developers modify the same lines of code simultaneously. Research shows that merge conflicts are prevalent: about 19% of all merges end up in a merge conflict [31, 12, 90].

Merge conflicts have an impact on the code quality [12, 51, 118] and are disruptive to the development workflow. To resolve a merge conflict, a developer has to stop what they are doing and focus on the resolution. Resolving a conflict requires the developer to understand the conflicting changes and craft a solution that satisfies both sets of requirements driving the change. This disruption to the workflow can be compounded if conflict resolution requires additional expertise [45, 51, 118]. These factors can prompt developers to postpone the conflict resolution, or "kick the can" further down the road. In fact, a study by Nelson et al. [117] found that 56.18% of developers have deferred at least once when responding to a merge conflict. However, the later a conflict is resolved, the harder

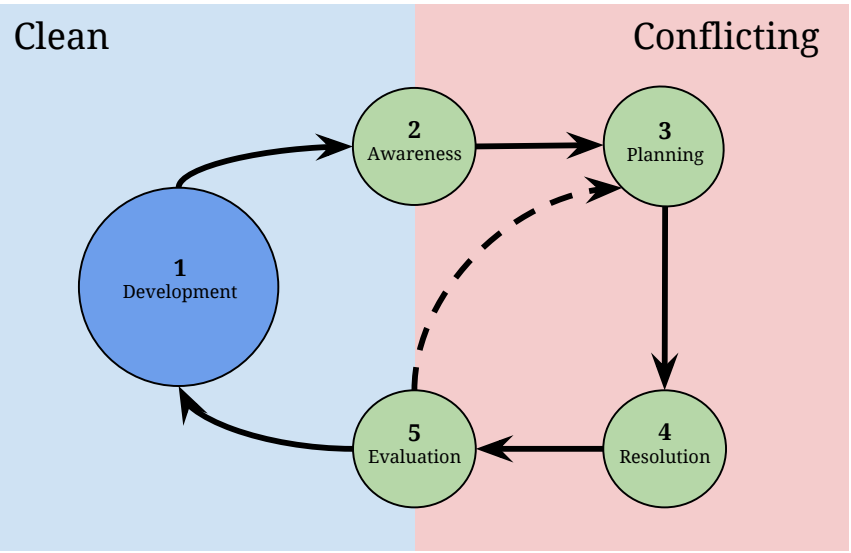

FIGURE 5.1: Model of Developer Processes for Managing Merge Conflicts, from Nelson et al. [117]

it is to recall the rationale of the changes. which makes the resolution process that much more difficult [20, 64]. As aptly put by a participant from the study by Nelson et al. [117]:

> *"Deferring a merge conflict simply kicks the can down the road (or off a cliff). Typically resolving the conflict only gets more difficult as time passes."*

However, sooner or later the conflict has to be resolved. To do so developers follow a process with four distinct resolution phases [117], as illustrated in Figure 5.1. Developers alternate between clean and conflicting states of code. Beginning from (1) the *development* stage, developers create an (2) *awareness* of conflicts within the codebase either passively when they face a conflict during a merge or by proactively monitoring ongoing changes. Once aware, developers begin (3) *planning* for a (4) *resolution* to fix the conflict. And finally, developers (5) *evaluate* the effectiveness of their deployed resolutions (returning to planning if the resolution fails).

Several research works exist to support parts of the conflict resolution process. For the Development and Awareness phase, developers can benefit from workspace awareness tools [137, 31, 24, 74]. When working on the Resolution phase, developers can utilize

different semi-automated merge techniques, such as unstructured merge [21, 40, 28, 112, 116], structured merge [34, 140, 150], semantics-based merge [23, 83], and hybrid merge [98, 15, 16]. The Evaluation phase has support through existing VCS (e.g., Git, SVN, TFS, CVS) and Continuous Integration systems (e.g., Jenkins, Travis CI, TeamCity).

None of these works, however, support the Planning phase of merge conflict resolution. We aim to close this gap and help developers plan their conflict resolution by predicting the difficulty of a merge conflict.

Our work can facilitate planning of conflict resolution in several ways. It can help developers plan: (1) when to resolve the conflict; if the conflict is simple they can resolve it instantaneously, otherwise they may need to allocate a longer resolution time period, (2) who to resolve it with; if a conflict is difficult they may need to coordinate a collaborative merge, (3) how much to review or test the merged code; if a conflict is difficult it may behoove the developers to more rigorously review and test the merged changes.

In this work, through a large scale empirical investigation we analyze what makes a merge conflict difficult, and whether we can predict the severity of a conflict from its underlying changes. More formally, we aim to answer the following research questions:

RQ1: How well can we predict the difficulty of merge conflicts?

RQ2: What makes a merge conflict difficult?

RQ3: How portable is our conflict difficulty prediction model?

To answer these research questions, we mine the characteristics of 6,380 merge conflicts from 128 Java projects in GitHub. To enable a prediction of merge conflict resolution, we gather a total of 16 process- and code-related metrics, such as the conflicting lines of code, differences in abstract syntax trees (AST), cyclomatic complexity (CC) etc. We use metrics that are available as the conflict develops (i.e., before the developer merges their changes), therefore, enabling developers lead time for their planning.

Our results show that we can predict the difficulty of merge conflicts accurately (an AUC of 0.76). Knowing which conflicts are difficult can help developers plan their conflict resolution. Our work also serves as a baseline prediction model for further research.

5.2 Related Work

Merge conflicts are a common side effect of concurrent development [159]. While the use of version control systems flags divergent changes and prevents one change from overwriting another, they cannot always automatically resolve conflicts. Researchers have tried different approaches to help developers deal with conflicts: (1) early detection of conflicts, (2) merging assistance, and (3) prevention of conflicts.

5.2.1 Early detection

Conflicts tend to grow over time. Therefore, early detection helps limit the files and the size of the changes involved in a conflict. Workspace awareness tools monitor ongoing work to detect emerging conflicts. The goal of these tools is to "catch" the conflicts early so that it is easier to resolve them. Biehl et al. [24] propose FastDash, which identifies developers modifying the same file and notifies them about potential merge conflicts as they emerge. Hattori and Lanza [78] propose Syde, a tool that analyzes changes made to the source code at the level of AST operations. Syde detects conflicts by comparing the (AST) tree operations. Guimaraes and Silva [74] introduce WeCode, which continuously merges changes in the background to detect merge conflicts. Tools such as Palantir [137] and Crystal [31] proactively detect both merge conflicts, as well as semantic conflicts; the latter being conflicts that are revealed by failed builds or tests. Servant et al. [139] propose CASI, a tool that allows developers to visualize the code that their changes impact, with the aim of detecting semantic conflicts.

While these tools notify developers of emerging conflicts, they do not provide any

assistance in resolving them. Our focus is on predicting the difficulty of merge conflicts so that developers can prioritize their resolution efforts.

5.2.2 Merging Assistance

Another major thread in merge conflict research is support for the resolution of merge conflicts. Mens [110] provides a comprehensive survey on the state of the art merging techniques. Apel et al. [16, 15] propose a new merging approach; called semi-structured merging. This technique considers the syntactical structure of the code that is to be merged. Lippe and van Oosterom [101] also propose a new merging technique, operational merging, which considers the changes that were done to the code, in addition to the end result.

While we do not attempt to provide resolution support to developers, our work may help developers choose one resolution strategy over other based on the difficulty of the conflict.

5.2.3 Prevention

Finally, another way to deal with conflicts is to prevent them from occurring in the first place. Kasi and Sarma [90] try to avoid merge conflicts altogether by scheduling tasks in a way that minimize the probability of conflict. Wloka et al. [152] introduce SafeCommit. It uses a static analysis approach to identify changes that can be committed safely, i.e. they do not cause any of the tests to fail. This allows developers to cherry pick the commits that are safe to commit (and avoid conflicts). Dewan and Hedge [55] propose a new development process that allows developers to synchronously collaborate on the conflicting code and solve the conflicts before finishing the task. Leßenich et al. [99] conducted a survey of 41 developers and inferred 7 indicators to predict the number of merge conflicts. Then they analyzed 163 open-source projects found that none of these 7 indicators suggested by the participating developer has a predictive power concerning

the frequency of merge conflicts.

While some conflicts can be prevented, others are bound to occur. For example, Kasi and Sarma in their approach have to relax some constraints to allow some conflicts to occur when the space becomes too constrained. Leßenich et al. [98] and Cavalcanti et al. [38] examined 50 and 60 projects, respectively, to compare semi-structured and unstructured merge techniques in terms of how many conflicts they report. Both studies found that semi-structured merge techniques can reduce the number of conflicts by approximately half, but not eliminate them. Our work can be useful as a guide for which constraints (or conflicts) can be relaxed.

5.2.4 Conflict difficulty

Different studies have investigated ways to measure the amount of effort required to resolve a conflict. Resolution time varies significantly across projects and ranged not in hours, but in days [**?**]; and it can be used as a proxy to measure the difficulty of conflicts (difficult conflicts take more time to solve). The Orion approach by Prudencio et. al [127] tried to minimize the number of files to be *locked* using the actions applied in the file, and the committed actions. Their end goal was to *minimize* the number of conflicts that would occur, at the cost of reduced development concurrency. McKee et al. [108] and Nelson et al. [117] interviewed developers and then performed a follow-up survey with 162 developers to build a detailed understanding of developer perceptions regarding merge conflicts. They found, among other things, that complexity of the conflicting lines of code and file as a whole, number of LOC involved in the conflict, and developers' familiarity with the lines of code in conflict all impact how difficult developers find a conflict to resolve. Menezes et al. [109] used number of conflicting chunks to determine patterns that occur in merge conflicts.

All in all, none of these related works deals with the main purpose of the our work: the prediction of difficulty of potential merge conflicts in order to help developers

prioritize merge conflicts to inspect, accomplish more things given tight schedule, and not waste reviewing effort on trivial merge conflict resolutions.

5.3 Methodology

To predict the difficulty of conflict resolutions, we collect a representative corpus of merge conflicts to be examined by four different machine learning classifiers. We use the following process during our study: (A) we collect a sample of Java projects hosted on GitHub; (B) we filter projects that do contain merge conflicts, are inactive, or toy projects; (C) we extract the relevant attributes needed for merge conflict analysis by conducting a literature survey; (D) we label a subset of merge conflicts manually; (E) we conduct supervised training with four machine learning classifiers; (F) we compare the results from each of the classifiers; (G) we perform feature selection; and, (H) we repeat steps (E) and (F) for cross-project merge conflict difficulty prediction. We describe each of these steps in further detail in the following subsections.

5.3.1 Project Selection Criteria

We made our project selections to be applicable to the requirements of the four machine learning classifiers and to be representative of code developed in the real world. Therefore, we only select active, open source projects from GitHub. We opted to select projects that use the same programming language to control for language-specific differences in the Lines of Code (LOC) metrics, program dependencies, and construct comparability. We use Java as our language of choice for two reasons: (1) Java is one of the most popular languages (according to the number of Java projects hosted on Github [68] and the Tiobe index [7]); and, (2) there are more mature analysis tools available for Java as compared to other programming languages.

We began by selecting 900 Java projects returned by GitHub search mechanism

without any filtering criteria. From these 900 projects, we eliminate projects that were forked copies of other projects to prevent skewed results, leaving 500 projects in the end. Since some projects do not compile, either due to syntax, build errors, or missing dependencies, we eliminated an additional 300 projects. After filtering, our corpus contained 200 Java projects we were successfully able to compile and run.

We follow the guidelines presented by Kalliamvakou et al. [89] for mining Git repositories. We removed projects that were too small (fewer than 10 files, or fewer than 500 lines of code), and those that were not active in the past 6 months. We also removed projects that do not contain merge conflicts. These selection criteria were required since there is a long tail of small and short lived projects on GitHub; including trial projects, projects with a single author, or projects with no parallel development history, which did not have any merge commits. We focus on collaborative software development for this work, and we therefore remove projects that are not collaborative in nature. Our final corpus had 128 projects. Table 5.1 provides a summary of project statistics for these projects, including: number of lines, total number of commits, total number of merges, total number of merge conflicts, number of developers, and number of days that project has existed on GitHub as of March 1, 2018.

TABLE 5.1: Mined Projects Statistics

Dimension	Max	Min	Average	Std. dev.
Line count (LOC)	542,571	751	75,795.04	105,280.1
Total Commits	30,519	16	3,894.48	5,070.73
Total Merges	4,916	1	252.60	522.73
Total Conflicts	227	1	25.86	39.49
# Developers	105	4	72.76	83.19
Duration (days)	6,386	42	1,674.54	1,112.11

We also manually categorized the domain of the projects by looking at the project description and using the categories used by Souza et al. [52]. Table 5.2 has the summary of the domains of the 128 projects and their percentage of representation within our corpus.

TABLE 5.2: Distribution of Projects by Domain

Domain	Percentage
Development	61.98%
System Administration	12.66%
Communications	6.42%
Business & Enterprise	8.10%
Home & Education	3.11%
Security & Utilities	2.61%
Games	3.08%
Audio & Video	2.04%

5.3.2 Conflict Identification

We queried the repository of the 128 projects, from which we extracted 556,911 commits. This included 36,122 merge commits. Since Git only stores information of commits, but does not record instances of merge conflicts we identified merge conflicts by following branch merges and analyzing the commits involved as shown in Figure 5.2. First, we identified merges as commits with two or more parents, such as commit AB. Then we merged the parents of that commit (A_n and B_n) using the `git merge` command. If the merge was unsuccessful then AB was marked as a merge conflict (m). Using this approach, we identified 6,380 merge conflicts.

We consider a merge conflict to be an instance when running `git merge` failed because of concurrent changes in the 2 (or more) branches being merged. A conflict can have multiple conflicting files, each with multiple conflicting chunks. For the purposes of this research, we focused on conflicts that occurred in source code files (`.java`).

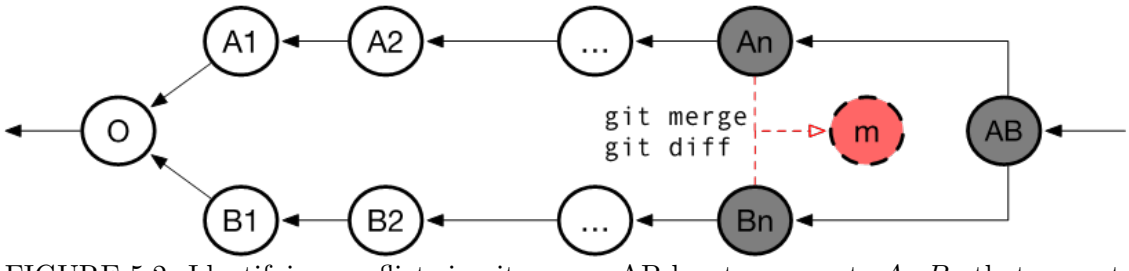

FIGURE 5.2: Identifying conflicts in git, merge AB has two parents A_n,B_n that cannot be merged automatically.

5.3.3 Data Collection

Once we identified a merge conflict, we extracted additional information relevant for the analysis of the conflict, such as: the authors involved in the commits, the commit message, the files that were edited, the changes that were made (by using the `git diff` command), and so on. This was done for the parent commits A_n and B_n, as well as all commits on either branch back to the base commit (i.e. the last shared parent commit). That is, from all commits A_1 to A_n and B_1 to B_n (in Figure 5.2).

Performance of any prediction is dependent on the features used. Therefore, we wanted to use a comprehensive set of features. In order to get an overview of the current state of the art research on merge conflict difficulty and metrics used, we conducted a literature survey. We targeted all full conference and journal papers related to merge conflict difficulty from 2008 to 2018 (inclusive) that appeared in top Software Engineering venues: ICSE, FSE, ASE, ICSM/ICSME, MSR, ESEM, TSE, TOSEM. Starting from the proceedings, we searched for a set of keywords including *"merge conflict difficulty," "merge conflict resolution effort"* etc. We found only 4 papers [127, 117, 109, 108] and we analyzed what metrics were used in the studies.

Based on the metrics reported in the identified papers and also based on intuition, we obtained 16 factors for each conflict, which we list in Table 5.3. We grouped these factors into five unique dimensions: size, complexity, diffusion, development pattern and comment [153]. We gathered these factors from either the Git repository, or we derived them by analyzing the source code, when the factors are related to the process and code metrics (characterized in numerical form).

Since certain metrics are calculated at different levels of granularity (e.g. complexity metrics are calculated at the method level), we aggregate all factors to a per conflict measurement using the average. In Table 5.3, the subscript is the operation used for aggregation. For example, for cyclomatic complexity CC_{sum} is the sum of the cyclomatic

complexities of all the files modified in both branches. We calculated file-based metrics using all modified files and conflicting files. Size metrics use LOC as the unit of measurement.

We also include branch and author commits patterns. They refer to the temporal order in which commits are ordered in separate branches. The branch pattern reflects how commits are interleaved between branches. For example, in Figure 5.3, commit A_1 in branch A followed by commit B_1 in branch B, which yields the pattern AB. We continue building the pattern, until we reach the final $ABAABBBA$. We then collapse identical letters, yielding the final pattern $ABABA$. We collapse the letters, because we are interested in the interleaving, and not the total number of commits. A longer pattern means that the commits were more interleaved (tangled).

Similarly, author pattern shows how commits were interweaved between different authors. In Figure 5.3 we have 3 authors: John (J), Alice (A), and Oscar (O). Applying the same rules as the branch pattern, we end up with $JAOAJ$. Our rational for using these patterns as features in our analysis is that, the more interleaved (tangled) a development patterns is, the more difficult it should be to untangle it when resolving the merge conflict.

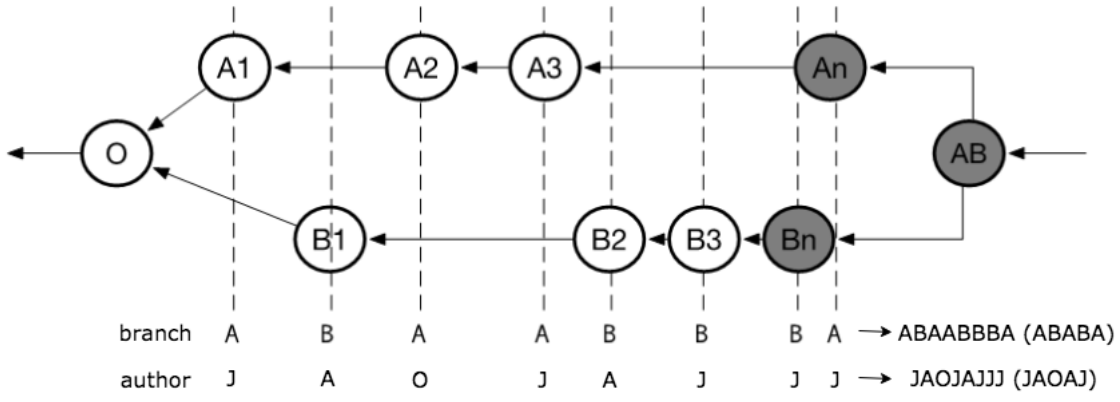

FIGURE 5.3: Example of calculating the branch and author patterns for a merge commit. Time flows from left to right and the arrows point to a commits parent(s).

TABLE 5.3: Collected Process and Product Metrics

Category	Metric	Description	Source
Size	*Number of authors*	Number of different authors involved in all commits $A_1..A_n$, and $B_1..B_n$.	[117]
	Number of edits	The number of edits of each file	[117, 109]
	LOC	Size in Lines of Code (LOC) of commit A (LOC_A), B (LOC_B)	[117, 109]
	LOC_{diff}	LOC differences between commits A_n and B_n	[117]
Complexity	CC_{sum}	The total (sum) cyclomatic complexity of all files modified in both branches	[117, 108]
	CC_{max}	The maximum cyclomatic complexity of all files modified in both branches	[117, 108]
	CC_{avg}	The average cyclomatic complexity of all files modified in both branches	[117, 108]
Dev. Pattern	$Pattern_{branch}$	Length of commit pattern between branches	[127]
	$Pattern_{author}$	Length of author pattern between branches	[127]
Diffusion	$Files_{java}$	Total number of *java* files modified in branches A, B	[109]
	$Files$	Total number of files modified in branches A, B	[108, 117]
	$Dependency_{sum}$	Total dependencies (sum) of all files modified in both branches	[108, 117]
	$Dependency_{max}$	The maximum dependencies of all files modified in both branches	[108, 117]
	$Dependency_{avg}$	The average dependencies of all files modified in both branches	[108, 117]
	AST_{diff}	The number of differences of the AST trees between all modified files in both branches	[111]
Comment	$Comments$	All the comments from commits $A_1 \dots A_n$, and $B_1 \dots B_n$.	[109]

```
476     Factory<byte[]> xidGlobalIdFactory = createXidGlobalIdFactory
            ();
        ... // no conflicts in these lines
482     txManager = new ReadOnlyTxManager( xaDataSourceManager,
            xidGlobalIdFactory, logging.getMessagesLog(
            ReadOnlyTxManager.class ) );
```

Listing 4: Authors' resolution of a *Severe* merge conflict. In this example, the developers made two concurrent refactorings to the `ReadOnlyTxManager` constructor, one of the refactorings introducing a new parameter.

5.3.4 Training Data Labeling

Before training the classifiers, we first manually labeled conflicts as either *severe* or *trivial* based on their difficulty of resolution. From the pool of 6,380 conflicts, we extracted and labeled a random sampling of 600 conflicts (approximately 10% of all conflicts). This random sample represents conflicts in 105 distinct projects out of the total of 128.

In order to validate our evaluation, the first two authors independently labeled 60 conflicts based on their difficulty. In order to evaluate the difficulty, the authors recreated each conflict and attempted the resolution. The authors used the time required to solve the conflict as well as the cognitive load, in order to classify the merge conflict as difficult or *Severe* or *Trivial*. In order to validate the resolution, we compared our merge conflict resolution with the one that was checked in the version control system. We considered the developer's merge conflict resolution as the oracle, as it is most likely to be correct, given their in depth knowledge of the code. In all cases, our resolution was functionally equivalent to the developer's resolution. Listing 4 shows part of the author's resolution for a *Severe* merge conflict, which is functionally identical to the developers' resolution[10].

We checked for agreement by using Cohen's Kappa, and we achieved an IRR of 0.8. The two authors then independently coded the rest of the conflicts, in order to build the training set.

[10] `https://github.com/neo4j/neo4j/commit/178be5393fcfaf860c72ba2b9f26acc05b621375\#diff-af31f64d35f8926cc7c52a61578eb839R482`https://github.com/neo4j/neo4j//commit/178be5

5.3.5 Feature Selection

To select the appropriate metrics we carry out an analysis of potential multicollinearity between the metrics. Previous research [19, 141] demonstrated that many process and source code metrics are correlated, both with each other, and with lines of code (LOC). Ignoring such correlations would lead to increased errors in the estimates of model performances, and increased standard errors of the predictions [76]. We checked for multicollinearity using the Variance Inflation Factor (VIF) [41] of each predictor in our model for our data set. VIF describes the correlation between predictors. A VIF score between 1 and 5 indicates moderate correlation with other factors, and these selected the predictors are with VIF < 5. This step was necessary since the presence of highly correlated factors forces the estimated regression coefficient of one variable to depend on other predictor variables that are included in the model. Out of the 16 factors, 5 had VIF ≥ 5, so we ended up using the remaining 11 factors for building the classifiers.

We further investigate the resulting eleven factors in their effectiveness in predicting the difficulty of a merge conflict, and report the results in Section 5.4. We find that prior work has also used a subset of these factors as a proxy for difficulty of a conflict [127, 90], which is encouraging for our own work.

5.3.6 Machine Learning

We trained and tested our sample using 4 different machine learning techniques: Support Vector Machines (SVM), Logistic Regression, Multi-Layer Perceptron (Perceptron), and Bayes Network (BayesNet). For all techniques, we used a 10-fold cross-validation on our sample of 600 labeled conflicts. We used a wide range of learning techniques to reduce the risk of dependence on a particular algorithm or implementation.

BayesNet

We use the Bayes Network algorithm as we expected features to not be independent and Bayesian Network does not have assumptions regarding independence. For example, LOC_{diff} and *files* (number of files) share a relation, as editing more files will also increase the LOC edited. Also, Bayesian Networks can represent richer models compared to naive Bayes classifiers. We used the SimpleEstimator to calculate the conditional probabilities used by the Bayes algorithm. Finally, BayesNet uses a hill-climbing greedy algorithm for evolving, combined with a K2 search algorithm to create its network. We configured with a batch size of 100.

Binomial Logistic Regression

For the Binomial Logistic Regression, we started with a full model using all 11 attributes from conflicts. This was followed by a model selection using *Akaike Information Criterion* (AIC), which estimates the information loss between models in comparison to the original. It ultimately selects the best model based on both the fit of the model and the information lost. We then use the selected attributes to build the new, final model, on which we evaluate.

Support Vector Machine (SVM)

Based upon an assumption that conflicts would be linearly separated across factors, we selected SVM. Our SVM uses the standard *Radial Bases Function* (RBF) Kernel and for the other parameters we performed a grid search to choose the best classification found. This configuration result had a 1,000-cache size, a 1,000 size, a gamma (γ) of 0.0001, a C of 1,000 and one maximum iterator.

Multi-Layer Perceptron

We used a Multi-Layer Perceptron to see if it could leverage hidden relationships not explored in the other algorithms. We configured our Perceptron with a 0.3 learning rate, a 0.2 momentum, and 500 epochs. The Perceptron would terminate its validation testing after not being able to reduce its error 20 times in a row.

Bagging

We also used Auto-WEKA [95] for identifying the best classifier, which automatically searches through the joint space of WEKA's learning algorithms and their respective hyperparameter settings to maximize performance, using sequential model-based optimization [30] (a Bayesian optimization method). Though there is one Python based implementation called Auto-sklearn [62], we chose Auto-WEKA because it comprises a larger space of models and hyperparameters [95] compared to Auto-sklearn. This ended up identifying "Bagging (Bootstrap aggregating)" [29] as the best technique. "Bagging" is an ensemble based approach that uses multiple models to fit the bootstrap samples generated from the original data and then uses voting for classification. Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [43] was identified as the base learner (accessible at *weka.classifiers.rules.JRip*) by Auto-WEKA. These results were generated by running Auto-WEKA with random seed 123 for 4 hours.

5.3.7 Evaluation

We report the standard precision, recall, and AUC (Area Under the receiver operating characteristic Curve) to asses the performance of the prediction models, because it is independent of prior probabilities [22]. Also, AUC is a better measure of classifier performance than accuracy because it is not biased by the size of test data. Moreover, AUC provides a "broader" view of the performance of the classifier since both sensitivity and specificity for all threshold levels are incorporated in calculating AUC. Other work

related to prediction have used AUC for comparison purposes [56, 66, 67, 156]. We list the formula used for calculating precision, recall and F-measure below. The AUC curve is created by plotting the recall against the false positive rate (FPR) at various threshold settings. We list the formula of FPR also.

- **Precision (P)**: A measure of whether the *Severe* predictions were *actually difficult*.

$$precision = \frac{t_p}{t_p + f_p} \tag{5.1}$$

- **Recall (R)**: A measure of the percentage of *Severe* instances that the approach managed to correctly predict.

$$recall = \frac{t_p}{t_p + f_n} \tag{5.2}$$

- **False positive rate (FPR)**: A measure of the ratio of the number of *Severe* conflict wrongly categorized and the total number of actual *Severe* conflicts.

$$FPR = \frac{f_p}{f_p + t_n} \tag{5.3}$$

5.3.8 Cross-project prediction

Cross-project prediction is important for some projects, specially projects that do not have historical data to perform any significant training. Hence, we investigated whether it is feasible to perform cross-project training following the method used by Rahman et al. [128]. We do so by training models on one project and testing on all other projects, ignoring time-ordering.

5.4 Results

Here we discuss the results of our study by placing them in the context of three research questions, which investigate the the ability to predict the difficulty of a conflict (RQ1), factors that are useful in determining the difficulty of a conflict (RQ2), and whether we can perform cross-project merge conflict difficulty prediction (RQ3).

5.4.1 RQ1: How well can we predict the difficulty of merge conflicts?

To answer this research question, we trained four different machine learning algorithms: Bayes Network (BayesNet), Logistic Regression, Support Vector Machine (SVM), and Multi-Layer Perceptron (Perceptron). We also used Auto-WEKA [95], which automatically searches through the joint space of WEKA's learning algorithms and their respective hyperparameter settings to maximize performance and identify the best classifier. The algorithm with the best performance according to Table 5.4 is *"Bagging (Bootstrap aggregating)"* with *"RIPPER"* as the base learner, which has the highest AUC (0.85). Bagging is closely followed by Bayes Network, which still performs better-than-chance with 0.78 AUC. On the other hand, SVM performs worst (0.56 AUC). Table 5.4 shows the results in terms of precision, recall, and AUC.

TABLE 5.4: Performance of the classifiers. Bagging has the highest AUC at 0.85.

	Precision	Recall	AUC
SVM	0.70	0.70	0.56
L.R.[i]	0.70	0.65	0.73
Perceptron	0.75	0.69	0.75
Bayes Network	0.75	0.75	0.78
Bagging	0.79	0.79	0.85

[i] Logistic Regression

Additionally, we use our Bagging model on the full corpus of 6,380 conflicts to see the characteristics of the merge conflicts; those that are predicted as *Severe* or *Trivial.*

The model identifies 21% of the conflicts as *Severe,* and the rest 79% are classified as *Trivial.*

In our context, precision shows how well we correctly predict *Severe* conflicts, recall shows how many of the *Severe* conflicts we are able to find. We posit that in our scenario, precision has a higher priority than recall. This is because incurring more false positives is likely to make the developer to distrust the tool. As Bagging outperformed all other classifiers, we report the precision, recall, and AUC of Bagging separately for the two classes (*Severe* and *Trivial*) in Table 5.5.

TABLE 5.5: Results of the Bagging classifier, per class

Class	Precision	Recall	AUC
Severe	0.80	0.49	0.76
Trivial	0.79	0.94	0.85

5.4.2 RQ2: Which characteristics of merge conflicts are associated with its difficulty?

In Section 5.4.1 we show that it is possible to predict the difficulty of a merge conflict with high accuracy. Our next step is to understand what are the characteristics of difficult conflicts. For this purpose we use *feature subset selection (FSS).* Specifically, we use "Wrapper" based methods, which considers the selection of a set of features as a search problem. Different combinations of features are prepared, evaluated and compared to other combinations [84]. "Wrapper" methods are also able to detect the possible interactions between features. In this technique, a predictive model is used to evaluate the combinations of features and a score is assigned based on accuracy of the model. We used RIPPER [43] as the predictive model for feature selection, since it was identified as best classifier for predicting merge conflict difficulty as explained in (Section 5.3.6).

Using FSS we obtain a set of ten factors from our initial set of 12. The ten selected factors encompass all the four metric categories (complexity, diffusion, size, and develop-

ment pattern) to which the original factors belonged (see Table 5.3). All these factors can be calculated before the conflict occurs. This suggests that each of these categories are relevant in predicting difficult merge conflicts, even before the developer faces the conflict.

Table 5.6 presents additional information about these ten factors. Two of these factors include complexity metrics, such the CC_{sum} and the CC_{avg} referring to the mean and the sum of the cyclomatic complexities of all files modified in both branches. This suggests that the complexity of the code is an important metric that affects the difficulty in resolving a conflict. However, calculating complexity metrics require specialized (standalone) analysis tools. So, in the worst case, developers therefore have to "guestimate" the complexity of the code based on their own experience.

TABLE 5.6: Feature Selection Results (Sorted based on relative importance)

Category	Metric	FSS	Human Perceived Importance [**?**]
Complexity	CC_{sum}	1	1
Diffusion	$Dependency_{max}$	2	7
Diffusion	$Dependency_{avg}$	3	7
Complexity	CC_{avg}	4	1
Diffusion	AST_{diff}	5	6
Size	LOC_{diff}	6	6
Size	LOC	7	4
Size	*Number of authors*	8	Not mentioned
Dev. Pattern	$Pattern_{branch}$	9	Not mentioned
Dev. Pattern	$Pattern_{author}$	10	Not mentioned

Table 5.6 also shows that development process related metrics are less influential (*Number of authors*, $Pattern_{branch}$ and $Pattern_{author}$) compared to code related metrics (CC_{sum}, $Dependency_{max}$, $Dependency_{avg}$, CC_{avg}, AST_{diff}, LOC_{diff} and LOC). This led us to investigate whether there is any difference between these two types of metrics when it comes to predicting merge conflict difficulty. We perform this analysis since both process and product metrics have known differences in prediction capability in the

context of defect prediction [128]. We built the same set of classifiers shown in Table 5.4, once using only the process related metrics and once using only code related metrics. Tables 5.7 and 5.8 shows the results in terms of precision, recall, and AUC. Surprisingly, both process and code related metrics have similar prediction capabilities (AUC values of 0.69 vs. 0.70), unlike defect prediction, where process related metrics were found to be more powerful [128].

TABLE 5.7: Performance of the classifiers built using only process related metrics (*Number of authors*, $Pattern_{branch}$ and $Pattern_{author}$).

	Precision	Recall	AUC
SVM	0.68	0.69	0.55
L.R.[i]	0.69	0.63	0.70
Perceptron	0.65	0.63	0.72
Bayes Network	0.70	0.70	0.69
Bagging	0.69	0.71	0.69

[i] Logistic Regression

TABLE 5.8: Performance of the classifiers built using only code related metrics (CC_{sum}, $Dependency_{max}$, $Dependency_{avg}$, CC_{avg}, AST_{diff}, LOC_{diff} and LOC).

	Precision	Recall	AUC
SVM	0.65	0.68	0.52
L.R.[i]	0.68	0.69	0.67
Perceptron	0.55	0.48	0.49
Bayes Network	0.70	0.71	0.72
Bagging	0.70	0.71	0.70

[i] Logistic Regression

In the last column of Table 5.6 we report the factors that McKee at al. [108] identify as factors used by software practitioners to gauge merge conflict difficulty. They identified these factors through a survey of software practitioners. Except for the complexity category, none of the other top features mentioned by practitioners are in the top features identified by FSS and vice versa. Clearly there is a disjoint between the human perceived features and machine learned features. We discuss this further in Section 2.5.

5.4.3 RQ3: Is cross-project training possible to predict difficult merge conflicts?

In RQ1 we tested our models using a 10-fold cross-validation with all our training data (600 conflicts). However, some projects may not have historical data to perform any significant training. Cross-project prediction has been investigated in other areas of software engineering such as defect prediction [102, 149, 158]. However, to the best of our knowledge, no one has investigated the applicability of cross-project merge conflict difficulty prediction. We followed the method used by Rahman et al. [128] to perform cross-project merge conflict difficulty prediction. We train models on one project using our best algorithm: "Bagging using RIPPER" and test on all other projects.

Figure 5.4 shows the portability of models across projects for different sets of evaluation metrics (precision, recall, F-measure and AUC). Performance clearly degrades in cross-project settings in comparison to "Bagging using RIPPER" algorithms performance of 0.85 AUC, shown in Table 5.4.

5.5 Discussion

Centrality Matters: The goal of our study is to investigate whether it is feasible to predict the difficulty level of a merge conflict by using automated (machine learning) techniques. One factor that emerged as relevant is *Dependency*. A file that has high *Dependency* is likely to be highly coupled with other parts of the code, and therefore has high centrality. This is problematic for two reasons. First, as the file is central, it has more reasons to change. For example, a class with multiple functionalities (i.e. a God class [**?**]) is more likely to be changed for any kind of modification of the software. Second, the more a file gets changed, the more it's likely to be involved in a merge conflict. Moreover, the conflict is likely to contain disparate changes. This presents a challenge as the developer

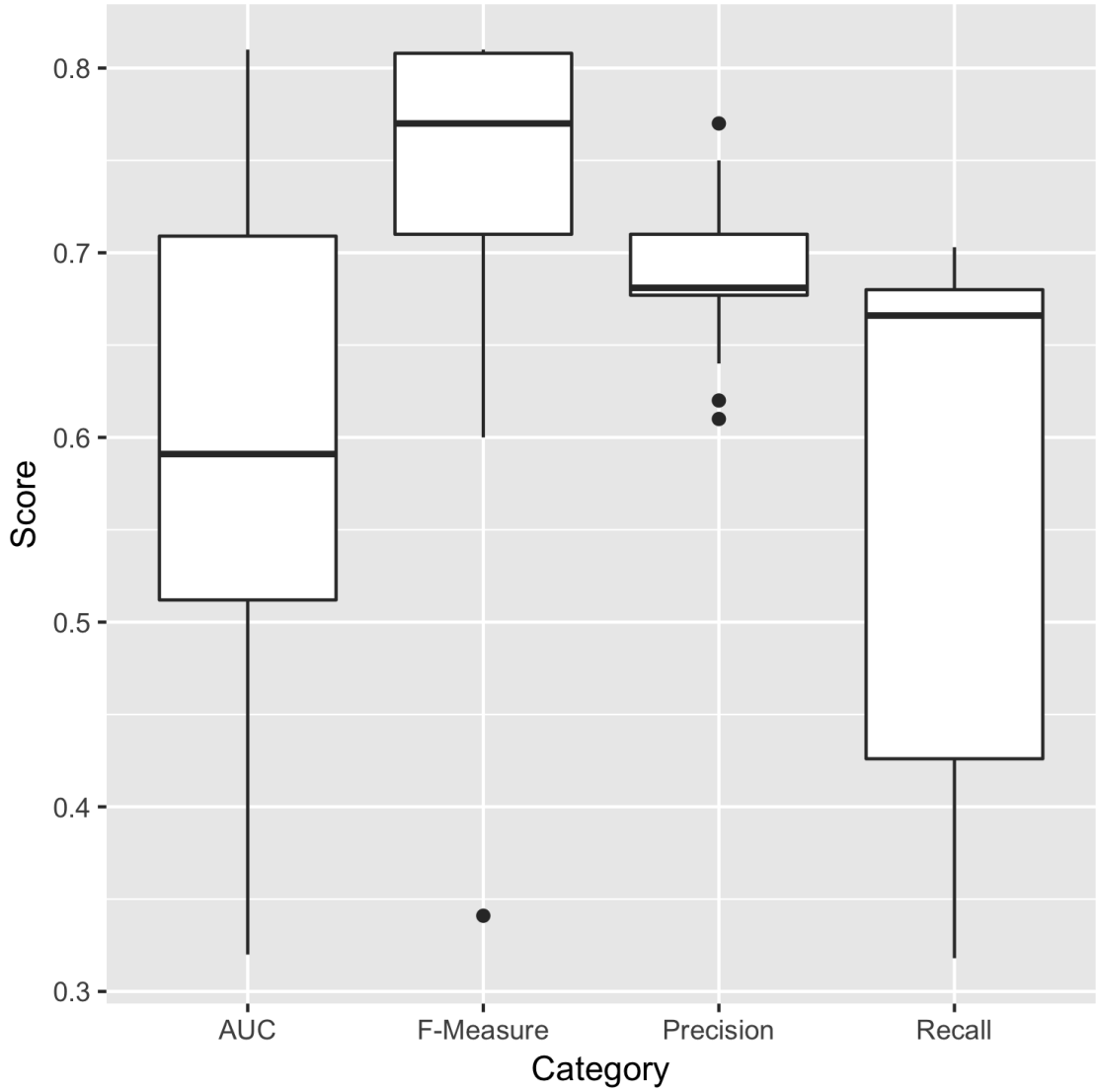

FIGURE 5.4: Precision, recall, f-measure and AUC for cross-project training.

has to understand all the changes involved before resolving the merge conflict. Both our machine learning classifier and developers agree that this is a factor that determines the merge conflict resolution difficulty.

Tangled Changes: In our analysis we find that $Pattern_{author}$ is a significant factor in predicting merge conflict difficulty. A reason for this might be that the more authors that are involved in the development process, the more disparate the conflicting changes

are—because each developer is likely working on a different functionality. So, whenever a conflict occurs, the developer has to understand the broader context of the changes before attempting to resolve the conflict.

Similarly, a longer branch pattern ($Pattern_{branch}$) means that the changes are more tightly tangled. This makes the changes more difficult to untangle when resolving the conflict. Interestingly, while our classifier identified these as important factors, the developers did not. This is an indication that developers are not aware that this could be a potential pain point.

Size does matter: Our classifier also identified the size difference (LOC_{diff}) between the two branches as a relevant factor for conflict difficulty prediction. This is intuitive, as the more lines are changed, the harder it is to understand the changes that were made (in that change set). This, in turn, makes it more difficult to the place that change set in the context of the rest of the code base. When dealing with the difference in terms of AST nodes (AST_{diff}), this becomes even more important, as the AST node difference is more likely to highlight semantic changes. In this case, both the classifier and the developers agree that the size difference between the two branches is an important factor in determining the merge conflict difficulty.

It's Complicated: It's well known that code with higher cyclomatic complexity is more difficult to understand. Therefore, it's not surprising that our classifier identified CC_{sum} as a significant factor for identifying the difficulty of a merge conflict. Another aspect is that code with high cyclomatic complexity is usually indicative of a complicated control structure. Therefore, conflicts in that area are more likely to be semantic in nature. Prior research has shown that areas of code affected by such conflicts are more likely to be buggy [12]. In this case, both the classifier and the developers agree that this is an important factor.

Learn from others: Our final observation is that models are portable between soft-

ware projects. This is an indication that the factors that contribute toward merge conflict difficulty are more or less project independent. Therefore, our technique can be applied to new projects, or to projects with little development history and still prove beneficial to developers.

5.6 Implications

Our findings have multiple implications for tool builders, researchers, and practitioners.

5.6.1 Researchers

Our model for predicting merge conflict difficulty achieved an AUC value of 0.76 when predicting the minority class, which is a high value compared to a baseline of random classification [106]. However, there is still room for improvement. The research community should focus on identifying different types of factors (social, product and process) and investigate their effect on overall prediction accuracy, with the goal of improving the overall prediction accuracy.

Our results inform future research by providing insights into the factors that are associated with the difficulty of a merge conflict. Projects share similar features, as demonstrated by the moderate performance of cross-project merge conflict difficulty prediction. This also indicates that the prediction process can be bootstrapped even for project that lacks history. We recommend that researchers should also focus on identifying the best project selection criteria for bootstrapping cross-project prediction. For bootstrapping, we should use similar projects. However, what constitutes as a similarity metric between the project, in this context, is still an open research question. Our results show that cyclomatic complexity is the most important metric when predicting difficulty. (Table 5.6), so projects with similar Cyclomatic complexity should could used for bootstrapping. How-

ever, further investigation is required to make any conclusive or definitive remarks.

We also found that Auto-WEKA, which automatically searches through the joint space of WEKA's learning algorithms, and their respective hyperparameters, helped us to identify the best classifier and increased the AUC value from 0.78 to 0.85 (Table 5.4). Our finding is inline with the findings of other researchers [148, 146] who have shown the benefit of parameter optimization in improving classifier performance. Therefore, we recommend that researchers using machine learning classifiers should seriously consider parameter optimization to ensure the best performance of the classifiers.

5.6.2 Tool builders

When looking at the types of factors that make a merge conflict difficult, we identified categories relating to the complexity of the code (*Complexity*), extent of the change (*Diffusion*) and the length of branch pattern (*Development Pattern*). The *Complexity* and *Diffusion* metrics are already used by researchers and tool builders for merge conflict prediction. However, we are the first to associate the length of a branch pattern to merge conflicts and their difficulty. Further research can help identify threshold of branch pattern after which a merge conflict becomes severe. Tool builders can use such thresholds as a criteria to filter and prioritize the notifications about potential conflicts. This will not only help users manage the information load, but also will have impact on the quality of the final product [**?**].

5.6.3 Developers

Our results indicate that the more *"tangled"* a piece of code is, the more difficult it will be to resolve a conflict related to that code. So developers can use the length of $Pattern_{author}$ and $Pattern_{branch}$ in deciding merge conflict resolution strategy. Moreover, it's more likely that a code with bigger pattern length has diverged a lot from the initial point and has become difficult for any individual to understand completely. In such

cases, it will be more productive to do a collaborative merge [45], instead of a developer performing the merge by herself.

We also found that all ten significant factors in determining merge conflict difficulty can be collected even before the merge conflict actually occurs. Current awareness tools are already collecting these information when predicting emerging conflicts. Therefore, without further overhead, awareness tools can use our model to predict the difficulty of the emerging conflict, which can be then used as a prioritization criteria when notifying users. Tools can also recommend developers who are most suited to resolve a conflict based on the historical data of merge resolutions. The rationale would be that developers with more experience of resolving difficult conflicts in the past are suitable candidates for resolving a *Severe* conflict, as compared to someone who lacks the experience.

5.7 Threats to Validity

Our research findings may be subject to the concerns that we list below. We have taken all possible steps to neutralize the impacts of these possible threats, but some couldn't be mitigated and it's possible that our mitigation strategies may not have been effective.

Our samples have been from a single source - Github. This may be a source of bias, and our findings may be limited to open source programs from Github. However, we believe that the large number of projects sampled more than adequately addresses this concern.

Another threat to our findings is that, Git tracks the version history of a project as it occurred but it also allows history rewriting using the "rebase" command. It is known that some development teams use "rebase" instead of "merge" to reintegrate branches [40] which means that may have missed merges in our analysis and the number of merges

we analyzed is a lower bound as compared to the actual total number of merges in the projects.

Although we use 24 factors spanning across six categories, there are likely other features that we did not measure. For example, we suspect that the design patterns of a program might influence the likelihood of a conflict resolution being difficult. We plan to expand our metric set to include additional categories in future work.

Our training and testing data had to be manually labeled since this information is not currently available in CM systems or issue trackers. Therefore, our labels may not accurately represent the real merging difficulty because of lack of domain expertise. Our labeling process included inter-rater reliability to prevent individual bias and to reduce this threat. Additionally, the experience and familiarity with the source code and the project can make a conflict difficult to resolve for one developer but simple for another. As we had multiple researchers and we also had a high inter-rater agreement, we assume this should minimize the aforementioned threat.

Another threat would be that we excluded non-source files from our manual analysis (e.g. configuration xml file etc.), but changes to non-source files can have impact on the program's execution if these files are involved in the build/deploy process or for code generation and ultimately make the merge resolution difficult.

5.8 Conclusions and Future Work

In this empirical study, the first of its kind, we investigated the different aspects that can impact the difficulty level of resolving merge conflicts. We evaluated five different classification techniques from different families and identified "Bagging using RIPPER" as the base learner to be the best model; with an AUC of 0.76. We also identified a set of ten metrics that are most influential while predicting the difficulty level of a conflict which

include metrics about the complexity of the code, the size of the change, and development pattern etc. We also found that there is a disconnect between the factors developers use to gauge the difficulty of a conflict and the factors our automatic classification technique identified as important. Finally, we showed that we are able to perform cross-project merge conflict difficulty prediction; with median AUC of 0.60. Therefore, our results show that we can bootstrap prediction in projects with no (labeled) data or only small amount of history, by training on other projects. Our study opens a new avenue in Software Engineering research related to predicting the difficulty level of a merge conflict and help developers plan the merge conflict management process efficiently.

We also provide actionable implications for researchers, tool builders, and practitioners to harness the results of our study. In future work, we hope to explore whether these factors can be seamlessly merged into tools or techniques to assist developers' workflows.

6 CONCLUSIONS AND FUTURE WORK

In this thesis we proposed three research goals. The first, was to understand the connection between software design, bugs and merge conflicts. Our results show a strong correlation between design smells, bugs and merge conflicts. This indicates that merge conflicts are a sort of a canary in the mine. If conflicts occur often in a certain part of the code, perhaps it's time for the code to be refactored. Not only will this make developers' lives easier, but it will also improve the quality of the code, and that of the product.

We also proposed a new taxonomy for merge conflicts, based on the root cause of the conflict. We find that, if the conflict involves changes that alter the behavior of the code, the lines involved in the conflict are 27 times more likely to be involved in a bug fix in the future.

Second, we aimed to understand why merge conflicts are difficult for developers Our qualitative analysis indicates that existing tool support is inadequate for merge conflict resolution. In many cases, the information required is spread across multiples sources, and it's up to the developer to to find it, link it, and make sense of it. This makes merge conflict resolution a challenging task. We also observed instances where the information required for the resolution was held by other team members.

Third, we proposed a novel approach to predict the difficulty of a merge conflict resolution. We find that we can predict the difficulty with good accuracy. This information can help developers plan ahead for merge conflict resolution. Hopefully, this will make the resolution less haphazard, and reduce the likely hood of bugs being introduced in the resolution.

Based on our findings, we propose the following recommendations to 3 key audiences:

- *Developers:* When encountering merge conflicts that require code changes, we recommend performing extensive code reviews. Our results show that those areas of

the code are likely places for bugs to hide in. Those parts of the code are also good candidates for extensive testing.

- *Researchers:* The strong relationship between merge conflicts, code smells and bug fixes suggests that there might be area of improvement for current bug-prediction tools. In particular, we recommend looking at areas that are frequently involved in merge conflicts, to determine why we see such high correlations with future bug fixes.

- *Project Managers:* We have shown a strong correlation between code smells and merge conflicts. We therefore recommend that project managers give code smells more priority when planning out software development, as it can yield benefits a lot earlier than expected.

For the future, we plan to take some the findings from our field study (detailed in Chapter 4 to better inform our merge conflict difficulty prediction. In particular, our findings for the number of information sources, and their *packing* can be used to improve the performance of our existing model.

Finally, in the long run, we plan to investigate where exactly existing tools fall short of developers' needs. We also plan to bridge those gaps through better tooling. Merge conflicts will be part of software development for the foreseeable future. However, we can make developers' lives easier, if we take the time to understand their needs and workflows, and continue to build tools that support them.

BIBLIOGRAPHY

1. Companion website. `http://caius.brindescu.com/research/mc-classification`.
2. Infusion. `http://www.intooitus.com/inFusion.html`. Accessed: 2014-01-01.
3. Tiobe. http://tiobe.com/index.php/content/paperinfo/tpci/index.html.
4. Apache Maven. The Apache Software Foundation, 2017.
5. Companion website. https://goo.gl/ORpLkU, 2017.
6. GitHub. https://www.github.com/, 2017.
7. Tiobe index for april 2017. `https://tiobe.com/tiobe-index//`, 2017. Accessed: 2017-04-18.
8. Understand^{TM} Static Code Analysis Tool. Scientific Toolworks, Inc., 2017.
9. P. Accioly. Understanding Conflicts Arising from Collaborative Development. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, volume 2, pages 775–777, May 2015.
10. Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. Understanding semi-structured merge conflict characteristics in open-source Java projects. *Empirical Software Engineering*, 23(4):2051–2085, August 2018.
11. John G Adair. The Hawthorne effect: A reconsideration of the methodological artifact. *Journal of applied psychology*, 69(2):334, 1984.
12. I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma. An Empirical Examination of the Relationship between Code Smells and Merge Conflicts. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 58–67, November 2017.
13. Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. Can Testedness Be Effectively Measured? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 547–558, Seattle, WA, USA, 2016. ACM.
14. Iftekhar Ahmed, Umme Ayda Mannan, Rahul Gopinath, and Carlos Jensen. An empirical study of design degradation: How software projects get worse over time. 08 2015.

15. Sven Apel, Olaf Leßenich, and Christian Lengauer. Structured merge with auto-tuning: balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 120–129. ACM, 2012.

16. Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 190–200. ACM, 2011.

17. Francesca Arcelli Fontana, Elia Mariani, Alessandro Morniroli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. pages 450 – 457, 04 2011.

18. Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. Clone smells in software evolution. pages 24 – 33, 11 2007.

19. Henrike Barkmann, Rüdiger Lincke, and Welf Löwe. Quantitative evaluation of software quality metrics in open-source projects. In *2009 International Conference on Advanced Information Networking and Applications Workshops*, pages 1067–1072. IEEE, 2009.

20. Stephen P Berczuk and Brad Appleton. *Software configuration management patterns: effective teamwork, practical integration*. Addison-Wesley Longman Publishing Co., Inc., 2002.

21. Thomas Berlage and Andreas Genau. A framework for shared applications with a replicated architecture. In *ACM Symposium on User Interface Software and Technology*, pages 249–257. Citeseer, 1993.

22. Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18. ACM, 2007.

23. Valdis Berzins. Software merge: semantics of combining changes to programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1875–1903, 1994.

24. Jacob T Biehl, Mary Czerwinski, Greg Smith, and George G Robertson. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1313–1322. ACM, 2007.

25. David Binkley, Susan Horwitz, and Thomas Reps. Program Integration for Languages with Procedure Calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, January 1995.

26. Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

27. Christian Bird and Thomas Zimmermann. Assessing the Value of Branches with What-if Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 45:1–45:11, New York, NY, USA, 2012. ACM.

28. Barry W Boehm, John R Brown, and Mlity Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press, 1976.

29. Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

30. Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

31. Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 168–178. ACM, 2011.

32. Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering*, 39(10):1358–1375, 2013.

33. Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005.

34. Jim Buffenbarger. Syntactic software merging. In *Software Configuration Management*, pages 153–172. Springer, 1995.

35. G. Canfora, L. Cerulo, and M. D. Penta. Identifying Changed Source Code Lines from Version Repositories. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 14–14, May 2007.

36. Marcelo Cataldo and James D. Herbsleb. Communication networks in geographically distributed software development. In *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*, CSCW '08, page 579–588, New York, NY, USA, 2008. Association for Computing Machinery.

37. Marcelo Cataldo and James D. Herbsleb. Coordination Breakdowns and Their Impact on Development Productivity and Software Failures. *IEEE Trans. Softw. Eng.*, 39(3):343–360, March 2013.

38. G. Cavalcanti, P. Accioly, and P. Borba. Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, October 2015.

39. Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.*, 1(OOPSLA):59:1–59:27, October 2017.

40. Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.

41. J. Cohen, P. Cohen, S.G. West, and L.S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Taylor & Francis, 2013.

42. Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

43. William W. Cohen. Fast effective rule induction. In *Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

44. Jonathan Corbet and Greg Kroah-Hartman. 2016 Linux Kernel Development Report. Technical report, The Linux Foundation, August 2016.

45. Catarina Costa, Jair Figueiredo, Anita Sarma, and Leonardo Murta. Tipmerge: recommending developers for merging branches. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 998–1002. ACM, 2016.

46. Catarina Costa, José J. C. Figueiredo, Gleiph Ghiotto, and Leonardo Gresta Paulino Murta. Characterizing the Problem of Developers' Assignment for Merging Branches. *International Journal of Software Engineering and Knowledge Engineering*, 24(10):1489–1508, 2014.

47. D. S. Cruzes and T. Dyba. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 275–284, September 2011.

48. Isabella A. da Silva, Ping H. Chen, Christopher Van der Westhuizen, Roger M. Ripley, and André van der Hoek. Lighthouse: Coordination Through Emerging Design. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, Eclipse '06, pages 11–15, New York, NY, USA, 2006. ACM, ACM.

49. C. R. B. de Souza and D. F. Redmiles. The Awareness Network, To Whom Should I Display My Actions? And, Whose Actions Should I Monitor? *IEEE Transactions on Software Engineering*, 37(3):325–340, May 2011.

50. Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. "Breaking the Code", Moving Between Private and Public Work in Collaborative Software Development. In *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*, GROUP '03, pages 105–114, New York, NY, USA, 2003. ACM.

51. Cleidson RB De Souza, David Redmiles, and Paul Dourish. Breaking the code, moving between private and public work in collaborative software development. In *Proceedings of the 2003 International ACM SIGGROUP conference on Supporting group work*, pages 105–114. ACM, 2003.

52. Lucas Batista Leite De Souza and Marcelo De Almeida Maia. Do software categories impact coupling metrics? In *Proceedings of the 10th working conference on mining software repositories*, MSR '13, pages 217–220, Piscataway, NJ, USA, 2013. IEEE Press, IEEE Press.

53. Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65:127–139, 02 2003.

54. Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72:129–143, 07 2004.

55. Prasun Dewan and Rajesh Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *ECSCW 2007*, pages 159–178. Springer, 2007.

56. Dario Di Nucci, Fabio Palomba, Giuseppe De Rosa, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24, 2018.

57. Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Trans. Softw. Eng.*, 34(3):321–335, May 2008.

58. Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. 01 1992.

59. Kalhed El Emam, Säı da Benlarbi, Nishith Goel, and Shesh N. Rai. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Trans. Softw. Eng.*, 27(7):630–650, July 2001.

60. H.C. Estler, M. Nordio, C.A. Furia, and B. Meyer. Awareness and Merge Conflicts in Distributed Software Development. In *2014 IEEE 9th International Conference on Global Software Engineering (ICGSE)*, pages 26–35, August 2014.

61. Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

62. Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

63. Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

64. M. Fowler. Continuous integration. `http://martinfowler.com/articles/continuousIntegration.html`. Accessed: 2017-03-1.

65. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 01 1999.

66. Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 171–180. ACM, 2012.

67. Emanuel Giger, Martin Pinzger, and Harald C Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 83–92. ACM, 2011.

68. GitHub Inc. Software repository. `http://www.github.com`.

69. Michael W. Godfrey and Lijie Zou. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, February 2005.

70. Ian Gorton and Anna Liu. Software Component Quality Assessment in Practice: Successes and Practical Impediments. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 555–558, New York, NY, USA, 2002. ACM.

71. Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, Kyle Rector, and Irwin Kwan. End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction*, 19(1):1–28, March 2012.

72. Rebecca E. Grinter. Using a Configuration Management Tool to Coordinate Software Development. In *Proceedings of Conference on Organizational Computing Systems*, COCS '95, pages 168–177, New York, NY, USA, 1995. ACM.

73. Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm Testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 78–88, Minneapolis, MN, USA, 2012. ACM.

74. Mário Luís Guimarães and António Rito Silva. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 342–352, Zurich, Switzerland, 2012. IEEE Press, IEEE Press.

75. Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–39, 09 2014.

76. Frank E Harrell Jr. *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer, 2015.

77. Lile Hattori and Michele Lanza. Syde: A Tool for Collaborative Software Development. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 235–238, Cape Town, South Africa, 2010. ACM.

78. Lile Hattori and Michele Lanza. Syde: a tool for collaborative software development. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 235–238. ACM, 2010.

79. D.A. Hensher and P.R. Stopher. *Behavioural Travel Modelling*. Croom Helm, 1979.

80. G. J. Holzmann, R. Joshi, and A. Groce. Swarm Verification Techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, November 2011.

81. Torsten Hothorn and Brian S Everitt. *A Handbook of Statistical Analyses Using R*. Chapman and Hall/CRC, 2014.

82. Riitta Jääskeläinen. Think-aloud protocol. *Handbook of translation studies*, 1:371–374, 2010.

83. D. Jackson and D.A. Ladd. Semantic Diff: A tool for summarizing the effects of modifications. In *, International Conference on Software Maintenance, 1994. Proceedings*, volume 94, pages 243–252, September 1994.

84. George H John, Ron Kohavi, and Karl Pfleger. Irrelevant features and the subset selection problem. In *Machine Learning Proceedings 1994*, pages 121–129. Elsevier, 1994.

85. Eric J. Johnson, Steven Bellman, and Gerald L. Lohse. Defaults, framing and privacy: Why opting in-opting out1. *Marketing Letters*, 13(1):5–15, Feb 2002.

86. Eric J Johnson and Daniel Goldstein. Do defaults save lives?, 2003.

87. Elmar Jürgens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 485–495. IEEE, 01 2009.

88. Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. pages 119–128, 10 2010.

89. Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

90. Bakhtiar Khan Kasi and Anita Sarma. Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 732–741, Piscataway, NJ, USA, 2013. IEEE Press.

91. Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17:243–275, 06 2012.

92. Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

93. Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic Identification of Bug-Introducing Changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

94. Rainer Koschke. Survey of research on software clones. 01 2006.

95. Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830, 2017.

96. Philippe Kruchten, Robert Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, 11 2012.

97. Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer-Verlag New York Inc, 2006.

98. Olaf Leßenich, Sven Apel, and Christian Lengauer. Balancing precision and performance in structured merge. *Automated Software Engineering*, 22(3):367–397, 2015.

99. Olaf Leßenich, Janet Siegmund, Sven Apel, Christian Kästner, and Claus Hunsen. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering*, 25(2):279–313, 2018.

100. Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80:1120–1128, 07 2007.

101. Ernst Lippe and Norbert Van Oosterom. Operation-based merging. In *ACM SIGSOFT Software Engineering Notes*, volume 17, pages 78–87. ACM, 1992.

102. Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248–256, 2012.

103. Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*, pages 173–182. IEEE, 2001.

104. Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 10 2004.

105. Raúl Marticorena, Carlos López, and Yania Crespo. Extending a taxonomy of bad code smells with metrics. In *7th ECCOP International Workshop on Object-Oriented Reengineering (WOOR)*, page 6. Citeseer, 2006.

106. Simon Mason and N.E. Graham. Areas beneath the relative operating characteristics (roc) and relative operating levels (rol) curves: Statistical signié cance and interpretation. *Quarterly Journal of the Royal Meteorological Society*, 128:2145 – 2166, 07 2002.

107. Mary L McHugh. Interrater reliability: The kappa statistic. *Biochemia medica: Biochemia medica*, 22(3):276–282, 2012.

108. S. McKee, N. Nelson, A. Sarma, and D. Dig. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 467–478, September 2017. ISSN:.

109. Gleiph Ghiotto Lima Menezes, Leonardo Gresta Paulino Murta, Marcio Oliveira Barros, and Andre Van Der Hoek. On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github. *IEEE Transactions on Software Engineering*, 2018.

110. T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, May 2002.

111. Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and AyBener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

112. Webb Miller and Eugene W Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.

113. A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April 2000.

114. Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, July 2002.

115. Naouel Moha, Jihene Rezgui, Petko Valtchev, and Ghizlane El-Boussaidi. Using fca to suggest refactorings to correct design defects. 4923:269–275, 01 2006.

116. Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

117. Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Software Engineering*, pages 1–44, 2019.

118. A. Nieminen. Real-time Collaborative Resolving of Merge Conflicts. In *Proceedings of the 2012 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2012)*, COLLABORATECOM '12, pages 540–543, Washington, DC, USA, 2012. IEEE, IEEE Computer Society.

119. Yuichi Nishimura and Katsuhisa Maruyama. Supporting merge conflict resolution by using fine-grained code change history. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 661–664. IEEE, 2016.

120. Steffen Olbrich, Daniela Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*, pages 390–400. IEEE Computer Society, 10 2009.

121. Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *Software maintenance and evolution (ICSME), 2014 IEEE international conference on*, pages 101–110. IEEE, 2014.

122. Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on*, pages 268–278. IEEE, 2013.

123. Jeffrey Parsons and Chad Saunders. Cognitive heuristics in software engineering applying and extending anchoring and adjustment to artifact reuse. *IEEE Transactions on Software Engineering*, 30(12):873–888, 2004.

124. Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

125. Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel Changes in Large-scale Software Development: An Observational Case Study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, July 2001.

126. Peter Pirolli and Stuart Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of International Conference on Intelligence Analysis*, volume 5, pages 2–4, 2005.

127. João Gustavo Prudêncio, Leonardo Murta, Cláudia Werner, and Rafael Cepêda. To lock, or not to lock: That is the question. *Journal of Systems and Software*, 85(2):277–289, 2012.

128. Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 432–441. IEEE, 2013.

129. Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

130. Roger M. Ripley, Ryan Y. Yasui, Anita Sarma, and André van der Hoek. Workspace Awareness in Application Development. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, Eclipse '04, pages 17–21, New York, NY, USA, 2004. ACM.

131. Chanchal Roy and James Cordy. A survey on software clone detection research. *School of Computing TR 2007-541*, 01 2007.

132. Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.

133. William Samuelson and Richard Zeckhauser. Status quo bias in decision making. *Journal of risk and uncertainty*, 1(1):7–59, 1988.

134. Anita Sarma, Gerald Bortis, and Andre van der Hoek. Towards Supporting Awareness of Indirect Conflicts Across Software Configuration Management Workspaces. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 94–103, Atlanta, Georgia, USA, 2007. ACM.

135. Anita Sarma, Zahra Noroozi, and André Van Der Hoek. Palantir: Raising Awareness among Configuration Management Workspaces. pages 444–454, 2003.

136. Anita Sarma, David Redmiles, and André van der Hoek. Empirical Evidence of the Benefits of Workspace Awareness in Software Configuration Management. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 113–123, New York, NY, USA, 2008. ACM.

137. Anita Sarma, David F Redmiles, and Andre Van Der Hoek. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, 38(4):889–908, 2012.

138. Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. Building empirical support for automated code smell detection. 01 2010.

139. Francisco Servant, James A. Jones, and André van der Hoek. CASI: Preventing Indirect Conflicts Through a Live Visualization. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '10, pages 39–46, New York, NY, USA, 2010. ACM.

140. Haifeng Shen and Chengzheng Sun. Syntax-based reconciliation for asynchronous collaborative writing. In *2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 10–pp. IEEE, 2005.

141. Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

142. Emad Shihab, Christian Bird, and Thomas Zimmermann. The Effect of Branching Strategies on Software Quality. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 301–310, New York, NY, USA, 2012. ACM.

143. Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 858–870, Seattle, WA, USA, 2016. ACM.

144. Zhongbin Sun, Qinbao Song, and Xiaoyan Zhu. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1806–1817, 2012.

145. Cass R Sunstein and Richard H Thaler. Libertarian paternalism is not an oxymoron. *The University of Chicago Law Review*, pages 1159–1202, 2003.

146. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 321–332. IEEE, 2016.

147. Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 386–396. IEEE, 2012.

148. Ayse Tosun and Ayse Bener. Reducing false alarms in software defect prediction by decision threshold optimization. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 477–480. IEEE Computer Society, 2009.

149. Burak Turhan, Ayse Tosun, and Ayse Bener. Empirical evaluation of mixed-project defect prediction models. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 396–403. IEEE, 2011.

150. Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Software Configuration Management Workshop: Proceedings of the 3 rd international workshop on Software configuration management*, volume 12, pages 68–79, 1991.

151. Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.

152. Jan Wloka, Barbara Ryder, Frank Tip, and Xiaoxia Ren. Safe-Commit Analysis to Facilitate Team Software Development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 507–517, Washington, DC, USA, 2009. IEEE, IEEE Computer Society.

153. Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. Predicting crashing releases of mobile applications. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 29. ACM, 2016.

154. Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *WCRE*, volume 13, pages 242–251, 10 2013.

155. Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 17–23. ACM, 2011.

156. Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 182–191. ACM, 2014.

157. Thomas Zimmermann. Mining Workspace Updates in CVS. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 11–, Washington, DC, USA, 2007. IEEE Computer Society.

158. Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

159. Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.