Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

What Could the Source Code History Tell UsAbout Errors

Luis Henrique Vieira Amaral

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2020

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

What Could the Source Code History Tell UsAbout Errors

Luis Henrique Vieira Amaral

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Uirá Kulesza	Prof. Dr. Paulo Meirelles
DIMAp/UFRN	FGA/UnB

Prof. Dr. Bruno Macchiavello
Coordenador do Programa de Pós-graduação em Informática

Brasília, 18 de Fevereiro de 2020

Dedicatória

Dedico este trabalho à minha família e aos meus amigos que sempre me apoiaram de forma incondicional e que estiveram presentes em todos os momentos dessa jornada.

Agradecimentos

Primeiramente, quero agradecer ao meu Orientador, Professor *Rodrigo Bonifácio*, pela dedicação, apoio incondicional e principalmente pela paciência que teve durante este trabalho. Se não fosse por você, eu teria desistido em vários momentos de dificuldades. Além de Orientador e professor, você foi meu amigo, meu parceiro de viagem, um irmão que sempre me apoiou e que tenho gratidão imensurável. Também desejo agradecer a professora *Edna Dias Canedo* que me ajudou de todas as maneiras ao longo dessa jornada. Muito obrigado pelos projetos de pesquisa, pelas experiências compartilhadas, pelos melhores bolos de chocolate e pelo feijão que você fazia pra gente. Pessoas como vocês me fazem acreditar que é possível mudar o mundo, melhorar a educação e a pesquisa do Brasil e que é importante ajudar as pessoas que estão em nossa volta.

Não menos importante, agradeço a minha mãe *Maria Cleide V. Amaral.* A pessoa mais importante da minha vida, que me ensinou a ler e escrever quando eu estava no Jardim. Que deixou de fazer muitas coisas ao longo da vida para investir na educação dos filhos. Que não me deixou desistir do mestrado quando estava sem rumo e perspectiva de vida. Uma mulher que sempre segura a barra quando as coisas estão difíceis e que tem um coração gigante. Espero que um dia eu possa retribuir pelo menos 10% do que você proporcionou na minha vida. Obrigado por ser a melhor mãe do mundo.

Agradeço ao meu pai, *Antônio dos Reis Amaral* e meu irmão, *Gustavo Henrique V. Amaral.* Sem dúvida, vocês foram essenciais para que este trabalho fosse concluído. Sempre estavam presentes quando eu precisei e sempre me apoiaram em minhas escolhas pessoais. Agradeço à *Delícia*, que estava comigo em quase todos os momentos e que aguentou meus estresses, dramas e a insegurança. Você é uma menina de ouro, que me ensinou a ver o mundo com outra perspectiva e que tenho muito orgulho de ter namorado. Por fim, agradeço aos meus amigos *Walter Lucas, Leomar Camargo e Ranielson* por estarem comigo durante todo o programa.

Resumo

Quase todos os desenvolvedores criam software usando uma abordagem de desenvolvimento colaborativo. Nesse cenário, após concluir a tarefa, desenvolvedores submetem suas contribuições a um repositório remoto— disponibilizando-os para outros colaboradores. Enquanto sequências de revisões e trabalho paralelo aumentam a produtividade do software, por outro lado, alterações simultâneas podem causar conflitos de mesclagem. Além disso, quando duas entidades de software (por exemplo, classes, métodos, campos) são mudados frequentemente de forma simultânea, eles se tornam dependentes de co-alteração um do outro— um tipo de dependência que geralmente está oculta dos desenvolvedores. Alguns estudos investigam como reduzir as dependências de co-alterações e de conflitos sintáticos em operações de mesclagem, mas existem algumas questões em aberto sobre esse tipo específico de dependência e se conflitos de mesclagem introduzem bugs. Neste trabalho, esclarecemos essas questões e apresentamos os resultados de uma avaliação empírica que explora os dados históricos de 34 projetos Apache, para verificar se as alterações que introduziram erros (BIC) se correlacionam com cenários de mesclagem conflitantes e commits que levam a dependências de co-alterações. Nosso estudo apresenta que o SZZ - um algoritmo para encontrar os commits que introduziram erros - rotulou 3,62 % dos cenários de mesclagem em conflito como um commit de introdução de erros e 18,77 % dos commits levam a dependências de co-alterações . Nossos resultados trazem várias implicações para pesquisadores e profissionais. Entre eles, evidenciamos que os desenvolvedores não devem ter medo de resolver conflitos, já que apenas uma pequena porcentagem de os cenários de mesclagem em conflito foi suspeita de ter introduzido bugs.

Palavras-chave: evolução de código-fonte, errors de software, SZZ, conflitos de mesclagem, dependências de co-alterações

Abstract

Almost all developers build software using a collaborative development approach. In this scenario, after concluding a task, developers commit their contributions to a remote repository—making them available to other contributors. While sequences of revisions and parallel working increase software productivity, on the other hand, concurrent changes might cause merge conflicts. Moreover, when two software entities (e.g. classes, methods, fields) are frequently changed together, they become co-change dependent on each other—a kind of dependency that is often hidden from the developers. Some studies investigate how to reduce co-change dependencies and syntactic conflicts in merge operations, but there are some open questions about whether this particular kind of dependency and merge conflicts introduce bugs. In this work, we shed light upon these questions and present the results of an empirical assessment that mine the historical data of 39 Apache projects, to verify if bug-introducing changes (BIC) correlate with conflicted merge scenarios and commits that lead to co-change dependencies. Our study presents that SZZ — an algorithm to find bug introducing commits — labeled 3.62% of conflicted merge scenarios as being a bug-introducing commit and 18.77% of the commits that lead to co-change dependencies. Our results bring several implications for both researchers and practitioners. Among them, we give evidence that developers should not be afraid of solving conflicts since just a small percentage of the conflicted merge scenarios are suspicious of having introduced bugs.

Keywords: source-code evolution, Mining Software Repositories, SZZ, merge conflicts, Co-change Dependencies

Contents

1 Introduction **1**

1.1 Problem Statement and Research Questions 2
1.2 Objectives 3
1.3 Document Organization 4

2 Background **5**

2.1 Version Control Systems 5
2.2 Bug Tracking Systems 8
2.3 The SZZ Algorithm 9
2.4 Co-change evolution 12

3 Methodology **14**

3.1 Project Selection Criteria 14
3.2 Finding Bug-introducing commits 15
3.3 Re-playing merge scenarios and collecting their characteristics 17
3.4 Tool support for collecting merge scenarios 19
3.4.1 Computing Co-change Dependencies 19

4 Results **23**

4.1 Exploratory data Analysis 23
4.2 Disclosing conflicting merge scenarios linked to BICs 29
4.2.1 R.Q.1: To what extent conflicting merge scenarios correspond to the bug introduction contributions? 33
4.2.2 What are the characteristics of the conflicted merge scenarios that introduced error? 36
4.3 RQ2 How to predict bugs on conflicted merge scenarios? 39
4.4 RQ3 To what extent commits with co-change dependencies relate to bug-introducing changes? 43

5 Conclusions **49**
5.1 Contributions 50
5.2 Threats to Validity 50
5.3 Future Works 51

6 Related Works **52**
6.1 Research on Merge conflicts 52
6.2 Research on Co-change Dependencies 54

References **56**

List of Figures

2.1 Example of Three-way line-based textual merge scenario [41] 7
2.2 Jira Issue Tracking System 9
2.3 Phase I of SZZ workflow.[8] 10
2.4 Overview of the SZZ-Unleashed workflow [8] 10
2.5 Table with the SZZ limitations collected from [49] 11
2.6 Draco refactoring approach overview [17] 13

3.1 Number of projects selected. 15
3.2 Workflow of SZZ-Unleashed [8] 16
3.3 Sample of git merge operation. 17

4.1 Log-scale box-plot with the number of closed bug issues and linked bug-fixing commits per project 24
4.2 Rate of the linked fixes over the number of issues per project 25
4.3 Log-scale box-plot with the number of BFC and BIC per project 25
4.4 Histogram with the rate value of (BFC - BIC) / BFC per project 26
4.5 Histogram with the rate value of BFCs that fixed errors introduced by BICs 27
4.6 Log-scale histograms with the number of merge scenarios before(left), and after filtering the set of projects (right) 28
4.7 Log-scale histograms with the number of conflicted merge scenarios before (left), and after filtering the set of projects (right) 28
4.8 Histograms with the rate value (Conflicts/Merges) before (left), and after filtering the set of projects (right)) 28
4.9 Rate of the linked BFCs over the total number of issues (labeled as bugs) per project 31
4.10 Log-scale box-plot with the number of all commits, merge scenarios, and conflicted merge scenarios per project 32
4.11 Histogram with the rate value of merges over commits per project 32
4.12 Histogram with the rate value of conflicted merge scenarios per project . . 33

4.13 Histogram with the rate value of bug-induced commits over all commits per project 34
4.14 Conflicting merge commits linked to BICs 34
4.15 Histogram with the log-scale number of files changed by the right branches of the conflicted merge commits linked to bug-introducing commits 37
4.16 Histogram with the log-scale number of files changed by the left branches of the conflicted merge commits linked to bug-introducing commits 38
4.17 Histogram with the log-scale number of lines additions by the right branches of the conflicted merge commits linked to bug-introducing commits 39
4.18 Histogram with the log-scale number of lines additions by the left branches of the conflicted merge commits linked to bug-introducing commits 40
4.19 Histogram with the log-scale number of lines removals by the right branches of the conflicted merge commits linked to bug-introducing commits 41
4.20 Histogram with the log-scale number of lines removals by the left branches of the conflicted merge commits linked to bug-introducing commits 42
4.21 Histogram with the log-scale number of active authors on the right branches of the conflicted merge commits linked to bug-introducing commits 43
4.22 Histogram with the log-scale number of active authors on the left branches of the conflicted merge commits linked to bug-introducing commits 44
4.23 Histogram with the log-scale number of commits on the right branches of the conflicted merge commits linked to bug-introducing commits 45
4.24 Histogram with the log-scale number of commits on the left branches of the conflicted merge commits linked to bug-introducing commits 46
4.25 (Log-scale) Histogram with the log-scale number of files in conflict for the induced merge commits 47
4.26 (Log-scale) Frequency of conflicted merge commits that introduced bugs over the week 47
4.27 Spearman correlation of the features collected from conflicted merge scenarios 48

List of Tables

3.1 Features collected from merge scenarios. 18
3.2 Collected values from the Induced merge commits of the project Spark. . . 19
3.3 Summary of the number of merges, conflicted merges, and conflicted rate . 20

4.1 Phase I of SZZ-Unleashed 23
4.2 Phase II of SZZ-Unleashed 24
4.3 Summary of the number of merges, conflicted merges, and conflicted rate for the filtered projects 27
4.4 Initial results of merge scenarios that Induced errors 29
4.5 Summary of projects with more than 5% of conflicting merge scenarios linked to BICs. #CM means number of conflicting merge scenarios, #BICs means the number of BICs related to conflicting merge scenarios, and Rate stands for the percentage of #CM over #BICs. 35
4.6 Summary of the characteristcs of our dataset with simple merge scenarios . 36
4.7 Number of files changed, lines additions and lines removals for both Right and Left branches of induced conflicted merges. 39
4.8 Performance of the training classifiers — all conflicting merge scenarios, simple merge scenarios, and complex merge scenarios 42
4.9 Summary of the metrics NOCC, SOMC, ... 45
4.10 Simple linear regression of *buggy ratio* on NOCC 46
4.11 Simple linear regression of *buggy ratio* on SOCC 46

Chapter 1

Introduction

Almost every software is built in a collaborative development environment, in which tasks are distributed to the developers, who work separately, on their local machine. After concluding a task, the project is updated and pushed to the remote repository to be available for other contributors. Even though modern Version Control Systems (VCS) are used to help the code integration, conflicts usually emerge due to concurrent work and become more complex as further developments are made without being integrated [28]. Earlier studies reported that such conflicts appear frequently and that developers need to spend a considerable amount of time and effort to solve them, considering that understanding integration conflicts might be a complicated and error-prone task. Regrettably, merging is burdensome and interrupts programming flow. Consequently, some developers do not merge as often as desired. Teams used to avoid parallel work because of complicated merges [27, 45], and developers rush their tasks to avoid being the ones responsible for the merge [19].

An important field of research in software engineering is to identify a bug or defect before it is pushed to the repository, and thus specialists in industry have proposed many best practices to control merge conflicts. For instance, code reviewing in pull-based development is a well-established practice where developers make pull requests after changing anything in the code, and other developers review and accept or reject them. Furthermore, Continuous Integration [23, 21] recommends frequent merges and check-ins to avoid conflicts staying undetected for too long. Finally, developer teams use code integration tools with automated tests to prevent bugs injection. Most development teams work around an issue tracking system. Bugs are recorded using an issue report, where each bug is often detailed using a description (name and summary), a severity (normal, critical, blocker), and a report date. A common approach to linking the code contributions to a specific issue (a bug report or a feature request), is to specify the id of the issue in the commit message after applying any change related to it to the software. Substantially issues are

reported as bugs after some changes are concluded. In this case, they are classified as bug introduction changes [52]. Previous studies relate that developers spend half of their time in bugs [36]. Thus software bugs are costly to be fixed [37]. Sliwerski et al. [52] have proposed a well-known approach to identify a bug-introducing change, named latter as SZZ framework [35]. Rodriguez et al. [50] points out that around 65%-77% of the Bug-Fixing commit (BFC) were caused by a Bug-Introducing Commit (BIC). Also, 10% of bug-fixes in ElasticSearch and 24% in Nova were caused by co-evolution, compatibility issues, or bugs in external API [50]. Software modularity is essential in order to be modified and get improvements and might be compromised if the developer does not take care of crosscutting patterns [18]. According to Cesar et al. [17], a special kind of dependency, is motivated by a set of co-changes between two software elements, assuming there is no static dependency between them. Besides, they implemented a framework that find co-change dependencies and suggest code refactoring to avoid them.

Even using all available tools to avoid bugs, after mining any Issue Tracking System (ITS) such as JIRA, it is possible to perceive that not all changes made in the past have been beneficial. Altogether, there are still some relevant research questions that require further investigation. What if these merge conflicts and co-change dependencies were responsible for introducing new bugs to the software? Can we predict and avoid them? If the answer is no, can we at least aware developers depending on some characteristics of their merge conflicts?

1.1 Problem Statement and Research Questions

Even though there are several studies related to merging conflicts and bug prediction, only a few works correlate these conflicts with bug introduction changes. Therefore, it is still not clear whether or not the merge conflicts induce bugs. Answering this question is important to characterize how the occurrence of merge conflicts could damage the overall quality of a system. Also, this work aims to investigate a related research topic, about the impact of co-change dependencies on the introduction of bugs. Altogether, we aim to answer the following general research questions:

(RQ1) *To what extent conflicting merge scenarios correspond to the bug introduction contributions?* Answering this research question is crucial because it reveals the impact of merging operations in the quality of the systems.

(RQ2) *How to predict bugs on conflicting merge scenarios?* Answering this research question is relevant because it brings a better understanding of the properties of merge

scenarios, as well might reveal how to explore the previous research questions further.

(RQ3) *To what extent commits with co-change dependencies relate to bug-introducing changes?* Answering this research question is essential because it could reveal a negative side of co-change dependencies, which has not been explored before.

1.2 Objectives

The main goal of this study was to build a general understanding of the effect of merge operations and co-change dependencies during the source code evolution, in terms of introducing new bugs. More specifically, in order to achieve the goal of this research, we have accomplished several more specific goals. For instance,

- Research Question 1 is related to an essential problem of software engineering research (verify which commits introduce bug during the software development). We used a well-known algorithm of automatic identification of bug-introducing changes named SZZ [52] with some adaptions. We built a dataset with the source-code history of open-source projects, which we used as input of SZZ-Unleashed [8], and as a result, we obtain the commits blamed to be a BIC, by verifying if SZZ-Unleashed blamed the id of the merge commit for being a Bug-introducing commit.

- Answering question RQ.2 is challenging because it involves a sophisticated study setting, in particular, because there are several approaches to analyze merge operation strategies and conflicts prediction. First of all, it is necessary to get a good comprehension of merge strategies, merge conflicts, attributes that might induce conflicts, and how to treat and analyze this information. Then, we conducted an empirical study on open-source projects, reproducing all merges from them and computing metrics of *size* of the commits (e.g., number of contributors, number of days, number of files in change). We applied some methodologies of Machine Learning and Statistics in a data set containing several merges from these projects. This stage aims to build a model to extract information and possible predictions about when a merge scenario will introduce bugs, when resolving the conflicts, which will allow us to answer additional questions, such as "do the number of commits involved in a merge scenario induce bugs?"

- We answered Research Question 3 by checking if the entities in commits that lead to co-change dependencies were also changed when fixing the bug. We used the framework Draco [17] to collect co-change dependencies over the same open-source projects we answered 2.

1.3 Document Organization

This Chapter briefly motivates the covered problems through this research. Then, Chapter 2 detail and review the essential concepts used throughout this work. Furthermore, chapter 3 presents related works. We described the study settings and performed procedures in Chapter 4. Moreover, in Chapter 5, we present the results of our research. Finally, in Chapter 6, we present the conclusions, threats to validity, and future works.

Chapter 2

Background

In this chapter we describe and explain concepts forming the basis to understand this work. First of all, in Section 2.1, we introduce Version Control Systems and explains an overview of merge techniques and merge conflicts. The SZZ, algorithm to find bug introducing change, is explained in section 3.2. Basic concepts of co-change evolution, necessary to answer 2, are presented in section 2.4.

2.1 Version Control Systems

Version Control Systems (VCS) and Issue Tracking Systems (ITS) contain huge volumes of historical information that can give deep perception of the software project evolution [22]. During development and maintenance of any codebase, bugs and regressions might happen frequently. In order to be solved, the developer may need to revert some code or configuration to earlier versions. In addition, it will be hard to see who is contributing with a patch and is almost impossible to avoid that developers break each other's code if there is no version control and all developers can work in only on one version [53]. Version Control Systems record file and or a set of files trough the time, being possible to return to a previous version if it is necessary. This is an approach widely used not only in software development environment but also in any projects context, where the possibility to tracking updates and to reverse something wrong is crucial for loosing nothing important. These systems were improved over the time and according to the GIT documentation [11] (a well known VCS), they are classified in three types: Local, Centralized and Distributed Version Control Systems.

Usually developers used to create new folder in their local machine to backup a project or to save previous versions of them. Nevertheless, doing so could be an error-prone strategy when the project is getting bigger and some mistakes could happen by overwriting old files and replacing functional version of it by a variant with bugs. In order to solve

this problem, Local VCSs were created, where a log with the changes in a project were saved in a local database. In addition, all files were under revision and could be traceable anytime. According to git[1] documentation [11], Revision Control System (RCS) was the most popular tool of LVCS and it is still available in many computers nowadays. The next major problem begins when more developers needed to contribute in a same project. To deal with this, CVCS (such as CVS, Subversion[2], and Perforce) was created, where a centralized server contained all the versioned files and the development contributors could check out the latest snapshots or files from the central place. This methodology was the standard of Version Control Systems for years [11].

The trouble here begin when the central server is down or if the hard drive containing the system has been corrupted. Maintaining a big system centralized without the properly backup means that it is possible to lose everything if the server fails. Distributed Version Control Systems (such as GIT, Mercurial, Bazaar or Darcs) arise to mitigate this issue. Instead of just checkout the latest updates or files, each client mirror the entire repository project, including all the historical versions and changes trough the time. Using this approach, it is possible to create several workflows such as hierarchical model that is not possible in centralized VCS [11].

The advantage of using a distributed version control system is that each developer mirror the entire project repository creating his/her personal copy. Parallel development increases the project efficiency and decreases external issues such as server breakdown or the requirement of working in a specific place and or machine. But there is no free lunch and the price of this approach is the necessity of conducting merge operations frequently, integrating the personal copies into a new version of the shared repository. Even though VCS generally have integration functionality for merging different versions, usually the merging of source code proceeds automatically, but if there are conflicts, it is necessary to be resolved manually. During the merge process, conflicts of parallel changes might occur and must be resolved at this moment as Tom Mens reported in his paper "A state-of-the-art Survey on Software Merging" [41, 40, 45]. As stated by him, a merge operation can be classified according to the approach that was used.

Two-way merge is in essence a diff operation between left-side and right-side commits, comparing all lines of each file and computing the differences between them. Differently, The three-way merge approach consider the base file that originate the both side commits, comparing them with the original one (see Figure 2.1). Depending on how software artifacts are represented and treated by the merging tools, it is possible to be categorized as following.

[1] `https://git-scm.com/`

[2] `https://subversion.apache.org/`

text-based when the software treats the source code as flat text files; and

structure-based when the software considers the parse tree of the files when performing the merge operations and sometimes also consider the programming language.

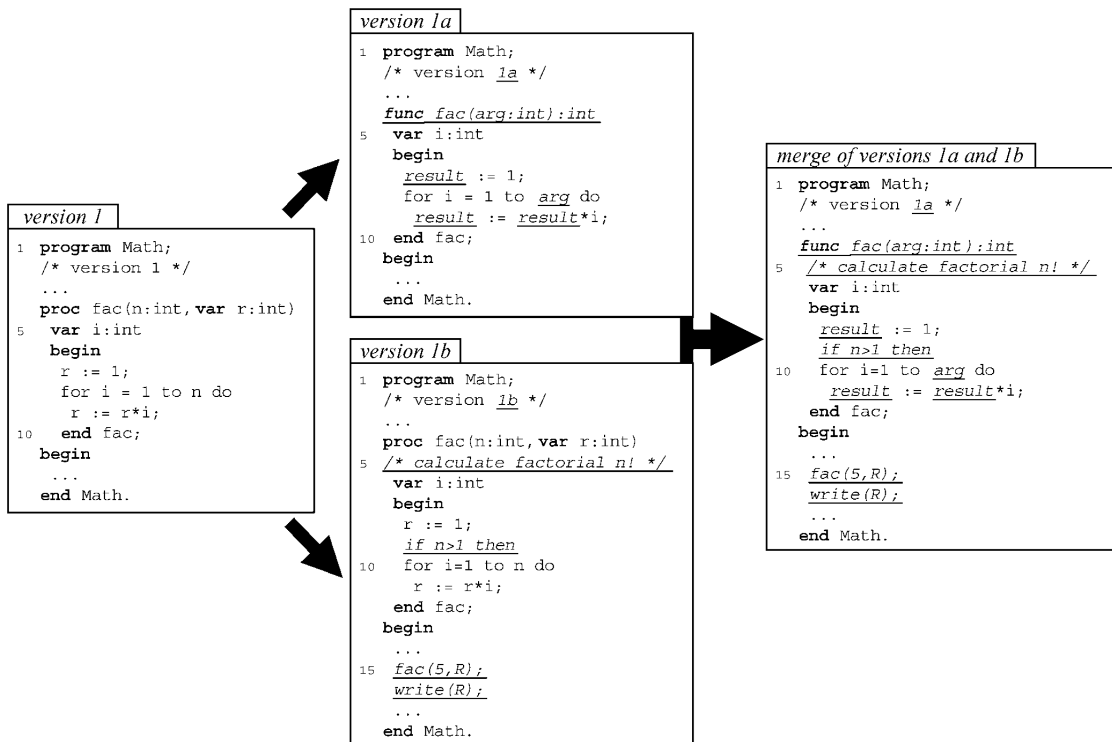

Figure 2.1: Example of Three-way line-based textual merge scenario [41]

Developers generally modify private working copies of the system avoiding them from knowing what part of the system are being modified by the co-workers at the same time. When two concurrent works are combined, merge conflicts might appear. These conflict can be textual or higher-order. Textual conflict appear when two developers make changes at the same chunk of the code. VCS allow the first developer to publish the change but in order to avoid overwriting, it prevents the second one, until the code integration have been completed [9].

According to Mens [41], a merge conflict do not necessarily need two developers to make changes in the same software file. It also could happen if one developer make changes in a specific part of the software that is reused by other parts of the software. In addition, Brun et al. says that "more damaging conflicts arise when the VCS can integrate the developers' textual changes, but the changes are semantically incompatible and can cause compilation errors, test failures, or other problems" [9].

It is harder to resolve conflicts that were later detected because they are no longer fresh in developers' minds [23, 6]. The number of software defects increases with parallel

work, which was found to be substantially and inadequately supported by merge tools, according to the conclusion of Perry et al. [47]. Their conclusion remains today, since a most recently studied [9, 40, 2, 45] affirmed that the current tools, the processes, and project management supported, is still insufficient for the high level of parallelism degree. In addition, Brun et al. [9] point out that merge conflicts "appear not only as overlapping textual edits but also as subsequent build and test failures".

To support developers to detect conflicts earlier, AWARENESS [20] helps by informing where in the code the co-workers are currently making changes. However, while programming, it is very tough for developers to identify conflicts by themselves, because of the complex semantics of the programming languages. Besides that, AWARENESS may overload developers with excessively information being more difficult to early detect conflicts [20, 16, 24]. Even with numerous merge techniques and tools, it has been stated that between 10% 20% of the merges operation results in conflicts that need to be manually revised [26]. This means that none of the tools can completely automate the merge processes.

2.2 Bug Tracking Systems

According to Just et al [33], a Bug tracking systems are the main tools for the users to communicate with the developers, reporting bugs and suggesting new features implementation. These systems work as the central repository for handling the progress of bug reports, where developers can track unresolved bugs, discuss potential solutions for fixing them and request more information from the users [57]. In addition, developers use the information provided from bug reports to identify the possible cause of the defects and the plausible files that needed to be fixed.

As stated by Jalbert et al. [31] in some projects, 25% of the bug reports are duplicated and this situation hamper the use of ITS because developers need to manually identify them. This identification process is time-consuming and increase the already high-cost software maintenance. Bettenburg et al. [7] argue that often, duplicated reports are closed and some valuable information are discarded and this is a bad practice. Instead, this important information should be merged to the master report that are chosen by the developers. As a result, they provided recommendations to improve efficiency for a better bug tracking system.

Figure 2.2: Jira Issue Tracking System

2.3 The SZZ Algorithm

The purpose of SZZ algorithm is to identify commits that introduce bugs by combining software repositories, version control systems (VCS) such as GIT, with bug tracking systems (BTS) such as Jira. It was created by Sliwerski et al. [52] and its name was given later by using the first letters of the last name of the authors (Sliwerski, Zimmermann, and Zeller). According to Kim et al. [35], in contrast to bug-fixes that are relatively easy to obtain, the extraction of bug-introducing changes is challenging. They call bug-introducing change the modification in which a bug was injected into the software. By getting this information means to discover when does it happen, who made this modification to the code, and possibly prevent some similar bugs injection.

Basically the SZZ algorithm is divided in two steps. First of all, the issue in bug tracking system or BTS is linked to the bug fix commit. This is done by using regular expressions to search commit messages for references with the BTS issues. In case there is no BTS or it is poorly maintained, commits with messages containing the word "fix" or similar are defined as a bug-fix commit. Using a *diff* command it is possible to extract the changed lines known as hunks from these bug-fixes commits.

Secondly, SZZ uses these changed lines extracted previously to trace down all commits that also have changed the same lines previously than the bug-fix commit. This is possible, for instance, thanks to the Blame GIT functionality that is a way to track down which commit has made the last change in a specific line of code. By doing this, it is possible to discover which commit potentially introduced a bug. The date of the "blamed commits"

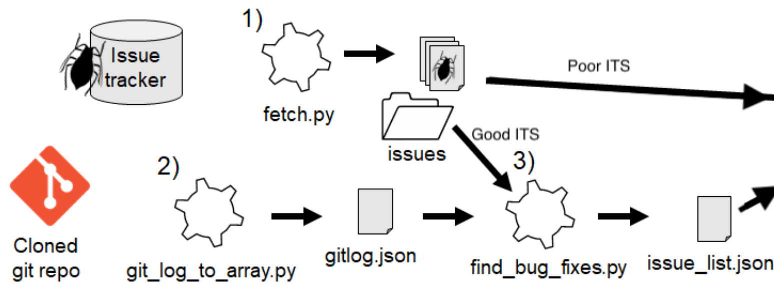

Figure 2.3: Phase I of SZZ workflow.[8]

is checked and compared with the date when the bug was reported. It is labeled as a bug-introducing commit when it happens before the report date. If the date is after the report, this commit could only be labeled as bug-introducing if it is a partial fix or if it is possibly responsible for a new bug 2.4.

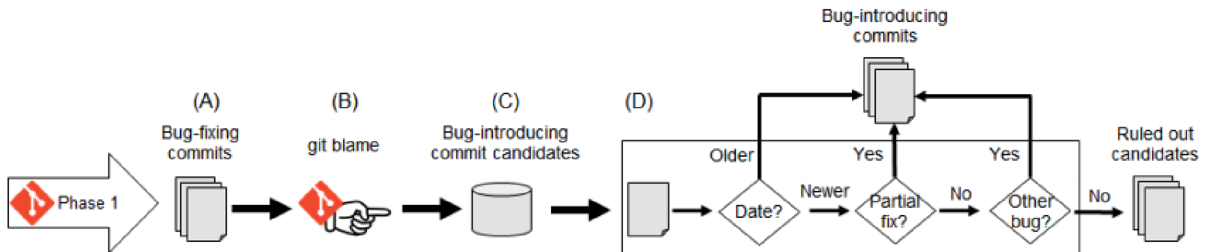

Figure 2.4: Overview of the SZZ-Unleashed workflow [8]

The first SZZ [52] has several problems. For instance, it considers stylish modifications as indentation, blank lines, comments, as possible bug-introducing change. This is incoherent because stylish changes do not modify the software behavior. For this reason, some versions with improvements of SZZ were implemented. According to Willans et al [55], a major remaining open question about SZZ is to guarantee that the lines indicated as fix-induces by SZZ algorithm are actually the source of defects. Costa et al. [13] proposed a framework to evaluate and compare five different versions of SZZ including their own version. The main idea was to generate a means for exploring the SZZ-generated data highlighting subsets that are more inaccurate, and suggesting manual inspection to them. In addition, they explains it is really difficult to have a perfect SZZ algorithim since there is no ground truth dataset to train and validate a model. This ground truth dataset should be manually created by specialists and would demand a lot of effort.

In a case study based on Systematic literature Review (SLR), Rodriguez-Perez et al [49] analyzed 187 papers that made use of the SZZ algorithm to evaluate the reproducibility and credibility of these publications in Empirical Software Engineering. Significant

contributions were presented in this study, such as, an overview of the impact that SZZ has had so far in ESE, an analysis of how these studies manage the limitations of this algorithm and how their addresses the reproducibility in their research work. Despite SZZ being largely used in ESE to locate bugs, it suffers from multiple limitations which make it error prone as reported in SLR. Considering the first part of the algorithm, the limitation relies in how bug reports are linked to commits, i.e., if the bug fix is not identified, the bug commit cannot been determined and this cause a false negative. False positive happens when a bug report does not describe a real bug, but a fixing commit is is linked to it. As reported by early studies 33.8% [29] to 40% [48] of the bugs in issue tracking system are miss-classified. The Second part of the algorithm, which is concerned with the identification of the bug-introducing commit(s), can also produce false positives and negatives, and addressing these limitations requires a manual and tedious validation process [49]. A summary of the SZZ limitations presented in SLR study is in figure 2.5:

Part	Type	Description
First part	Incomplete mapping [SLR[14]	The fixing commit cannot be linked to the bug
	Inaccurate mapping [SLR[15]]	The fixing commit has been linked to a wrong bug report, they don't correspond to each other
	Systematic bias [SLR[14]	Linking fixing commit with no *real* bug report
Second part	Cosmetic changes, comments, blank lines [SLR[16]]	Variable renaming, indentation, split lines, etc.
	Added lines in fixing commits [SLR[8]]	The new lines can not be tracked back
	Long fixing commits [SLR[8]]	The larger the fix, the more false positives
	Semantic level is weak [SLR[11]]	Changes with the same behavior are being blamed
	Correct changes at the time of being committed [SLR[8]]	Changes in other parts of the source code base trigger a bug issue in another part
	Commit Squashing [17]	Might hide the bug introducing commit, loosing authorship information

Figure 2.5: Table with the SZZ limitations collected from [49]

Even thought there are limitations in the algorithm, the impact of SZZ is significant since 458 publications cite SZZ and their variations and some of them are often been published in high quality conferences and top journals. A problem presented in the SLR is that only 13% of the SZZ publications provide a replication package and carefully describe each step that might contribute to make reproduction feasible.

To mitigate this replication problem, Borg et al. introduced SZZ Unleashed, an open implementation of SZZ algorithm, and made it available on GitHub under MIT license since June 2018 [8]. This is a Java implementation with some supporting Python scripts

to collect bug records from Issue Tracking Systems and use JGit library to facilitate interaction with git repositories. It is based on the seminal paper by Sliwerski et al. [52] and later enhancements by Williams and Spacco [55]. The workflow of SZZ Unleashed is based in three steps. Fist we should extract from issue tracker records stated as bugs and status of fixed and or closed. Than, we need to verify which of these records are linked with real bug-fixing commits. Finally we need to identify bug-introducing commits for the fixing commits. The steps related to the phase of finding bug-introducing commits of the SZZ Unleashed is show in figure 2.4. The output of the SZZ Unleashed was used as a ground truth of a classification training set to indicate commits that might require particularly careful code reviews. As a result this study found a classification score with accuracy F1 of 15% and they discussed about the small number, relating it with the necessity of oversampling for imbalanced classes and the solely use of cross validation that is not appropriate to evaluate classifiers in software engineering data with timestamps.

In a framework to evaluate the results of the SZZ Approach, da Costa et al. [13] provided a systematic mean for evaluating the data that is generated by a given SZZ Implementation. They compared five SZZ implementation using data from 10 open source projects by doing an evaluation considering three criteria: realism of bug introduction, the impact in future changes and the earliest bug appearance. In addition, they founded that the proposed improvements to SZZ tends to inflate the number of incorrectly identified bug-introducing changes. The results demonstrated that SZZ implementations still lack of mechanisms to identify bug-introducing commits in a more precisely method. In a recent empirical study, Neto et al. evaluate the impact of refactoring changes on the SZZ Algorithm. They founded that 6.5% of the lines that were flagged to be a bug-introducing change were in fact refactoring changes [44].

2.4 Co-change evolution

David Parnas, in the paper "On the Criteria To Be Used in Decomposing Systems into Modules" [46], explains the advantage of the design module implementation and define module as a work assignment unit that is flexible and could be changed without interfering other modules. Murphy et al. [43] presents another idea of module, in which it is expected to be rebuild the modularity in terms of work assignments instead of treating software decomposition considering language structures such as Java packages, classes, and interfaces. We considered in this study that software decomposition is a problem of graph partitioning. By Mitchell et al. [42] definition, the software are represented as a graph named Module Dependency Graph (MDG), where the vertices represent the source-code entities and the edges represent the dependencies between them. Existing techniques that

recommend software decomposition change refactorings, (such as a move method) usually do not explore co-change dependencies [17]. Oliveira et al. [18] considered that two entities are co-change dependents when they frequently change together and there is no static dependency, leading to a hidden dependency between them. In addition they report the advantages of adding co-change dependencies to a coarse-grained MDG. In Figure 2.6 it is presented how the framework recommend a refactoring based on co-change dependencies.

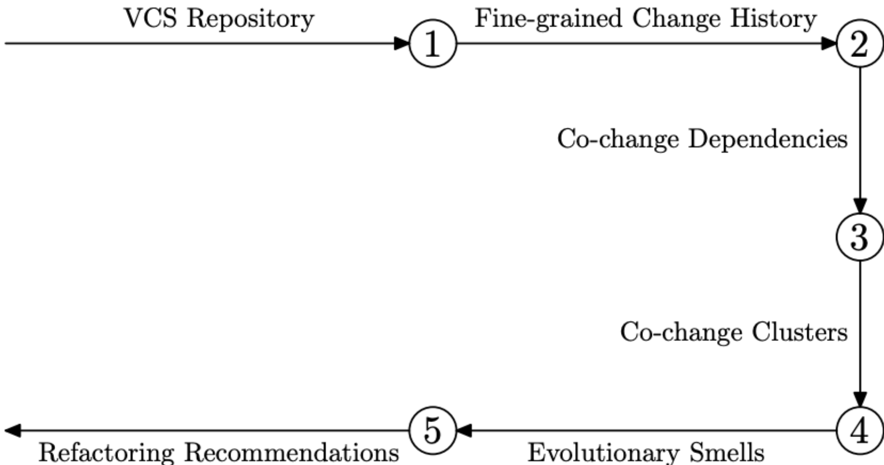

Figure 2.6: Draco refactoring approach overview [17]

Chapter 3

Methodology

In this chapter, we discuss the settings of our study. The main goal of this dissertation is to answer whether conflicts in merge operations can be held responsible for introducing defects in software along with the source code evolution. Considering the taxonomy detailed by Gerhardt et al. [25], we classify our research as a quantitative study, with applied nature, exploratory objectives, and experimental procedures. In the exploratory stage, we conduct a literature review to increase our knowledge and the background necessary to conduct the research. Furthermore, in the problem stage, we collected and read related works to see what research questions have not been answered yet. By completing this step, we defined our research questions, hypotheses, and started to construct the analysis models to answer them. Besides answering the research questions defined in the first chapter, we organized the approach in three subgroups. Data collection is a crucial stage that was necessary to conduct the study and investigate the research questions.

3.1 Project Selection Criteria

We searched for popular real-world open-source projects hosted on GitHub [12], with a large number of commits and that use JIRA [39] as its issue tracking system (ITS). More specifically, we collected Java Apache projects that meet these specifications. The decision about Java projects was because we found several tools that reproduce merge scenarios and reduce the occurrence of merge conflicts in Java projects [3, 4, 10]. Besides, we opted for the JIRA issue tracking system because the SZZ Unleashed was developed with some python scripts to collect issues from JIRA, and several Apache projects have moved from Bugzilla to JIRA. As a result, we cloned 101 most popular Java Apache projects hosted on GitHub to use in both exploratory studies. Figure 3.1 shows the number of projects that we use in each part of the study and the intersection of both studies, with 39 projects.

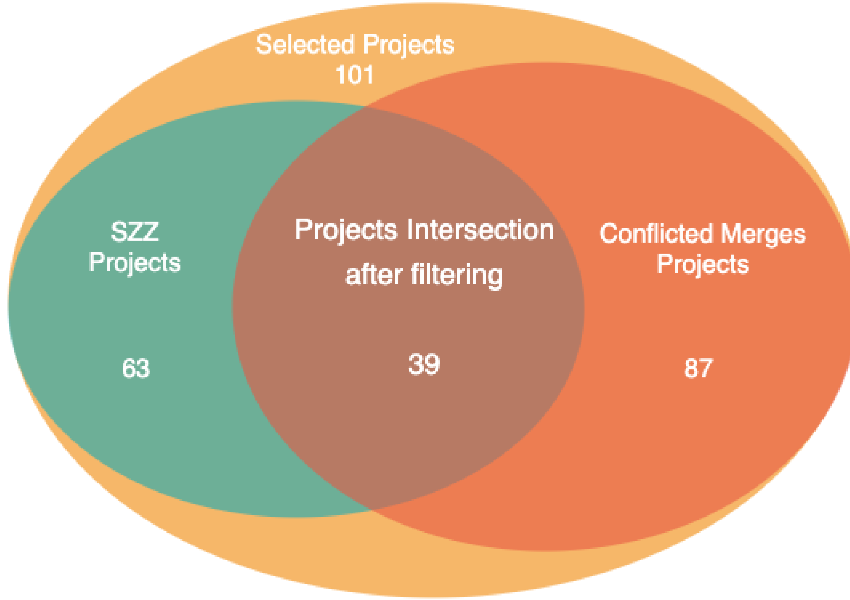

Figure 3.1: Number of projects selected.

3.2 Finding Bug-introducing commits

We conducted an activity of *mining software repositories* in order to detect bug introduction commits that happened during the evolution of the selected projects. By doing so, we used SZZ-Unleashed [8]. Some reasons support this choice. Fist of all, there is a lack of SZZ implementations publicly available [49], and the work of Borg et al. [8] is one of that we consider stable and well documented. Besides that, this implementation supports Git repositories, and being relatively simple to replicate their study and check its usage scenarios and features. Finally, that work also made available scripts to prepare the data to be used as input of SZZ-Unleashed. Therefore, we collect the information from SZZ-Unleashed to populate a dataset with commits that are likely to introduce bugs.

In order to validate the approach and experiment with the SZZ-Unleashed tool, we conduct a pilot study. This pilot study also allows us to detail our research methodology, based on the SZZ-Unleashed, using a more concrete example. We chose the Apache Spark project to run this pilot study, in which it is hosted on GitHub and uses Jira as the issue tracking system. We follow the following steps (see Figure 3.2 [8]):

Step (1) **Fetch bugs issues:** The first step is to collect bugs issues from Jira, using the REST API and filtering the issues using the issue type = **bug**, the status either **resolved** or **closed**, and the resolution = **fixed**. As an output, we collected 6907 issues from Apache Spark in 7 JSON files (because there is a limit of 1000 issues per page).

Step (2) **Convert Git-log to an array:** The second step is to clone the project repository and convert the Git log to an array. The result is a JSON file with all source-code history of the project.

Step (3) **Find Bug Issues:** The third step is to use the resulting files from previous steps to link *bug fixes commits* to *issues*. In this case, it necessary to specify how a bug fix looks like in a commit, and the framework tries to find some patterns, such as the word "fixed" and the issue number, to decide whether or not a commit is a bug-fix. As a result, we obtain a file containing all bug-fixes commits necessary as input to the second Phase of SZZ-Unleashed. For the pilot study, we found 6424 bug-fixes commits, meaning that in this stage, we mapped 93% of the issues from Jira to bug-fix commits on Git-log.

Step (4) **Run SZZ-Unleashed:** Finally, after preparing all the necessary files, we ran the SZZ-Unleashed for project Spark. As a result, we obtained 6835 pairs of bug-fixes commits and their respective bug introducing commits (BFC-BIC). Notice that a bug-introducing commit might be responsible for causing more than one bug-fix commit, and one bug-fix commit might have more the one BIC.

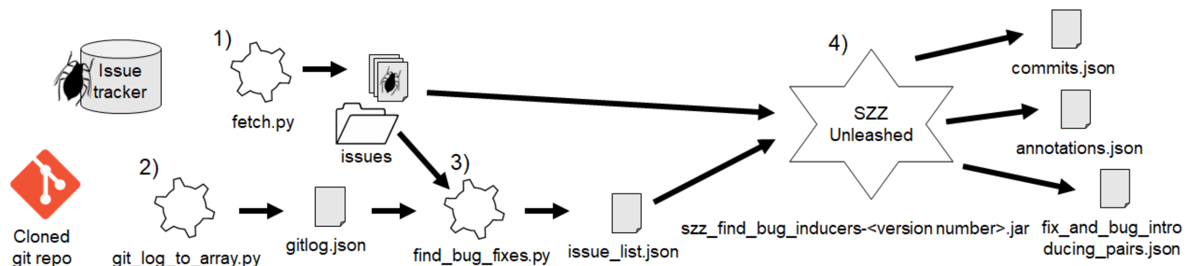

Figure 3.2: Workflow of SZZ-Unleashed [8]

From SZZ-Unleashed on Apache Spark, we got 1802 bug-fix commits that fixed changes from 2382 bug-introducing commits. In summary, using SZZ-Unleashed, we found 6424 fixes (out of 6907 issues). The analysis reveals 6835 pairs of BFC-BIC (bug fix commits and bug introducing commits)— having 1802 unique BFCs and 2382 unique BICs.

The first stage of our pilot study validate our approach, so we created python scripts to replicate the same study, running SZZ-Unleashed for the remaining 100 project repositories. The results of this study are presented in Section 4.1.

3.3 Re-playing merge scenarios and collecting their characteristics

The goal of this study is to collect different merge scenarios from Open Source Project repositories, in order to better understand the possible causes of conflicts in merge scenarios and the induction of errors. Based on some characteristics of the merge, the idea is to build a model that can be used to aware developers of the possibility of bug introduction when concluding a merge operation. To get this information, it was necessary to check all merges scenarios to know when a conflict has happened. Since Git does not record merge conflicts, in order to get this information, we need to recreate each merge scenario for all projects and analyze some patterns in the data.

Considering we are interested in verifying the relation of bug-introduction—the data collected in the previous section- with conflicted merge scenarios, we needed to collect the merge scenarios for the same projects. In the second phase of our pilot study, we mined the labeled merge commits from Git history of the Apache Spark project. Despite this, not all information about the merge is available in a Git repository's log data. Since we are interested in conflicted merges, we needed to get this information by re-playing the merge to discover additional properties (e.g., number of files in conflict and number of days). Also, we used our pilot study with Project Apache Spark to collect the information about each commit and re-create merge scenarios.

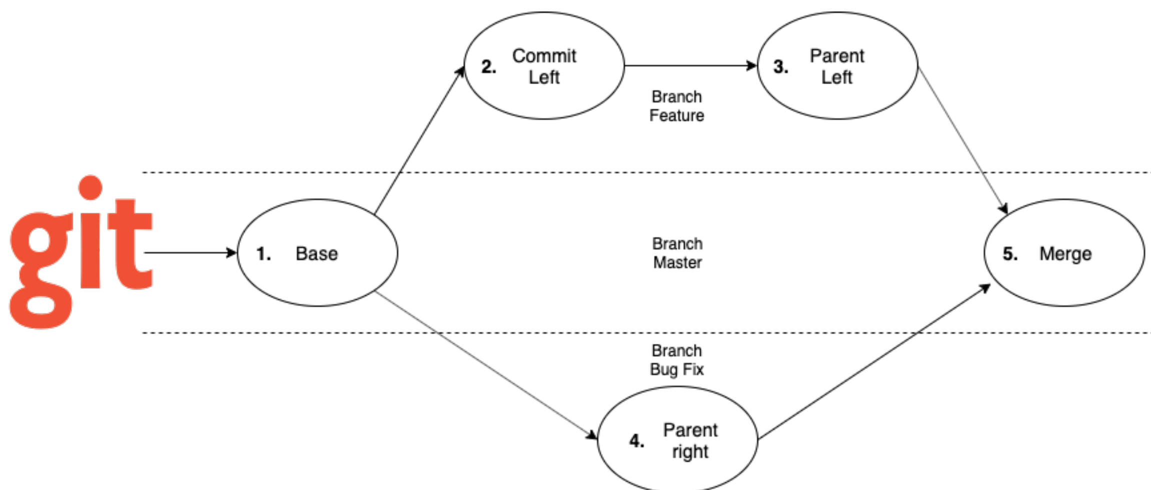

Figure 3.3: Sample of git merge operation.

Figure 3.3 presents an example of a three-way merge scenario that we considered in this study, and the steps performed with Apache Spark are presented below:

Step (1) **Get all commits:** The first step, is to collect the **hash value**, the **date**, and the **author's name** of all commits. We collected this information using the following git command in the terminal: **git log –pretty=format:%H,%cd,%cn**;

Step (2) **Get merge commits:** The second step is to collect the Hash of all merge commits and their respective parents' Hash as shown on Figure 3.3: **git log –merges –pretty=format:%H,%P**; This command brings the Hash of merge commits followed by a pair of two hashes: Parent Left and Parent Right

Step (3) **Find the Base:** After finding the hash of the parents commits, we are able to find the common ancestor, Base commit number 1 of figure 3.3 using the git command: **git merge-base –all 'leftP' 'rightP'**; We avoided octopus merge commits, by using only the first hash of the ancestor.

Step (4) **Re-play merge commits:** We can verify if there were files in conflict by hard resetting git to the base commit: **git reset –hard "Base_Hash"**, merge the base with parent right: **git merge "rightP_Hash"**, and merge the results with the parent left: **git merge "leftP_Hash"**.

Step (5) **Record the outcome:** Finally, when a conflict occurs, we collect and treat the outcome of Step 4 to get the number of files in conflict.

In the end, we recreated 1641 Merges scenarios for Apache Spark, from which, 231 let to conflicts (14,08%). By merging the dataset of bug introducing commits with the dataset containing the conflicting merge scenarios, we found that 8 conflicted merge scenarios were blamed by SZZ-Unleashed (that is, they are Bug-Introducing changes). Table 3.1 presents the features we collected from the merge scenarios, and Table 3.2 presents the features and their respective values that we collected from the induced merge commits of project Apache Spark.

	Features collected from merges		For both Right and left branches
1.	Files changed	4.	Number of Commits
2.	Lines Additions	5.	Number of active Authors
3.	Lines Removals	6.	Files in Conflict

Table 3.1: Features collected from merge scenarios.

Our second stage of the pilot study with Apache Spark was validated and we were ready to replicate the approach for the complete set of projects.

rFiles	rAdd	rRem	lFiles	lAdd	lRem	rCom	rAuth	lCom	lAuth	Conf
571	20859	3069	2	9	1	557	43	4	1	1
138	2854	2287	1056	75210	57891	54	5	745	51	86
10	109	42	1058	71559	69579	2	1	52	5	4
4	144	160	256	7801	4571	34	1	402	40	1
14	95	89	2	7	2	1	1	4	3	1
50	1529	515	53	649	453	35	1	14	7	1
28	1202	164	316	4905	3480	24	2	243	21	5
9	183	2	374	17720	8208	9	1	631	56	2

Table 3.2: Collected values from the Induced merge commits of the project Spark.

3.4 Tool support for collecting merge scenarios

We developed some Python scripts to analyze the history of commits of projects repositories using GitPython API and extract information from merge scenarios. Our idea is to use only features that could be easily collected using git commands that were supported by previous studies. The input of our program is the path of the directory containing target repositories. As an output, we generated four `csv` files per project.

(**File** 1) **All commits:** The first file contains all commits with the respective dates and authors;

(**File** 2) **Merge commits:** The second file is a CSV containing all merge scenarios with their respective features;

(**File** 3) **Between right Merge commits:** The third file contains the right side of commits between two merges. Commit number 4 of Figure 3.3 is an example of between right commit.

(**File** 4) **Between left Merge commits:** The fourth file contains the left side of commits between merges. Commits number 2 and 3 of Figure 3.3 are examples of between left commit.

After running our scripts for the 100 Java Apache projects, we verify that it was possible to obtain information for 91, since 10 of them did not contain any merging scenario. We recreated 59 503 merge scenarios from which, 9410 (15.81%) led to a conflict, see Table 3.3.

3.4.1 Computing Co-change Dependencies

A co-change dependency arises when two source-code entities, such as classes, interfaces, methods, or fields, frequently change together. When two source-code entities are co-

Projects: 91	Merges	Conflicted	Rate
Min. :	1.0	0.0	0.00000
1st Qu.:	26.0	1.5	0.03989
Median :	110.0	12.0	0.09924
Mean :	653.9	103.4	0.14614
3rd Qu.:	595.0	48.0	0.18237
Max. :	10143.0	4137.0	1.00000
Total:	59503.0	9410.0	0.15814

Table 3.3: Summary of the number of merges, conflicted merges, and conflicted rate

change dependent on each other, we refer to the set of commits that modified both entities as "commits related to the co-change dependency." In this study, we are interested in finding commits related to co-change dependencies that also introduced bugs. In particular, we are interested in co-change dependencies between *fine-grained* entities (e.g., methods or fields), since the bug detection tool informs the lines changed by the bug introducing commit, and we want to detect which method or field contains the bug.

Popular VCSs such as GIT maintain the evolution of source-code artifacts (typically files). The history of changes submitted to a VCS can be described as a sequence of commits $H = (c_1, c_2, \ldots, c_n)$, where each commit contains a subset of artifacts in the form $c_i \subseteq A$. Since in this work, we are interested in the change history of fine-grained source-code entities (e.g., methods or fields), instead of coarse-grained entities (e.g., files or classes), here we first have to preprocess the original change history to produce a more detailed one (which we call *fine-grained change history*). This detailed change history can be described as a sequence $H' = (c'_1, c'_2, \ldots, c'_n)$, where each commit is a subset of fine-grained source-code entities $c'_i \subseteq F$ that changed together. To transform a change history (H) into a fine-grained change history (H'), we analyze each source-code artifact of a commit to discover which fine-grained entities have been modified. We take advantage of Kenja[1], a software utility that produces fine-grained change history from git repositories.

As discussed before, two source-code entities are co-change dependent upon each other when they *frequently* change together. Certainly, the precise definition of *frequently* depends upon how often these two entities changed together, and we compute this information considering the fine-grained change history. More specifically, we use two metrics to determine if two entities e_a and e_b change frequently together: *support count* and *confidence*. The first counts the number of commits in which both e_a and e_b appear together; while the second corresponds to the ratio of the *support count* between e_a and e_b and the number of commits containing e_a. Note that, while the support count is commutative, i.e., the support count between e_a and e_b is the same of the support count between e_b

[1] `https://github.com/niyaton/kenja`

and e_a, the confidence is not, i.e., the confidence between e_a and e_b can be different from the confidence between e_b and e_a. We consider that e_a and e_b change frequently if their support count and confidence are above the threshold for supporting count S_{min} and confidence C_{min} at least in one direction. Several studies on co-change dependencies use the values $S_{min} = 2$ and $0.4 \leq C_{min} \leq 0.5$ (e.g., [5, 18, 51]). Finally, we list the commits related to each co-change dependency.

In our running example with project Apache Spark, we demonstrate our methodology to collect commits that lead to co-change dependencies and how to correlate these commits whit bug-introducing commits.

(S1) **Clone the repository and create a destination repo for the fine-grained commits:** After cloning the Spark repository, we need to create the destination folder to the converted repository.

```
git clone https://github.com/apache/spark
mkdir spark-hr
cd spark-hr
git init --bare
cd ..
```

(S2) **Convert to fine-grained repository:** The second step is to convert all commits to fine-grained using the docker image and specifying the origin and destiny directory paths. This procedure took around 16 hours to be completed for the project Spark in a regular computer.

```
docker run --rm -v $PWD/spark:/source -v $PWD/spark-hr:/dest
 projectdraco/g2h converter.sh /source /dest
```

(S3) **Get all commits for both repositories:** After converting the repository, we can collect the **hash value** and the **date** of all commits of the origin and destiny repositories using git log command. This information is relevant because Kenja creates new hashes when converting the commits and we need to link the timestamp to correlate them.

```
cd spark
git log --pretty=format:\%H,\%cd > spark-commits.csv
cd ../spark-hr
git log --pretty=format:\%H,\%cd > conv-spark-commits.csv
```

(S4) **Collect the commits that lead to co-change dependencies:** Using another Docker image we can collect all commits that lead to co-change dependencies for Apache Spark (using the command bellow). This operation took 5 hours to be completed in a regular computer.

```
docker run -it --rm -v $PWD:/repo projectdraco/mining-cochange
 --output=rules-and-commits > spark-cochange.mdg
```

(S5) **Treat the output file:** Finally, we have collected all necessary data from Apache Spark. The columns of the output file are:

```
1. Field representing the origin vertex of the edge;
2. Field representing the destination vertex of the edge;
3. Support count;
4. Confidence;
5. --
6. Total of commits;
7. Commits hashes.
```

In this study, we filtered for co-change dependencies with support count greater than one and confidence greater or equal to 0.5. For Apache Spark, 1293 commits meet these requirements (support count from 2 to 22 and confidence from 0.5 to 1). By combining the commits that leads to co-change dependencies with commits that SZZ-Unleashed blamed to introduce bugs, we found that 13.76% of commits that contribute to co-change dependencies are related to bug-introducing commits. With this initial result, we created bash scripts to replicate our data collection all over the projects.

Chapter 4

Results

In this chapter, we present the results of our empirical study. We first report the outcomes of an exploratory data analysis 4.1, and then we answer our research questions using either hypothesis

4.1 Exploratory data Analysis

We first report the results of income and outcomes of SZZ-Unleashed, which contains information about the number of issues, bug-fix commits, and bug-introducing commits over the set of projects of the section 3.2. In phase I of SZZ (collecting bug issues from Jira), we verified that it was not possible to find a relevant number of closed issues for some projects. Then we filtered for projects in which it was possible to collect at least 200 closed bug-issues, first quartile, to guarantee that we would have linked a substantial number of issues to Bug-Fixing commits. Table 4.1 shows the number of projects that were possible to collect at least 200 issues to run SZZ-Unleashed.

Projects	Issues	Fixes	Rate
63	131044	104636	76.49%

Table 4.1: Phase I of SZZ-Unleashed

In Figure 4.1, we present a log-scale of the number of collected issues, and the number of linked bug-fixes commits over the projects. The mean of collected issues is 2080.063, and the mean of bug-fix commits is 1660.889 per project – while in *Apache Ambari*, we have mined 15,465 closed bug issues and linked 14,333 to a bug-fix commit, in *Apache Fineract* we got 158 bug-fix commits for 203 issues collected from *Jira*. We removed project *Apache Log4j 2* because it was possible to link only 2 BFCs over the 794 issues collected from JIRA.

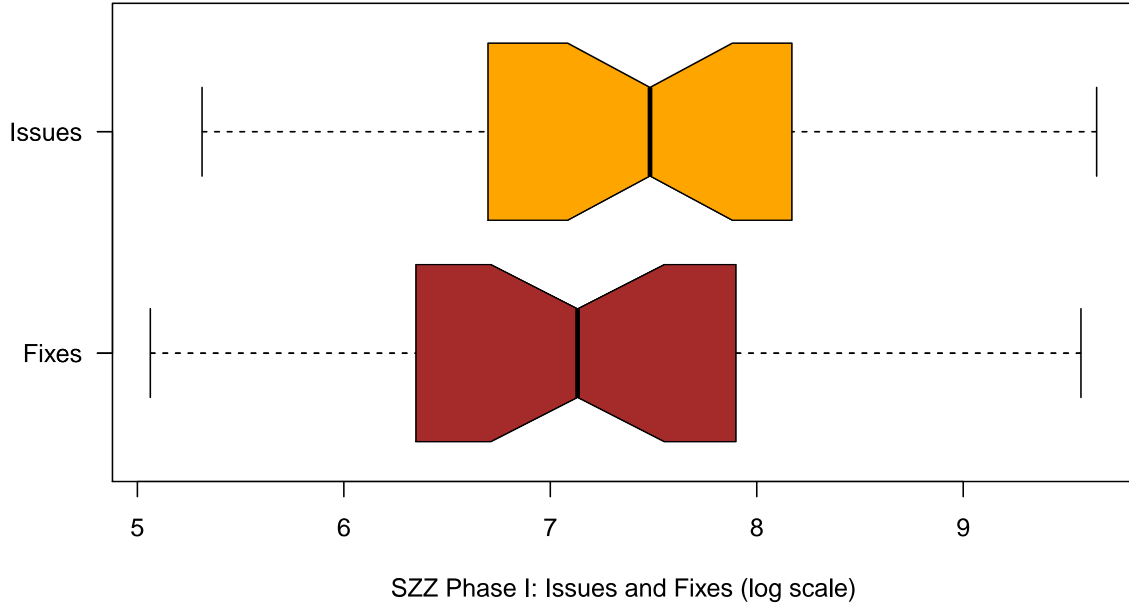

Figure 4.1: Log-scale box-plot with the number of closed bug issues and linked bug-fixing commits per project

Figure 4.9 shows a histogram that considers the rate of bug-fix commits over the number of issues per project. Overall, *SZZ Phase I* have linked 80.33% of the issues to bug-fix commits. In 14 projects, we have more than 90% of linked issues. Nonetheless, in project *Apache Cordova-Android*, only 508 linked bug-fixes over 4709 issues (which represents 10.79%). This situation occur because *Apache Cordova* has different repositories on *GitHub*, such as *Cordova-Windows, Cordova-IOS, Cordova-browser*, and all of them use the same pattern on *Jira*.

Table 4.2 presents the outcome of the second phase of SZZ-Unleashed. We found 339,252 pairs of BFC-BIC in 62 Java projects, composed by a set of 72,072 bug-fix commits and 69,616 bug-introducing commits. It is important to remember that a bug-introducing commit might introduce bugs in more than one place, and a BFC might fix bugs introduced by multiple BICs.

Projects	Pair(BFC-BIC)	BFC	BIC
63	339,250	72,072	69,616

Table 4.2: Phase II of SZZ-Unleashed

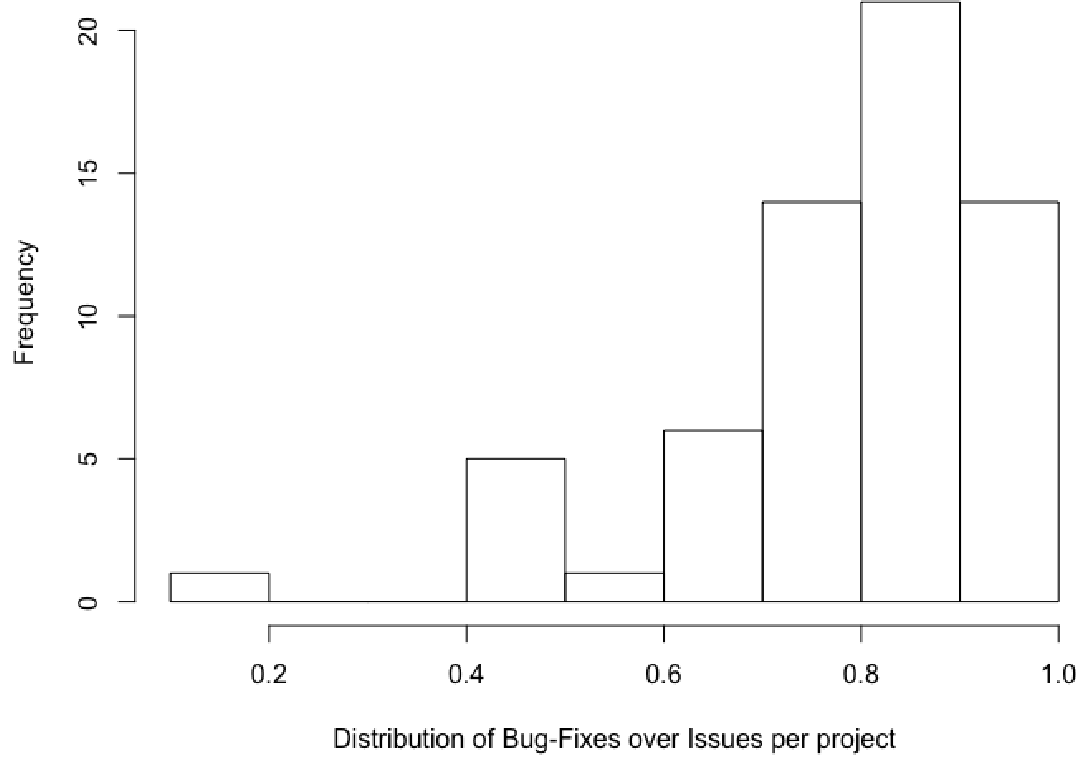

Figure 4.2: Rate of the linked fixes over the number of issues per project

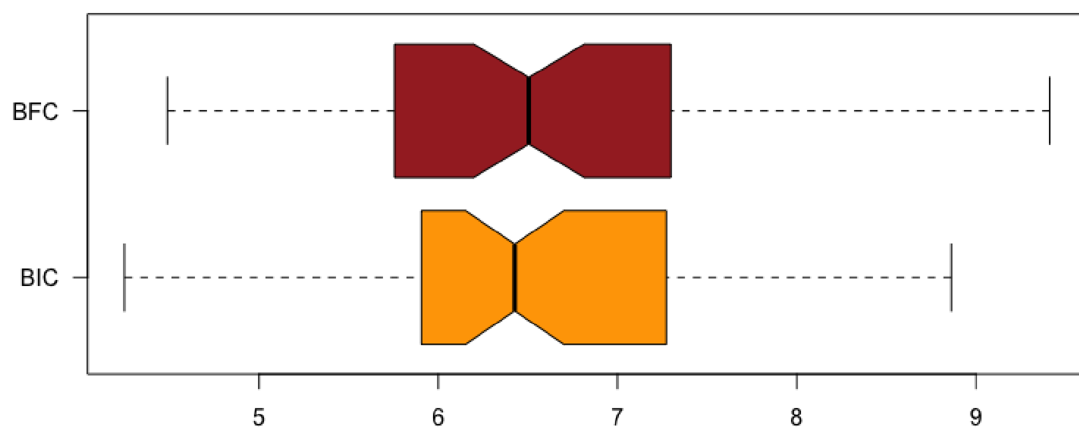

Figure 4.3: Log-scale box-plot with the number of BFC and BIC per project

As a result of phase II of SZZ-unleashed, Figure 4.3 shows a log-scale box-plot of the distribution of bug-fix commits and bug-introducing commits over the projects. As expected, *Apache Ambari* is the project with more BICs, (7,051) introduced errors in 12,227 BFCs. By comparing with Phase I, where we found 14,333 bug-fixes commits, it means that SZZ-Unleashed could not find BICs for 2,106 BFCs. On the other hand, the same does not occur with the project with less BICs: In the project *Netbeans*, 70 BIC introduced errors in 218 bug-fix commits while 178 BICs were responsible for introducing errors in 89 BFCs on project *Apache Collections*.

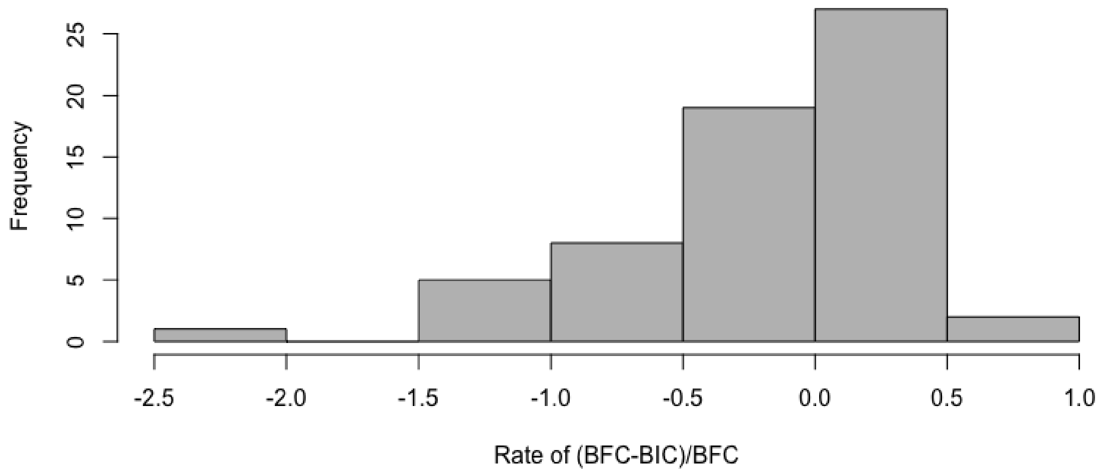

Figure 4.4: Histogram with the rate value of (BFC - BIC) / BFC per project

Figure 4.4 presents the rate of the difference between BFCs and BICs per project. 33 projects have more BICs than BFCs (negative value) and 29 have more BFCs than BICs (positive Value). SZZ-Unleashed blamed 417 BICs for introducing error(s) solved by 138 BFCs in project *Apache Fineract* and blamed 2152 BICs for 986 BFCs in project *Apache Beam*. Nonetheless, in *Apache Bigtop*, 853 bug-fix commits fixed the errors of 367 bug-introducing commits.

The histogram 4.5 shows the rate of bug-fix commits that the outcome of SZZ Phase II linked to bug-introducing commits over all bug-fix commits (outcome of Phase I of SZZ). Overall, 68.88% of BFCs fixed errors caused by Bug-Introducing commits, with a rate higher than 0.8 on 30 projects, such as project *Apache Tinkerpop* had 229 BFCs as outcome of SZZ and 260 as income (88.08%). The lowest rate value occurred in *Apache Hadoop*, in which only 420 BFCs were linked to BICs when it has 4163 as input (10.09%). Other causes of bugs induction are co-evolution, compatibility issues, or bugs in external APIs, but we cannot guarantee that this has happened in the five projects with a rate

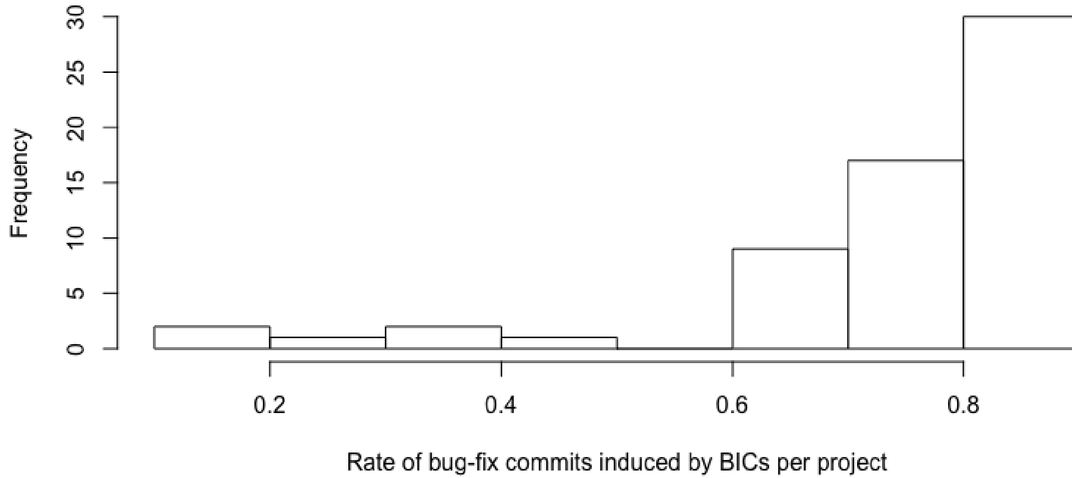

Figure 4.5: Histogram with the rate value of BFCs that fixed errors introduced by BICs

lower than 0.4. To avoid bias in our merge scenarios dataset and errors in our analysis results, we removed projects that did not have conflicted merge scenarios. Furthermore, we eliminated projects that have not a significant number of merge scenarios, filtering for projects that have more than 26 (first quartile), according to the table 3.3. We present the summary of filtered projects in Table 4.3:

Projects: 63	Merges	Conflicted	Rate
Min. :	27.0	1.0	0.003774
1st Qu.:	65.0	11.0	0.065201
Median :	327.0	28.0	0.104737
Mean :	894.0	148.8	0.141374
3rd Qu.:	804.5	98.0	0.187843
Max. :	10143.0	4137.0	0.545454
Total:	56322.0	9377.0	0.166489

Table 4.3: Summary of the number of merges, conflicted merges, and conflicted rate for the filtered projects

Figures 4.6, 4.7, and 4.8 present the distribution of the merge scenarios, conflicted merge scenarios and the rate of conflicted merge scenarios before and after we applied the filters. We observed interesting things here: project Apache Cassandra has 10146 merge commits in which 4137 of those were conflicted (40.79%) while project Apache Beam has 7367 with only 45 conflicted ones (0.61%). Moreover, project Apache ActiveMQ Artemis has 2619 merge scenarios and none of those had conflicts.

Some questions come to our mind after this step:

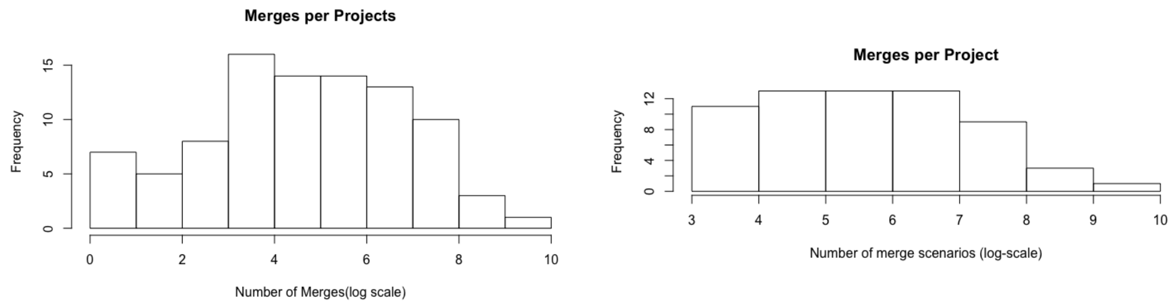

Figure 4.6: Log-scale histograms with the number of merge scenarios before(left), and after filtering the set of projects (right)

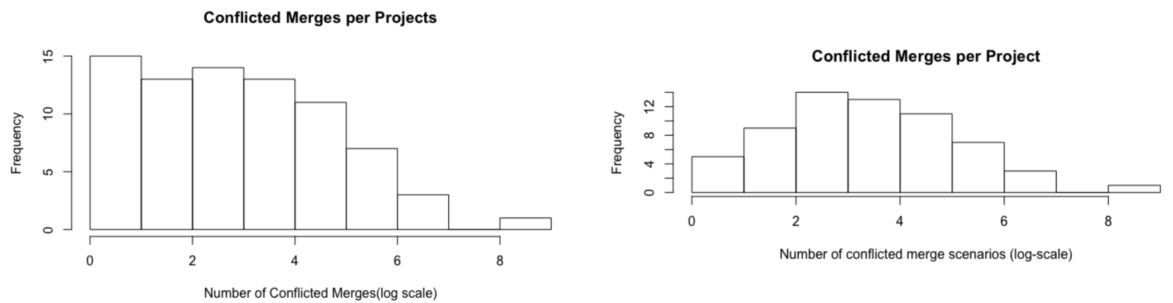

Figure 4.7: Log-scale histograms with the number of conflicted merge scenarios before (left), and after filtering the set of projects (right)

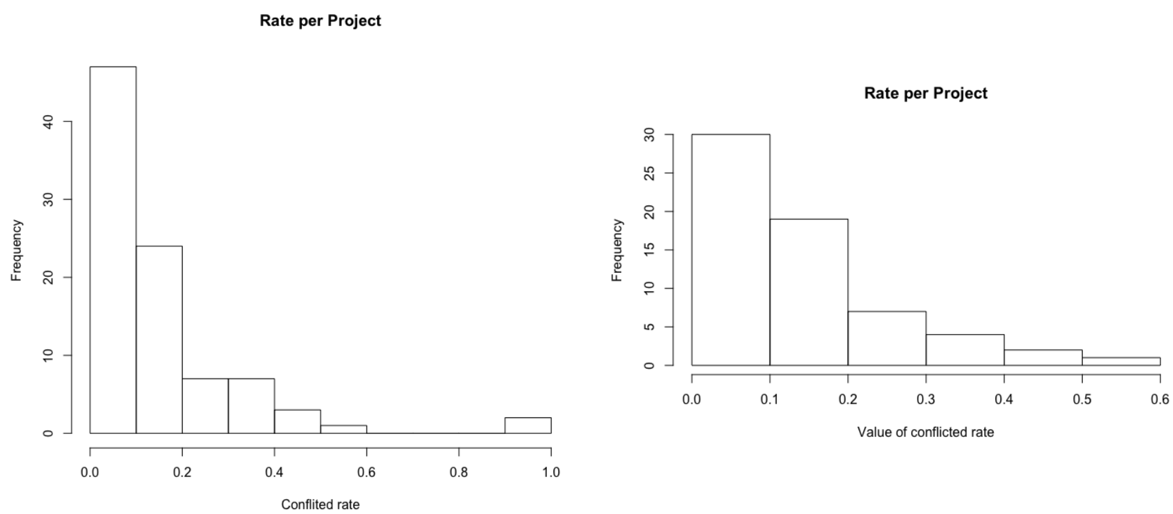

Figure 4.8: Histograms with the rate value (Conflicts/Merges) before (left), and after filtering the set of projects (right))

1. In 9 projects, we did not find any merge commit and, another 7, have less than three merges commits. Is there any agreement not to use merge approaches on these projects?
2. What are the best-practices the developers use in order to drastically reduce or even eliminate the rate of conflicted merges scenarios?

When we first merged the output of SZZ-Unleashed with our recreated merge scenarios, we found that SZZ blamed 288 merge commits for introducing errors into 26 different projects. At this point, we discovered that 21 of the merge commits blamed by SZZ did not have textual conflict, which contrast the intuition that a merge commit needed to introduce new code to be labeled as bug-introducing commit. By analyzing these particular cases, in 2 specific scenarios, the left or right parent were also blamed for introducing error, and SZZ might have propagated this information to the merge commit. Nonetheless, in the other 19 not-conflicted merge scenarios or 6.597%, SZZ-Unleashed blamed the merge commit but did not blame the right nor the left parent commits. This mislabeling were probably caused by limitations of SZZ algorithm, since we used depth search with 3 levels in this study and, in some cases, the part of the code that the bug-fix commit changed, might be introduced by commits with higher level of depth.

Blamed Merge Commits	With conflicted files	Without conflicted files
288	267	21

Table 4.4: Initial results of merge scenarios that Induced errors

4.2 Disclosing conflicting merge scenarios linked to BICs

We conduct an exploratory data analysis to get a general understanding about the frequency of merge scenarios and conflicting merge scenarios, as well as to build a curated dataset. That is, our goal is to avoid bias in our merge scenarios dataset that could lead to errors in our analysis results. Accordingly, we removed projects that did not have either merge scenarios or conflicted merge scenarios. Interesting, in 9 projects, we did not find any merge commit (e.g., COMMONS-IO). It is still not clear to us why some projects do not employ merge operations. Furthermore, we eliminated projects that do not have at least 26 (first quartile) merge scenarios and filtered out projects in which it was not possible to collect at least 200 (first quartile) closed bug-issues, in order to guarantee that we would have linked a substantial number of issues to bug-introducing commits. Finally, we classified the merge scenarios either as **simple** or **complex**. To this end, we defined a rough

estimation for the complexity of a merge scenario (Cm) as the geometric mean between the number of changed files from its parents (left and right). In our dataset, complex merge scenarios are those scenarios with $Cm > 32.296$ (third quartile). This separation is necessary because we found many scenarios changing a huge number of files. For instance, the merge scenario with commit ID `3b21d1db4109939450dc400faebe568222ab4758` from NETBEANS changed more than 70 000 files. Altogether, our curate dataset, which is the intersection of the outcomes generated by our three studies (see Sections 3.2, 3.3, and 3.4.1), contains information about 40 092 merge scenarios of 34 Java Apache projects, from which we collected 49 678 bug-introducing commits and 38 752 commits that lead to co-change dependencies. In Figure 4.1, we present the number of closed issues (bugs) and the number of linked bug-fixing commits over the projects (using a log-scale). The average number of issues and bug-fixing commits per project is 2661 and 2134.2, respectively. While in APACHE AMBARI, we have mined 15 465 closed bug issues and linked 14 333 bug-fixing commits, in APACHE FINERACT we got 158 bug-fixing commits for 203 closed bug issues collected from JIRA.

Figure 4.9 shows a histogram that considers the rate of bug-fixing commits over the number of issues per project. Overall, the first phase of SZZ linked 77.28% of the issues to bug-fixing commits. In 9 projects, SZZ linked more 90% of the issues to BFCs (e.g., ACCUMULO and LUCENE-SORL). Nonetheless, in project APACHE CORDOVA-ANDROID, SZZ linked only 508 bug-fixing commits to a total of 4709 issues (which represents 10.79%). This situation occurs because APACHE CORDOVA-ANDROID is a submodule of APACHE CORDOVA, which shares the same JIRA repository with other modules. Nonetheless, in our analysis we only considered APACHE CORDOVA-ANDROID.

The outcomes of the second phase of SZZ revealed 249 041 pairs of BFC-BIC over the projects, composed by a set of 50 925 bug-fixing commits and 49 678 bug-introducing commits. It is important to remember that a bug-introducing commit might introduce bugs in more than one place, and a bug-fixing commit might fix bugs introduced by multiple BICs.

For instance, APACHE AMBARI is the project with more BICs—SZZ blamed 7051 commits for 12 227 BFCs. By comparing with its first phase, where SZZ linked 14 333 bug-fixing commits for APACHE AMBARI, it means that SZZ could not find BICs for 2106 BFCs. Considering the NETBEANS project, SZZ revealed 70 BICs for 218 bug-fixing commits while 178 BICs were responsible for introducing errors in 89 BFCs on project APACHE COLLECTIONS. Overall, 83.42% of BFCs fixed errors caused by bug-introducing commits, with a rate higher than 0.8 on 15 projects, such as FINERACT and BEAM. The lowest rate value happened in APACHE SPARK, in which SZZ linked only 1802 BFCs to bug-introducing commits (26.07% of the total number of BFCs).

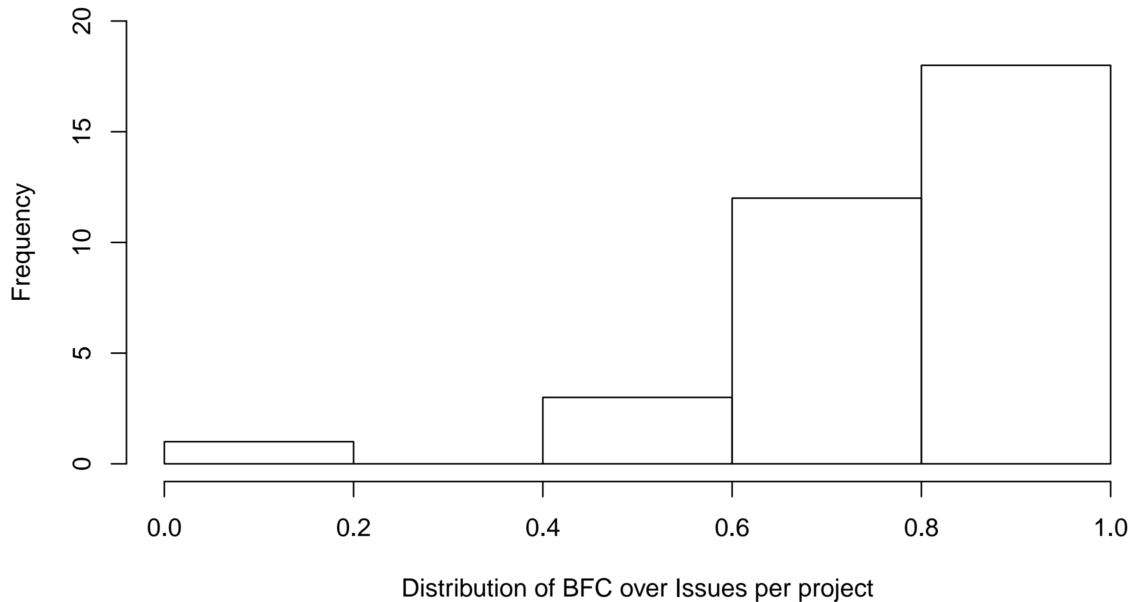

Figure 4.9: Rate of the linked BFCs over the total number of issues (labeled as bugs) per project

We found 7453 conflicting merge scenarios (18.58% of the total number of merge scenarios). Figure 4.10 presents the log-scale distribution of merge commits and conflicting merge commits over the 34 projects. Considering the merge scenarios, Apache Avro has 47 (the lowest) and Cassandra has 10 143 (the highest). Finally, Apache James has only one conflicting merge scenario, while Cassandra presents 4137 conflicting merge scenarios. More than 40% of the merge scenarios of Cassandra led to a conflict. Considering the rate of conflicted merge scenarios over the number of merges, Figure 4.12 shows that, in most of the projects (79.41%), conflicts occur in less than 20% of merge scenarios.

Most of the projects have low rate of merge scenarios over the total number of commits, 21 projects with rate less than 5%. However, in project *Apache Cassandra*, 40.19% over all commits are merge scenarios and other two projects have percentages between 30 and 35 of merge scenarios over the number of commits, as showed in Figure 4.11. Considering the rate of conflicted merge scenarios over the number of merges, Figure 4.12 demonstrates that the percentages of conflicted merge scenarios are lower than 15% in 27 projects (which represents 69.23% of the number of analyzed projects). On the other hand, two projects had conflicted merge scenarios in values between 40% to 50% over the number

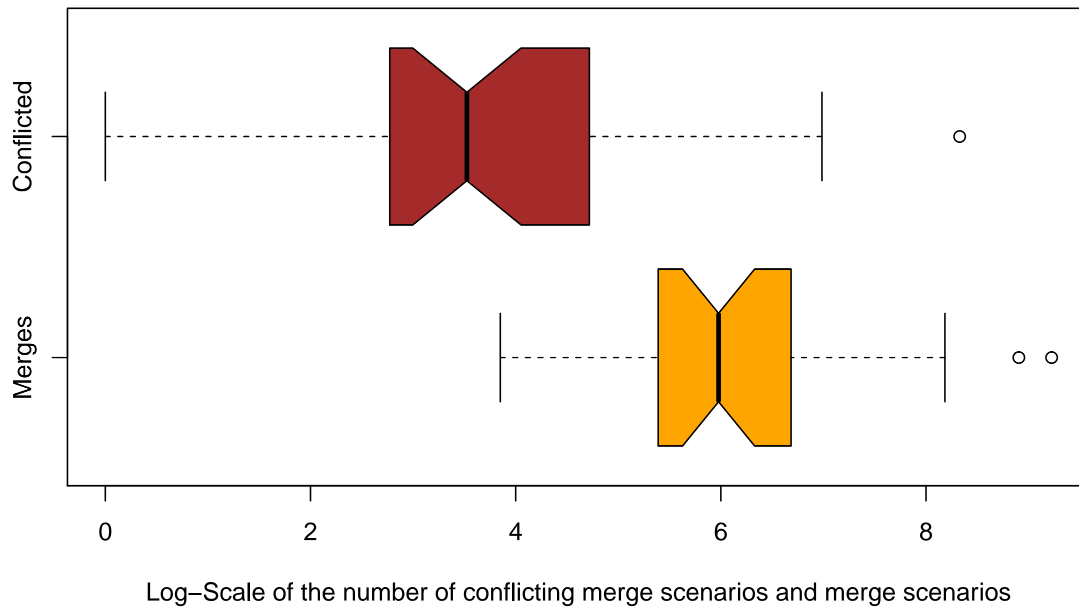

Figure 4.10: Log-scale box-plot with the number of all commits, merge scenarios, and conflicted merge scenarios per project

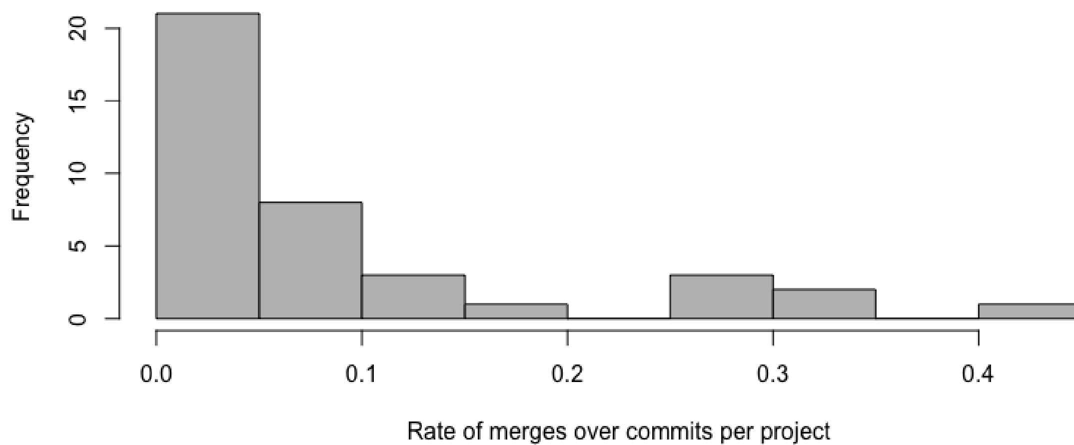

Figure 4.11: Histogram with the rate value of merges over commits per project

of merges, and this is not a good approach, since some errors might occur when resolving these conflicts.

Figure 4.13 shows the rate of bug-introducing commits over the total of commits

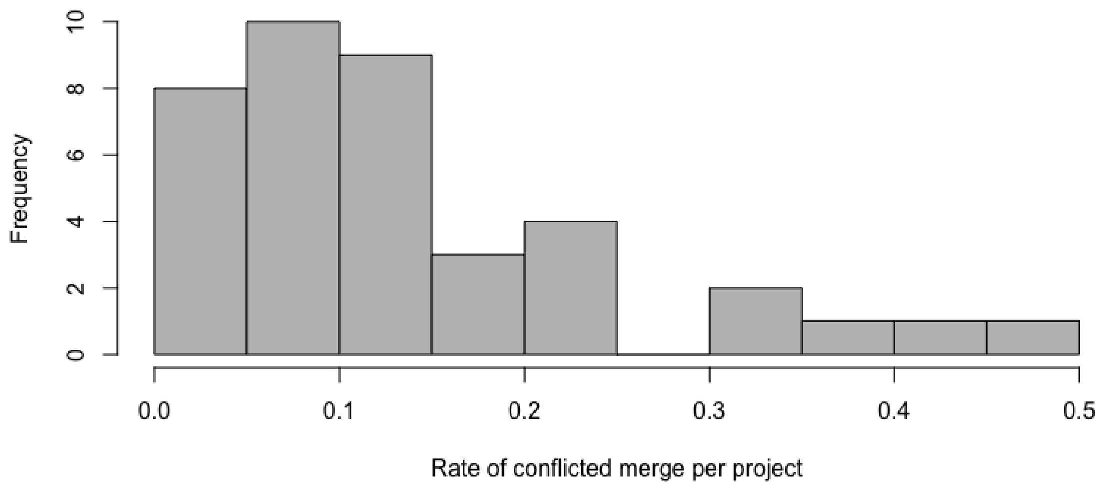

Figure 4.12: Histogram with the rate value of conflicted merge scenarios per project

per project. Considering all projects, SZZ-unleashed blamed 12.02% over all commits for being responsible for introducing errors. The project *Apache Calcite* presented the highest percentage of bug-introducing commits (SZZ blamed 35.54% of the commits for introducing errors). The second is *Ambari* with 28.69% of commits that introduced errors, and third, *Apache Hive* with 26.28%. On the other side, we found that *Apache Netbeans, Apache Jena, and Apache Ofbiz* contain less than 5% of the commits introduced bugs.

4.2.1 R.Q.1: To what extent conflicting merge scenarios correspond to the bug introduction contributions?

From the conflicting merges scenarios (7453 observations), SZZ blamed 265 commits (3.56%) as bug-introducing; introducing errors in 22 projects. Figure 4.14 shows the distribution of BICs linked to conflicting merge scenarios over the projects. Apache Cassandra is the project with the highest number of bug-introducing commits linked to merge scenarios (137), followed by Apache Accumulo with 25 conflicting merge commits that introduced errors. SZZ did not blame any conflicting merge scenario in 12 projects, and other five projects have only one blamed merge commit. Most of the projects (64%) had less than 5% of BICs linked to conflicting merge scenarios—even in Cassandra, the one with highest value of bug-introducing commits linked to conflicting merge scenarios, the rate is 3.31%. Moreover, 12 projects have a rate value greater than 5% (see Table 4.5), and 2 projects presented values greater 15%. Project Apache

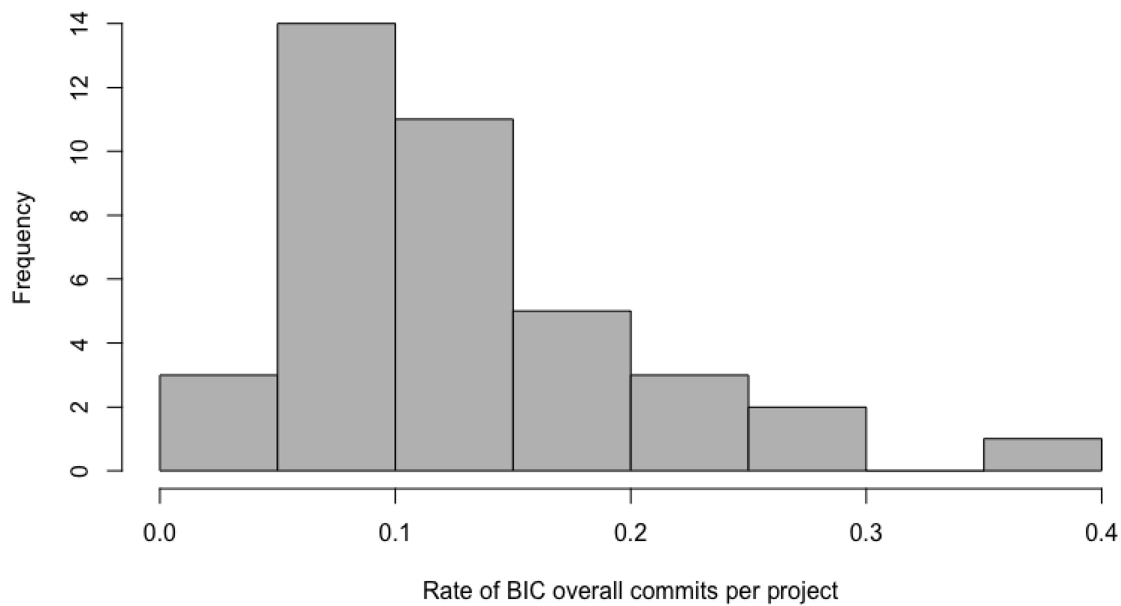

Figure 4.13: Histogram with the rate value of bug-induced commits over all commits per project

Phoenix presents the highest rate value, SZZ blamed 2 conflicting commits over a total of 6 (33.33%) followed by Apache Mahout with 2 over 12 (16.67%).

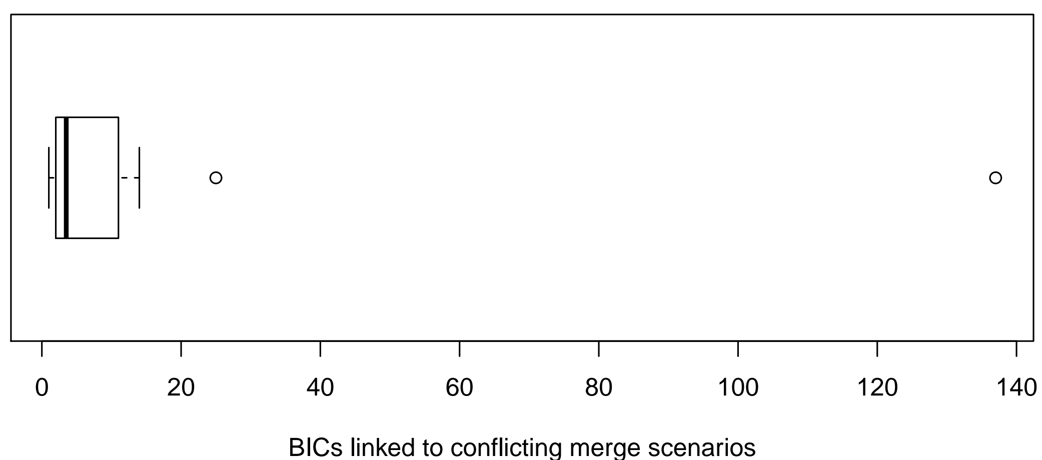

Figure 4.14: Conflicting merge commits linked to BICs

Project	#Commits	#Merges	#CM	#BICs	Rate
phoenix	2917	63	6	2.00	33.33%
mahout	4116	62	12	2.00	16.67%
geode	8131	412	75	11.00	14.67%
jena	7963	630	32	4.00	12.50%
groovy	16129	787	105	12.00	11.43%
nifi	5197	369	32	3.00	9.38 %
ambari	24580	509	66	6.00	9.09%
lucene-solr	32156	627	134	11.00	8.21%
kylin	7859	828	112	9.00	8.04%
camel	38634	214	16	1.00	6.25%
storm	10071	2713	241	14.00	5.81%
hive	13625	327	159	8.00	5.03%

Table 4.5: Summary of projects with more than 5% of conflicting merge scenarios linked to BICs. #CM means number of conflicting merge scenarios, #BICs means the number of BICs related to conflicting merge scenarios, and Rate stands for the percentage of #CM over #BICs.

The contribution of conflicting merge scenarios represents 1.78% of the total number of commits. In the same way, the contribution of conflicting merge scenarios that were blamed by SZZ represents 0.53% when considering all bug-introducing commits. These percentages indicate that the occurrence of a bug introduced by a conflicting merge scenario is approximately three times lower than the occurrence of a conflicted merge commit. We applied the paired `t-test` over the sample of conflicting merge scenarios compared to the sample of all commits (after checking all assumptions necessary to run this test), where observations in both samples are the percentage of bug-introducing commits for each project. According to the analysis, we found that conflict merge scenarios are 7.59% less likely to introduce bugs than usual commits (between 5.04% and 10.13% with confidence interval of 95% and p-value = 8.233e-07).

We also replicated our analyzes considering only **simple merge scenarios** and **complex merge scenarios**, separately. Interesting, when considering only simple merge scenarios—which corresponds to 70% of our curated dataset of merge scenarios, the number of bug-introducing commits linked to conflicting merge scenarios drops from 264 to 28, 10.57% of all BICs linked to conflicting merge scenarios. This suggests that almost 90% of conflicting merge scenarios linked to bug-introducing commits are caused by complex merge scenarios. Nonetheless, it is important to note that even the **simple dataset** contains merge scenarios involving more than 97 files on the average, with contributions made by more than 7 authors (also on average). Table 4.6 summarizes some features of the simple merge scenarios. Finally, since the number of BICs linked to conflicting merge scenarios appears more frequently in complex scenarios, we ran a new hypothesis testing

on this group. The results of the paired `t-test` over this sample compared to the sample of all commits, show that conflicting merge scenarios (the complex ones) are 6.38% less likely to introduce bugs than usual commits (between 3.41% and 9.35% with confidence interval of 95%.

Statistic	Mean	St. Dev.	Min	Max
Number of files changed	97.744	310.892	0	25,142
Number of contributors	7.852	9.626	2	175
Number of commits	101.855	257.169	2	2,776

Table 4.6: Summary of the characteristcs of our dataset with simple merge scenarios

Summary of R.Q.1: According to the outcomes of the SZZ algorithm, conflicting merge scenarios rarely introduce bugs—that is, only 3.56% of the conflicting merge scenarios introduce bugs—representing 0.53% of all bug-introducing commits.

Although our results suggest that only a small number of bug-introducing commits arise from merge conflict resolution, it is worth to investigate new methods to detect and avoid conflicts—since the source of these errors only relate to the tasks of resolving conflicting merge scenarios.

4.2.2 What are the characteristics of the conflicted merge scenarios that introduced error?

Here we demonstrate some characteristics of the 267 conflicted merge scenarios that SZZ-Unleashed blamed to introduce bugs, such as, number of developers, files changed, files in conflict. In Table 3.1, the features 1 to 5 are in a branch-level granularity, with values for both right and left branches, while feature 6 is in a merge-level. Some studies tried to investigate if these features correlate with safe versus conflicted merge scenarios. We also try to perceive some patterns of these commits to verify if it would be possible to prevent the error(s) based on their characteristics. We first present the collected results for the branch-level features, with the right branch on the top histogram and left branch on the bottom histogram of each figure, all histograms are in log-scale.

In Figures 4.15 and 4.16, we present two histograms that show the log-scale of **files changed** by the right and the left branches. In most of the right branches, the number of files changed was between 400 and 3000, and the summation is 200,525 files changed over the 267 entities. Nonetheless, most of the left branches changed between 1 to 20 files, with

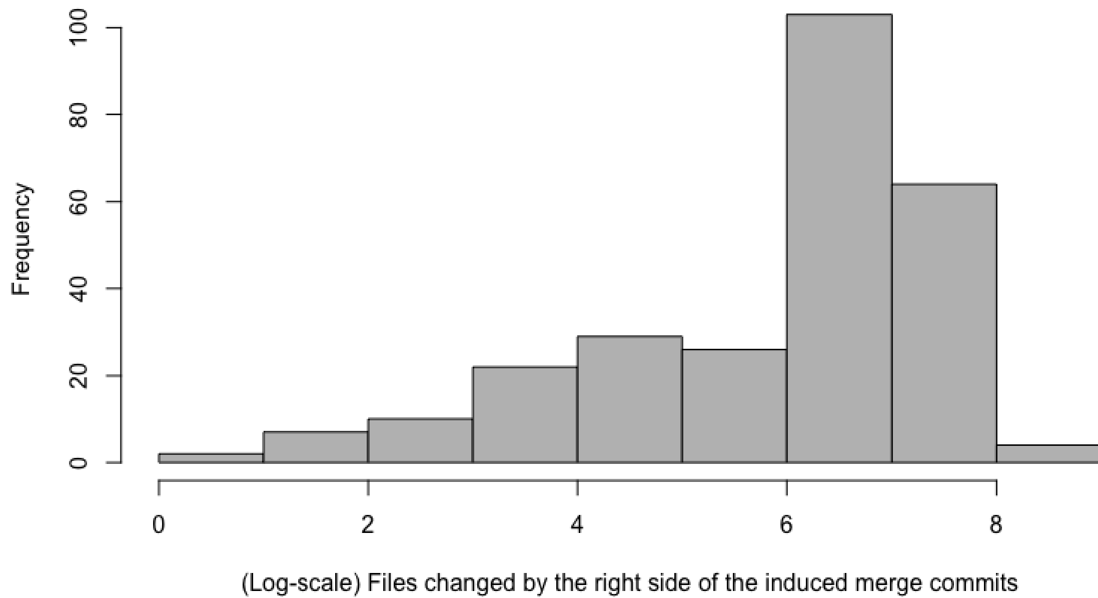

Figure 4.15: Histogram with the log-scale number of files changed by the right branches of the conflicted merge commits linked to bug-introducing commits

86,427 files changed by the 267 entities. Figures 4.17 and 4.18 present the distribution of **lines additions** for both sides over the branches. It is possible to perceive that more than a half of the merge commits that introduce errors had contributions between 22,000 to 162,000 lines additions on the right branches, while had contributions between 20 to 1000 lines additions on the left sides. The same pattern occur with lines removals, Figures 4.19 and 4.20, where more than a half of the right branches contributed between 8,000 to 22,000 of **removed lines**, while the majority of the left sides contribute with less than 1,000 lines removals. Totally, the right sides contributed with 15,725,329 lines additions and 8,861,187 lines removals, while the left branches contributed with 6,104,363 lines additions and 3,341,356 lines removals on the 267 conflicted merge scenarios that were blamed by SZZ-Unleashed to introduce error(s).

Figures 4.21 and 4.22 show the log distribution of the number of **active authors** over the branches of the both sides. In the right branches 4.21, there are a considerably well distribution between 1 to 112 contributors, 40 branches with only 1 developer, 40 branches with 7–12 developers, and 48 branches with 20–33 contributors. On the other hand, most of the left sides 4.22 had only one contributor (143 branches) and only 13 branches had more that 20 developers. At the same way, Figures 4.23 and 4.24 presents the log distribution of the **number of commits** of the branches of both sides, emphasizing the

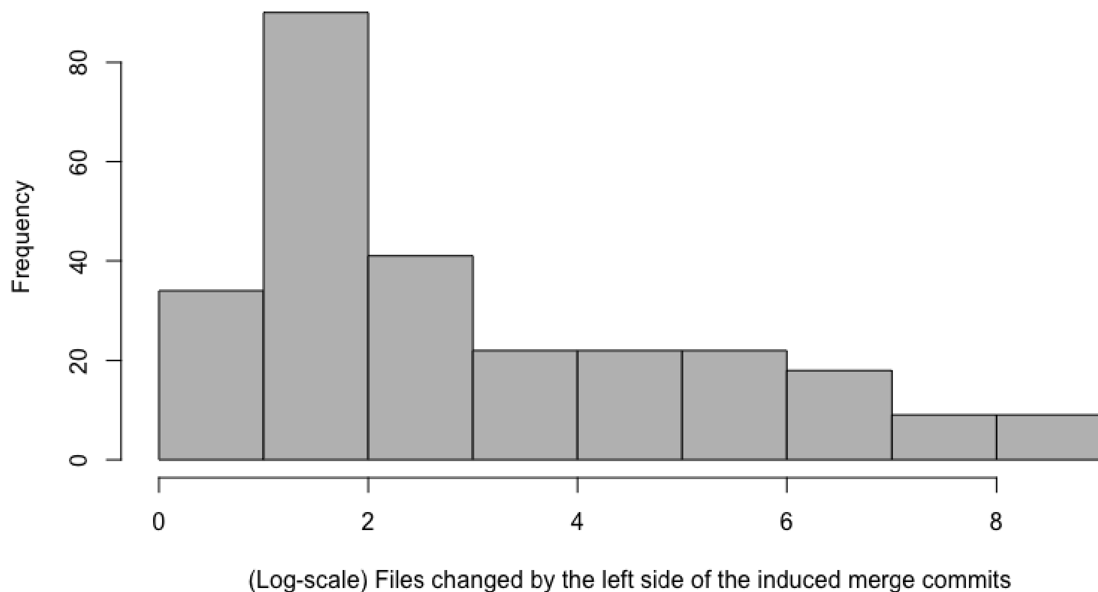

Figure 4.16: Histogram with the log-scale number of files changed by the left branches of the conflicted merge commits linked to bug-introducing commits

differences between the right and the left sides. Four branches of the right side presented only 1 commit and more than 140 branches had contribution grater than 400 commits while, in the left side, 140 branches had only one commit and only 8 branches had more than 400 commits of contribution.

Figure 4.25 represents the merge-level feature **Files in conflict** and, according to our intuition, the one that possibly has a higher value of correlation with the introduction of bug by a conflicted merge commit. From the conflicted merge commits that introduced bugs, 69 had only one file in conflict (25.843%), and 40 merges had two files in conflict (14.981%). Furthermore, 67 have three or four conflicted files (25.094%), and 38 had five, six, or seven files in conflict (14.232%). Finally, seven merges had more than 100 conflicted files, ten merges with values between 30 and 100, and 25 merges with values between 10 and 30. In Figure 4.26, we demonstrate that the conflicted merge commits blamed by SZZ-Unleashed to introduce errors were well distributed over the weekdays with few representation on weekends. The table 4.7 present the summary of the features *Files changed, lines additions, and lines removals* for the both sides of merge commits that introduced errors.

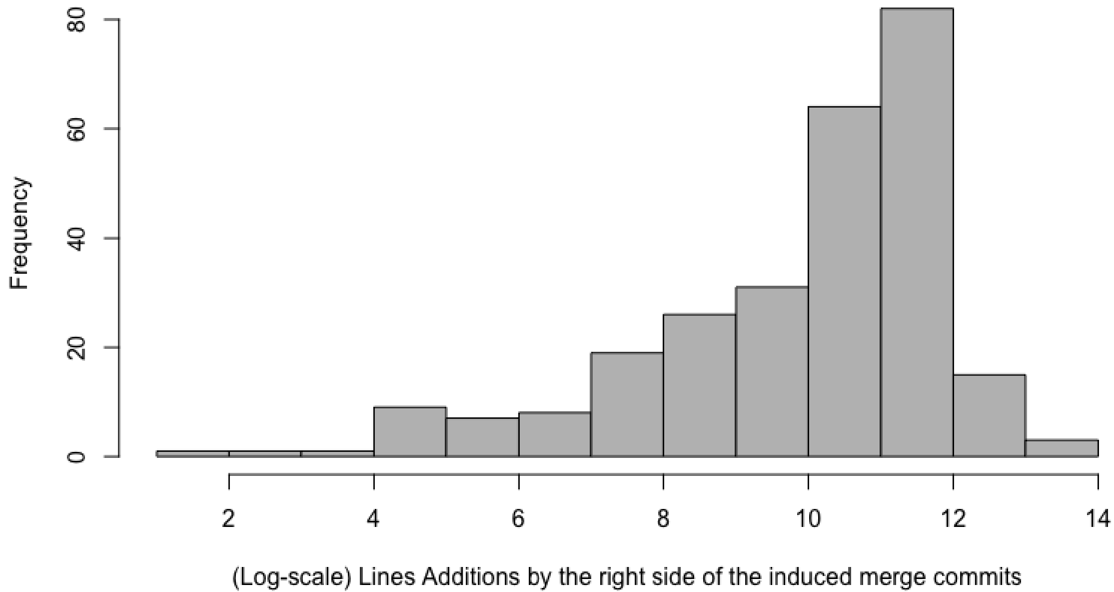

Figure 4.17: Histogram with the log-scale number of lines additions by the right branches of the conflicted merge commits linked to bug-introducing commits

	Right branches			Left branches		
Metric	Files	Additions	Removals	Files	Additions	Removals
Min. :	1	3	1	1.0	1.0	0
1st Qu.:	142	6730	2245	4.0	51.5	13
Median :	650	41085	18848	8.0	215.0	70
Mean :	751	58896	33188	323.7	22862.8	12514
3rd Qu.:	1105	73838	38892	100.0	2898.5	777
Max. :	4432	653754	368349	7903.0	961094.0	738069

Table 4.7: Number of files changed, lines additions and lines removals for both Right and Left branches of induced conflicted merges.

4.3 RQ2 How to predict bugs on conflicted merge scenarios?

In our second research question, our goal is to identify the most important characteristics to predict when a conflict merge scenario is more likely to introduce bugs (according to SZZ algorithm).

Method. To this end, we first filter out the non-conflicting merge scenarios of our curated dataset and selected a couple of features from the literature [1, 38, 45] (as follows) to use as predictors of bugs using data from merge scenarios.

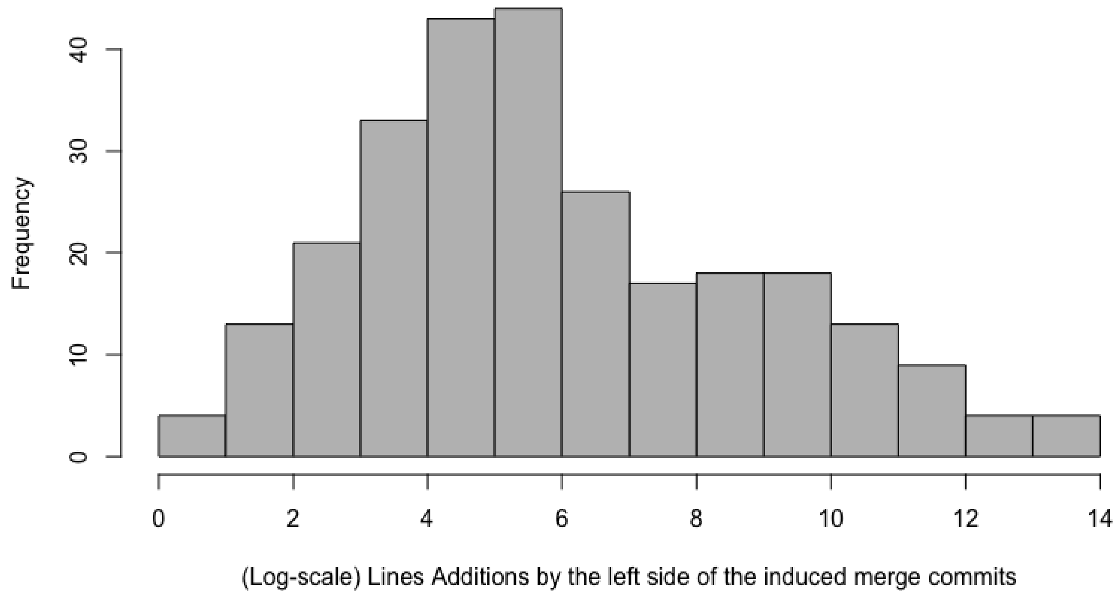

Figure 4.18: Histogram with the log-scale number of lines additions by the left branches of the conflicted merge commits linked to bug-introducing commits

- Total of files changed in both branches
- Total number of contributors in both branches
- Total number of commits in both branches
- Total number of conflicting files

As mentioned before, we compute these features by replaying all merge scenarios. We then investigate the Spearman correlation among these (see the results in Figure 4.27. Similar to previews studies [2], the number of changed files have a extremely low correlation coefficient (p-value = 0.11) while the number of active authors and the number of commits do not have correlation with the number of conflicting files (p-value <= 0.05). Otherwise, we found a strong correlation (coefficients >= 0.75) among the other features number of changed files, contributors, and commits.

In the process of data preparation and feature engineering, we explore our dataset to treat skewness on the predictors, and we decided to run the classification models for each class of merge scenarios (one for simple merge scenarios and other for complex merge scenarios). According to the curated dataset, SZZ linked 1.87% of simple merge scenarios to BICs and linked 4.15% of bug-introducing commits to complex merge scenarios. Finally, we experiment with different classifiers (e.g., Logistic Regression, Decision Trees, and

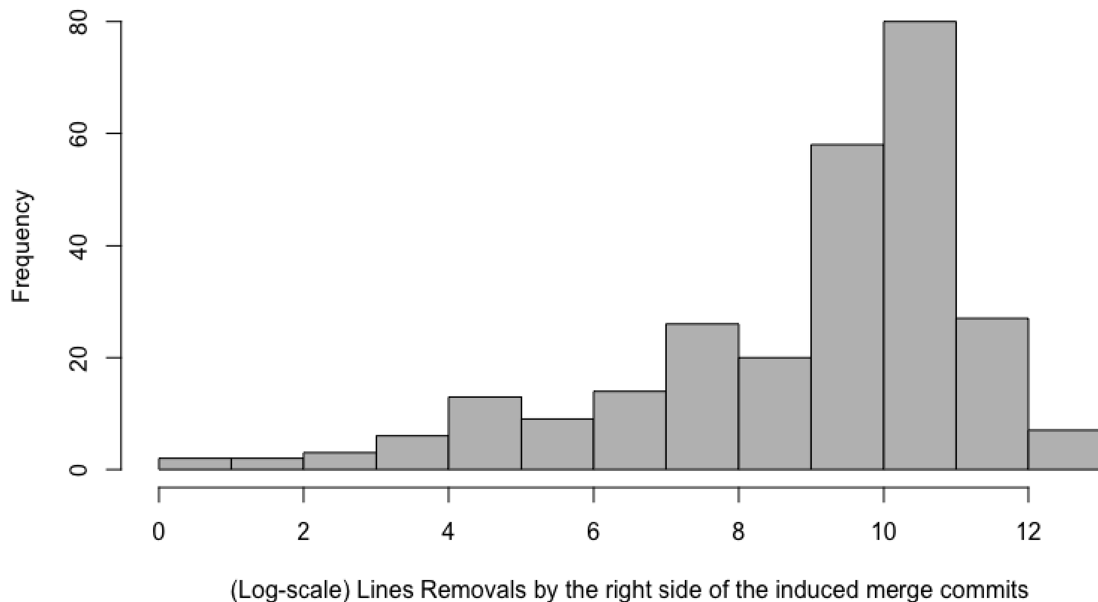

Figure 4.19: Histogram with the log-scale number of lines removals by the right branches of the conflicted merge commits linked to bug-introducing commits

Random Forest), considering all merge scenarios, simple merge scenarios, and complex merge scenarios.

Results. Table 4.8 shows the performance results of the classifiers we trained to answer 2. The results show that, overall, based on the outcomes of the three classifiers, it is hard to predict if a conflicting merge scenario will be responsible for introducing bugs. That is, when considering all conflicting merge scenarios, the Random Forest classifier presented the best performance with a f1-score of 0.1286, followed by Decision Trees (f1-score = 0.1151). On simple merge scenarios, Random Forest also presented the best performance (0.5 of recall and f1-score = 0.1739), but now, followed by Logistic Regression, with higher precision (0.125) and recall = 0.25. Finally, Decision Trees led to a best performance when considering the complex merge scenarios (f1-score = 0.20833 and precision = 0.17045), while Random forest presented higher recall (46.43) with almost the same f1-score (0.208).

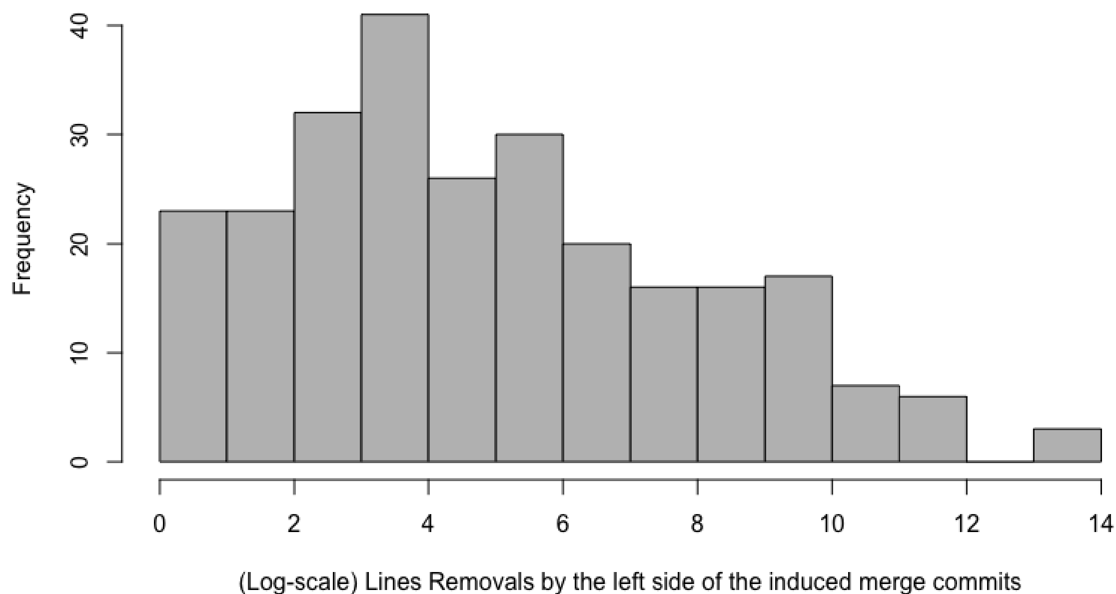

Figure 4.20: Histogram with the log-scale number of lines removals by the left branches of the conflicted merge commits linked to bug-introducing commits

Classifier	Accuracy	Precision	Recall	f1-score
Logistic Regression	0.9383	0.0606	0.03333	0.04161
Decision Trees	0.7122	0.06601	0.45	0.11514
Random Forest	0.7275	0.07417	0.48333	0.1286
Simple	Merges	Scenarios		
Logistic Regression	0.9699	0.125	0.25	0.16667
Decision Trees	0.7892	0.028571	0.5	0.05105
Random Forest	0.9428	0.10526	0.5	0.1739
Complex	Merges	Scenarios		
Logistic Regression	0.5369	0.07593	0.73214	0.13758
Decision Trees	0.8973	0.17045	0.26786	0.20833
Random Forest	0.8216	0.13402	0.46429	0.208

Table 4.8: Performance of the training classifiers — all conflicting merge scenarios, simple merge scenarios, and complex merge scenarios

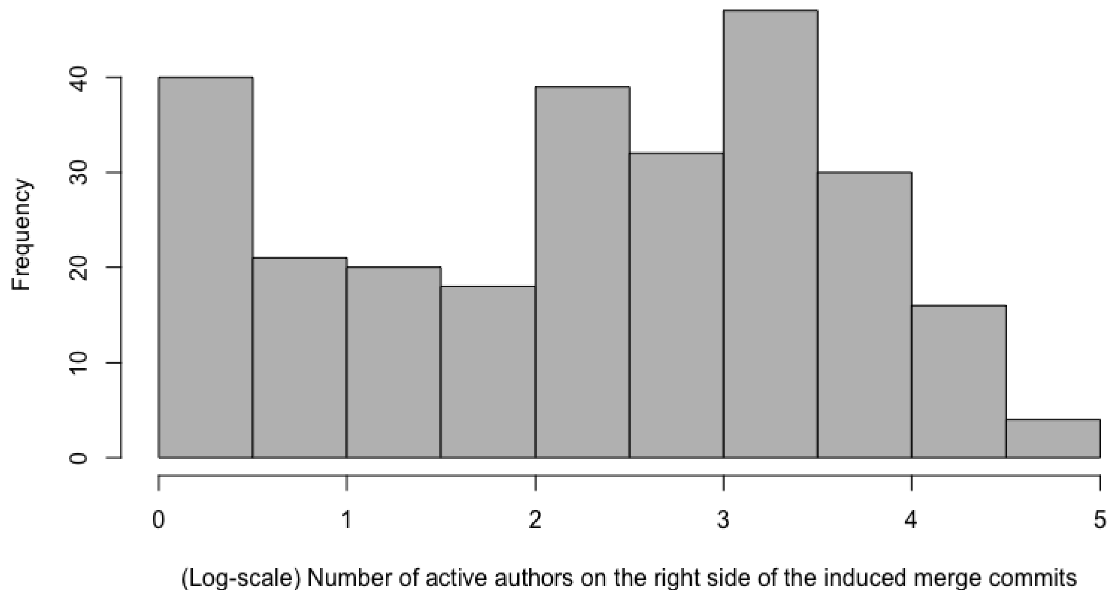

Figure 4.21: Histogram with the log-scale number of active authors on the right branches of the conflicted merge commits linked to bug-introducing commits

4.4 RQ3 To what extent commits with co-change dependencies relate to bug-introducing changes?

The goal of this research question is to investigate if we can explain bug incidence using co-change dependencies. This question has been investigated before by [14], though using a smaller number of systems and applying a different method for relating bugs to components. According to their findings, bug predictions models can be improved with change-coupling (co-change dependencies) information.

Method. To answer this research question, we first use the change history of a system to compute the co-change dependencies between software components (see Section 3.4.1)—either at the coarse-grained level (e.g., classes) or at the fine-grained level (e.g., methods). From the co-change dependencies, we compute two additional metrics [14]: Number of Coupled Classes (NOCC) and Sum of Class Coupling (SOCC). The first computes the *number of classes n-coupled with a given class*—where n specifies a dependency threshold corresponding to the minimum number of changes between two components. The second is the *sum of the shared transactions between a given class c and all the classes n-coupled with c*, and thus SOCC considers the strength of the coupling between two components. When working with fine-grained components, we have

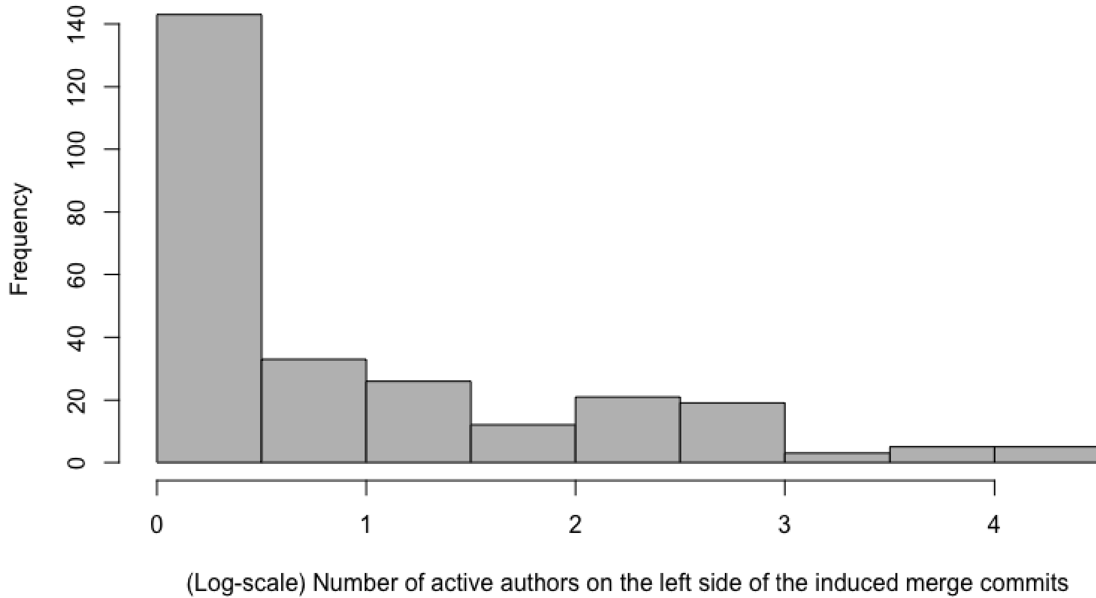

Figure 4.22: Histogram with the log-scale number of active authors on the left branches of the conflicted merge commits linked to bug-introducing commits

the corresponding Number of Coupled Methods (NOCM) and Sum of Method Coupling (SOMC).

We build two datasets with the co-change data (one for coarse-grained components and one for fine-grained components), consisting of the name of the component and the metrics NOCC (NOCM) and SOCC (SOMC). We also compute, using a historage repository, datasets with the change history of all components—where each row corresponds to an observation that a commit changed a given component. We use the first phase of the SZZ algorithm to compute all bug-introducing commits. We then merge the datasets with the change history of all components with the dataset with all BFCs, and computed the number of non bug-fixing (NBC) and bug-fixing commits (BC) of a given component. After that, we estimate the buggy ratio (Br) of a component c using Eq. (1). .

$$Br(c) = \frac{BC(c)}{NBC(c) + BC(c)} \tag{4.1}$$

Finally, we use simple linear regression analysis to estimate the strength of the relationship between NOCC (NOCM) and SOCC (SOMC), with the buggy rate of a component. Simple linear regression allow us to answer the questions (a) *Is there a relationship be-*

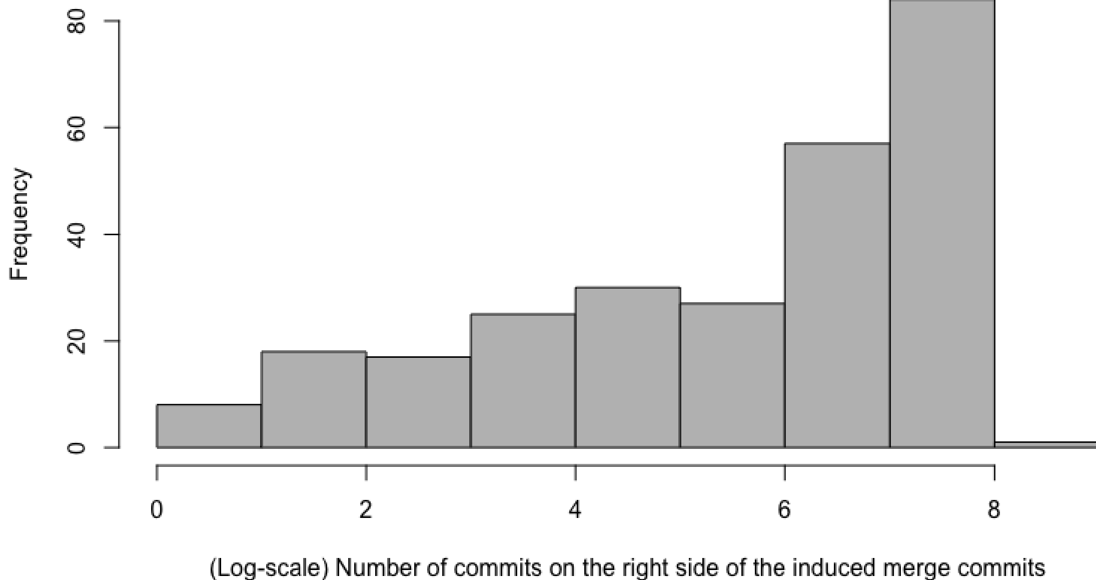

Figure 4.23: Histogram with the log-scale number of commits on the right branches of the conflicted merge commits linked to bug-introducing commits

tween NOCC (NOCM) and SOCC (SOMC) with buggy ratio? and (b) *How strong is the relationship between these features and the buggy ratio?* [32]

Metric	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
NOCC	1	2	5	37.77	22	780
SOCC	5	14	40	270.3	156	5003

Table 4.9: Summary of the metrics NOCC, SOMC, ...

Results. Table 4.9 shows some descriptive statistics from the co-change metrics observations. Interesting, considering the coarse-grained scenario (NOCC and SOCC), most of the observations rely on the interval from 2 (1st Qu.) to 22 (3rd Qu.) co-change dependencies—although we found a specific component with 780 co-change dependencies. Since these unusual observations are increasing the mean value of NOCC, we decided to remove the components having NOCC > 22 from our coarse grained dataset. Tables 4.10 and 4.10 show the results of the simple linear regression analysis. These results suggest that there is **a small relationship between NOCC and SOCC** with the buggy ratio of a class. We also compute the Spearman correlation between NOCC and SOCC with the number of BFCs related to a component, leading to a p-value = 0.26 and 0.28, respectively. These findings contrast with the results of a previous research [14].

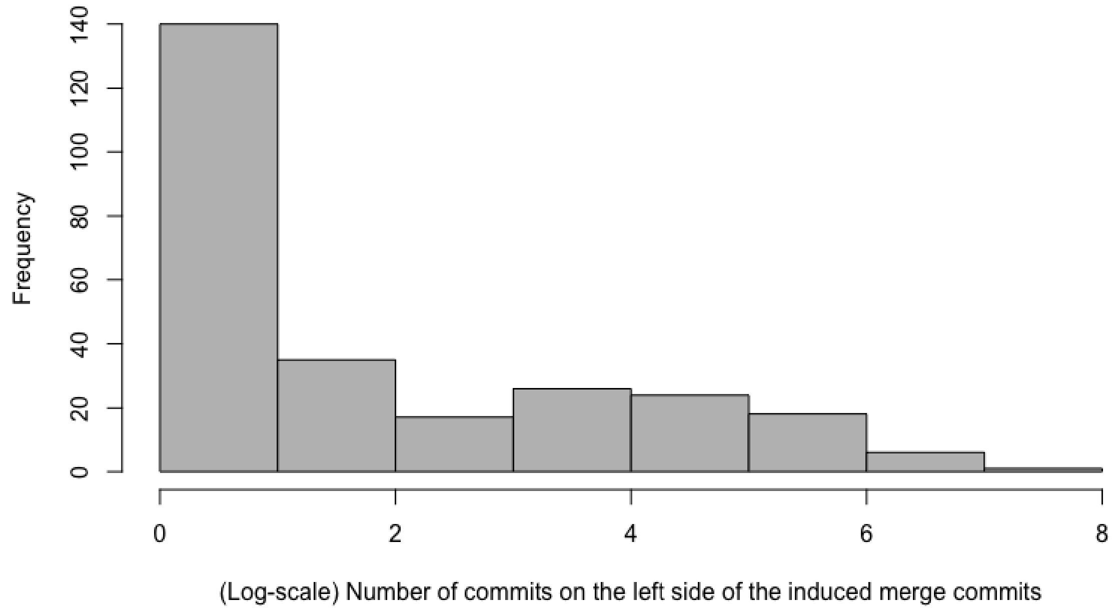

Figure 4.24: Histogram with the log-scale number of commits on the left branches of the conflicted merge commits linked to bug-introducing commits

	Estimate	Std. Error	t value	Pr(>|t|)	
(Intercept)	0.0689608	0.0005255	131.237	<2e-16	(*)
NOCC	0.0003501	0.0001559	2.245	0.0247	

Table 4.10: Simple linear regression of *buggy ratio* on NOCC

	Estimate	Std. Error	t value	Pr(>|t|)	
(Intercept)	6.871e-02	5.200e-04	132.144	< 2e-16	(*)
SOCC	7.515e-05	1.803e-05	4.168	3.08e-05	(*)

Table 4.11: Simple linear regression of *buggy ratio* on SOCC

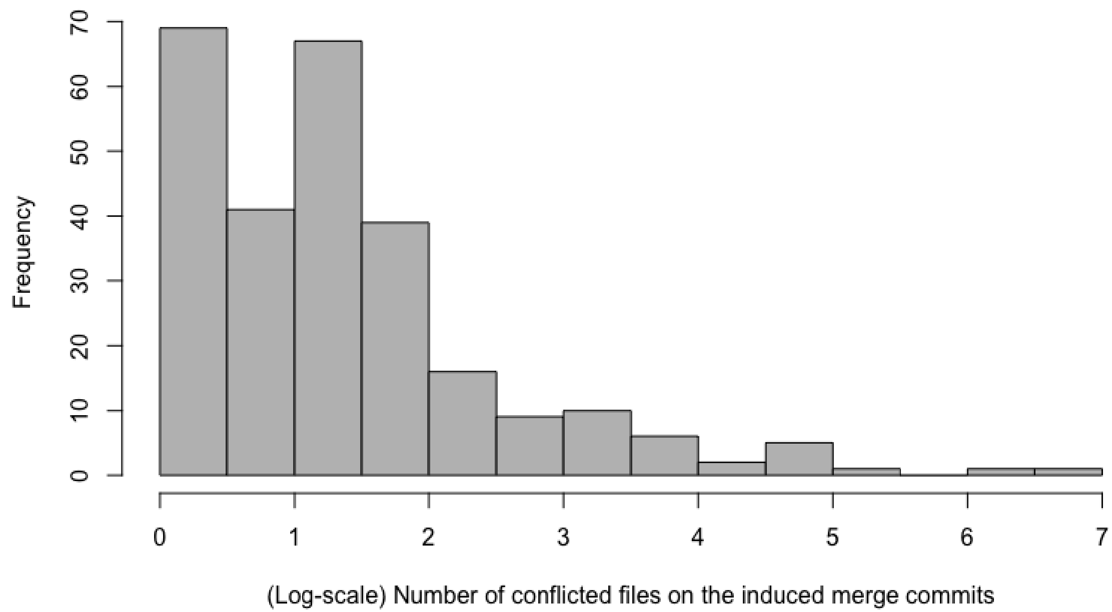

Figure 4.25: (Log-scale) Histogram with the log-scale number of files in conflict for the induced merge commits

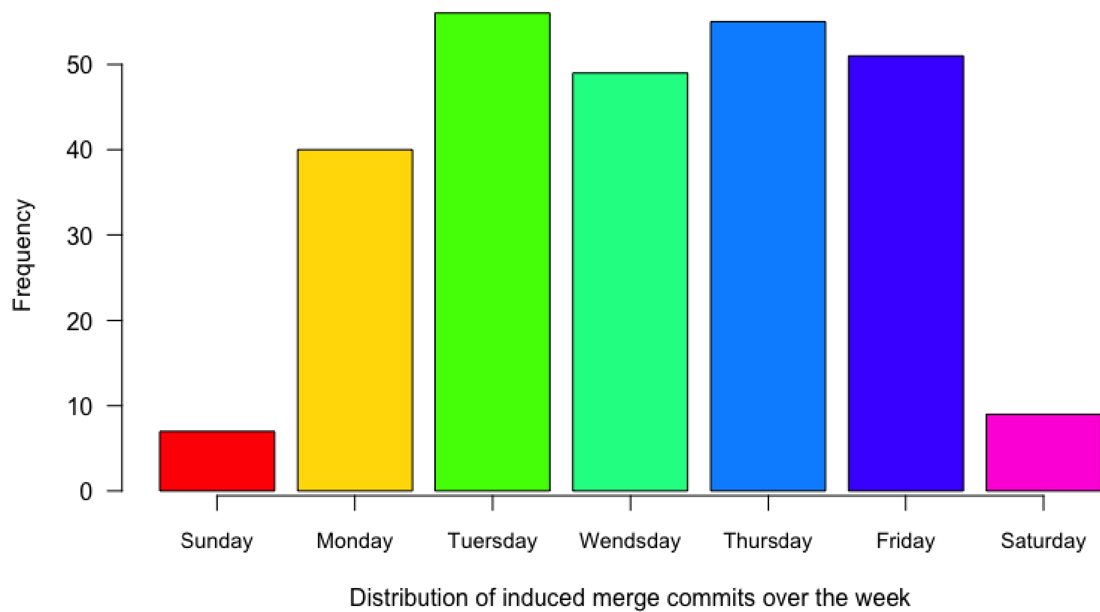

Figure 4.26: (Log-scale) Frequency of conflicted merge commits that introduced bugs over the week

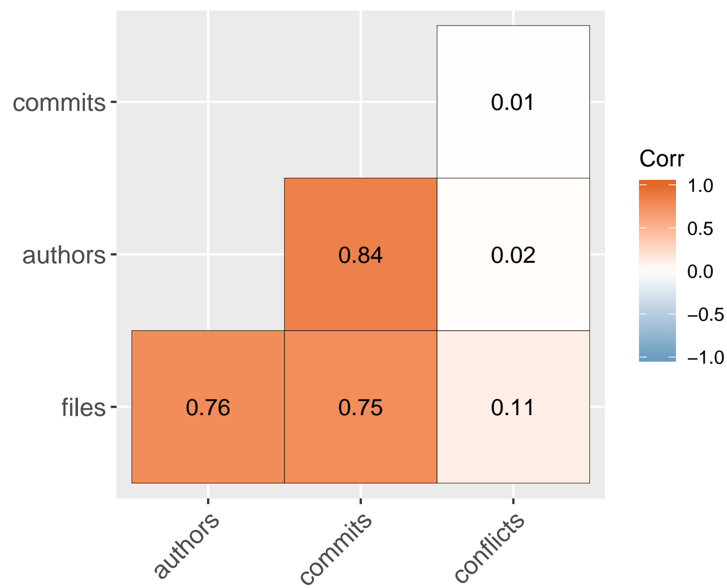

Figure 4.27: Spearman correlation of the features collected from conflicted merge scenarios

Chapter 5

Conclusions

In this work, our goals were to investigate the relationship between different merge scenarios with bug-introducing commits and also verify if we could explain the bug incidence by using co-change dependencies. For doing so, we first conducted an empirical study with a set of Java Apache projects and collected information about bug-fix commits and bug-introducing commits — using SZZ Algorithm. Then, we recreated all merge scenarios of these projects to get conflicting merge scenarios and some characteristics of them — such as the number of changed files, the number of active developers, number of files with conflicts. Moreover, we collected the entities with co-change dependencies for both coarse-grained repositories (in which an entity is a file or a Java class) and fine-grained repositories (e.g., methods or fields). According to the analysis in our set of projects, SZZ-Unleashed blamed 3.62% of conflicting merge scenarios of being responsible for introducing bugs. Moreover, our results indicate that the occurrence of a bug introduced by a conflicting merge scenario is approximately three times lower than the occurrence of a conflicted merge commit. Although our results suggest that only a small number of bug-introducing commits arise from merge conflict resolution, it is worth investigating new methods to detect and avoid conflicts—since the source of these errors only relates to the tasks of resolving conflicting merge scenarios. Furthermore, the framework that recreates merge scenarios and the infrastructure we developed for the quantitative study could be used, for instance, to derive more advanced conflict and bug-introducing prediction models. Practitioners can benefit from these results by worrying less about the bug introduction when resolving merge conflicts and not postpone merge-operations. Differently from previous studies that considered only few study cases, When considering a large set of projects, our study shows that there are not a relation between the metrics of co-change dependencies and the bug introduction changes.

5.1 Contributions

We summarize our contributions as follows:

- Empirical evidence that conflicting merge scenarios are less likely associated with bug-introducing commits occurrence;
- Empirical evidence that it is difficult to predict the introduction of bugs when resolving conflicting merge scenarios based on characteristics extracted from GIT;
- Quantitative evidence that suggests that there is **a small relationship between NOCC and SOCC** with the buggy ratio of a class, contrasting with the results of a previous research [14];
- A Framework to collect all merge scenarios and their characteristics;
- We provide the infrastructure we developed for the quantitative study online, allowing its replication.

5.2 Threats to Validity

The main limitation of our work is related to the SZZ-Unleashed framework. First, since Borg et al. [8] developed it to be language-independent, it does not treat cosmetic changes such as comments and blank spaces. This possibly increased the number of false-positives, but it does not implicate in changing our conclusion that conflicting merge scenarios are less likely associated with bug-introducing commits than other commits. Secondly, in the first phase of SZZ, some bug issues might not be linked correctly with the commits associated with them — perhaps developers might have forgotten to mention the related issue to the changes they were fixing. Even by not collecting all bug issues and linking some of them correctly, we believe that this situation would not change our findings because we collected a massive number of issues, and more than 75% of them were linked to bug-fixing commits. Considering the merge scenarios, we probably did not collect all of them because, in some cases, developers might be used git rebase, and some historical changes were lost. In order to avoid threats related replaying merge scenarios, we cross-validated our results with previous work and tool [45], [4]. Finally, we cannot generalize our results because we have limited our study to Java Apache projects, and further investigation should be necessary with other programming languages.

5.3 Future Works

We aim to cross-validate the output of SZZ-Unleashed with other SZZ implementations [13] and also perform manual verification in a sample of bug-introducing commits to check the reliability of the framework. In addition, we will implement some improvements to SZZ-Unleashed and make it available to help the community with a replication studies. Another possibility, is to investigate the contribution of all kinds of merge conflicts, besides textual, with bug-introducing changes.

Chapter 6

Related Works

6.1 Research on Merge conflicts

Leßenich et al. [38] inferred seven potential indicators to predict merge conflicts based on a survey of 41 developers, in which they shared their typical problem during merge operations and the reasons for conflicts in their projects. Almost all developers agreed that late merging is one of the most causes that lead to negative implications and merge conflicts. Moreover, 38% of them stated that sometimes, they avoid synchronization, leading to late merging, because they fear to face conflicts. Also, the responses suggest that merge conflicts are still a common problem in software development. Some of the indicators that formed the basis for the analysis of the empirical study are: "branches with more commits, larger commits, and more scattered changes are more likely to cause merge conflicts," since more developers are working in parallel. This study aims to verify if merge conflict prediction based on these indicators can be used to help developers during the concurrent code integration. In this study, they analyzed 21,488 real merge scenarios of 163 open-source Java projects in order to test the predictive power and the usability of these indicators to predict challenging merge scenarios. As a result of the empirical analysis, they found that none of the seven indicators that have been suggested by the developers' survey have a predictive power concerning the number of conflicts. By analyzing the correlation of the indicators per project, they found that, for 14 projects, the number of changed lines shows a strong correlation. Similarly, the number of simultaneously changed files has a strong correlation for five projects and a medium correlation for 86 projects. Finally, even this overall negative result, their study formed a solid basis for replication and follow-up studies such as conflict-avoidance strategies (e.g., speculative merging), and the results serve as a warning to practitioners and researchers when making assumptions about merge conflicts.

In the paper "Predicting merge conflicts in Collaborative Software Development,"

Owhadi-Kareshk et al. [45] built some classifiers to decrease the cost of speculative merging running in the background, by avoiding to perform speculative merging in the safe scenarios. They have found this opportunity because proactive conflict detection is based on speculative merging, by pulled and combining all available branches and merging them in background. While it is cheap to perform a single textual merge operation, the cost can increase exponentially according to the number of active branches. Differently of the previous study that measured the correlation between merge conflicts and the features collected from Git, they argue that a lack of correlation does not mean that it is not possible to classify safe versus conflicting merge scenarios. In this study, they collected 267,657 merge scenarios from 744 well-engineered repositories of 7 different programming language, and build separate classifiers for repositories from each programming language. Their results confirmed the lack of a significant correlation between the features and the number of conflicts, but their prediction results show that their classifiers did not perform poorly. While the predictors can detect conflict merge scenarios with a precision of 0.48 to 0.63 and a recall of 0.68 to 0.83 over the programming languages, the safe scenarios presented a precision of 0.97 to 0.98 and recall between 0.93 to 0.96. Even with not so good performance to predict conflict merge scenarios, the results are useful to check safe merge scenarios and reduce the costs of proactive speculative merging, reducing the computational costs.

Accioly et al. [2], conducted an empirical study that analyzes 5,647 merge scenarios from 45 Java-maven-Travis projects from GitHub to collect textual, build, and test conflicts in order to analyze how frequently a predictor occurrence is associated with a conflict occurrence and if they could be considered good predictors. For this, they considered two predictors: EditSameMC (editions to the same method) and EditDepMC (editions to directly dependent methods). In order to reproduce all merge scenarios and collect their information, they used the Conflict Analyzer tool and FSTMerge tool — a semi-structured merge tool that automatically resolves ordering conflicts and spurious conflicts often reported by unstructured merge tools [1]. In the sample, 290 merge scenarios presented merge conflicts, and 508 have conflict predictors — when removing spacing instances, they had 251 merges with conflicts and 469 merges containing the predictors. When they crossed both datasets, there were 286 merge conflicts with at least one predictor and 272 when removing spacing predictors – 45 with EditDepMC and 266 with EditSameMC instances. As a result of the conflict awareness tool considering EditSameMC and EditDepMC, the precision indicates that the tool triggered the alarm 57.99% of the merge scenarios. Moreover, the recall indicates they have captured 82.67% of the merge scenarios with conflicts (merge, build, or test) when considering both predictors. Also, they did not find many build and test conflicts, because they started to

collect data after the implementation of Travis CI on their set of projects — according to the previous study [56], the adoption of CI practices improve the quality of software and developers resolve most of the conflicts locally. For this reason, when analyzing the predictors individually, EditSameMC has a precision of 56.71% and EditDepMC, only 8.85%, and recall of 80.85% and 13.15% respectively. As a conclusion, they say that their study is useful to guide conflict awareness strategies and provide a better notion of the real frequency of merge conflicts.

In the paper "An Empirical Examination of the Relationship Between Code Smells and Merge Conflicts", Ahmed et al. [3] analyzed 143 open source java projects with a total of 36111 merges where 6979 scenarios caused conflicts and 7467 code smell instances in the scope. They also investigated whether there were a connection between entities that contains code smells, the code smells they contain, and the merge conflicts surrounded by smelly entities with the purpose to obtain metrics about code changes and conflicts. First, they divided conflict merge scenarios into two categories — semantic conflicts, (requires to understand the logic of the program to resolve, such as variable name changed), and non-semantic, (easier and less risky to resolve, such as comments and white space). As a result they founded that on average, elements involved in merge conflicts presents three times more code smell than elements not involved in merge conflicts. The mean number of smells in conflicting scenarios is 6.54 while the mean of smells on non-conflicting scenarios is 1.92 — statistically significant with Mann-Whitney test for population not normally distributed. Furthermore, not all code smells are equally correlated to merge conflicts, and the precense of code smells on the lines of code in a merge conflict has a significant impact on its bugginess. Since code smells are more expected to be related with bugs in the future [34], they concludes that entities involving code smells and merge conflicts were more likely to be buggy, and practitioners should pay more attention on code smells to reduce the number of merge conflicts.

6.2 Research on Co-change Dependencies

Marco D'Ambros et al. [15] defined several measures of change coupling to verify their correlations with software defects. In an empirical study with three large Java software systems, they provide evidence that change coupling correlates with defects extracted from the issue tracking system. Moreover, they have investigated, based on the severity of the reported bugs, the relationship between co-change dependencies and software defects. Furthermore, they found a better correlation connecting coupling changes and defects than complexity metrics. Finally, they showed that it is possible to improve the performance of defect prediction models based on complexity metrics by adding co-change information.

In this work, they considered an entity as a class, meaning that each entity received change coupling measures coarse-grained — they defined the measures concerning the coupling of a class with the entire system. As a result, they say that change coupling correlates with defects, more than object-oriented metrics, but less than the number of changes. Another result is that regression models based on objected-oriented metrics, and change coupling information have greater explanatory and predictive than models based only on metrics. As threats to validity, they considered that using a coarse-grained analysis is not the best option because they cannot consider inner classes. Finally, considering the commit messages to find bug-fix commits cannot guarantee that all bug fixing information was collected.

Wise et al. [54] empirically investigated the relationship between strong change coupling, defined by them using historical and social metrics, and the number of defects associated with them. More than 50% of the releases with more change couplings were associated with the defect, and 3/4 of the change couplings are associated with at least one defect. Based on historical and social metrics, they build classification models to identify strong change couplings with 70-99% F-measure and 88-99% AUC. Also, they have built a defect prediction model based on strong change dependencies and correctly predicted 45.7% of the defects. They have defined change coupling as strong for all couplings with support higher than the third quartile, and, otherwise, defined as weak. They have considered as a dataset, the six releases of the project Apache Aries, collecting the number of issues and the number of change coupling per release. As threats to validity, the first concern to generalize the results, since they have analyzed only one case study. Another one is the possibility of tangled code changes [30] in the commits they have mined.

References

[1] P. Accioly, P. Borba, and G. Cavalcanti. Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Software Engineering*, 23(4):2051–2085, 2018. 39, 53

[2] P. Accioly, P. Borba, L. Silva, and G. Cavalcanti. Analyzing conflict predictors in open-source java projects. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 576–586, 2018. 8, 40, 53

[3] 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. IEEE, 2017. 14, 54

[4] S. Apel, O. Leßenich, and C. 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. 14, 50

[5] F. Beck and S. Diehl. On the impact of software evolution on software clustering. *Empirical Software Engineering*, 18(5):970–1004, 2013. 21

[6] S. P. Berczuk and B. Appleton. *Software configuration management patterns: effective teamwork, practical integration*. Addison-Wesley Longman Publishing Co., Inc., 2002. 7

[7] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful... really? In *2008 IEEE International Conference on Software Maintenance*, pages 337–345. IEEE, 2008. 8

[8] M. Borg, O. Svensson, K. Berg, and D. Hansson. Szz unleashed: An open implementation of the szz algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project. *arXiv preprint arXiv:1903.01742*, 2019. ix, 3, 10, 11, 15, 16, 50

[9] Y. Brun, R. Holmes, M. D. Ernst, and D. 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. 7, 8

[10] G. Cavalcanti, P. Borba, and P. Accioly. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):59, 2017. 14

[11] S. Chacon and B. Straub. *Pro git*. Apress, 2014. 5, 6

[12] P. Charles. Project title. `https://github.com/charlespwd/project-title`, 2013. 14

[13] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017. 10, 12, 51

[14] M. D'Ambros, M. Lanza, and R. Robbes. On the relationship between change coupling and software defects. In *2009 16th Working Conference on Reverse Engineering*, pages 135–144, Oct 2009. 43, 45, 50

[15] M. D'Ambros, M. Lanza, and R. Robbes. On the relationship between change coupling and software defects. In *2009 16th Working Conference on Reverse Engineering*, pages 135–144. IEEE, 2009. 54

[16] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *International Conference on Global Software Engineering (ICGSE 2007)*, pages 81–90. IEEE, 2007. 8

[17] M. C. de Oliveira, R. Bonifácio, D. Freitas, G. Pinto, and D. Lo. Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings. 2019. ix, 2, 3, 13

[18] M. C. de Oliveira, R. Bonifácio, G. N. Ramos, and M. Ribeiro. Unveiling and reasoning about co-change dependencies. In *Proceedings of the 15th International Conference on Modularity*, pages 25–36. ACM, 2016. 2, 13, 21

[19] C. R. De Souza, D. Redmiles, and P. 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. 1

[20] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *CSCW*, volume 92, pages 107–114, 1992. 8

[21] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007. 1

[22] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 23–32. IEEE, 2003. 5

[23] M. Fowler and M. Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122:14, 2006. 1, 7

[24] S. R. Fussell, R. E. Kraut, F. J. Lerch, W. L. Scherlis, M. M. McNally, and J. J. Cadiz. Coordination, overload and team performance: effects of team communication strategies. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 275–284. ACM, 1998. 8

[25] T. E. Gerhardt and D. T. Silveira. *Métodos de pesquisa.* Plageder, 2009. 14

[26] G. Ghiotto, L. Murta, M. Barros, and A. 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. IEEE*, 2018. 8

[27] R. E. Grinter. Using a configuration management tool to coordinate software development. In *Proceedings of conference on Organizational computing systems*, pages 168–177. ACM, 1995. 1

[28] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering*, pages 342–352. IEEE Press, 2012. 1

[29] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 international conference on software engineering*, pages 392–401. IEEE Press, 2013. 11

[30] K. Herzig and A. Zeller. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 121–130. IEEE, 2013. 55

[31] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 52–61. IEEE, 2008. 8

[32] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: With Applications in R.* Springer Publishing Company, Incorporated, 2014. 45

[33] S. Just, R. Premraj, and T. Zimmermann. Towards the next generation of bug tracking systems. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 82–85. IEEE, 2008. 8

[34] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, 2012. 54

[35] S. Kim, T. Zimmermann, K. Pan, E. James Jr, et al. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 81–90. IEEE, 2006. 2, 9

[36] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006. 2

[37] M. Lerner. Software maintenance crisis resolution: The new ieee standard. *Software Development*, 2(8):65–72, 1994. 2

[38] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering*, 25(2):279–313, 2018. 39, 52

[39] P. Li. *JIRA 5.2 Essentials*. Packt Publishing Ltd, 2013. 14

[40] 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. IEEE, 2017. 6, 8

[41] T. Mens. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, 28(5):449–462, 2002. ix, 6, 7

[42] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006. 12

[43] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić. The emergent structure of development tasks. In *European Conference on Object-Oriented Programming*, pages 33–48. Springer, 2005. 12

[44] E. C. Neto, D. A. da Costa, and U. Kulesza. The impact of refactoring changes on the szz algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390. IEEE, 2018. 12

[45] M. Owhadi-Kareshk, S. Nadi, and J. Rubin. Predicting merge conflicts in collaborative software development. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019. 1, 6, 8, 39, 50, 53

[46] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. 12

[47] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(3):308–337, 2001. 8

[48] G. Rodríguez-Pérez, J. M. Gonzalez-Barahona, G. Robles, D. Dalipaj, and N. Sekitoleko. Bugtracking: A tool to assist in the identification of bug reports. In *IFIP International Conference on Open Source Systems*, pages 192–198. Springer, 2016. 11

[49] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology*, 99:164–176, 2018. ix, 10, 11, 15

[50] G. Rodríguez-Pérez, A. Zaidman, A. Serebrenik, G. Robles, and J. M. González-Barahona. What if a bug has a different origin?: making sense of bugs without an explicit bug introducing change. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 52. ACM, 2018. 2

[51] L. L. Silva, M. T. Valente, and M. d. A. Maia. Co-change clusters: Extraction and application on assessing software modularity. In *Transactions on Aspect-Oriented Software Development XII*, pages 96–131. Springer, 2015. 21

[52] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005. 2, 3, 9, 10, 12

[53] J. Visser, S. Rigal, G. Wijnholds, and Z. Lubsen. *Building Software Teams: Ten Best Practices for Effective Software Development.* " O'Reilly Media, Inc.", 2016. 5

[54] I. S. Wiese, R. T. Kuroda, R. Re, G. A. Oliva, and M. A. Gerosa. An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project. In *IFIP International Conference on Open Source Systems*, pages 3–12. Springer, 2015. 55

[55] C. Williams and J. Spacco. Szz revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 32–36, New York, NY, USA, 2008. ACM. 10, 12

[56] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 60–71. IEEE, 2017. 54

[57] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu. Improving bug tracking systems. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 247–250. IEEE, 2009. 8