Chair of Software Engineering

Software Practitioners Perspective on Merge Conflicts and Resolution

Masterarbeit von

Muhammad Zohaib Brohi

1. Prüfer	2. Prüfer
Prof. Dr. Sven Apel	Prof. Dr. Gordon Fraser

17. Juli 2019

Abstract

Merge conflicts are undeniable events in collaborative software development. Merge conflicts arise when developers perform concurrent changes in the same part of code. Researcher and practitioners seek to minimize the number of merge conflicts because resolving them is difficult, time consuming, and often an error-prone task. Despite the number of studies investigating merge conflicts, there are only few studies focusing on merge conflicts resolution. The goal of this study is to investigate which factors make conflicting merge scenarios harder to resolve in practice. To achieve the goal, we analyzed 31 projects developed in 13 programming languages containing around 45 thousand merge scenarios involving 1.2 million files, and 6.3 million chunks. Methodologically, we use rank correlation, principal component analysis, multivariate regression model, and effect size to investigate which independent variables (e.g. number of conflicting chunks and files) influence most our dependent variable (i.e., number of seconds to merge). As result, we find that the number of chunks (*#Chunks*) and the code complexity of conflicting code (*CodeComplexity*) have a negative influence while number of lines of code (*#LoC*), number of conflicting chunks (*#ConfChunks*), number of developers involved in a merge scenario (*#Devs*), and the percentage of the files that the integrator had knowledge before solving the merge conflict (*%IntegratorKnowledge*) have a positive influence. Our *effect size analysis* reveals that *#Chunks*, *#LoC*, and *#Devs* have a medium effect size on the merge conflict resolution time and *#ConfChunks*, *CodeComplexity*, and

%IntegratorKnowledge have a small effect size on the merge conflict resolution time. In further analysis we discuss reasons of why some projects are prone to have formatting changes, why integrators that had knowledge on changed files before face with the merge conflicts need more time to solve merge conflicts than integrators that did not had previous knowledge on the conflicting code, and why merge scenario size metrics have a stronger correlation with the merge conflict resolution time than merge conflict size metrics.

Acknowledgments

I want to thank my supervisor Gustavo do Vale for his guidance and support throughout the thesis work. Without his encouragement guidance, this thesis would not have materialized.

Contents

List of Figures

List of Tables

1 Introduction

Collaborative software development is more often than not, a team effort in which the success depends on the ability to coordinate social and technical assets [JAM17]. Multiple developers may work and update a single project concurrently by working on a separate version of the mainline project. The changes in the separate version of the project are pushed to the mainline project, which is updated by other developers to ensure that they have the latest version of the current project.

Version control systems (VCSs) is a tool used to achieve collaborative software development [Whi10]. The project evolves incrementally by developers performing tasks. With the support of version control systems changes in the project are saved overtime in a repository. It allows tracking all the changes made in the past (e.g. detect what, why, and when changes were introduced, and who did them). In case any changes went wrong, developers can go back in time and revert them to its working version. With the concept of branching, VCSs allow developers to create a branch to implement new features, refactor existing features, or to fix bugs simultaneously without affecting the project branch [PSW11]. Once the task is completed in the working branch, developers *merge* it to the main repository branch.. Overwriting changes in this manner is often referred to as *Merging Branches* [Roc17] and the whole process is often referred pull-based model or three-way merge [Les18][GPD14].

However, merging particular code changes to the project may introduce merge conflicts. Merge conflicts occur when one or multiple developers change the same chunk of code in different branches. Merge conflicts should be resolved before continuing working on the project as the code will not be able to run. According to Kasi and Sarma [KS13], merge conflicts occur on an average of 23% of the merges. Researchers seek to minimize the number of merge conflicts, because resolving them is difficult, time-consuming, and often error-prone [Les18][Men02]. Studies related to merge conflicts involves: (i) merge strategies (e.g., structured [ALL12] or semi-structured [Ape11]), (ii) prediction strategies (e.g., continuous integration [GS12] and speculative merging [Bru11]), (iii) awareness tools (e.g., CollabVS [DH07], Palantír [SRv12], Cassandra [KS13], and FASTDash [Bie07]), (iv) studies to understand their nature (e.g., type of code changes that lead to conflicts) [ABC18][Les18], and (v) merge conflict resolution strategies.

Despite the number of studies investigating merge conflicts, the resolution of merge conflicts is in its initial phase [Nel19] [S M17]. For instance, Nelson et al. [Nel19] have presented 9 factors that impact the evaluation and resolution of a merge conflict, these factors include: complexity of conflicting line of code, expertise in area of conflicting code, complexity of files with conflicts, number of conflicting lines of code, time to resolve a conflict, number of files in the conflict. They asked developers to point out factors that make a conflicting merge scenario harder to solve. However, there is no empirical study to see if these factors reflect the working practice.

The goal of this study is to investigate which factors do make merge conflicts harder to resolve in practice. Knowing that we may provide opportunities for future research by presenting practical insights into the factors that influence the conflicting merge scenarios resolution. Without such knowledge, tool builders may build tools on wrong assumptions, and researchers may miss opportunities for improving the state of the art.

To achieve our goal, we extract merge scenario information from 31 GitHub projects chosen based on their popularity. Merge scenario information includes variables: the number of files (*#Files*), number of conflicted files (*#ConfFiles*), number of lines of code (*#LoC*), number of conflicting lines of code (*#CLoC*), number of chunks (*#Chunks*), number of conflicting chunks (*#ConfChunks*), number of developers (*#Devs*), developer knowledge in the area of the conflicting file (*%IntegratorsKnowledge*), formatting changes (*%FormattingChanges*), code complexity of the chunk (*CodeComplexity*) and time (in seconds) taken to resolve the merge conflict (*#SecondsToMerge*). We use time as our proxy of difficulty, hence, we create a matrix correlation among all covariables and present the covariables concerning their *rank correlations* with other variables. We additional use *principal component analysis* and to reduce the dimensionality of the dataset to provide better visualization of the variables. We also use *multivariate regression model analysis* to build the model with the highest variance with all independent variables and groups that we obtained from the *principal component analysis*. Finally, we perform *effect size analysis* to answer our research question by classifying independent variables which have a larger effect on merge conflict resolution time.

As a result of our study, we find that *#Chunks*, *#LoC*, and *#Devs* are top 3 variables that make conflicting merge scenarios harder to resolve in practice. We find strong correlation among *#LoC*, *#Files*, *#Chunks*, and moderate correlation among *#ConfLoC*, *#ConfFiles*, and *#ConfChunks*. Finally, we also find that for a fixed amount of *#ConfLoC*, *#Chunks*, *#Devs*, *CodeComplexity*, and developers knowledge in the area of conflicting code (*%IntegratorKnowledge*), adding 100 LoC in the merge scenario leads to an increase in time by approximately 8 minutes to solve the merge conflicts, on average. Our discussions show the reasons why some projects are prone to have formatting changes, why integrators that had knowledge on changed files before face with the merge conflicts need more time to solve merge conflicts than integrators that did not had previous knowledge on the conflicting code, and why merge scenario size metrics have a stronger correlation with the merge conflict resolution time than merge conflict size metrics.

This thesis makes the following main contributions:

- We provide evidence that *#Chunks*, *#LoC*, *#Devs*, *#ConfChunks*, *CodeComplexity*, *%IntegratorKnowledge* has effect on the merge conflict resolution time.
- We provide evidence that variables indirectly related to merge conflicts, such as the number of lines of code changed in a merge scenario has high impact on time, and variables directly related to merge conflict, such as the number of lines of code in conflict.
- We found positive correlation coefficient among *#LoC*, *#ConfLoC*, and *#Devs*, and negative correlation coefficient among *#Chunks* and *CodeComplexity*, and *%IntegratorKnowledge* and *CodeComplexity*.
- We make our infrastructure publicly available to mine fine-grained merge scenario information from software repositories. [Bro19].
- We make all data publicly available for replication and follow-up studies on a supplementary website [Bro19].

This thesis is structured as follows: Chapter 2 contains background on collaborative software development, version control systems, pull-based model, and merge conflicts. Chapter 3 describes related work. Chapter 4 describes the methodology used to gather and analyze the merge conflicts. Chapter 5 presents the results of the study. Chapter 6 discusses the outcome of the study and threats to validity. Chapter 7 concludes the study and discusses future work.

2 Background

Software development is a collaborative and distributed activity in which success depends on the ability to coordinate social and technical assets [JAM17]. Coordination of social and technical assets is one of the greatest challenges in collaborative software development, especially in open source systems where many developers contribute voluntarily and are normally geographically distributed [SRP17]. To achieve high productivity and support daily tasks, developers normally choose well-defined development models (e.g., pull-based model) and version control systems (e.g., Git, Mercurial, and SVN). However, with great number of code contributions simultaneously, problems are prone to occur. Among these problems, merge conflicts have attracted the attention of researchers and practitioners because solving them is a difficult, time-consuming, and often an error-prone task [Men02].

To get deeper in concepts and technologies of collaborative software development, the following sections present an overview of *version control systems*, the *pull-based model*, and *merge conflicts*.

2.1 Version Control Systems

Version control systems support collaborative software development by tracking all source code changes over time. This way, if something goes wrong, developers can revert previous changes and fix the unexpected behavior [Zim04]. Version control systems can be decentralized or centralized [DH07]. In *centralized version control systems*, there is a single centralized master copy of the code base, and changes are made to this central copy. Part of the changed code is locked, and only one developer is allowed to work on that part of the code [AS09]. Examples of centralized version control systems are concurrent version systems (CVS[1]) and Subversion[2]. In *distributed version control systems*, a project is cloned locally, with its full history. To work on distributed version control systems, developers make code changes locally and, when they finish their tasks, they merge it back to the main project branch. Hence, in distributed version control systems multiple developers can work on the same part of the code concurrently. Examples of distributed version control systems are Git[3] and Mercurial[4].

Distributed version control systems allow developers to share code without adopting the main repository. On the other hand, centralized version control systems use the main repository through which developers share their code. Distributed version control systems offer flexibility for reviewing and fixing changes ahead of submitting to the main repository. This way, they can be considered an evolution of centralized version control systems and are used more often, especially in open source projects. The huge use of distributed version control systems on open source projects comes basically because it allows complex work-flows, tasks being distributed among several developers, and does not block the work of others contributing to the project [AS09].

[1]https://www.nongnu.org/cvs/
[2]https://subversion.apache.org/
[3]https://git-scm.com/
[4]https://www.mercurial-scm.org/

2.2 Pull-Based Model

To extract the best of version control systems and achieve high productivity with multiple developers contributing to the project, it is also important choosing an efficient development model. The pull-based development model is a widely used development model of which only GitHub platform hosts more than a million of projects [GSB16]. The pull-based model consists of merge scenarios and it is basically based on three steps: (i) contributors create new branches from the main repository of the project they want to contribute to, (ii) contributors make their changes independently to solve a task (e.g., new feature implementation, bug fixing), and (iii) they integrate it back to the main repository [GSB16]. The integration is normally done by means of merge commits or pull requests. Integrations using pull requests happen when directly in one hosting platform, such as GitHub[5], GitLab[6], and Bitbucket[7]. This process is also called three-way merge [GPD14] [Les18].

A good practice when using this model is to create different branches for release, development, implementing a new feature, or fixing bugs [PSW11]. Figure 2.1 exemplifies this model of which each dot represents a commit. In this example, the merge scenario consists of the integration of two branches (left and right), however, more branches can be integrated simultaneously in a merge scenario. To retrieve information from a merge scenario, there are three important commits: *base commit*, *parent commit*, and *merge commit*. The base commit, the one most to the left on the top of Figure 2.1 (commit hash: f5a31ed), is the common ancestor between the two branches. The parent commit is the last commit of a branch before the integration. In this example we see one parent commit to the left (commit hash: 32cc0f8) and another to the right branch (com-

[5]https://github.com
[6]https://about.gitlab.com
[7]https://bitbucket.org

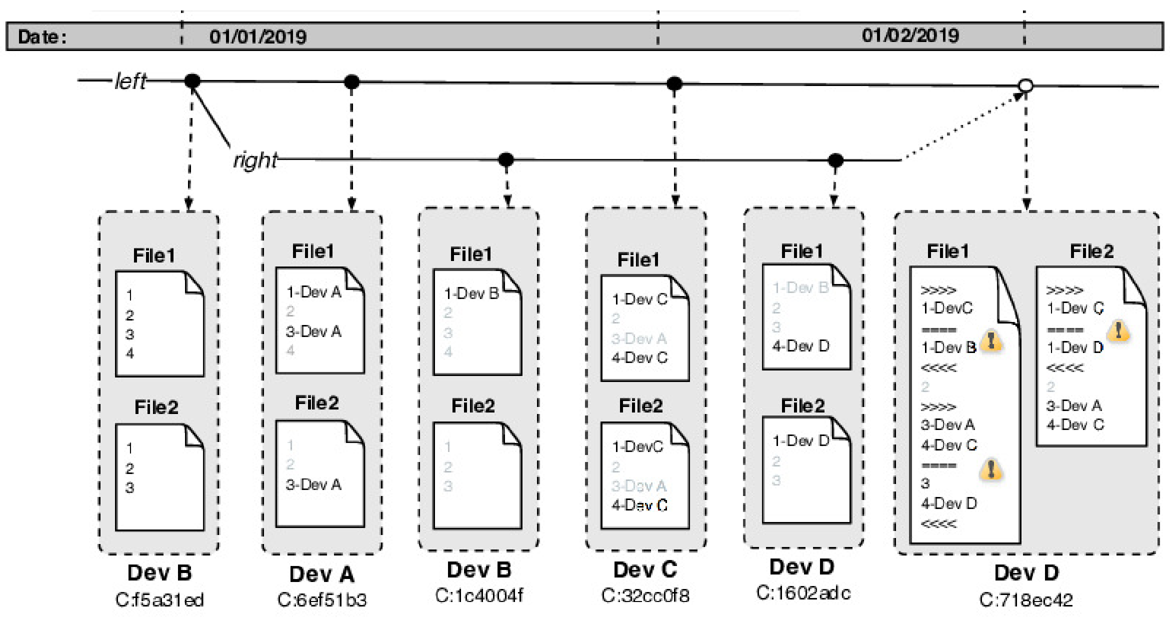

Figure 2.1: Overview of the Pull-based Model by Means of a Merge Scenario

mit hash: 1602adc). The merge commit is the commit that integrates both branches (commit hash: 718ec42).

2.3 Merge Conflicts

As mentioned, merge conflicts occur if developers change the same part of chunk code in the branches that are going to be merged. As merge conflicts are unexpected events, they have a negative effect on project's objectives compromising the project success, especially when arising frequently [GS12][JAM17][SRv12]. In Chapter 3, we give more details about how researchers have been investigating merge conflicts over the past years.

As mentioned, Figure 2.1 presents a merge scenario. In this scenario developers A and C changed the left branch while developers B and D changed the right branch. As there is concurrent changes in the same chunk of code, this is a conflicting merge scenario (i.e., there are merge conflicts). In this merge scenario we can see three merge conflicts highlighted by the exclamation mark. For instance, as developer C changed line 1 of File1 on the left branch while developer B changed the same line of code on the right branch a merge conflict emerged when trying to merge both branches.

In Figure 2.1, much information can be extracted from a merge scenario. Such information includes, for instance, the number of developers involved, the number of files, chunks, and the number of conflicting chunks in the merge scenario. In Chapter 4, we present all information extracted for each merge scenario investigated in this thesis.

3 Related Work

This chapter describes previous work in the area of merge conflicts. It is essential to learn some of the studies performed in this area to get benefit from work done in this study. Section 3.1 contains studies in the field of merge strategies. Section 3.2 contains studies to detect merge conflict in the early stage. Section 3.3 presents studies that provide tools to support developers awareness to minimize the number of merge conflicts. Section 3.4 explains the nature of merge conflicts. Section 3.5 presents studies that investigate how developers approach merge conflicts, how some merge conflicts are harder to resolve, and insights into how developers resolve merge conflicts.

3.1 Merge Strategies

Merging strategies can be classified into unstructured, structured and semi-structured. Unstructured strategies treat software artifacts as sequences of text lines and they are known to be fast, however not efficient in avoiding merge conflicts. Structured strategies are based on the artifacts syntactic structure and they are known as efficient in avoiding merge conflicts, however, computationally expensive. Semi-structured strategy is trade-off which inherits strengths of structured and unstructured strategies. This way, Apel et al. [ALL12] offers a software merging strategy that obtains a balance between precision and performance. Thereby, semi-structured strategies represent artifacts as trees, however an artifact is only partly exposed in the tree, and the rest is treated as plain text [ALL12]. In what follows we describe how the semi-structured strategy is proposed. First, software artifacts are represented as trees. When developers merge code changes, this strategy merge them using tree matching and amalgamation. Then, for performance enhancement, they rely on auto-tuning. To apply expensive differencing and merge operations in suitable situations, this strategy used unstructured merge when it is free of merge conflict, and structure merge when conflicts are detected. In a empirical study [Ape11] noticed that: semi-structured strategy reduced the number of merge conflicts by 21±34 %, the number of conflicting lines of code by 22±61%, and the number of conflicting files by 12±28%.

3.2 Prediction Strategies

When merge conflicts are detected late, software defects may appear and it requires expensive resolution since developers may have forgotten the reasons of their code changes. Guimaraes et al. [GS12] propose a solution for early detection of the merge conflicts on behalf of developers. Their solution introduces the concept of continuous merging inside the IDE. Analyzing merge conflicts earlier provides information to developers taking better decisions regarding how to proceed, whether it will be a safe merge (i.e., free of merge conflicts), to publish a safe change, to immediately address a new merge conflict, and to interact with developers [Bru11]. Brun et al. [Bru11] provide an approach to support developers classifying and resolving merge conflicts early before those conflicts become difficult and before developers forget about the notable changes. They explain the design of Crystal[1], a publicly-available tool that uses speculative analysis to deliver

[1]http://crystalvc.googlecode.com/

concrete advice unobtrusively accessible to developers, helping them identify, maintain, and prevent merge conflicts. Their study verifies that: (i) conflicts are the norm rather than the exception, (ii) 16% of all merges required human effort to solve textual conflicts, (iii) 33% of merges that were reported to contain no textual conflicts by the VCSs, included higher-order merge conflicts, and (iv) conflicts endure, on average, for ten days.

3.3 Awareness Tools

Merge conflicts arise when developers make concurrent changes to the code-base. The primary goal of awareness tools is to identify potential merge conflicts early, while conflicts are small and easier to resolve. Some studies provide supporting tools when developing software. These tools aim at minimizing the number of merge conflicts and notify developers of emerging conflicts [SNv03]. Kasi et al. [KS13] present Cassandra. Cassandra proactively identifies merge conflicts and other constraints to manage the task that will avoid the incidence of merge conflicts. Biehl et al. [Bie07] present FASTDash. It provides an interactive visualization which enhances team activity awareness. FASTDash determines which team members have files checkout, which files were viewed and which classes and methods were changed. It gives insights into potential conflict situations, such as two or more developers working on the same file.

3.4 Understanding Merge Conflicts Nature

Merge conflicts occur when merging concurrent changes into the main project, and developers have to complete the merge manually. The amount of pain when solving the merge conflicts can be reduced if the developer knows the nature of the merge conflict. In what follows, we present three studies which investigate the nature of the merge conflicts.

Lessenich et al. [Les18] analyzed seven potential indicators to predict the number of merge conflicts. These indicators are: (i) number of commits, (ii) commit density, (iii) number of files changed by both branches, (iv) number of changed lines of code, (v) number of code chunks changed, (vi) number of changes inside class declaration, and (vii) scattering degree (number of changed classes/number of all classes). They computed correlations between the seven potential indicators and the number of conflicts in a merge scenario. They found that none of the seven indicators can predict the number of merge conflicts.

Accioly et al. [ABC18] derive a merge conflict catalog with nine patterns to understand the structure of the changes that lead to merge conflicts. These patterns are: (i) edits to the same or consecutive lines of the same method, (ii) different edits to the same class field declaration, (iii) methods added with the same signature and different bodies, (iv) class fields declarations added with the same identifier and different types or modifiers, (v) different edits to the modifier list of the same type declaration, (vi) different edits to the same implements declaration, (vii) different edits to the same extends declaration, (viii) different edits to the same Enum constant declaration, and (ix) different edits to the same annotation method default value declaration. Their results showed that 84.57% of merge conflicts occur because developers edit the same lines or consecutive lines of the same method. Editing methods, class fields, or modifier lists have similar probability of leading to merge conflicts. Merge conflicts can be avoided if awareness tools are improved to alert developers of these cases.

Ghiotto et al. [Ghi18] focus on the investigation of the nature of merge conflicts, and how the developers resolved them. They analyzed thousands of merge scenarios from five different open source projects. Then, they identified merge scenarios that led to merge conflicts and chunks which are involved in merge conflicts. Finally, they analyzed the resolution strategy used by developers to resolve the merge conflicts. They analyzed different perspectives that make a merge conflict more difficult to resolve than others. These perspectives included the size and the number of conflicting chunks, the presence of language constructs in the conflict chunks, the language constructs' patterns, the relating language constructs, and the developer's decisions. Their results showed that: (i) 87% of conflicting chunks (automated analysis; 83% manual analysis) had all the information needed to resolve merge conflicts without writing any new code, (ii) 60% of merge conflicts involve multiple conflicting chunks which can depend on the project and, (iii) 14% to 16% of those chunks are dependent on other chunks.

3.5 Merge Conflict Resolution

Resolving merge conflicts are painful, particularly when changes diverge significantly [Bru11]. In additional, merge conflict resolution can be a tedious task and cause delays on the project production. As we aim at investigating factors that make merge conflicts harder to resolve in practice, the studies of this section are closer to our study. We found only two studies of merge conflict resolution of which one is an extension of the other.

Nelson et al. [Nel19] present insights into developers‘ process and perceptions on merge conflicts resolution. Nelson et al. [Nel19] is an evolution of McKee et al. [S M17] the main difference is that they did a semi-structured interview and examined barriers to approach merge conflicts resolution. To learn the effects and implications of software developers' processes and strategies, they answered six questions: (i) how do software developers manage merge conflicts?, (ii) how do software developers become aware of merge conflicts?, (iii) how do software developers plan for merge conflict resolutions?, (iv) how do software developers evaluate merge conflict resolutions?, (v) What difficulties do software developers experience when managing merge conflicts?, and (vi) how well do tools support developer's needs for managing merge conflicts?

To answer these questions they investigate nine factors: (i) complexity of conflicting lines of code, (ii) expertise in area of conflicting code, (iii) complexity of files with conflicts, (iv) number of conflicting lines of code, (v) time to resolve a conflict, (vi) atomicity of changesets in conflict, (vii) dependencies of conflicting code, (viii) number of files in the conflict, and (ix) non-functional changes in codebase.

As results, they found that developers rely on the code complexity of the conflicting lines and their knowledge in the area of the conflict as the top two factors when estimating the difficulty of a merge conflict resolution. They also found that most developers use the reactive process when observing merge conflicts i.e., 87.72% of those participants rely on version control systems (e.g. Git[2], SVN[3], CVS[4]), while 21.05% use continuous integration systems (e.g. Jenkins[5], Travis CI[6]). The developers rely on their expertise and the complexity of the conflicting code. In addition, they found that when the merge conflict resolution fails, developers generally rely on their knowledge to resolve a merge conflict but also seek help from other practitioners to resolve the conflict if they feel that their experience is not sufficient to resolve that conflict.

Our study complements these studies because we examine most of their factors that led the merge conflict resolution in the practice of collaborative software development environment. While Mckee et al. [S M17] and Nelson et al. [Nel19] surveyed developers

[2]https://git-scm.com/
[3]https://subversion.apache.org/
[4]https://www.nongnu.org/cvs/
[5]https://jenkins.io/
[6]https://travis-ci.org/

to find out factors that effects on the resolution of merge conflicts, our study investigates their factors and additional factors by analyzing time merge conflicts took to be resolved. Details of our methodology can be seen in Chapter 4, and a comparision of their results with our results are discussed in Chapter 6.

4 Methodology

This chapter presents our empirical study methodology, whose overall goal is to *investigate which factors do make conflicting merge scenarios harder to resolve in practice of the collaborative software development.* First, we describe our research question followed by explanations of how we selected subject projects. Then, we present our approach to retrieve contribution data. Finally, we explain how we answered the research question.

4.1 Research Question

Our discussion of the literature has shown how painful merge conflicts are for practitioners achieving the project objectives (see Chapter 2) and how researchers have been investigating merge conflicts over the years (see Chapter 3). Despite the number of studies investigating merge conflicts only few of them investigate how merge conflicts are solved and factors that make merge conflicts harder to resolve. Considering that knowing in practice which factors lead to a longer conflicting merge scenarios resolution time is useful for tool builders developing more useful tools to better support practitioners and, for researchers investigate opportunities for improving the state of art of merge conflict resolution, we investigate this topic in practice based on factors presented by previous studies [S M17] [Nel19]. Our study is guided by the following research question:

RQ: What factors does make conflicting merge scenarios harder to resolve in practice of collaborative software development?

This research question investigates factors that make a conflicting merge scenario harder to solve. To measure the difficult of resolving conflicting merge scenarios, we use the difference of time between the merge commit and the latest commit of the merged branches. To know which factors may influence the time of resolving merge conflicts, we defined a set of ten variables (e.g., the number of conflicting chunks, the number of developers involved, the size and complexity of the conflicting code) inspired on factors presented on related work [Nel19] [S M17]. As mentioned in the related work chapter, the main difference of our study and previous studies is that we rebuild thousands merge scenarios and look whether these ten variables influence the merge scenario resolution time while previous studies asked developers which factors (variables) make the merge conflict resolution harder.

Table 4.1 describes all factors we explore in this study divided into dependent and independent variables. In what follows, we explain the reasons of investigating these variables. To make our description short, we organize these variables into three groups: *time*, *variables directly related to merge conflicts*, and *variables indirectly related to the merge conflicts*.

Time. *#SecondsToMerge* captures how much time (in seconds) was taken to resolve the merge conflict. This is our dependent variable for that reason we give more details about how this variable measures difficulty. We think that this variable is a good indicator of difficulty because it is natural that trivial tasks (e.g., simple, short) will take less time than large and complex tasks. In addition, time has been used to investigate difficulty of executing tasks in usability methods [Thy09] and crowdsourcing [Fan18].

Table 4.1: Investigate Variables of Our Study

Measure	**Description**
	Dependent variable
#SecondsToMerge	The shortest time difference between the parent commits of the merged branches and the merge commit of the merge scenario
	Independent variables
#LoC	The sum of lines of code of the merge scenario
#ConfLoC	The sum of conflicting lines of code of the merge scenario
#Chunks	Number of chunks of the merge scenario
#ConfChunks	Number of conflicting chunks of the merge scenario
#Files	Number of files of the merge scenario
#ConfFiles	Number of conflicting files of the merge scenario
#Devs	Number of developers involved in code changes of the merge scenario
%IntegratorKnowledge	Percentage of files which exists conflicting chunks that the merge scenario integrator had committed before integrate branches of a merge scenario among all chunks of the merge scenario
%FormattingChanges	Percentage of formatting changes of conflicting chunks among all chunks changed in the merge scenario
CodeComplexity	Sum of the cyclomatic complexity of conflicting chunks of the merge scenario

Variables directly related to merge conflicts. This group contains the majority of variables investigated in this study since our goal is about finding factors that leads to longer merge conflict resolution time. Therefore, nothing more adequate than chose measures that directly measures the size, complexity, and knowledge on the conflicting code. *#ConfLoC*, *#ConfChunks*, *#ConfFiles*, *%IntegratorKnowledge*, *%FormattingChanges*, and *CodeComplexity* are the six variables that composes this group.

Variables indirectly related to merge conflicts. A prove that merge conflict resolution is not trivial is the number of studies investigating it in the whole life-cycle. Considering that the resolution of a merge conflict may depend on other code changes not in conflict, we opted to measure code changes not related to the merge conflict, but to the whole merge scenario. *#LoC*, *#Chunks*, *#Files*, and *#Devs* are the four variables that composes this group.

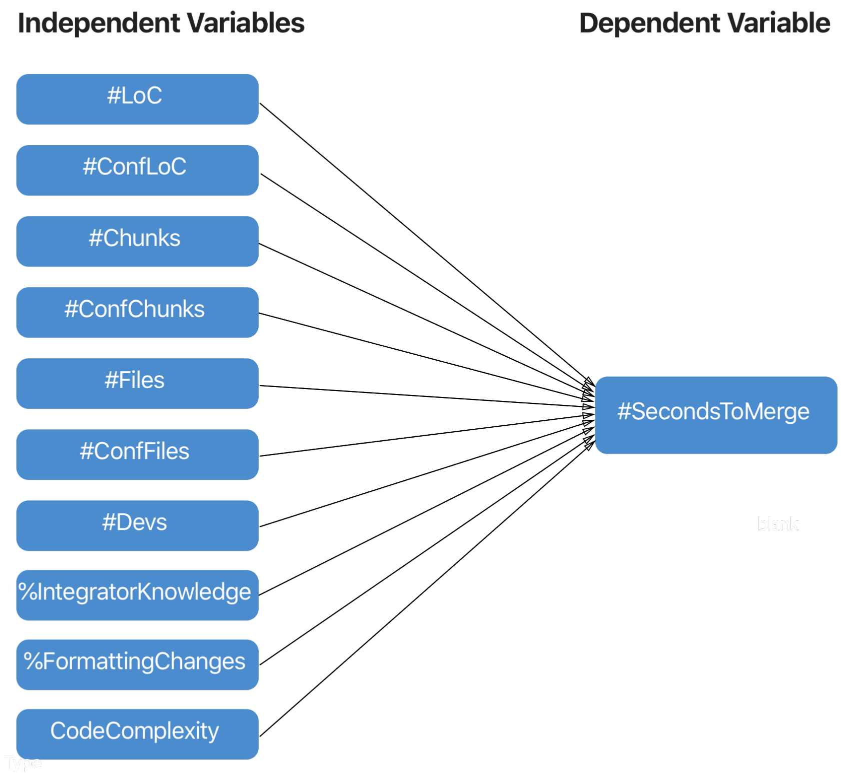

Figure 4.1: Investigated Relationship of Our Study

Figure 4.1 illustrates the relationship we investigate in our study by means of the variables presented in Table 4.1.

4.2 Subject Projects

Overall, we selected 31 subject projects from a variety of domains from the hosting platform GitHub. We chose to limit our analysis to Git repositories because it simplifies the identification of merge scenarios in retrospect. We restrict our selection of projects on GitHub because it is one of the most popular platforms to host repositories and have been investigated and used by several studies [Sto14][Lei13][Dab12][TDH14][GSB16][LLH16]. We selected the corpus as follows. First, we retrieved the 50 most popular projects on GitHub, as determined by the number of stars [BV18]. Then, we applied the

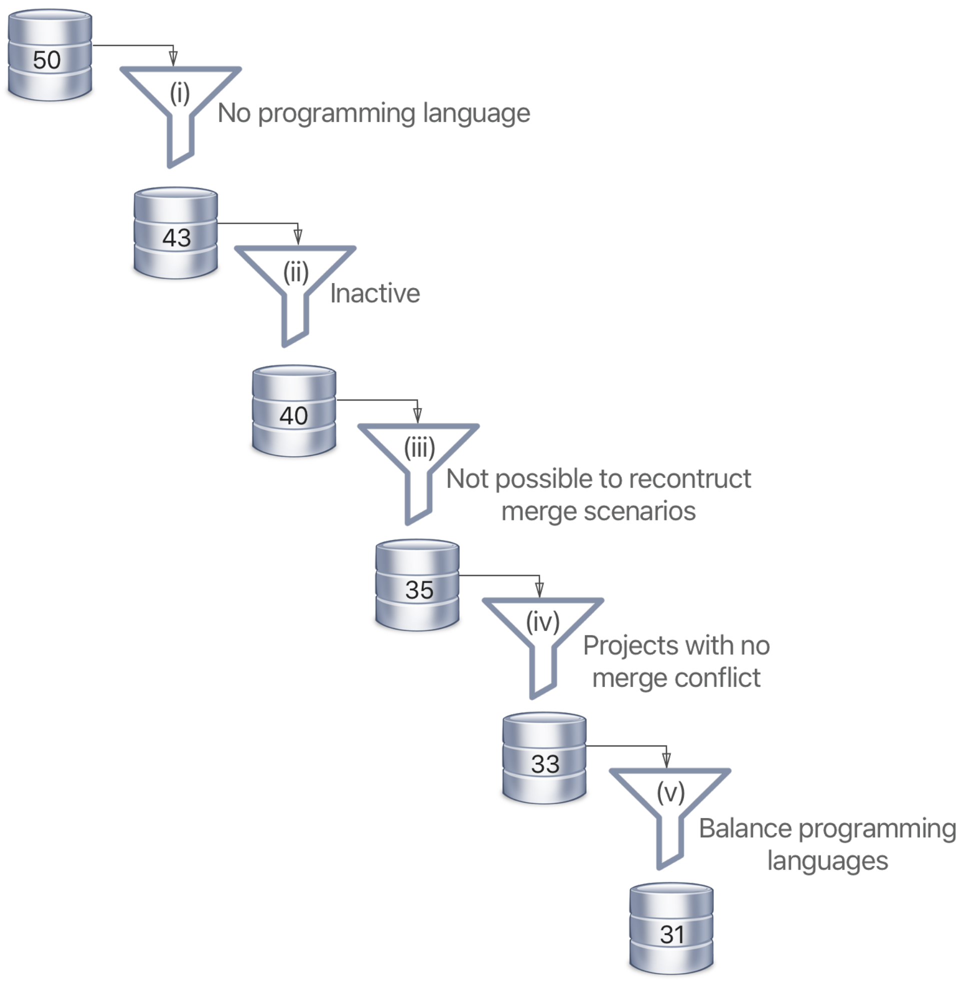
50
(i)
No programming language
43
(ii)
Inactive
40
(iii)
Not possible to recontruct merge scenarios
35
(iv)
Projects with no merge conflict
33
(v)
Balance programming languages
31

Figure 4.2: Projects Filter

following five filters: (i) projects that do not have a programming language classified as the main language (i.e., the main file extension), (ii) projects with less than two commits per month in the last six months, (iii) projects in which it was not possible to reconstruct most of the merge scenarios, (iv) projects with no merge conflicts and (v) the balance of the main programming language of the subject projects. Figure 4.2 presents the amount of projects remaining after each filter.

We created these filters inspired by Kalliamvakou et al. [Kal14]. These filters aim at selecting active projects in terms of code contributions with an active community and at increasing internal validity. For example, the second filter captures active community projects on GitHub. The third filter excludes projects such as kubernetes[1] and moby[2] because we considered that these projects do not mostly use the pull-based model (i.e., do not follow the three-way merge [GPD14]) and they could bias our analyses. Details of how we rebuild merge scenarios are provided in Section 4.3. As most of the popular projects are developed in JavaScript, in the fourth filter, we excluded two less popular JavaScripts projects ordered by the number of stars until they accounted for less than half of the subject projects. After applying all filters, we obtained 33 projects developed in 13 programming languages (i.e., a project can be developed using more than one programming language), such as JavaScript, CSS, and C++, containing around 45 766 merge scenarios that involve 1.2 million files changed, and 6.3 million chunks. Table 4.2 provides information and statistics of each subject project. More details, such as the subject project's URLs, are available on the supplementary website.

Some #MS and #CMS are blank in Table 4.2. This is due (i) seven project does not have any main programming language (i.e., we cannot get all the merge scenarios form that project), (ii) three projects are inactive, and (iii) not possible to reconstruct most of the merge scenarios of five projects.

4.3 Data Acquisition

Note that we have excluded merge scenarios that do not have a base commit (e.g., rebase, fast-forward, or squash integrations [S J16]) and we ignore binary files, because we cannot track changes from them, and we retrieve data only for merge scenarios that resulted on merge conflicts.

[1]https://github.com/kubernetes/kubernetes
[2]https://github.com/moby/moby

Table 4.2: Overview of the Subject Projects

id	Subject Project Name	Main Prog. Language	#Stars	#MS	#CMS	Excluded by filter
1	freeCodeCamp	JavaScript	302551	4499	81	
2	996.ICU	Rust	243370	1280	134	
3	vue	JavaScript	137975	140	5	
4	bootstrap	JavaScript	133212	4355	184	
5	react	JavaScript	128811	2486	11	
6	tensorflow	C++	127312	5831	-	(iii)
7	free-programming-books	No prog lang	122687	-	-	(i)
8	awesome	No prog lang	108144	-	-	(i)
9	You-Dont-Know-JS	No prog lang	101509	-	-	(i)
10	oh-my-zsh	Shell	87925	1579	5	
11	javascript	JavaScript	84945	311	0	(iv)
12	d3	JavaScript	84554	688	286	
13	gitignore	No prog lang	83537	-	-	(i)
14	developer-roadmap	No prog lang	80910	-	-	(i)
15	coding-interview-university	No prog lang	77388	-	-	(i)
16	react-native	JavaScript	76955	650	3	
17	vscode	TypeScript	75689	3857	20	
18	linux	C	74733	61720	-	(iii)
19	electron	C++	73088	4198	32	
20	create-react-app	JavaScript	67352	80	8	
21	awesome-python	Python	67037	452	26	
22	flutter	Dart	63213	1842	0	(iv)
23	system-design-primer	Python	62818	5	1	
24	CS-Notes	Java	61165	493	35	
25	node	JavaScript	61047	391	57	
26	Font-Awesome	JavaScript	59749	156	4	
27	angular.js	JavaScript	59509	36	6	
28	animate.css	CSS	59336	111	5	
29	axios	JavaScript	59120	188	5	
30	go	Go	57868	147	4	
31	public-apis	Python	56704	624	46	
32	moby	Go	53239	15808	-	(iii)
33	models	Python	52510	970	20	
34	kubernetes	Go	52454	36960	-	(iii)
35	laravel	PHP	52169	1455	14	
36	jquery	JavaScript	51510	250	1	
37	three.js	JavaScript	51325	6083	-	(iii)
38	youtube-dl	Python	50264	1381	79	
39	free-programming-books	No prog lang	49712	-	-	(i)
40	puppeteer	JavaScript	48965	-	-	(ii)
41	TypeScript	TypeScript	48760	7616	7	
42	webpack	JavaScript	48729	2269	86	
43	javascript-algorithms	JavaScript	48676	-	-	(ii)
44	atom	JavaScript	48675	4336	134	
45	redux	JavaScript	48489	675	9	
46	angular	TypeScript	47939	4	2	
47	swift	C++	47677	28342	-	(ii)
48	java-design-patterns	Java	47481	393	25	
49	material-ui	JavaScript	46700	2115	64	(v)
50	socket.io	JavaScript	46218	342	33	(v)

#Stars, #MS, and #CMS denote the number of GitHub stars, the number of merge scenarios, the number of conflicting merge scenarios we were able to compute.

We rebuilt merge scenarios from the subject projects since their creation until May 2019. Our strategy for merge scenario data acquisition consists of five steps. First, we clone a subject project's repository. Second, as merge commits can be identified in Git when the number of parent commits is greater than one, we identify merge scenarios by filtering commits with multiple parent commits. Third, for each merge commit, we retrieve a common ancestor for both parent commits (i.e., the base commit). Fourth, we (re)merge the parent commits and retrieve the measures for the variables in Table 4.1 by comparing the changes that occurred since the base commit until the merge commit. Finally, we store all data and repeat steps 3 to 5 for each merge scenario found in the step 2.

Figure 4.3 presents a class diagram containing all the information required for our study. As can be seen, for each *project* we build *merge scenarios.* Each merge scenario contains one or more *files* and each file contains one or more *chunks* of code. For each conflicting chunk, we get the changed code that resulted in the merge conflict, the changed code that resolved the merge conflict, and time difference between the parent commit (seen in Figure 4.3 as *merge_conflict_info).* Then, we compute variables for these classes and extract each one of them at merge scenario level. Following we describe how we compute these variables:

- ***#SecondsToMerge.*** We compute it by taking the shortest time difference between the parent commits of the merged branches and the merge commit of a merge scenario.
- ***#LoC.*** We calculate it by counting lines of code of all the chunks from the files that were involved in a merge scenario.
- ***#ConfLoC.*** We compute it by counting line of codes of all the chunks in conflict from the files that were involved in a merge scenario.

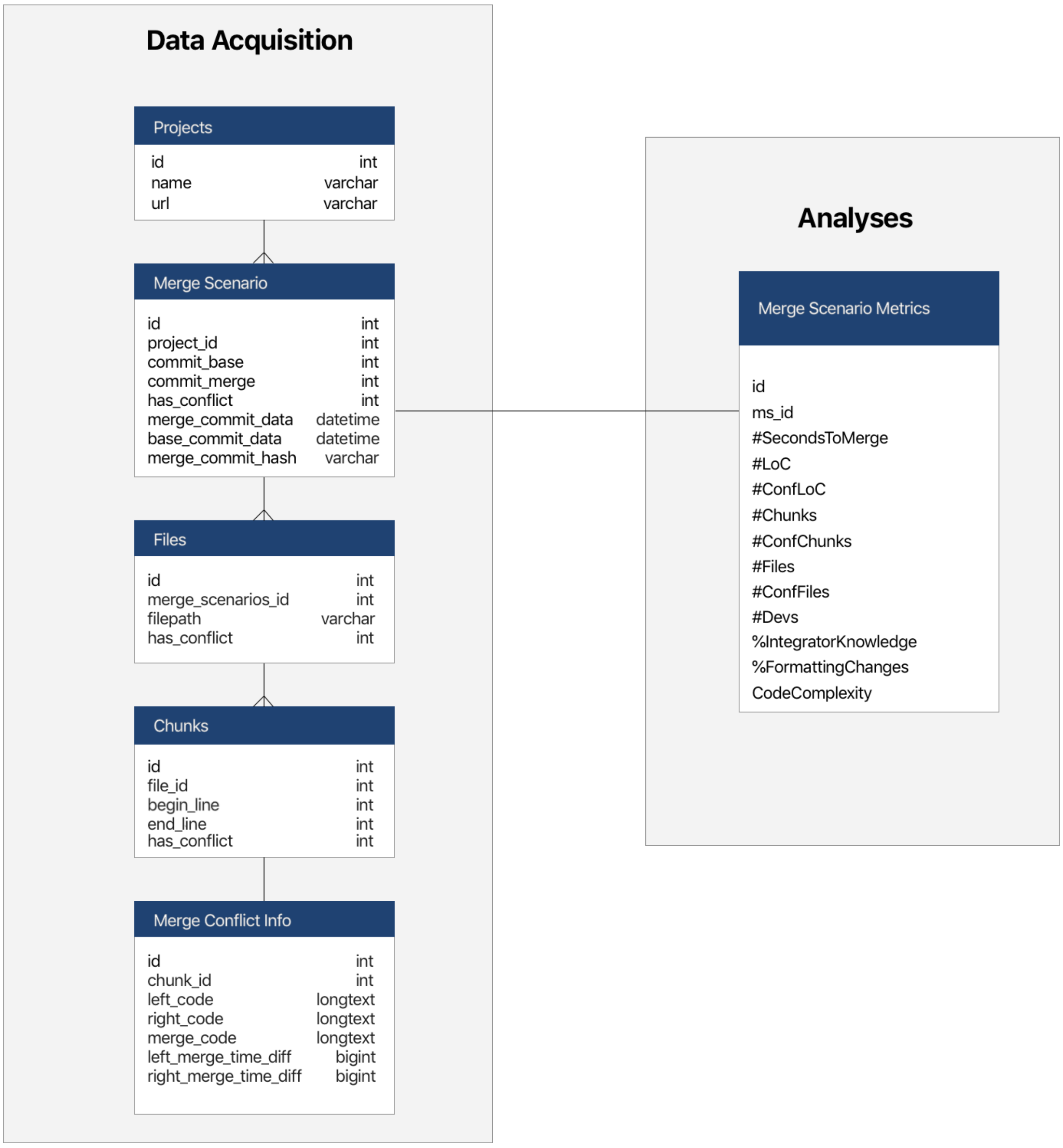

Figure 4.3: Class Diagram

- ***#Chunks.*** We compute it by counting all the pieces of code where it was added or modified in a merge scenarios
- ***#ConfChunks.*** We compute it by counting all the places where the code is added or modified and had conflict in a merge scenarios
- ***#Files.*** We calculate it by counting all the files added, removed or modified in a merge scenarios
- ***#ConfFiles.*** We calculate it by counting all the files added or modified and has conflict in a merge scenarios
- ***#Dev.*** We calculate it by counting all developers who committed to a chunk of code in each file involved in a merge scenario.
- ***%IntegratorKnowledge.*** We calculate it by checking if the developer had previously committed in the file before integrating the code in a merge conflict. We then take percentage dividing it by the number of merge conflicts in a merge scenario.
- ***%FormattingChanges.*** The percentage of conflicting chunks that had only formatting changes among all conflicting chunks in a merge scenario. We calculate it by analyzing chunks in a file resulted from formatting changes in a merge conflict. We then take percentage dividing it by number of chunks in a merge scenario.
- ***CodeComplexity.*** We compute McCabe cyclomatic complexity of each chunk in a merge conflict. We then take sum of all cyclomatic complexity of each chunk in a merge scenario.

Exemplifying the subject variables. Taking the merge scenarios exemplified in Figure 2.1, developers changed 10 lines of code (*#LoC*) in this merge scenario, of which

8 of them are in conflict (*#ConfLoC*). These lines of code changed 4 chunks *#Chunks*, in which 3 are in conflict (*#ConfChunks*). This chunks belongs to 2 files (File1 and File2), hence *#Files* equal to 2 and as the conflicts are in both files, it also has 2 conflicting files (*#ConfFiles*). Overall 4 developers (Dev A, Dev B, Dev C, and Dev D) committed to this merge scenario. As Dev D is the developer who solved the merge conflict and he had committed before in both files, we consider that this developer knew the code before lead with the merge conflicts. Therefore, Dev D had 100% of knowledge on the changed files (*%IntegratorKnowledge*). As Figure 2.1 is illustrative example, it is not possible calculating *%FormattingChanges*, *CodeComplexity*, and *#SecondsToMerge*. However, we can say that the right branch commit has the shortest distance to the merge commit when compared with the commit of the left branch. Hence, we would have gotten the difference among the right commit (commit hash: 1602adc) and the merge commit (commit hash: 718ec42)

Framework and Data Availability. Our analysis framework (Java and Python) and analyses scripts (R and Python) are open-source. All data necessary for replicating this study are stored in a MySQL database and replicated on spreadsheets (.csv files). All tools, links to the subject projects, and data are available at the supplementary Web site [Bro19].

4.4 Operationalization

The operationalization of our study to answer the research question consists of 5 steps. First, we check the *distribution of covariates among projects* to see if it may influence our results. Second, we compute the rank correlation of all covariables, using the Spearman rank-based correlation, which is invariant to any linear transformations of the covariates.

Third, we compute the principal component analysis to have a better visualization of the correlation among covariables and remove some of them of our regression model avoiding noise in our analysis. In other words, we want to have the highest variance with the minimum number of covariables. Fourth, we build and compute the multivariate regression model for predicting factors that make the merge conflict harder to resolve. To define the used model, we compare the model with all independent variables with a model with a simplified number of independent variables until we optimize the variance of our model. Finally, aware that independent variables change differently and, even with the results of the regression model they are not able to classify the most influencing factors, in the fifth step, we compute the Cohen's f^2 effect size. In what follows we provide an overview of the last four analysis we did since the first one consist only on plotting the values of each variable to see how they are distributed.

4.4.1 Rank Correlation

Spearman's correlation coefficient is represented as the Pearson correlation coefficient among the rank variables [LW03]. This way, for size n, the n raw scores $X_i, Y_i X_i, Y_i$ are converted to ranks $\mathrm{rg}\, X_i, \mathrm{rg}\, Y_i$, and r_s calculated from the following equation:

$$r_s = \rho_{rg_X, rg_Y} = \frac{cov(rg_X, rg_Y)}{\sigma_{rg_X}\sigma_{rg_Y}} \tag{4.1}$$

where ρ indicates the Pearson correlation coefficient but applied to the rank variables, $\mathrm{cov}(\mathrm{rg}_X, \mathrm{rg}_Y)$ is the covariance of the rank variables, and $\sigma_{\mathrm{rg}_X}\sigma_{\mathrm{rg}_X}$ and $\sigma_{\mathrm{rg}_Y}\sigma_{\mathrm{rg}_Y}$ are the standard deviations of the rank variables.

4.4.2 Principal Component Analysis (PCA)

Principal component analysis is a dimensionality-reduction method [Bha14], we use it to reduce the number of variables, but preserve the essential variables . Because small data sets are more comfortable to explore and visualize, we also analyze how variables are correlated with each other. The principal component analysis is based on a correlation matrix. To avoid bias due to distribution of the data, we use the rank transformed data in correlation estimation, as mentioned before. Based on the covariance matrix for the original variables in a matrix form is defined as:

$$Q \propto X^T X = W \Lambda W^T \tag{4.2}$$

The empirical covariance matrix within the principal components becomes

$$W^T Q W \propto W^T W \Lambda W^T W = \Lambda \tag{4.3}$$

where Λ is the diagonal matrix of eigenvalues $\lambda_{(k)}$ of $X^T X$. $\lambda_{(k)}$ is equivalent to the sum of the squares across the dataset associated with various component k, i.e. $\lambda_{(k)} = \Sigma_i \; t_k^2 i$ $= \Sigma_i \; (x_i w_k)^2$.

4.4.3 Multivariate Regression Model

Multiple linear regression is used to predict a dependent variable y (also called outcome variable) on the basis of multiple distinct independent variables x (also called predictor variables). The following equation represents a multiple linear regression model:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n + \varepsilon_0 \tag{4.4}$$

The β values are called the regression weights (or beta coefficients). They measure the association between the predictor variable and the outcome variable. β_j can be interpreted as the average effect on y of a one unit increase in X_j, holding all other predictors fixed [Jam13].

4.4.4 Effect Size

Effect size is a model of a kind of standardized average effect in the population across all the levels of the independent variable [Coh88]. We analyze which factor have more effect on the time taken to solve the merge conflict by using effect size analysis. We use Cohen's f^2, in the context of an ANOVA (Analysis of variance), which is one of many effect size measures like F-test and multiple regression. The f^2 effect size measure for sequential multiple regression and PLS (partial least squares path modeling) modeling is defined as:

$$f^2 = \frac{R^2_{AB} - R^2_A}{1 - R^2_{AB}} \tag{4.5}$$

where B is the variable of interest, A is the set of all other variables , R^2_{AB} the proportion of variance accounted for by A and B together (relative to a model with no regressors), and R^2_A is the proportion of variance accounted for by A (relative to a model with no regressors). By custom, $f^2 B$ effect sizes of 0.02, 0.15, and 0.35 are termed small, medium, and large, respectively [Coh88]. Cohen's $\hat{f}$ can be found for factorial analysis of variance (ANOVA) using:

$$\hat{f}_{\text{effect}} = \sqrt{(df_{\text{effect}}/N)(F_{\text{effect}} - 1)} \tag{4.6}$$

In a symmetrical design of ANOVA, the similar population parameter of f^2 is

$$\frac{SS(\mu_1, \mu_2, \ldots, \mu_K)}{K \times \sigma^2,}\frac{SS(\mu_1, \mu_2, \ldots, \mu_K)}{K \times \sigma^2,} \tag{4.7}$$

where μ_j indicates the population means within the jth group of the total K groups, the equivalent population standard deviations within each group. SS is the sum of squares in ANOVA.

5 Results

In this chapter, we present the results of our empirical study which goal is to present factors that make conflicting merge scenarios harder to resolve in practice. We structure this chapter according to the five steps presented in Section 4.4 plus a section to summarize our results.

5.1 Distribution of Covariates Among Projects Analysis

Figure 5.1 presents the distribution of each variable investigated in this study among the subject projects. Each variable is shown in a different boxplot. X-axis shows the project id, which an mapping of these ids can be found in Table 4.2. Y-axis shows the values of a target investigated variable. Hence eleven different plots can be found.

As can be seen in Figure 5.1, all variables have a similar behavior among projects, the only exception is *%FormattingChanges* variable since it is not so frequent for most of the projects. Besides, there are few outliers which are beyond the first and third quartile, most covariates seem to be sufficiently equally distributed among the subject projects. This is a statistically a good choice, as unequally distributed covariates could result in

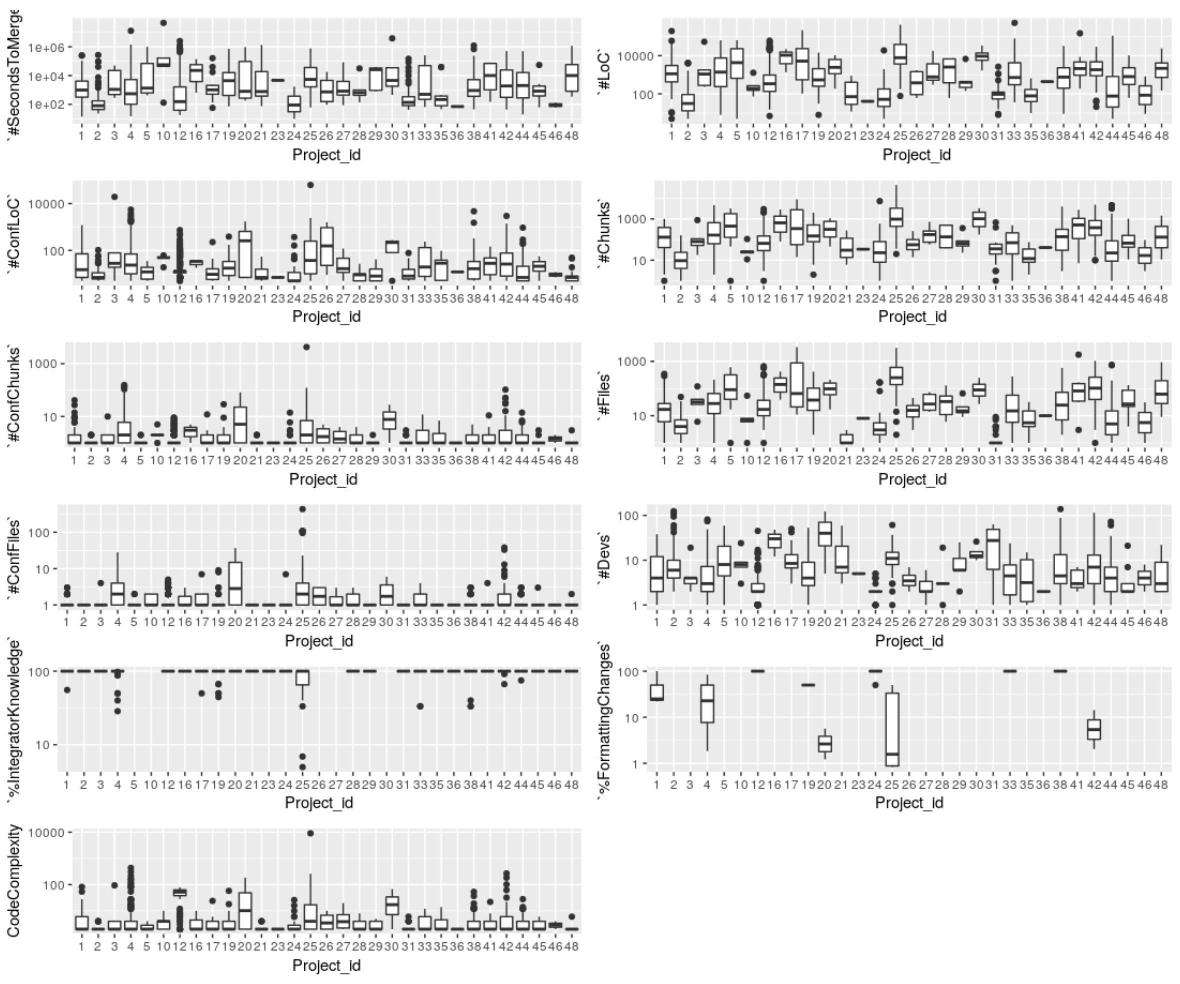

Figure 5.1: Distribution of the Covariates Among Projects

an erroneous interpretation. Therefore, in general, there is no project concentrating a behavior that may bias our analysis.

5.2 Rank Correlation of All Covariables Analysis

Figure 5.2 presents a matrix correlation among all covariables in analysis. As expected merge scenario size metrics (i.e., *#LoC*, *#Files*, and *#Chunks*) has a high correlation (above 0.8) among them. Merge conflict size metrics (*#ConfLoC*, *#ConfFiles*, and

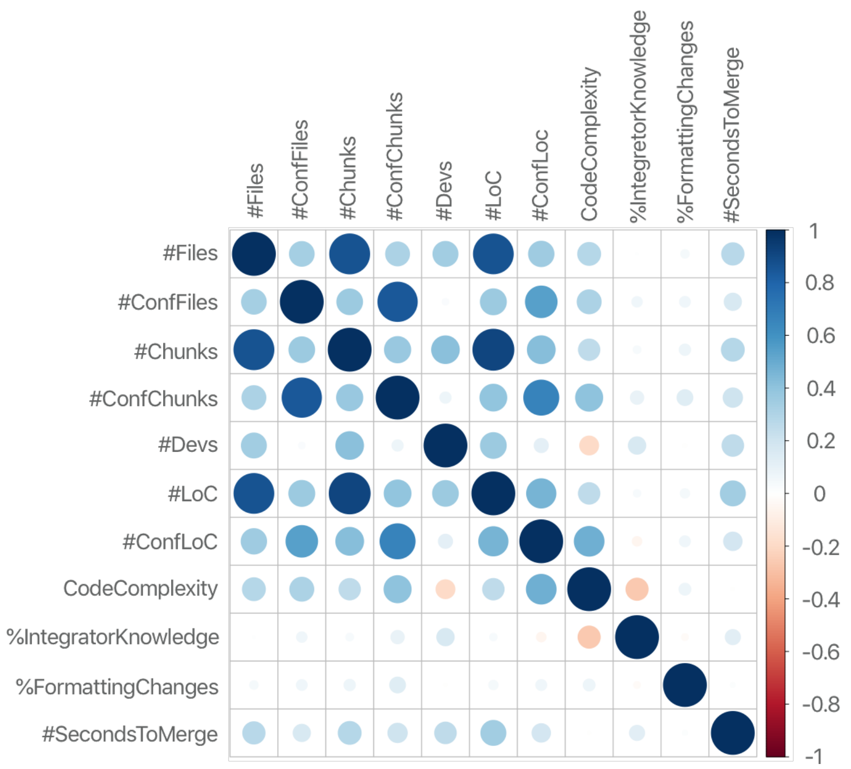

Figure 5.2: Rank Correlation of Covariables

#ConfChunks) also have a moderate to high correlation (above 0.5) among them. Interesting not all metrics related to the merge conflicts correlate, for instance, *%Integrator-Knowledge* and *%FormattingChanges* have a correlation coefficient smaller than 0.1 for most of the merge conflict. The two exceptions are in the case of *%IntegratorKnowledge* with a negative correlation coefficient with *CodeComplexity* equals to -0.269 and positive correlation coefficient with *#Devs* equals to 0.167. Both cases seem to be reasonable since once the integrator knows about the conflicting chunks, it does not matter the complexity, they do not have to understand the code, hence, the merge conflict resolution becomes faster. Regarding *#Devs*, more developers may add different code style and it will impact on the code understanding as well the merge conflict resolution.

We also give more attention to the correlation among our dependent variable and each independent variable. For short, the coefficient correlation is significant with a confidence interval of 95% for all independent variables except *#CodeComplexity* and *%FormattingChanges*. For the significant ones the correlation coefficient are 0.342, 0.188, 0.286, 0.207, 0.272, 0.164, 0.256, 0.127 for *#LoC*, *#ConfLoC*, *#Chunks*, *#ConfChunks*, *#Files*, *#ConfFiles*, *#Devs*, *%IntegratorKnowledge*, respectively. Therefore, the top three variables with highest correlation coefficient are the three variables that measures the merge scenario size (*#LoC*, *#Chunks*, and *#Files*). In the next sections we investigate whether this correlation remains true to be able to say that these variables influence the most with merge conflict resolution time.

5.3 Principal Component Analysis

Figure 5.3 shows the two dimensional output from the principal component analysis for each communication approach, which covers 54.3% (37.7% + 16.6%) of the total variance of our data. The arrows represent the weights of each variable in the respective principal component and its color represents the square cosine (cos2). The square cosine represents the share of original variation in the variable that is retained in the dimensionality reduction. The longer the arrow, the larger is the share of a variable's variance. Arrows pointing to the same direction have a large share of common variance and can be assumed to belong to the same group.

Figure 5.3 analysis suggests to classify the confounding variables into four groups (size, conflicting size, social activity, and type of change). The arrows representing *#Chunks*, *#Files*, and *#LoC* point in to the same direction; they represent the size of a merge scenario. Pointing to another direction, *#ConfLoC*, *#ConfFiles*, and *#ConfChunks* represent the conflicting size. The two factors that involve social assets (*#Devs* and *%IntegratorKnowledge*) point to a close direction, hence, we call them social activity.

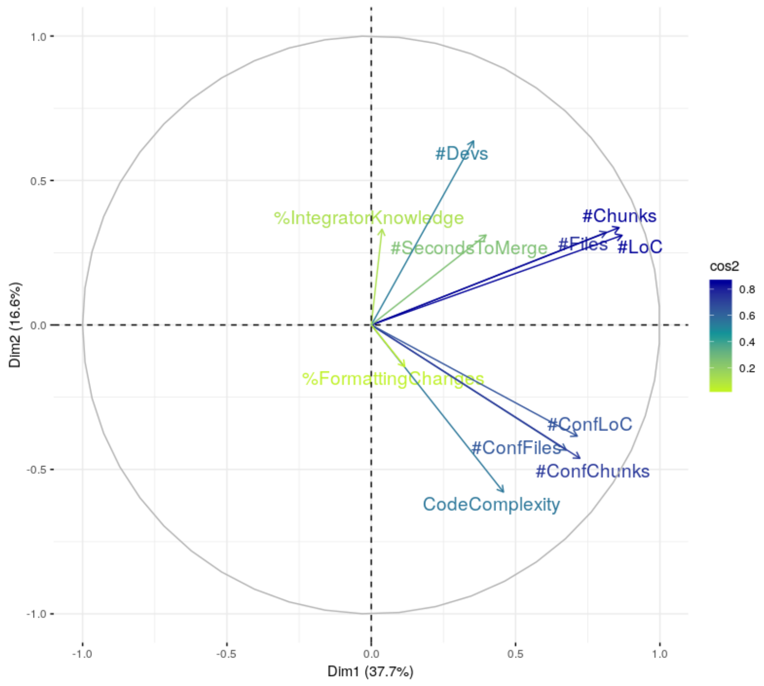

Figure 5.3: Pricipal Component Analysis of Our Covariables

The factors *CodeComplexity* and *%FormattingChanges* composes the fourth group which we named type of change since it represents formatting and the cyclomatic complexity of the conflicting code.

From the previous rank correlation analysis, it was already possible to see that not all variables that are directly relationed with merge conflicts belong together. The principal component analysis support us to easily group them. These four groups will be useful to check the variance of our regression model with a model with all covariables.

5.4 Multivariate Regression Model Analysis

As mentioned, our multivariate regression model analysis consists of building the model with the highest variance possible for our scenario. Hence, we first build two models,

Table 5.1: Correlation Coefficients Between Subject Measures

Measure	**Correlation Coefficient**	**p-value**
#LoC	0.4741	3.4e-14
#Chunks	-0.2406	0.00016
#ConfChunks	0.1473	5.4e-05
#Devs	0.1539	2.8e-07
CodeComplexity	-0.0724	0.03039
%IntegratorKnowledge	0.0773	0.01456

the former with all independent variables and the later with the four groups found by the principal component analysis. Performing the analysis of variance, we saw that the model with all independent variables has a higher variance than the other one. Hence, we build a third intermediate model with highest variance. This model is composed by the dependent variable *#SecondsToMerge* and 6 independent variables (i.e., *#Loc*, *#Chunks*, *#ConfChunks*, *#Devs*, *CodeComplexity*, and *%IntegratorKnowledge*). This model contains variables from all groups, excluded the suspicious variable found in the distribution of covariable among projects (*%FormattingChanges*), and excluded independent variables with high correlation coefficient with other variables in the model such as *#Files* that strongly correlate with *#Chunks*. These three points, in addition with the variance let us to think this is the best model we can build with the covariables investigated. The other models as well as their correlation coefficients can be found in our supplementary website [Bro19].

Table 5.1 presents the correlation coefficients obtained by our regression model as well as their p-value. Note that all independent variables are significant for a confidence interval of 95%. The multiple R-squared and the p-value of our model are equal to 0.166 and $< 2e - 16$, respectively. Despite the multiple R-squared is small, what is not always bad, it has a very small p-value which means that our model is significant for 99.9% confidence interval.

The most interesting point analyzing Table 5.1 is that *#LoC*, *#ConfLoC*, and *#Devs* has a positive correlation coefficient. Hence, if these variables increase the time to resolve merge conflicts also increases. On the other hand, *#Chunks* and *CodeComplexity* has a negative correlation coefficient is negative. Hence, increasing code complexity of conflicting chunks *CodeComplexity* and the number of chunks (#Chunks) leads to less time to resolve merge conflicts. Note that two out of the three variables with the highest correlation coefficient with the time to resolve the merge conflict found in the rank correlation analysis remains with the highest correlation in the multivariante regression model analysis. As #Files was removed for our best fit model it is supposed that this variable does not appear in this analysis. As mentioned, #LoC and #Chunks already represent the variance that #Files could represent to our model.

A simple way to interpret the results of or regression model is described as follows. For a given independent variable (predictor), the coefficient β can be interpreted as the average effect on $\hat{y}$ of a one unit increase in predictor, holding all other predictors fixed. For example, for a fixed amount of *#ConfLoC*, *#Chunks*, *#Devs*, *CodeComplexity*, and *%IntegratorKnowledge*, adding 100 LoC in the merge scenario leads to an increase in time by approximately $0.4741 * 100 = 474$ seconds or 8 minutes to solve the merge conflicts, on average. As another example, for a fixed amount of #LoC, #ConfLoC, #Devs, CodeComplexity, and %IntegratorKnowledge, adding 100 chunks in the merge scenario leads to an decrease in time by approximately 240 seconds or 4 minutes. We discuss this controversial counter intuitive in Chapter 6.

Table 5.2: Effect Size Analysis

Measure	**Cohen's f^2**
#Chunks	0.3136
#LoC	0.2097
#Devs	0.1773
#ConfChunks	0.1168
CodeComplexity	0.0838
%IntegratorKnowledge	0.0671

5.5 Effect Size Analysis

Finally to answer our research question and be able to classify independent variables for the one the influence *#SecondsToMerge* most, we performed an effect size analysis. As describe in Section 4.4, we choose Cohen's f^2 effect size since it is more adequate when using multivariate regression models. Table 5.2 presents the results of our effect size analysis ordered by the highest effect to the lowest effect. Note that as *#ConfLoC*, *#Files*, *#ConfFiles*, *%FormattingChanges* variables do not compose the effect size analysis since they were removed from the multivariate regression analysis.

Looking at Table 5.2 we see that the effect size of *#Chunks*, *#LoC*, *#Devs*, *#ConfChunks*, *CodeComplexity*, and *%IntegratorKnowledge* are 0.3136, 0.2097, 0.1773, 0.1168, 0.0838, and 0.0671, respectively. Following Cohen's classification, *#Chunks*, *#LoC*, and *#Devs* have a medium effect size on merge conflict resolution time and the bottom three variables (*#ConfChunks*, *CodeComplexity*, and *%IntegratorKnowledge*) have a small effect size. Note that, despite #Chunks has a medium effect size its Cohen's f^2 is close to the large effect size group.

Surprisingly the three variables with highest effect size are not directly related with the merge conflicts (i.e., they are related to the merge scenario changes). In addition, to that looking at the regression analysis, we can see that the number of chunks has a negative correlation coefficient while the other two variables indirectly related with merge conflicts (*#LoC* and *#Devs*) has a positive coefficient. Hence, we assume that more chunks in the merge scenario leads to shorter merge conflict resolution time with a medium effect size. On the other hand, more lines of code and developers lead to more time to resolve the merge conflicts with a medium effect size.

5.6 Research Question Answer Summary

Our *distribution of covariates among projects analysis* indicate that there is no bias evaluating all merge scenarios together since variables have a similar distribution among projects and *%FormattingChanges* may be not a good metric for our analysis since it is absent in most projects. Our *rank correlation analysis* enforced by the *principal component analysis* shows that some variables are strongly correlated. With that, they should be grouped to remove noise from our multivariate regression model and keep high variance. Our *multivariate regression model analysis* shows that *#LoC*, *#Chunks*, *#ConfChunks*, *#Devs*, *CodeComplexity* and *%IntegratorKnowledge* are correlated with the *#SecondsToMerge* with a confidence interval of 95% of significance. #Chunks and *CodeComplexity* have a negative influence while the other four variables have a positive influence. Our *effect size analysis* reveals that ***#Chunks*, *#LoC*, and *#Devs* have a medium effect size on the merge conflict resolution time and *#ConfChunks*, *CodeComplexity*, and *%IntegratorKnowledge* have a small effect size on the merge conflict resolution time.**

6 Discussion

This chapter discusses the results of the analysis of factors that make merge conflict resolution harder in practice. Furthermore, it presents the threat to validity of the study. The chapter begins with comparing our results with previous studies. Then it ends with the discussion of the threats to validity. This chapters discusses reflections on upon our study (6.1), compares our results with previous studies (6.2) and present limitation and threats to validity to our study.

6.1 Reflection Upon Our Results

Why do we investigate the difficulty to resolve conflicting merge scenarios instead of the difficulty to resolve each merge conflict separated? We investigate the difficulty to resolve conflicting merge scenarios instead of the difficulty to resolve each merge conflict separated because when the merge conflicts occur, the integrator should resolves all the merge conflicts present in the merge scenario. Hence, as some of them depend on other merge conflicts and might be larger or more complex than others it would be hard to estimate how much time the integrator took to solve each specific merge conflict. This approach of looking at the merge scenario changes and all conflicting merge scenarios are also used by related work [S M17] [Nel19] [Ghi18].

Why formatting changes are common in some software projects and absent in the majority of the subject projects? To investigate this question we looked at four measures: *domain*, *popularity*, *programming language*, and *rules to contribute to the project*. In addition, we use two merge scenario metrics to obtain an overview of formatting changes. *#CMS* and *#CMS100%* which stands for the number of conflicting merge scenarios with one or more formatting changes and the number of conflicting merge scenarios with 100% of formatting changes. Table 6.1 presents an overview of these measures for the subject projects containing formatting changes. Regarding the domain, we can see that three projects were classified to framework, three to tools, two to learning, and two to library domain. Learning domain has *#CMS* and *#CMS100%* equal to 11 and 8, respectively. Even that this domain has only two projects, they called our attention and we looked at deep into them. Looking at them, we could perceive that software projects in the learning domain consist of projects with theoretical content as well as practical exercises. Hence, a great amount of no programming language code can be found which may lead to merge conflicts given formatting changes. Regarding popularity, we use the project id from Table 4.2. In this table they were order by popularity. Considering it, we cannot see a notable difference since the most popular project and also the 42th most popular project belongs to this list. Regarding programming language, we can see that 6 out of the 10 projects that has merge scenarios with formatting changes use JavaScript. As our subject projects consist of almost half of JavaScript projects, it is hard to conclude that JavaScript is a programming language that normally involves formatting changes. Regarding the rules to contribute to the project, *CS-Notes* project with no rule to contribute to project defined has 8 and 7 conflicting merge scenarios with formatting changes and with 100% of formatting changes, respectively. As the other two projects without rules to contribute to the project do not have many formatting changes, it is hard to draw some conclusion. However, as stated by previous

Table 6.1: An Overview of the Projects with Formatting Changes

Id	**Project Name**	**Domain**	**Programming Languages**	**#CMS**	**#CMS100%**	**Rules to contribute to the project**
1	freeCodeCamp	Learning	JavaScript	3	1	Yes, well defined
4	bootstrap	Framework	JavaScript	1	0	Yes, well defined
12	d3	Library	JavaScript	1	1	No
19	electron	Framework	C++	5	0	Yes, well defined
20	create-react-app	Tool	JavaScript	2	0	Yes, well defined
24	CS-Notes	Learning	Java	8	7	No
25	node	Framework	JavaScript	1	0	Yes, well defined
33	Models	Library	Python	1	1	Yes, but not very clear
38	youtube-dl	Tool	Python	1	1	No
42	webpack	Tool	JavaScript	5	0	Yes, well defined

studies [ALL12] many merge conflicts could not half existed by avoiding formatting and syntactic changes. With all, looking at the measures together, we can see that learning domain and not defining rules to contribute to the project may increase the number of conflicting merge scenarios with formatting changes.

Do integrators normally know the conflicting merge scenario that they are going to integrate? Are these merge scenarios more complicated than conflicting merge scenarios integrated by integrators without knowledge on the files in conflict? To answer these two questions, we create a subset with only merge scenarios that the integrator had some knowledge and compared with the merge scenarios that the integrator had no knowledge. As result, in 38.62% of the conflicting merge scenarios the integrator had some knowledge on the files in conflict (516 conflicting merge scenarios out of 1336). In addition, in 35.63% of the conflicting merge scenarios the integrator knew all the files in conflict (conflicting merge scenarios 476 out of 1336). This is an interesting finding because in more then one than one third of the conflicting merge scenarios the integrator already knew something about the files that she integrated. As our regression model does not present a negative correlation coefficient for this metric with the *#SecondsToMerge*, we did a further analysis to see whether the merge scenarios that they integrated were larger, more complex, or with more developers involved.

Table 6.2 shows the mean and average of the two datasets mentioned including all investigated variables in this study, except *%FormattingChanges* since this metric is in percentage. As we can see, integrators that had knowledge on the merge scenario take less time on average than the ones that do not have any knowledge, however, the median is greater for them. In other words, when integrators know about the files that they are merging, they are fast, however, for half of the merge scenarios that they integrate they took more time. It should be given the fact that developers normally ask for helping when doing complicated tasks [Nel19]. For that reason waiting for some support on the merge conflict resolution make it longer. Looking at the other metrics we can see that integrators with previous knowledge on the modified files normally integrate larger merge scenarios (i.e., *#LoC*, *#Chunks*, and *#Files*), with more developers involved (*#Devs*), and larger number of conflicting files (*#ConfFiles*). However, integrators without previous knowledge normally resolve conflicting merge scenarios with larger *#ConfLoC*, *#ConfChunks*, and *CodeComplexity*.

Why merge scenario size metrics are more correlated with the merge conflict resolution time than merge conflict size metrics? Solving merge conflicts is not something trivial, for that reasons we have so many studies investigating it, and some with unexpected results [Les18]. One may say that it depends on the type of the conflict [ABC18], others will say that it depends on the language constructs that composes the conflict [Ghi18]. What none may question is that resolving merge conflicts and avoid unexpected behavior as well bugs and passing in the tests do not involve look only at the conflicting code. As we can see in our data, normally the conflicting code is only a fraction of the changes in the merge scenario, however, if the conflicting code depends of many not conflicting code the integrator have to understand the non conflicting code.

Table 6.2: Comparision Between the Dataset which Integrators Had Knowledge before Resolving the Merge Conflicts and the Dataset which Integrators Had No Knowledge before Resolving the Merge Conflicts

Dataset	**Statistics**	**Measures**								
		#SecondsToMerge	**#LoC**	**#ConfLoC**	**#Chunks**	**#ConfChunks**	**#Files**	**#ConfFiles**	**#Devs**	**CodeComplexity**
Had Knowledge	mean	36943	7509.4	126.9	648.5	4.7	115.0	3.1	10.0	8.9
	median	773	596	12	87	1	18	1	5	2
Did not have Knowledge	mean	109409.0	5557.1	146.1	376.6	8.0	62.1	2.2	8.0	33.3
	median	364	359	12	67	1	15	1	3	4

Therefore, our answer to the question is regarding the dependencies among changed code. Integrators have to understand what changed in the merge scenario to only then decide the best option in solving each specific merge conflict.

Why does the number of chunks have a negative correlation coefficient with the number of seconds to resolve conflicting merge scenarios? The answer of the previous discussion point is also related to this question. So, arguments from the previous point remain true for this point. However, in the case of the number of chunks, we agree with Ghioto et al., [Ghi18] that the language constructors will have a fundamental role on the merge conflict resolution. Looking at our analysis, we can see that the impact of increasing the number of chunks will be greater than increasing lines of code. The reasonable explanation is that having small chunks make the code understanding fast, and in many cases, when there is no dependence, the integrator can only ignore that chunk. On the other hand, when the chunk is large many functionality is there and it increases also the dependence with other pieces of code.

6.2 Comparing Our Results with Previous Studies

In this section we compare our findings on merge conflict resolution with related work. As it is hard to compare values, we only provide a descriptive answer for each of them showing when they agree with each other or disagree with each other.

Table 6.3 shows the comparison of our results with previous studies. As we can see, 13 factors are explored by previous studies and our study explore 3 new factors. By comparing our results with Nelson et al. [Nel19], we see that complexity of conflicting lines of code (*CodeComplexity*) and non-functional changes (*%FormatingChanges*) are factors that make the merge conflict resolution harder in their study. However, in our study we

Table 6.3: Comparison of Our Results with Previous Studies

Factors	**Nelson**	**Ghiotto**	**Our Study**
Complexity of conflicting lines of code	V	-	X
Expertise in area of conflicting code	V	-	V
Complexity of files with conflicts	V	-	-
Number of conflicting lines of code	V	-	V
Time to resolve a conflict	V	-	-
Atomicity of changesets in conflict	V	-	-
Dependencies of conflicting code	V	-	-
Number of files in the conflict	V	-	V
Non-functional changes in codebase	V	-	X
Number of chunks	-	-	X
Number of lines of code	-	V	V
Number of developers	-	-	V
Number of files	-	-	V
Number of conflicted chunks	-	V	V
Language constructs involved in conflicting chunks	-	V	-
Language constructs involving multiple conflicting chunks	-	V	-

Nelson = Nelson et al. [Nel19] study, Ghiotto = Ghiotto et al. [Ghi18] study

"V" means that the factor makes merge conflict resolution harder.

"X" means that the factor makes the merge conflict resolution easier.

"-" the factor is not explored by the study.

found that code complexity and non-functional changes has a negative effect on merge conflict resolution. Comparing our results with Ghiotto et al. [Ghi18] study, we agree on the two common factors that we both explore. In other words, both studies agree that number of lines of code (*#LoC*) and number of conflicting chunks (*#ConfChunks*) make the merge conflict resolution harder.

6.3 Threats To Validity

In this section, we discuss limitations and internal and external threats to the validity of our study. We try to eliminate threats when possible and decrease the effect when the elimination of threats was not possible.

6.3.1 Internal Validity

There are basically three main factors for internal validity. First, we are not able to measure the experience of the integrator. This is a limitation of our study since more experience developers may need less time to solve the same merge conflict than less experience developers.

Second, despite our analysis is not dependent of programming language, our script to calculate the complexity of the code supports a limited number of programming languages. Hence, the study does not generalize to other groups or open source software projects written in programming languages that out script does not support. In our supplementary website, we provide details which programming languages the analysis script covers.

Third, we are not able to capture unexpected events that happened on the merge conflict resolution. We believe that it does not influence our results because it would change all the analyzed variables and not only some of them. However, we could not check if these merge scenarios are outliers without asking developers.

6.3.2 External Validity

External validity is threatened mainly by four factors. First, our restriction to Git and GitHub as platform as well as to the pull-based model. Generalizability to other platforms, projects, and development model is limited. This limitation of the sample was necessary to reduce the influence of confounds, increasing internal validity, though [Sie14]. While more research is needed to generalize to other version control systems and development models, we are confident that we selected and analyzed a practically relevant platform and a substantial number of software projects from various domains, programming languages, longevity, size, and coordination practices. In addition, our filters applied during subject project selection guarantee, for instance, that we sampled projects that actively use GitHub as a communication tool and that we do not let a single programming language dominate our dataset (see Section 4.2).

Second, developers may use informal work practices, awareness-tools, or prediction strategies (e.g., continuous integration and rebase) that we are not able to measure. To minimize this threat, we manually looked at 50 issues randomly selected of each subject project searching for terms that point to such practices and tools, but we did not find any indication. One may also claim that rebased scenarios bias our analysis, however, since the commit(s) that a developer wants to integrate into another branch will be added on the top of the branch. it will make the repository's history linear avoiding merge conflicts. In addition, as mentioned in Sections 4.2–4.3, the rebase scenarios that damage the repository's history and so we are not able to retrieve the common ancestor of two (parent) commits were excluded from our analysis. Considering that our research is only about the pull-based model (i.e., three-way merge), together with the previous actions, there is no bias.

Third, we are not able to retrieve information from binary files, hence, we may miss a piece of information from some merge scenarios. Unfortunately we cannot do anything about that, however, the number of binary files is normally small in software projects.

Fourth, the need of triangulation through interview data. Interviewing developers could make our analyses and findings more reliable, however, as some our results intrigued us, we believe this would also happen to other developers. To mitigate this threat, we provided triangulation through observational data for every topic that deals with counter-intuitive findings, as we discussed in Sections 6.1.

7 Final Remarks

In the collaborative software development developers work simultaneously in a project addressing different tasks through version control systems. Simultaneously contributions to a common code base may introduce problems during integration, often manifesting as merge conflicts. Merge conflicts occur when one or multiple developers change the same chunk of code in different branches. There are many studies investigating merge conflicts, however, only few of them investigated the merge conflict resolution and none of them has investigated merge conflict resolution in practice. However, these factors are not investigated in practice.

In this study, we present factors that make conflicting merge scenarios harder to resolve in practice. To achieve our goal, we extract merge scenario information from 31 GitHub projects chosen based on their popularity, developed in 13 different programming languages containing around 45 766 merge scenarios involving 1.2 million files and 6.3 million chunks. We investigate variables directly related to merge conflict (i.e., *#ConfLoC*, *#ConfChunks*, *#ConfFiles*, *%IntegratorKnowledge*, *%FormattingChanges*, and *CodeComplexity*) and variables indirectly related to merge conflict (i.e., *#LoC*, *#Chunks*, *#Files*, and *#Devs*). For operationalization of our study, we first check the distribution of covariates among projects to see if it may influence our results. Second, we compute the rank correlation of all covariables using the Spearman rank-based correlation. Third, we compute principal component analysis to have a better visualization of the correlation among covariables and remove some of them to reduce noise in our analysis. Fourth, we build and compute the multivariate regression model for predicting factors that influence the merge conflict resolution time. Finally, we compute Cohen's f^2 effect size to measure the influence of the investigated factors on the merge conflict resolution time.

As result of our *rank correlation analysis* together with our *principal component analysis*, we found a strong correlation among merge conflict size metrics (i.e., *#LoC*, *#Files*, and *#Chunks*), medium correlation among merge conflict size metrics (i.e., *#ConfLoC*, *#ConfFiles*, and *#ConfChunks*), and small correlation among *%IntegratorKnowledge* and *%FormatingChanges* with the merge conflict resolution time metric (*#SecondsToMerge*). The two exceptions are in the case of *%IntegratorKnowledge* with a negative correlation coefficient with *CodeComplexity* and positive correlation coefficient with *#Devs*. By using *multivariate regression analysis*, we found that increasing *#LoC*, *#ConfLoC*, *#Devs*, and *%IntegratorKnowledge*, the time to resolve the merge conflicts also increases. However, when increasing *#Chunks* and *CodeComplexity*, the time to resolve the merge conflicts decreases. Our *effect size analysis* indicates that the 6 factors that make conflicting merge scenarios to resolve in practice are: *#Chunks*, *#LoC*, *#Devs*, *#ConfChunks*, *CodeComplexity* and *%IntegratorKnowledge* ordered by the effect size.

Reflecting upon our study we investigated explanations for some unexpected results. These explanations show the reasons of why some projects are prone to have formatting changes, why integrators that had knowledge on changed files before face with the merge conflicts need more time to solve merge conflicts than integrators that did not had previous knowledge on the conflicting code, and why merge scenario size metrics have a stronger correlation with the merge conflict resolution time than merge conflict size metrics.

As future work, we suggest a study that involves more variables to understand their influence on the merge conflict resolution time. These variables include factors that we are not able to measure for instance the dependencies among files, chunks, conflicting files, and conflicting chunks. We also suggest interviewing developers asking them about specific outliers merge scenarios. This investigation would explain strange values of the factors that we analyzed in this study.

Bibliography

[ABC18] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. “Understanding Semi-structured Merge Conflict Characteristics in Open-source Java Projects”. In: *Empirical Softw. Engg.* 23.4 (Aug. 2018), pp. 2051–2085. ISSN: 1382-3256. DOI: `10.1007/s10664-017-9586-1`. URL: `https://doi.org/10.1007/s10664-017-9586-1` (cit. on pp. 9, 20, 51).

[AS09] Brian de Alwis and Jonathan Sillito. “Why Are Software Projects Moving From Centralized to Decentralized Version Control Systems?” In: Jan. 2009, pp. 36–39. ISBN: 978-1-4244-3712-2. DOI: `10.1109/CHASE.2009.5071408` (cit. on p. 13).

[Ape11] Apel, Sven and Liebig, Jörg and Brandl, Benjamin and Lengauer, Christian and Kästner, Christian. “Semistructured Merge: Rethinking Merge in Revision Control Systems”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.* ESEC/FSE ’11. Szeged, Hungary: ACM, 2011, pp. 190–200. ISBN: 978-1-4503-0443-6. DOI: `10.1145/2025113.2025141`. URL: `http://doi.acm.org/10.1145/2025113.2025141` (cit. on pp. 9, 18).

[ALL12] Sven Apel, Olaf Lessenich, and Christian Lengauer. “Structured Merge with Auto-tuning: Balancing Precision and Performance”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, 2012, pp. 120–129. ISBN: 978-1-4503-1204-2. DOI: 10.1145/2351676.2351694. URL: http://doi.acm.org/10.1145/2351676.2351694 (cit. on pp. 9, 17, 18, 50).

[Bha14] Sanjiv Bhadauria. “Introduction to Principal Component Analysis in Applied Research”. In: *New Man International Journal of Multidisciplinary Studies* Vol.1 (Dec. 2014), pp. 67–75 (cit. on p. 36).

[Bie07] Biehl, Jacob T. and Czerwinski, Mary and Czerwinski, Mary and Smith, Greg and Robertson, George G. “FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’07. San Jose, California, USA: ACM, 2007, pp. 1313–1322. ISBN: 978-1-59593-593-9. DOI: 10.1145/1240624.1240823. URL: http://doi.acm.org/10.1145/1240624.1240823 (cit. on pp. 9, 19).

[BV18] Hudson Borges and Marco Valente. “Whats in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform”. In: *Journal of Systems and Software* 146 (Sept. 2018), pp. 112–129. DOI: 10.1016/j.jss.2018.09.016 (cit. on p. 27).

[Bro19] Muhammad Zohaib Brohi. “Software Practitioners Perspective on Merge Conflicts and Resolution”. In: (2019). URL: https://sites.google.com/view/perspective-on-merge-conflicts (cit. on pp. 11, 34, 44).

[Bru11] Brun, Yuriy and Holmes, Reid and Ernst, Michael D. and Notkin, David. “Proactive Detection of Collaboration Conflicts”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Founda-*

tions of Software Engineering. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 168–178. ISBN: 978-1-4503-0443-6. DOI: `10.1145/2025113.2025139`. URL: `http://doi.acm.org/10.1145/2025113.2025139` (cit. on pp. 9, 18, 21).

[Coh88] J Cohen. "Statistical power analysis for the behavioral sciences. Rev ed, Academic Press". In: *New York. xv* (Jan. 1988) (cit. on p. 37).

[Dab12] Dabbish, Laura and Stuart, Colleen and Tsay, Jason and Herbsleb, Jim. "Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository". In: *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. CSCW '12. Seattle, Washington, USA: ACM, 2012, pp. 1277–1286. ISBN: 978-1-4503-1086-4. DOI: `10.1145/2145204.2145396`. URL: `http://doi.acm.org/10.1145/2145204.2145396` (cit. on p. 27).

[DH07] Prasun Dewan and Rajesh Hegde. "Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development." In: Jan. 2007, pp. 159–178. DOI: `10.1007/978-1-84800-031-5_9` (cit. on pp. 9, 13).

[Fan18] Fang, Yili and Sun, Hailong and Li, Guoliang and Zhang, Richong and Huai, Jingpeng. "Context-Aware Result Inference in Crowdsourcing". In: *Information Sciences* 460 (May 2018). DOI: `10.1016/j.ins.2018.05.050` (cit. on p. 25).

[Ghi18] Ghiotto lima de Menezes, Gleiph and Murta, Leonardo and Barros, MÃand van der Hoek, Andre. "On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub". In: *IEEE Transactions on Software Engineering* PP (Sept. 2018), pp. 1–1. DOI: `10.1109/TSE.2018.2871083` (cit. on pp. 20, 48, 51, 53, 54).

[GPD14] Georgios Gousios, Martin Pinzger, and Arie van Deursen. "An Exploratory Study of the Pull-based Software Development Model". In: *Proceedings of* *the 36th International Conference on Software Engineering.* ICSE 2014. Hyderabad, India: ACM, 2014, pp. 345–355. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568260. URL: http://doi.acm.org/10.1145/2568225.2568260 (cit. on pp. 8, 14, 29).

[GSB16] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. "Work Practices and Challenges in Pull-based Development: The Contributor's Perspective". In: *Proceedings of the 38th International Conference on Software Engineering.* ICSE '16. Austin, Texas: ACM, 2016, pp. 285–296. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884826. URL: http://doi.acm.org/10.1145/2884781.2884826 (cit. on pp. 14, 27).

[GS12] Mário Luís Guimarães and António Rito Silva. "Improving Early Detection of Software Merge Conflicts". In: *Proceedings of the 34th International Conference on Software Engineering.* ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 342–352. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337264 (cit. on pp. 9, 15, 18).

[Jam13] James, Gareth and Witten, Daniela and Hastie, Trevor and Tibshirani, Robert. *An Introduction to Statistical Learning.* Jan. 2013, p. 426. ISBN: 1461471389 (cit. on p. 37).

[JAM17] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. "Evolutionary Trends of Developer Coordination: A Network Approach". In: *Empirical Softw. Engg.* 22.4 (Aug. 2017), pp. 2050–2094. ISSN: 1382-3256. DOI: 10.1007/s10664-016-9478-9. URL: https://doi.org/10.1007/s10664-016-9478-9 (cit. on pp. 8, 12, 15).

[Kal14] Kalliamvakou, Eirini and Gousios, Georgios and Blincoe, Kelly and Singer, Leif and German, Daniel M. and Damian, Daniela. "The Promises and Perils of Mining GitHub". In: *Proceedings of the 11th Working Conference on Mining*

Software Repositories. MSR 2014. Hyderabad, India: ACM, 2014, pp. 92–101. ISBN: 978-1-4503-2863-0. DOI: `10.1145/2597073.2597074`. URL: `/pub/promises-perils-github.pdf` (cit. on p. 28).

[KS13] Bakhtiar Khan Kasi and Anita Sarma. "Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 732–741. ISBN: 978-1-4673-3076-3. URL: `http://dl.acm.org/citation.cfm?id=2486788.2486884` (cit. on pp. 9, 19).

[LW03] J L. Myers and Arnold Well. "Research Design Statistical Analysis". In: XVII (Jan. 2003) (cit. on p. 35).

[Lei13] Leif Singer and Fernando Marques Figueira Filho and Brendan Cleary and Christoph Treude and Margaret-Anne D. Storey and Klaus Schneider. "Mutual assessment in the social programmer ecosystem: an empirical investigation of developer profile aggregators". In: *CSCW*. 2013 (cit. on p. 27).

[Les18] Lessenich, Olaf and Siegmund, Janet and Apel, Sven and Kästner, Christian and Hunsen, Claus. "Indicators for Merge Conflicts in the Wild: Survey and Empirical Study". In: *Automated Software Engg.* 25.2 (June 2018), pp. 279–313. ISSN: 0928-8910. DOI: `10.1007/s10515-017-0227-0`. URL: `https://doi.org/10.1007/s10515-017-0227-0` (cit. on pp. 8, 9, 14, 20, 51).

[LLH16] J. Liu, J. Li, and L. He. "A Comparative Study of the Effects of Pull Request on GitHub Projects". In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. June 2016, pp. 313–322. DOI: `10.1109/COMPSAC.2016.27` (cit. on p. 27).

[Men02] T. Mens. “A State-of-the-Art Survey on Software Merging”. In: *IEEE Trans. Softw. Eng.* 28.5 (May 2002), pp. 449–462. ISSN: 0098-5589. DOI: `10.1109/TSE.2002.1000449`. URL: `https://doi.org/10.1109/TSE.2002.1000449` (cit. on pp. 9, 12).

[Nel19] Nelson, Nicholas and Brindescu, Caius and McKee, Shane and Sarma, Anita and Dig, Danny. “The life-cycle of merge conflicts: processes, barriers, and strategies”. In: *Empirical Software Engineering* (2019), pp. 1–44. DOI: `10.1007/s10664-018-9674-x`. URL: `https://app.dimensions.ai/details/publication/pub.1111934867` (cit. on pp. 9, 21, 22, 24, 25, 48, 51, 53, 54).

[PSW11] Shaun Phillips, Jonathan Sillito, and Rob Walker. “Branching and Merging: An Investigation into Current Version Control Practices”. In: *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering.* CHASE ’11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 9–15. ISBN: 978-1-4503-0576-1. DOI: `10.1145/1984642.1984645`. URL: `http://doi.acm.org/10.1145/1984642.1984645` (cit. on pp. 8, 14).

[Roc17] Rocha, Fabio and Menezes, Pablo and Nascimento, Rogerio and JÃ, Methanias and Oliveira, Adicineia. “CONTINUOUS INTEGRATION AND VERSION CONTROL: A SYSTEMATIC REVIEW”. In: May 2017. DOI: `10.5748/9788599693131-14CONTECSI/RF-4435` (cit. on p. 8).

[S J16] S. Just, K. Herzig, J. Czerwonka and B. Murphy. “Switching to Git: The Good, the Bad, and the Ugly”. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE).* Oct. 2016, pp. 400–411. DOI: `10.1109/ISSRE.2016.38` (cit. on p. 29).

[S M17] S. McKee and N. Nelson and A. Sarma and D. Dig. “Software Practitioner Perspectives on Merge Conflicts and Resolutions”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* Sept.

2017, pp. 467–478. DOI: 10.1109/ICSME.2017.53 (cit. on pp. 9, 21, 22, 24, 25, 48).

[SNv03] A. Sarma, Z. Noroozi, and A. van der Hoek. "Palantir: raising awareness among configuration management workspaces". In: *25th International Conference on Software Engineering, 2003. Proceedings.* May 2003, pp. 444–454. DOI: 10.1109/ICSE.2003.1201222 (cit. on p. 19).

[SRv12] A. Sarma, D. F. Redmiles, and A. van der Hoek. "Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes". In: *IEEE Transactions on Software Engineering* 38.4 (July 2012), pp. 889–908. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.64 (cit. on pp. 9, 15).

[SRP17] Todd Sedano, Paul Ralph, and Cécile Péraire. "Software Development Waste". In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 130–140. ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.20. URL: https://doi.org/10.1109/ICSE.2017.20 (cit. on p. 12).

[Sie14] Siegmund, Janet and Schumann, Jana. "Confounding parameters on program comprehension: a literature survey". In: *Empirical Software Engineering* 20 (Aug. 2014). DOI: 10.1007/s10664-014-9318-8 (cit. on p. 55).

[Sto14] Storey, Margaret-Anne and Singer, Leif and Cleary, Brendan and Figueira Filho, Fernando and Zagalsky, Alexey. "The (R) Evolution of Social Media in Software Engineering". In: *Proceedings of the on Future of Software Engineering*. FOSE 2014. Hyderabad, India: ACM, 2014, pp. 100–116. ISBN: 978-1-4503-2865-4. DOI: 10.1145/2593882.2593887. URL: http://doi.acm.org/10.1145/2593882.2593887 (cit. on p. 27).

[Thy09] Thyvalikakath, Thankam and Monaco, Valerie and Thambuganipalle, Himabindu and Schleyer, Titus. “Comparative study of heuristic evaluation and usability testing methods”. In: *Studies in health technology and informatics* 143 (Feb. 2009), pp. 322–7. DOI: `10.3233/978-1-58603-979-0-322` (cit. on p. 25).

[TDH14] Jason Tsay, Laura Dabbish, and James Herbsleb. “Influence of Social and Technical Factors for Evaluating Contribution in GitHub”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 356–366. ISBN: 978-1-4503-2756-5. DOI: `10.1145/2568225.2568315`. URL: `http://doi.acm.org/10.1145/2568225.2568315` (cit. on p. 27).

[Whi10] Whitehead, Jim and Mistrík, Ivan and Grundy, John and van der Hoek, André. “Collaborative Software Engineering: Concepts and Techniques”. In: *Collaborative Software Engineering*. Ed. by Ivan Mistrík et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–30. ISBN: 978-3-642-10294-3. DOI: `10.1007/978-3-642-10294-3_1`. URL: `https://doi.org/10.1007/978-3-642-10294-3_1` (cit. on p. 8).

[Zim04] Zimmermann, Thomas and Weisgerber, Peter and Diehl, Stephan and Zeller, Andreas. “Mining Version Histories to Guide Software Changes”. In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572. ISBN: 0-7695-2163-0. URL: `http://dl.acm.org/citation.cfm?id=998675.999460` (cit. on p. 13).

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Masterarbeitin gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, 17. Juli 2019

AUTHOR