UNIVERSITÄT
DES
SAARLANDES

Investigating the Merge Conflict Life-Cycle Taking the Social Dimension into Account

by
Gustavo Andrade do Vale

A dissertation submitted towards the degree

Doctor of Engineering (Dr.-Ing.)

of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken, 2023

Dean of the Faculty	Prof. Dr. Jürgen Steimle
Day of Colloquium	11.03.2024
Chair of the Committee	Prof. Dr. Ingmar Weber
Reviewers	Prof. Dr. Sven Apel
	Prof. Anita Sarma, PhD
Academic Assistant	Dr. Andreas Schmidt

Abstract

Context. Software development is a collaborative and distributed activity in which success depends on the ability to coordinate social and technical assets. Version control systems help developers to manage technical assets (e.g., code changes) over time by tracking code contributions, especially when involving collaborations of multiple developers. This allows developers to address different programming tasks (e.g., bug fixing and adding new features) simultaneously without losing changes. After fulfilling their tasks, developers can merge their possibly concurrent changes to the main repository to have their work incorporated into the mainstream. A merge scenario includes the whole timeline of creating a project branch, committing changes independently to the project main branch, and creating a merge commit.

Problem. In collaborative software development, merge conflicts arise when developers integrate concurrent code changes. Merge conflicts are a notorious problem in collaborative software development. Whereas merge conflicts are common to introduce, they bring several issues to software projects. For instance, merge conflicts: (i) distract developers from their workflow, (ii) negatively impact team productivity, motivation, and keeping the schedule, and (iii) resolving them is a difficult, time-consuming, and often error-prone task. Despite a substantial number of studies investigating merge conflicts, the social dimension of the problem (e.g., the influence of developer communication and developer roles related to conflicts) is often ignored.

Goals. In this thesis, we seek out to understand the role the social dimension plays in the merge conflict life-cycle. The assumption is that, by deeply understanding the social dimension of the problem, we are able to provide actionable directions for researchers, tool builders, and practitioners to efficiently avoid, predict, and resolve merge conflicts.

Method and Results. To reach our goals, we conducted a series of empirical studies investigating the merge conflict life-cycle. As a technical foundation, we created a framework, 4CsNet, to automatically mine and rebuild historical information of open source repositories that follow the three-way merge pattern. 4CsNet retrieves technical information using the Git version control system and social information mainly using GitHub issues and events. With the data collected by 4CsNet, we conducted four empirical studies.

In the first empirical study, we investigated the relation between the communication activity and merge conflicts motivated by the popular belief that communication and collaboration successes are mutually dependent. We found that active GitHub communication is not associated with the emergence or avoidance of merge conflicts even though developers communicate with each other.

In the second empirical study, we investigated whether it is possible to predict merge conflicts with social measures (i.e., developer roles at coarse- and fine-grained levels). The motivation is that, effectively predicting merge conflicts decreases the cost of constantly

pulling and merging a large number of branch combinations (i.e., speculative merging), which makes awareness tools reliable and feasible in practice. Our results show that it is possible to predict merge conflicts taking the social perspective into account with 100% of recall (i.e., all real conflicts are correctly identified). However, to achieve state-of-the-art performance (i.e., combination of recall, prediction, accuracy, Area Under the Curve (AUC)), technical measures (e.g., the number of lines of code in conflict or the number of chunks) are necessary. On one hand, our results highlight the importance of investigating the social perspective, especially when developers coordinate themselves without sophisticated tool support. On the other hand, this study highlights that the technical perspective is still essential to predict merge conflicts.

In the third empirical study, we investigated developer roles and specific developer code changes. The motivation is that understanding social aspects of conflicting contributors and their activity on changing source files can help managers or developers themselves to decide which developers to instruct to avoid merge conflicts. Our results show that 80% of contributors are involved in one or two merge scenarios, and only 3.8% of developers are involved in more than 10 conflicting scenarios. We also found that 48% of the project's top contributors participated in more than 50% of the conflicting merge scenarios in their project. This is evidence that these developers are related to merge conflicts, and a better coordination of these developers might reduce the number of merge conflicts.

In the fourth empirical study, we moved to the end of the merge conflict life-cycle investigating the challenges and factors related to the merge conflict resolution. The premise is that understanding which kind of merge conflicts are time-consuming to resolve, developers should focus on avoiding these kinds of conflicts mostly. Our results show that measures indirectly related to merge conflicts (i.e., measures related to the merge scenario changes) are more strongly correlated with merge conflict resolution time than measures directly related to merge conflicts (i.e., merge conflict characteristics). Aiming at cross-validating our results and searching for new findings, we surveyed 140 developers. We found four main challenges on merge conflict resolution: *lack of coordination*, *lack of tool support*, *flaws in the system architecture*, and *lack of testing suite or pipeline for continuous integration*.

Conclusion. In this thesis, we call the attention of researchers, tool builders, and practitioners to the importance of including the social dimension when investigating merge conflicts. Our findings also provide evidence that they should also look at the technical dimension more closely. We present below the main contributions of our thesis.

- GitHub communication activity itself does not influence the occurrence or avoidance of merge conflicts.
- It is possible to correctly predict all real merge conflicts using only social measures (e.g., the number of top or occasional contributors touching the source branch).
- The top conflicting contributor of a project is often related to the majority of merge conflicts of a project, which is evidence that properly coordinating this developer reduces the number of merge conflicts of a project.
- The branch that developers are touching is an important factor when investigating merge conflicts. For instance, changes in the *source branch* are 3 times more conflict-prone than changes in the *target branch* when occasional contributors are involved

in the code changes. Hence, branches in the source branch should be more closely monitored.

- Changes indirectly related to merge conflicts (e.g., the number of chunks changed in the merge scenario) have a greater impact on the merge conflict resolution time than *changes directly* related to merge conflicts (e.g., the number of chunks in conflict). Hence, researchers should consider changes indirectly related to merge conflicts to predict and avoid the merge conflicts, for instance.
- We identified four main challenges on merge conflict resolution reported by software developers (*lack of coordination*, *lack of tool support*, *flaws in the system architecture*, and *lack of testing suite or pipeline for continuous integration*) and purpose a set of solutions/guidelines to support developers to get around these challenges, minimise the emergence of merge conflicts, and make conflict resolutions faster.

Zusammenfassung

Kontext. Softwareentwicklung ist eine kollaborative und verteilte Aktivität, bei der der Erfolg von der Fähigkeit abhängt, soziale und technische Ressourcen zu koordinieren. Versionskontrollsysteme helfen Entwicklern, technische Assets (z. B. Codeänderungen) im Laufe der Zeit zu verwalten, indem sie Codebeiträge verfolgen, insbesondere wenn mehrere Entwickler zusammenarbeiten. Dadurch können Entwickler verschiedene Programmieraufgaben (z.B., Fehlerbehebung und Hinzufügen neuer Funktionen) gleichzeitig bearbeiten, ohne dass Änderungen verloren gehen. Nach Erfüllung ihrer Aufgaben, können Entwickler ihre möglicherweise gleichzeitigen Änderungen im Haupt-Repository zusammenführen, um ihre Arbeit in den Mainstream zu integrieren. Ein Zusammenführungsszenario umfasst die gesamte Zeitleiste der Erstellung eines Projektz branch, der unabhängigen Übertragung von Änderungen an den Haupt Branch des Projekts und der Erstellung eines Merge-Commits.

Problem. Bei der kollaborativen Softwareentwicklung entstehen Merge Konflikte, wenn Entwickler gleichzeitige Codeänderungen integrieren. Merge Konflikte sind ein bekanntes Problem bei der kollaborativen Softwareentwicklung. Während Merge Konflikte häufig auftreten, bringen sie bei Softwareprojekten mehrere Probleme. Merge Konflikte zum Beispiel: (i) lenken Entwickler von ihrem Arbeitsablauf ab, (ii) wirken sich negativ auf die Teamproduktivität, Motivation und Einhaltung des Zeitplans aus und (iii) sie zu lösen ist ein schwieriger, zeitaufwändiger und oft ein Fehler -anfällige Aufgabe. Trotz einer beträchtlichen Anzahl von Studien, die Merge-Konflikte untersuchen, wird die soziale Dimension des Problems (z.B., der Einfluss der Entwicklerkommunikation und der Entwicklerrollen im Zusammenhang mit Konflikten) häufig ignoriert.

Ziele. In dieser Arbeit versuchen wir zu verstehen, welche Rolle die soziale Dimension im Lebenszyklus von Merge Konflikte spielt. Wir gehen davon aus, dass wir durch ein tiefes Verständnis der sozialen Dimension des Problems in der Lage sind, Forschern, Werkzeugbauern und Praktikern umsetzbare Anweisungen zu geben, um Merge Konflikte effizient zu vermeiden, vorherzusagen und zu lösen.

Methode und Ergebnisse. Um unsere Ziele zu erreichen, haben wir eine Reihe empirischer Studien durchgeführt, die den Lebenszyklus von Merge Konflikte untersuchten. Als technische Grundlage haben wir ein Framework, 4CsNet, erstellt, um historische Informationen von Open-Source-Repositories, die dem Drei-Wege-Merge-Muster folgen, automatisch zu extrahieren und neu zu erstellen. 4CsNet ruft technische Informationen mithilfe des Versionskontrollsystems Git und soziale Informationen hauptsächlich mithilfe von GitHub-Problemen und -Ereignissen ab. Mit den von 4CsNet gesammelten Daten führten wir vier empirische Studien durch.

In der ersten empirischen Studie untersuchten wir den Zusammenhang zwischen der Kommunikationsaktivität und Merge Konflikte, die durch die weit verbreitete Überzeugung motiviert sind, dass Kommunikations- und Kollaborationserfolge einander bedingen. Wir

stellten fest, dass eine aktive GitHub-Kommunikation nicht mit der Entstehung oder Vermeidung von Merge-Konflikten verbunden ist, obwohl Entwickler miteinander kommunizieren.

In der zweiten empirischen Studie untersuchten wir, ob es möglich ist, Merge Konflikte mit sozialen Maßnahmen (d. h. Entwicklerrollen auf grob- und feinkörniger Ebene) vorherzusagen. Die Motivation besteht darin, dass durch eine effektive Vorhersage von Merge Konflikte die Kosten für das ständige Ziehen und Zusammenführen einer großen Anzahl von Zweigkombinationen (d. h. spekulatives Zusammenführen) gesenkt werden, was Awareness-Tools zuverlässig und in der Praxis umsetzbar macht. Unsere Ergebnisse zeigen, dass es möglich ist, Merge Konflikte unter Berücksichtigung der sozialen Perspektive mit 100% Rückruf vorherzusagen (d. h. alle realen Konflikte werden korrekt identifiziert). Um jedoch eine Leistung auf dem neuesten Stand der Technik (d.h., Kombination aus Rückruf, Vorhersage, Genauigkeit, AUC) zu erreichen, sind technische Maßnahmen (z. B. die Anzahl der in Konflikt stehenden Codezeilen oder die Anzahl der Chunks) notwendig. Einerseits unterstreichen unsere Ergebnisse, wie wichtig es ist, die soziale Perspektive zu untersuchen, insbesondere wenn Entwickler sich ohne ausgefeilte Toolunterstützung koordinieren. Andererseits unterstreicht diese Studie, dass die technische Perspektive immer noch von entscheidender Bedeutung ist, um Merge Konflikte vorherzusagen.

In der dritten empirischen Studie untersuchten wir Entwicklerrollen und spezifische Änderungen am Entwicklercode. Die Motivation besteht darin, dass das Verständnis der sozialen Aspekte widersprüchlicher Mitwirkender und ihrer Aktivitäten beim Ändern von Quelldateien Managern oder Entwicklern dabei helfen kann, selbst zu entscheiden, welche Entwickler sie anweisen sollten, um Merge Konflikte zu vermeiden. Unsere Ergebnisse zeigen, dass 80% der Mitwirkenden an einem oder zwei Zusammenführungsszenarien beteiligt sind und nur 3,8% der Entwickler an mehr als 10 widersprüchlichen Szenarien beteiligt sind. Wir haben außerdem herausgefunden, dass 48% der Top-Mitwirkenden des Projekts an mehr als 50% der widersprüchlichen Zusammenführungsszenarien in ihrem Projekt beteiligt waren. Dies ist ein Beweis dafür, dass diese Entwickler mit Merge Konflikte in Zusammenhang stehen, und eine bessere Koordination dieser Entwickler könnte die Anzahl der Merge Konflikte verringern.

In der vierten empirischen Studie näherten wir uns dem Ende des Lebenszyklus des Merge-Conflites und untersuchten die Herausforderungen und Faktoren im Zusammenhang mit der Lösung des Merge-Konfliktes. Die Prämisse ist, dass Entwickler sich darauf konzentrieren sollten, diese Art von Konflikten weitgehend zu vermeiden, wenn sie verstehen, welche Art von Merge Konflikte zeitaufwändig zu lösen sind. Unsere Ergebnisse zeigen, dass Maßnahmen, die sich indirekt auf Merge Konflikte beziehen (d.h., Maßnahmen im Zusammenhang mit Änderungen des Zusammenführungsszenarios), stärker mit der Lösungszeit für Merge Konflikte korrelieren als Maßnahmen, die direkt mit Merge Konflikte in Zusammenhang stehen (d.h., Merkmale von Merge Cinflicts). Um unsere Ergebnisse gegenseitig zu validieren und nach neuen Erkenntnissen zu suchen, befragten wir 140 Entwickler. Wir stellten vier Hauptherausforderungen bei der Lösung von Zusammenführungskonflikten fest: *Mangel an Koordination*, *Mangel an Tool-Unterstützung*, *Mängel in der Systemarchitektur* und *Mangel an Testsuite oder Pipeline für kontinuierliche Integration*.

Schlussfolgerung. In dieser Arbeit machen wir Forscher, Werkzeugbauer und Praktiker darauf aufmerksam, wie wichtig es ist, die soziale Dimension bei der Untersuchung von Merge Konflikte einzubeziehen. Unsere Erkenntnisse belegen zudem, dass auch die technis-

che Dimension stärker in den Blick genommen werden sollte. Im Folgenden stellen wir die Hauptbeiträge unserer Dissertation vor.

- Die GitHub-Kommunikationsaktivität selbst hat keinen Einfluss auf das Auftreten oder die Vermeidung von Merge Konflikte.
- Es ist möglich, alle echten Merge Konflikte korrekt vorherzusagen, indem man nur soziale Kennzahlen verwendet (z. B. die Anzahl der Top- oder gelegentlichen Mitwirkenden, die den Quellzweig berühren).
- Der Mitwirkende mit den größten Konflikten in einem Projekt steht häufig im Zusammenhang mit den meisten Merge Konflikte eines Projekts. Dies ist ein Beweis dafür, dass die richtige Koordination dieses Entwicklers die Anzahl der Merge Konflikte eines Projekts verringert.
- Der Branch, den Entwickler berühren, ist ein wichtiger Faktor bei der Untersuchung von Merge Konflikte. Beispielsweise sind Änderungen im *Source-Branch* dreimal konfliktanfälliger als Änderungen im *Target-Branch*, wenn gelegentliche Mitwirkende an den Codeänderungen beteiligt sind. Daher sollten Branches im Source-Branch genauer überwacht werden.
- Änderungen, die indirekt mit Merge Konflikte zusammenhängen (z.B., die Anzahl der im Zusammenführungsszenario geänderten Blöcke), haben einen größeren Einfluss auf die Auflösungszeit der Merge-Konflikte als Änderungen, die direkt mit Merge Konflikte zusammenhängen (z.B., die Anzahl der im Konflikt befindlichen). Daher sollten Forscher Änderungen berücksichtigen, die indirekt mit Merge Konflikte zusammenhängen, um beispielsweise Merge Konflikte vorherzusagen und zu vermeiden.
- Wir identifizierten vier Hauptherausforderungen bei der Lösung von Merge Konflikte, die von Softwareentwicklern gemeldet wurden (*mangelnde Koordination*, *mangelnde Toolunterstützung*, *Mängel in der Systemarchitektur* und *fehlende Testsuite bzw Pipeline für kontinuierliche Integration*) und wir entwickelten eine Reihe von Lösungen/Richtlinien, um Entwickler dabei zu unterstützen, diese Herausforderungen zu meistern, das Auftreten von Merge Konflikte zu minimieren und Konfliktlösungen schneller zu gestalten.

It is the long history of humankind (and animal kind, too)
that those who learned to collaborate and improvise
most effectively have prevailed.
Charles Darwin

Acknowledgments

This work would not have been possible without the support of many people. No matter how large the involvement of each one, it is the whole and not parts that made this dissertation possible.

First and foremost, I thank God for providing me the discipline and persistence to reach a Dr.-Ing (PhD) degree. I thank my beloved wife Fernanda, who has had patience in difficult moments and has always been by my side. I thank Laura and Alice, our daughters, who gave me a lot of helpful input, not directly related to the topic, but nevertheless very positive and inspiring. I thank my whole family — especially my father Fernando, my mother Márcia, my sister Letícia, my father-in-law Tom, and my mother-in-law Regina for having always supported me.

I would like to express my gratitude to my advisor Sven Apel. A great idea often begins in a fragile state and can be swiftly crushed by only a few words. Sven's ability to foster an idea, patiently develop it and deliver criticism with a degree of sensitivity, is a truly valuable and unique quality. I got to know Sven as a student in a conference and have always admired his openness to students and colleagues. Sven's personality was one of the main reasons why I decided to become part of his research group. It has been an absolute privilege to work with Sven.

I thank all my co-authors and colleagues from the research group for teaching me something new, all in their own way. Especially Eduardo Figueiredo and Angelika Schmid. Eduardo Figueiredo contributed conceptually and technically in multiple parts of this study, including a proof-reading of the whole thesis.

I am grateful for the funding of the Brazilian National Council for the Scientific and Technological Development (Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq) and the German Academic Exchange Service (Deutscher Akademischer Austauschdienst - DAAD).

Finally, I thank all participants who took the survey of one of our empirical studies.

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
APR	Automated Program Repair
AST	Abstract Syntax Tree
AUC	Area Under the Curve
BS	Borderline Synthetic Minority Oversampling TEchnique
CD	Continuous Delivery
CI	Continuous Integration
CLA	Contribution Licence Agreement
CRDT	Commutative Replicated Data Types
CVS	Concurrent Versions System
DAO	Data Access Object
DR	Difficulty Ratio
DSEE	Domain Software Engineering Environment
ID	Identifier
IDE	Integrated Development Environment
IQR	Inter-Quartile Range
IRC	Internet Relay Chat
KNN	K-Nearest Neighbours
LOC	Lines Of Code
MR	Modification Request
MVC	Model-View-Controller
NPM	Node Package Manager
OSS	Open-Source Software
OT	Operational Transformation
PCA	Principal Component Analysis
PR	Pull Request
RCE	Rarely Concurrently Edited
RCS	Revision Control System
RQ	Research Question

SCM Software Configuration Management
SMOTE Synthetic Minority Oversampling TEchnique
SVM Support Vector Machine
SVMS Support Vector Machine SMOTE
SVN Subversion
TOM Testing on Merges
UML Unified Modeling Language
UUT Unit Under Testing
VCS Version Control System
VFJ Versioned Featherweight Java
XML Extensible Markup Language

1 Introduction

1.1 Problem and Motivation

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 [157]. Multiple developers may work and update a single project concurrently by working on a separate version of the project. Version Control Systems (VCSs) are tools used to facilitate collaborative software development. With the support of a VCS, changes in the project are saved over time in a repository. It allows tracking all the changes made in the past (e.g., to detect what, why, and when changes were introduced, and who did them). In the case changes went wrong, developers can go back in time and revert them to a working version. VCSs allow developers to create a branch to implement new features, refactor existing features, or to fix bugs simultaneously without affecting the project mainline branch. Once the task is completed in the working branch, developers can merge it to the main repository branch. The whole process of creating a branch, addressing a task, and merging it back is often referred to as a merge scenario [116]. A *merge scenario* includes the whole timeline of creating a project branch, committing changes independently to the branch, and creating a merge commit.

Merging particular code changes may introduce merge conflicts. A *merge conflict* occurs when one or multiple developers make concurrent changes in different branches [203]. By concurrent changes, we refer to changes that work correctly separated, but when merged these changes alter the order of chunks of code, syntax, semantic, or the behaviour of a component. Merge conflicts must be resolved before continuing working on the project as the code will not be able to build or pass in the tests. Several studies have investigated the conflict rate (i.e., the percentage of conflicting merge scenarios among all merge scenarios), which varies from 0% to 87.84% when looking at projects individually [216] and from 8% to 14% when analysing sets of projects [2, 60, 110, 199]. Researchers and practitioners seek to minimise the number of merge conflicts, as merge conflicts distract developers from their workflow and resolving them is difficult, time-consuming, and often error-prone [180, 203].

There is a multitude of studies investigating the whole merge conflict life-cycle. Aiming at avoiding the emergence of merge conflicts, studies have investigated merge strategies (e.g., [10, 11]), prediction strategies (e.g., [49, 123]), and awareness tools (e.g., [32, 81, 252, 266]). Studies have investigated their cause and nature learning how they look like exactly [110]), which type of code changes lead to each type of merge conflict [1, 41, 192], and aiming at predicting them [82, 180, 226]. More recently, we can find studies investigating merge conflict resolution [42, 196, 214].

Although previous studies have provided mechanisms to avoid, minimise, understand, and remove merge conflicts, they largely ignore the social dimension of the problem. Considering that software development is a social task since developers (as a team) create, resolve, or avoid merge conflicts, the thesis' key assumption is that, by deeply understanding the social dimension, we will be able to understand scenarios that lead to merge conflicts more clearly, predict them, and make the merge conflict resolution simpler.

1.2 Thesis Goal

The goal of the thesis is to deeply understand the role the social dimension plays in the merge conflict life-cycle. To achieve this goal, we created a framework that mines GitHub repositories (called 4CsNet). 4CsNet allows us to retrieve fine-grained technical and social information about merge scenarios in the history of software projects. Based on the data retrieved by 4CsNet, we performed four empirical studies with complementary goals. First, we explored the relation between communication activity and merge conflicts motivated by the popular belief that communication and collaboration success are mutually dependent. Second, we investigated the influence of top and occasional contributors (named developer roles) on the occurrence of merge conflicts. The motivation is that, effectively predicting merge conflicts, decreases the cost of constantly pulling and merging a large number of branch combinations (i.e., speculative merging), which makes awareness tools reliable and feasible in practice. Third, after getting evidence that a few developers normally are involved in several merge conflicts (named top-conflicting contributors), we performed an exploratory study to identify conflict-prone actions of these developers. The motivation is that understanding social aspects of conflicting contributors and their activity on changing source files can help managers or developers themselves to decide which developers to instruct to avoid merge conflicts. Finally, we performed a two-step study to understand the main challenges developers face when resolving merge conflicts. The premise is that understanding which kind of merge conflicts are time-consuming to resolve, developers should focus on avoiding these kinds of conflicts mostly. In Figure 1.1, we show an overview of the conducted work and the mapping to chapters of the thesis. Note that 4CsNet is the framework used to collect data for all of our empirical studies, and, in all empirical studies, we investigate technical and social assets.

1.3 Contributions

In this thesis, we make several contributions to facilitate the analysis and understanding of the social dimension in the merge conflict life-cycle. We mention and explain the extent of our contributions below.

1. **A taxonomy of merge conflicts**. Our first contribution is a taxonomy of merge conflicts, for which we identified studies related to merge conflicts extracting four types of conflicts: textual, syntactic, semantic, and behavioural. Furthermore, we present an extensive literature review on merge conflicts including a characterization of merge

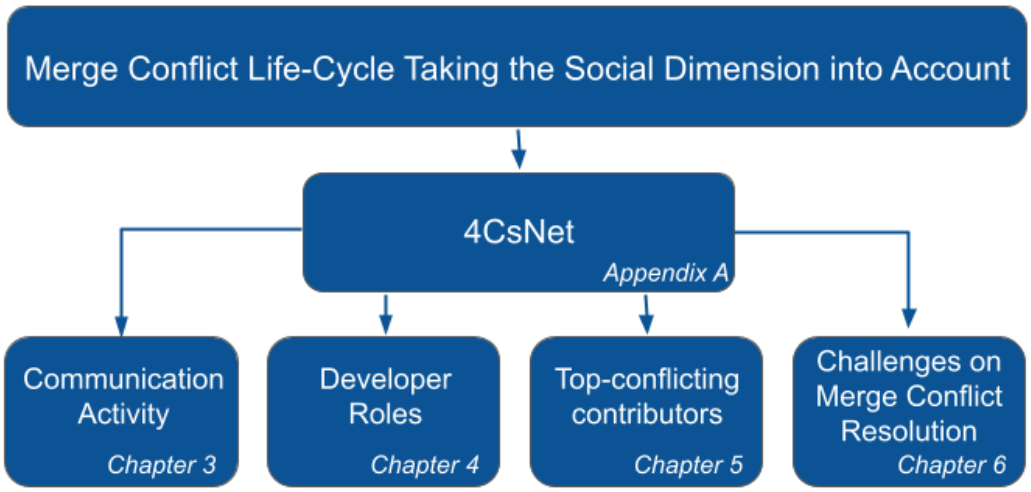

Figure 1.1: Overview of Conducted Work

conflicts, factors related to merge conflicts, and strategies to merge, to avoid, and to resolve merge conflicts.

2. **4CsNet**. Our second contribution is our methodological framework for studies mining GitHub repositories. Since we are interested in understanding the evolution of open-source repositories, we develop 4CsNet which is able to collect technical and social measures for our empirical studies. To the best of our knowledge, there is no tool to integrate technical and social perspectives. This integration supports studies investigating the influence of social factors on technical factors and vice-versa. This kind of study provides a broader view of software development evolution of open source software repositories and brings new findings and directions to studies on software coordination, especially when investigating the life-cycle of merge conflicts (seen in our empirical studies).

3. **The relation between GitHub Communication and Merge Conflicts**. Our third contribution is the results and findings from the study which understands the relation between GitHub communication and merge conflicts. There is a popular belief that communication and collaboration success are mutually dependent [256, 271, 280]. So, it is believed that proper communication activity helps to avoid merge conflicts. However, in practice, the role of communication for merge conflicts to occur or to be avoided has not been thoroughly investigated. To better understand this relation, we analysed the history of 30 popular open-source projects involving 19 thousand merge scenarios. Methodologically, we used a bivariate (Spearman's rank correlation) and a multivariate (principal component analysis and partial correlations) analysis to quantify their relation. In the bivariate analysis, we found a weak positive correlation between GitHub communication activity and the number of merge conflicts. However, in the multivariate analysis, the positive correlation disappeared, which counters the intuition that active GitHub communication helps to avoid merge conflicts. Interestingly, we found that the strength of this relationship depends on the merge scenarios' characteristics, such as the number of lines changed within the scenario. Puzzled by

these unexpected results, we investigated each covariate, which provided justifications for our findings. The main conclusion from this study is that GitHub communication activity itself is not directly associated with the emergence or avoidance of merge conflicts per se, even though such communication is associated only with merge scenario code changes or among developers.

4. **Predicting Merge Conflicts Considering Social and Technical Assets**. Our fourth contribution is the results and findings from a study that aims at understanding the role of social assets on the merge conflicts predictions. The overall goal is to understand and predict merge conflicts considering social and technical assets. We devise three models for predicting merge conflicts based on common measures used by developers. The first model focuses on the social assets, the second on technical assets, and the third on the combination of technical and social assets. To evaluate our predictors, we conducted a large-scale empirical study analysing the histories of 66 real-world software systems. Specifically, we categorise developers into *top* or *occasional* contributors at project and merge-scenario level. We found that top contributors at project level and occasional contributors at merge-scenario level are involved in more merge conflicts than developers taking other roles. Hence, the coordination of top contributors at project level and occasional contributors at merge-scenario level is a good starting point to minimise the occurrence of merge conflicts (especially because when these two developers work on the source branch, the chances of merge conflicts are 32.31%). Overall, we show that predicting merge conflicts incorporating developer roles is possible in practice with high accuracy (0.92) and recall (1.00) when combining technical and social assets, which is vital information to guide improvements on speculative merging techniques.

5. **Deeply Investigating Conflicting Contributors**. Our fifth contribution is a study on who are the developers contributing to merge conflicts and possible actions they are ignoring to be the developers dealing with so many merge conflicts. Contributions to open-source software projects typically depend on simultaneous contributions of several developers. These contributions often affect the same source files and may lead to merge conflicts when integrated. Previous studies investigated the reduction of conflicting merge scenarios. However, empirical evidence on the involvement of open-source software contributors in conflicting merge scenarios is scarce. We aim to fill this gap with a large-scale quantitative study with the goal of understanding: (i) the extent in which open-source software contributors are involved in conflicting merge scenarios; (ii) characteristics of these contributors; and (iii) characteristics of changing source files. We collected both contributor data and contribution data from 66 popular GitHub projects and analysed data of 2972 distinct contributors who were involved in, at least, one conflicting merge scenario. We rely on both descriptive and inference statistics to address our research questions. We found that about 80% of analysed contributors are involved in only one or two conflicting merge scenarios. 42 out of the 66 projects had its top-one contributor as the one mostly involved in conflicting merge scenarios. Finally, only a small set of changing source files are involved in conflicting merge scenarios, so training the typically small group of contributors could significantly reduce the number of merge conflicts.

6. **Challenges of Resolving Merge Conflicts**. Our sixth contribution is related to the challenges developers face when dealing with merge conflicts. Despite a substantial number of studies investigating merge conflicts, the challenges in *merge conflict resolution* are not well understood. Our goal is to investigate which factors make merge conflicts longer to resolve in practice. To this end, we performed a two-phase study. First, we analysed 66 projects containing around 81 thousand merge scenarios, involving 2 million files and over 10 million chunks. For this analysis, we used rank correlation, principal component analysis, multiple regression, and effect-size analysis to investigate which independent variables (e.g., number of conflicting chunks and files) influence our dependent variable (i.e., time to merge). We found that the *number of chunks*, *lines of code*, *conflicting chunks*, *developers involved*, *conflicting lines of code*, *conflicting files*, and the *complexity of the conflicting code* influence the merge conflict resolution time. Second, we surveyed 140 developers from our subject projects aiming at cross-validating our results from the first phase of our study. As main results, (i) we found that committing small chunks makes merge conflict resolution faster when leaving other independent variables untouched, (ii) we gathered evidence that merge scenario characteristics (e.g., the number of lines of code or chunks changed in the merge scenario) are stronger correlated with our dependent variable than merge conflict characteristics (e.g., the number of lines of code or chunks in conflict), (iii) we devise a taxonomy of four types of challenges in merge conflict resolution, and (iv) we observed that the inherent dependencies among conflicting and non-conflicting code is one of the main factors influencing the merge conflict resolution time.

Considering all six contributions, we can see that technical factors directly related to merge conflicts (e.g., the number of lines of code in conflict or chunks in conflict) are important to understand, avoid, predict, and resolve merge conflicts. However, we found evidence that looking a bit broader on factors indirectly related to merge conflicts (e.g., the number of lines of code and chunks touched in a merge scenario or the touched branch) and social assets (e.g., classifying developers into top and occasional contributors), often ignored by researchers, provide useful information to deeply understand the merge conflict life-cycle. Such information is useful for practitioners to make their workday tasks more fluid and pleasant and for researchers and tool builders to improve the state-of-the-art of merge conflicts. Our findings call the attention of the software engineering community to investigate merge conflicts broader than it is usually investigated. Furthermore, we highlight that sometimes the environment might confuse our perceptions leading to popular beliefs which are not always true. In another direction, performing analyses from different perspectives (e.g., based on data and based on developers´ perception) help us to deeply understand a phenomenon. In the end, we are confident that we provide substantial and innovative contributions for researchers, tool builders, and practitioners in the merge conflict life-cycle, which is a critical contribution to the future and success of coordination on software engineering.

1.4 Outline

We begin (Chapter 2) by describing the established concepts and techniques related to coordination in collaborative software development, how the communication flow in GitHub is, human factor investigations, and a wide overview on studies investigating merge conflicts. After presenting a robust background, we present the four empirical studies shown in Figure 1.1. In Chapter 3, we investigate the relation between GitHub communication and merge conflicts. In Chapter 4, we investigate whether it is possible to predict merge conflicts using social measures and the role of technical measures on merge conflict prediction. In Chapter 5, we investigate the interplay of top and occasional contributors on merge conflicts and deeply investigated possible reasons that a few conflicting contributors are involved in so many merge conflicts. In Chapter 6, we investigate the main challenges developers face when resolving merge conflicts. Finally, in Chapter 7, we discuss the broader implications of our work for investigating social and technical aspects when analysing merge conflicts and provide opportunities and directions for future work.

2

Background and Related Work

In this chapter, we present a background for the understanding of the thesis. In Section 2.1, we present an overview of coordination in software engineering. In Section 2.2, we describe what is collaborative software development. In Section 2.3, we describe the communication flow in collaborative software development. In Section 2.4, we present studies characterising merge conflicts. In Section 2.5, we present merging techniques, algorithms, tools, and studies comparing tools and strategies. In Section 2.6, we present studies presenting factors related to merge conflicts. In Section 2.7, we describe studies and tools proposed to avoid merge conflicts. In Section 2.8, we discuss studies related to merge conflict resolution. In Section 2.9, we briefly describe studies investigating human factors. Finally, in Section 2.10, we conclude this chapter and present perspectives on the discussed topics.

2.1 Coordination in Software Engineering

Coordination challenges were identified very early in the nascent field of Software Engineering. Brooks [160] observed that software development was "a complex interpersonal exercise". Curtis et al. [73] recognized that breakdowns in communication and coordination efforts constituted a major problem in large-scale software development. Humphrey [144] concluded that people are the organisation's most important asset. Humphrey's rationale was based on the principle that expecting the introduction of defects to software and then employing a reactive strategy to fix them is inefficient and costly. By changing the focus to people and how they performed their work, quality could be assured earlier in the software process resulting in cost and efficiency advantages. Staudenmayer [278] recognized that good coordination of teams of developers was correlated with high team performance. Over the years, the importance of people and how they perform their work has only increased. As a response to severe pressure to reduce time-to-market and the desire to remain competitive by tapping into global talent pools, organisations are increasingly adopting global software development practices. These organisations now face more and more challenges stemming from the need to coordinate large numbers of individuals working on tasks that span geographic, cultural, and language boundaries [135, 136, 224].

Researchers and practitioners have proposed a large number of strategies to facilitate the coordination and collaboration required of software development efforts including tools (e.g., Concurrent Versions System (CVS) and Git), approaches (e.g., software process models), and techniques (e.g., pair programming). Many of the problems faced by software developers are the same as problems faced by professionals in other domains: communication breakdowns, coordination problems, lack of knowledge about colleagues' activities, and so on [254].

2.2 Collaborative Software Development

Version control systems, such as Git, help developers to manage source-code changes over time by tracking all code modifications [323]. This allows developers to make concurrent contributions without losing changes. This way, multiple contributors may add new features or fix bugs simultaneously. After fulfilling their tasks, developers merge the proposed changes into the main repository. Developing software by means of merging changes into the main repository is a widely collaborative development pattern, called the *pull-based development model* [115][116]. A *merge scenario* includes the whole timeline of creating a project branch, committing changes independently to the branch, and creating a merge commit (e.g., using a Pull Request (PR)). It is also called three-way merge [116] [180].

There are other ways than the three-way pattern to integrate code to the repository, such as fast-forward, rebase, or squash integrations [161]. However, these integrations damage the project's history, hindering the understanding of how the changes were made in practice. Hence, to understand the evolution of the project, we use the three-way merge pattern. Even though a branch lives longer, it is considered just the changes from the fork up to the integration. If the branch is forked and integrated again, it sets up another merge scenario.

In Figure 2.1, we exemplify a merge scenario in the three-way pattern. The light-gray boxes in the communication layer stand for issues and white boxes inside these light-gray boxes denote GitHub events (e.g., comments). We highlight three pieces of information: the developer who created the event, related commits, and related issues. Commits are highlighted by the commit hash and issues by "#" and the GitHub issue number. Regarding the contribution layer each black dot represents a commit. We highlight four pieces of information: the file name, the changed lines, the commit author, and the commit hash. Chunks in conflict are evidenced by the exclamation icon. The contribution layer (bottom) illustrates a merge scenario involving four developers of which DevA and DevC were fixing a bug while DevB and DevD were adding a new feature to the project. Developers changed four chunks of code of two files (File1 and File2). Three of these four chunks give rise to merge conflicts. Merge conflict is a notorious problem in collaborative software development [203]. Before we discuss the problems when integrating concurrent code changes, we discuss software communication on collaborative software development.

2.3 Communication Flow

As software development often requires social interaction, it is no surprise that software engineers spend a large part of their workday communicating with co-workers [25]. Numerous studies highlight the importance of communication. For instance, Souza et al. [271] provide evidence that communication among contributors is required for the success of software projects. Bird et al. [34], Grinter et al. [122], and Sedano et al. [256] stress that the lack of communication is a critical problem in distributed software development. So, if communication is uncertain, inaccurate, or slow, misunderstandings among developers become likely, compromising the project budget and schedule. In this sense, a proper communication culture is fundamental for stakeholders being aware of the project progress.

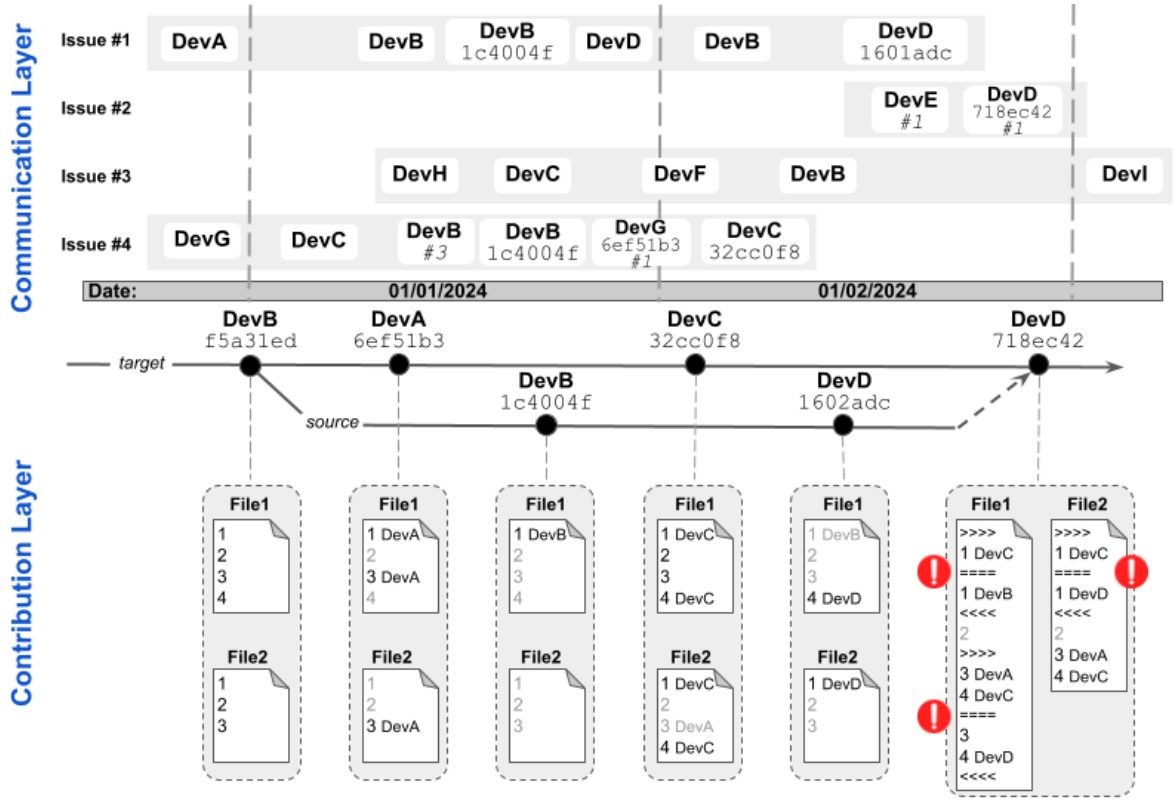

Figure 2.1: Communication and Contribution Layers of a Merge Scenario

Communication channels play an essential role in supporting communication and collaboration activities within a community of practice [280]. Various researchers have investigated developer collaboration through communication channels and tools, such as mailing lists, Internet Relay Chat (IRC) logs, issue trackers, and social networks (e.g., GitHub and Stack Overflow) [36] [75] [115] [124] [158] [172] [188] [229] [266] [296]. For instance, Bird et al. [36] explored the relationship between communication structure and code modularity. They found a relation between communication and code collaboration behavior for subcommunities. Guzzi et al. [124] analysed a large sample of e-mail threads from Apache Lucene's development mailing list. They found that developers participate in less than 75% of the threads, and in only about 35% of the threads source-code details are discussed. LaToza et al. [172] interviewed eleven developers to learn about common practices in software development. They found several barriers preventing e-mail usage and they highlighted advantages of face-to-face communication and that the use of more interactive communication channels is more desirable than e-mails. Panichella et al. [229] analysed three communication channels (mailing lists, issue trackers, and IRCs) and code changes of seven projects. They found that not all developers use all communication channels, and socio-technical relationships may change when using different communication channels and tools.

In an extensive study, Storey et al. [280] mapped different communication tools, such as e-mail lists, IRCs, SourceForge, GitHub, and Stack Overflow. They hypothesised that knowledge in software engineering is embedded in: (i) people's heads, (ii) project artefacts,

(iii) community resources, such as forums, blogs, and discussion groups, and, (iv) social networks. According to their study, GitHub is the only tool able to represent (the last) three types of knowledge.

The popularity of pull-based development models and GitHub has attracted the interest of researchers. For example, Singer et al. [266] and Dabbish et al. [75] explored the value of social mechanisms in GitHub. Both studies found that transparency helps developers to connect, collaborate, create communities, share knowledge, and discover new technologies. Tsay et al. [296] analysed the association of various technical and social measures with the likelihood of contribution acceptance. They found that PR acceptance is related to: (i) the strength of the social connection between the submitter and the project manager, (ii) the submitter's prior interaction, (iii) the number of comments, and (iv) the current stage of the project. Gousios et al. [115] analysed millions of PRs to study the effectiveness and efficiency of contributors handling PRs. They discovered that the time to merge a PR is influenced by the developer's previous track record, the size of the project and its test coverage, and the project's openness to external contributions. Liu et al. [188] conducted a quantitative study on the specific effects of PRs in the project. They found that PRs helps increase the social impact, resulting in more coordinated development activity.

In Figure 2.1, we illustrate the communication of 9 developers (Dev A, Dev B, Dev C, Dev D, Dev E, Dev F, Dev G, Dev H, and Dev I) distributed into four GitHub Issues. A GitHub issue is a way developers describe tasks that need to be done in the software repository. Hence, developers can assign themselves to these issues (i.e., tasks) and fix a bug or introduce new features, for instance. Issues can be of normal type or pull-request. Issues might have events, labels, commits, or other issues related to it. Events are changes in the state of an issue or commentaries added, for instance. Labels are tags added to an issue. Commits are commit hashes linked to an issue, and other issues are references to other GitHub issues indicating that these issues are somehow related. Note that in the illustrated example, developers communicate in a 4-day time-frame and multiple tasks are running at the same time. For instance, Dev A reported an issue (Issue #1) on August 1^{st} and Dev B and Dev D started working on this issue in the following day and finished working on it on August 3^{rd}. At the same time, these developers worked on other issues and communicated with other developers. We highlight that developers can also refer to commits and issues in the comments of the issues they are working to make others aware of concomitant tasks. For instance, in Issue #4, Dev G refereed to issue #1 and commit hash 6ef51b3. Next we discuss the problem at the end of the merge scenario.

2.4 Characterising Merge Conflicts

Merge conflicts occur during the integration of changes made by one or more developers in different branches. Therefore, by this definition we see that the number of developers, code changes, and branches might influence the number of merge conflicts as well as the location of the change. In this section, we show the results of studies that investigate the characteristics of merge conflicts.

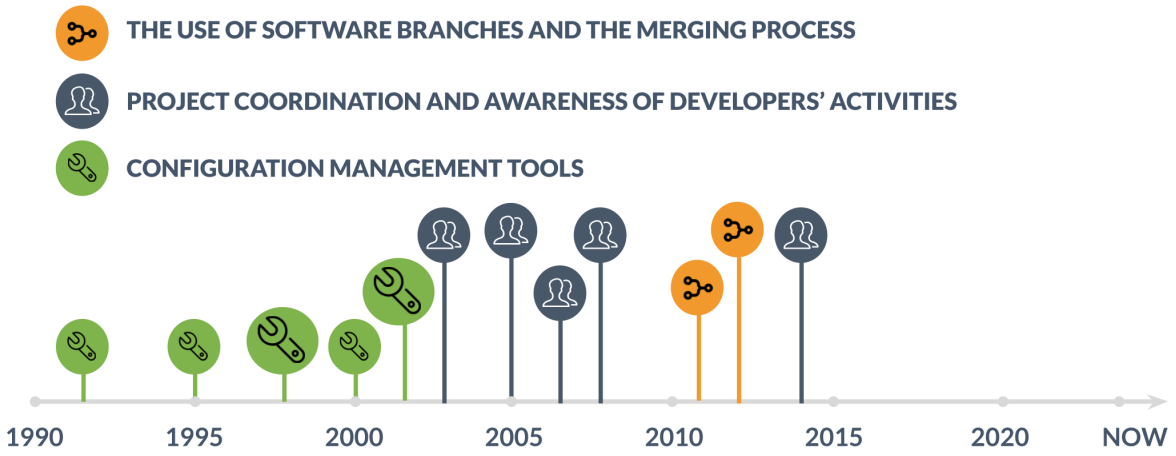

Figure 2.2: Merge Conflict Studies showing Negative Impact Timeline

2.4.1 Injury and Loss Caused by Merge Conflicts

Studies investigating the impact of merge conflicts in software development follow three main lines: (i) to understand configuration management tools, (ii) to understand project coordination and awareness of developers' activities, and (iii) to understand the use of software branches and the merging process. In Figure 2.2, we see the distribution of studies showing the impact of merge conflicts distributed into these three groups.

Configuration management tools. The first studies mentioning merge conflicts came from qualitative studies [72, 120, 121, 136, 137, 232, 233] trying to understand the configuration management tools. For instance, in two studies Grinter [120, 121] conducted a three month on-site and conducted over 100 interviews to capture how parallel development involving multiple developers working on the same module at the same time works. As a result, she found that: (i) the difficulty of coordinating the efforts of multiple developers lead to code integration problems [121] and (ii) developers avoid parallel development because of the potential complexities of merging [120]. Perry et al. [232, 233] conducted an observational case study in which they analysed the change and configuration management history of a legacy system to understand the problems encountered in parallel development. As a result, they found that the number of defects rises as the amount of parallel work increases.

Project coordination and awareness of developers' activities. Once configuration management tools and collaborative software development were better understood, researchers started investigating project coordination and awareness of developers' activities [97, 99, 270, 272, 274]. In a set of studies, Souza et al. understood the process of migrating between private and public work [272], how developers manage dependencies and changes [274], and which developers should be aware of project changes [270]. Their main findings related to merge conflicts are that: (i) merge conflicts displaces developers from their workflow; (ii) merge conflicts saddles developers with the potentially high complexity cognitive task of understanding another developer's changes; (iii) merge conflicts force developers into making code changes that may introduce bugs, which can be especially true for novice developers; and, (iv) merge conflicts make developers rush their work into the trunk to avoid being the developer who would have to resolve conflicts. A couple of years later,

Estler et al. [97] investigated the impact of awareness in the context of globally distributed software development. Based on an analysis of data from 105 student developers, they found that: (i) the likelihood of incurring into merge conflicts is not significantly affected by the location (co-located vs. remote) of developers within the same team and (ii) merge conflicts impact negatively the team productivity, motivation, and keeping the schedule.

The use of software branches and the merging process. In another moment, researchers started to understand the use of branches and the merging process [37, 234]. Bird and Zimmermann [37] surveyed 124 Microsoft engineers to understand the use of branches. They highlighted that the process of integrating changes from multiple branches can be difficult and error-prone, especially if changes on different branches conflict, either syntactically or semantically. In addition, this process takes time, which can slow teams on different branches that are dependent on each other or features which are related. Thus, branches incur an overhead in both developer effort and time, which, if not monitored and managed, can have severe impact on the project in the form of missed deadlines and increased failures. Phillips et al. [234] conducted a survey among 140 version control users and asked how branching and merging are used in practice and what defines a successful branching strategy in terms of user satisfaction. Their results show that 54% of the respondents think that the most significant problem about merging is conflict. Three respondent comments show their dissatisfaction with the merging process: (i) "*Merge conflicts are tedious to fix manually and often the mechanic[al] approach makes one miss the one crucial change that should have been kept*"; (ii) "*Usually in non source code file[s] the merge conflicts are a pain*"; and (iii) "*Code complexity and the project size can make the resolution process quite complex, especially when the platforms changes [sic]*".

In a nutshell, we see that merge conflicts are costly, a social issue, and normally related to coordination problems. The classification in the three groups (Figure 2.2) supports us seeing that researchers' concerns have changed over the years following the most recent tools to support software development. The diversity of studies and the absence of recent studies shows that the negative impact of merge conflicts is well-understood from different kinds of studies and it is a solid problem. As we are going to see in the next sections, researchers have investigated other topics related to merge conflicts instead of proving its negative impact on software development.

2.4.2 Types of Merge Conflicts

Researchers have categorised conflicts related to how it is identified (i.e., version control system, build failure, test failure, or production issue [49, 284, 311]), the type of inconsistency (i. e., textual, syntactic, semantic, and behavioural [74, 311, 318]), or simply differentiating textual conflicts from other conflicts (e.g.,lower-order vs higher-order conflicts [311], direct vs indirect conflicts [2, 251], first-level vs second-level conflicts [49]). Note that proximity of concurrent modifications is an important factor distinguishing the first- and second-level conflicts.

In Figure 2.3, we show a taxonomy of the types of conflicts found in the literature. *Textual conflicts* refer to concurrent code changes among the merged branches (normally from a line-based analysis). This type of conflict is normally identified by version control systems

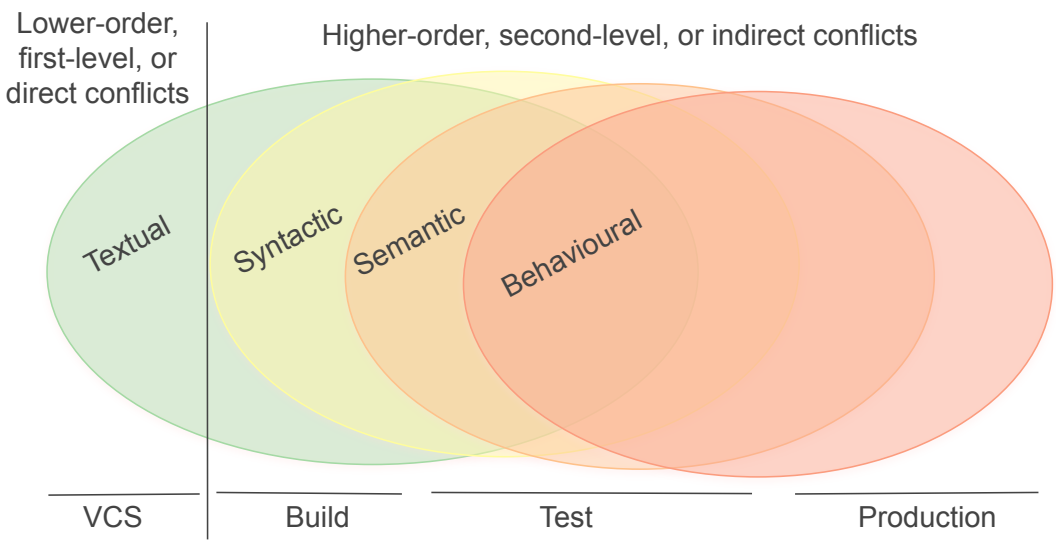

Figure 2.3: Merge Conflict Taxonomy

and are considered simpler to resolve than the other types of conflicts. *Syntactic conflicts* comes from concurrent changes and result in structural failures (e.g., the wrong order of words and symbols or missing a closing bracket). Most syntactic conflicts are found when building the project. *Semantic conflicts* are due to changes that alter the meaning of a statement (e.g., changing a keyword or renaming variables). *Behavioural conflicts* are similar to semantic conflicts, but are related to issues sensitive to the final user and cannot be statically identified (e.g., a button in a web application performing an action that does not match with its description). Most semantic and behavioural conflicts are identified in the review process or testing the project. When previous checks fail, issues from concurrent changes, often manifesting as bugs, are found on production.

While textual and syntactic conflicts are caused by modifications in close proximity to each other, code modifications responsible for semantic and behavioural conflicts can also be “far apart" [311]. Most studies related to merge conflicts investigate textual conflicts and consider these terms as synonyms [74]. In this study, we use the terms defined and presented in Figure 2.3.

Three studies [41, 216, 249] mix some previous definitions, defining conflicts at a finer-grained level. It made it hard to include all of their conflict types in our taxonomy presented in Figure 2.3. Hence, to keep this figure simple, we describe their conflict types as follows.

Nguyen et al. [216] classify conflicts as content conflicts, remove/update conflicts, and naming conflicts. *Content conflicts* are conflicts inside a file. *Remove/update conflicts* come from concurrent removals and updates of a file. *Naming conflicts* are related to renaming the same file or of two files with the same name. Note that their definition is not only related to textual/low-order conflicts. Naming conflicts will not be identified by a version control system that is line-based. We consider it closer to language constructs in merge conflicts and not types of conflicts (see Section 2.4.6).

Santos et al. [249] classify conflicts as direct, indirect, and pseudo conflicts. *Direct conflicts* represent a pair of code changes applied to the same code elements (e.g., attributes and methods). They can also be called structural conflicts. *Indirect conflicts* occur when code

changes applied to the source system are in the call graph of other code changes in the target system. They can also be named semantic conflicts. *Pseudo conflicts* occur when developers working on the source and target branches modify the same class or interface, but different and independent code elements (attribute or method). The authors also call them lexical conflicts. In our classification they are semantic or behavioural conflicts.

Brindescu et al. [41] classify conflicts into six types: semantic, disjoint, delete, formatting, comments, other. *Semantic conflicts* are two conflicting semantic changes where two different changes in the program logic overlap. *Disjoint conflicts* are semantically unrelated changes that overlap textually. *Delete conflicts* are conflicts in which one of the branches deletes code modified on the other branch. *Formatting conflicts* are conflicting changes due to formatting (white-space changes). *Comment conflicts* are conflicting changes limited to comments only. *Other conflicts* are conflicts not belonging to any of the other conflict types. Note that formatting and comment conflicts can be a subset of syntactic conflicts since there is no change in the program logic. Semantic, disjoint, and delete conflicts often change the logic of a program and we would classify them as semantic or behavioural conflicts.

It is important to highlight that, similar to recent work [257, 284], we consider build failures, test failures, and production issues ways practitioners identify the conflicts. They are not types of conflicts. A classification differentiating merge conflicts into two groups (e.g., low-order and higher-order conflicts) is too simplistic. Hence, our taxonomy is composed of four types of merge conflicts: textual, syntactic, semantic, and behavioural conflicts. Brindescu et al. [41] suggest classifying conflicts in the most “severe" category to properly prioritise them. Severe conflicts (i.e., semantic and behavioural conflicts) require reasoning about the goals of the changes and the best way to integrate them. Hence, they also require more from the developer who is resolving the conflicts as we discuss in Section 2.8.

2.4.3 Conflict Rate

Several studies [1, 2, 5, 11, 41, 49, 50, 60, 82, 110, 154, 163, 180, 192, 199, 200, 216, 217, 249, 257, 265, 317, 322] reported the rate of conflicting commits among merge commits (also known as conflict rate). This rate varies from 0.00% to 87.84% depending on aspects like the conflict type, version control system, merging strategy, number of projects, projects domain, programming language, coordination practices, contribution rules, and etc. In this Section, we show the conflict rate based on how researchers identified them (see Section 2.4.2). We found conflicts identified by the version control system, build failures, and test failures. We did not find any reports of conflicts causing production issues. In addition, we highlighted some factors that might have influenced the conflict rates in different studies and for each identification type.

Version control systems. As seen in Figure 2.3, merge conflicts identified by common version control systems using simple diff algorithms are textual conflicts. Several empirical studies investigating a great number of projects [2, 60, 82, 110, 180, 192, 199, 200] found the conflict rate between 8 and 14%. For instance, Accioly et al. [2] conducted an empirical study reproducing 70047 merge scenarios from 123 Java projects hosted on GitHub. Their results show that merge conflicts happened in 9.38% of the analysed merges, with a median of 6.64%, and an Inter-Quartile Range (IQR) of 8.81%. Once they looked at projects individually,

the conflict rate varied from 0.00% to 42.21%. These studies show that the general conflict rate is about 1 for each 10 merge scenarios. However, some projects are neglecting coordination practices and having conflicts in 1 out of 3 merge scenarios.

Our initial thoughts were that studies published several years ago would have conflict rates greater than recently published studies since over the years coordination practices and tools were incorporated in the development process. In fact, in 2007, Zimmermann et al. [322] analysed data of four large Open-Source Software (OSS) projects (GCC, JBoss, JEdit, and Python). They found that the conflict rate is between 22.75% (for GCC) and 46.62% (for JBoss). Later in 2013, Kasi and Sarma [163] also analysed four popular OSS projects (Perl, Storm, Jenkins, and Voldemort). They found a conflict rate between 7.6% (for Perl) and 19.3% (for Storm). More recently in 2018, Nguyen et al. [216] also analysed four popular OSS projects (Rails, IkiWiki, Samba, and Linux Kernel). They found conflict rates varying from 4.86% (for Linux Kernel) to 87.84% (for Samba). The main assumption of the authors to justify the huge conflict rate for Samba is their integration model. While the Linux Kernel uses a pull-based model via mailing-list, Samba uses a shared repository among registered contributors. An interesting finding is that the integration rate is not positively related to the conflict rate. In other words, integrating the project often is not related to high conflict rates. We discuss it when presenting strategies to avoid merge conflicts in Section 2.7.1.

Build Failures. A few studies [1, 49, 50, 163, 257, 265] investigated merge conflicts causing building failures. We will name conflict rate, but in this case we mean the number of merge scenarios failed due to build failures. The conflict rate vary from 0.13% to 14.7%. We describe each of the studies as follows. Brun et al. [49, 50] analysed nine popular OSS projects (Gallery3, Git, Insoshi, jQuery, MaNGOS, Perl5, Rails, Samba, and Voldemort), however, for identifying build failures, they had data for only three projects (Git, Perl5, and Volvemort). Their conflict rate varies from 0.15% (for Git) and 10% (for Voldemort). As the number of merges for Git is much greater than the other two projects (1 362 against 185 and 147 for Perl5 and Voldemort, respectively) the average conflict rate for these three projects is 1.42%. Kasi and Sarma [163] analysed four popular OSS projects (Perl, Storm, Jenkins, and Voldemort). They found a conflict rate between 2.1% (for Perl) and 14.7% (for Jenkins). Their average conflict rate is 8.89%. Acciloy et al. [1] analysed 64 445 merge scenarios from 422 GitHub OSS Java projects using Travis CI and Maven. They found a conflict rate of 0.13%. Silva et al. [265] analysed 451 GitHub OSS Java projects using Travis CI. They found a conflict rate of 7.50%. Finally, Seibt et al. [257] analysed 7 727 merge scenarios from 10 GitHub OSS Java projects. They integrated merge scenarios using three merging strategies (unstructured, semi-structured, and structured) and possible combinations of them (e.g., first merging with an unstructured strategy, then a semi-structured strategy, and finally a structured strategy). We detail merging strategies in Section 2.5. Their conflict rate due to build failure vary from 1.79% (using an unstructured strategy) and 2.09% using the sequence of merging with the unstructured, semi-structured, and structured merging strategies.

Note that the conflict rate due to build failures is sparse. We have studies reporting conflict rates close to 0% and more than 7%. The two studies with a large dataset that differ mostly the conflict rate presented reasons for this difference in their conflict rates. Acciloy et al. [1] justified such contrasting numbers by three main reasons. First, the previous studies [49, 50, 163] rely only on the merge commit build status. They do not consider

parent commit build status. This way, false positives might have been introduced. Second, because previous studies perform build and tests locally, some part of errored and failed builds might have been caused by external or configuration problems, for example, due to unsolved project dependencies. Third, their decision of analysing merge commits that occurred after the project has adopted Travis CI might have impacted the conflicts frequency. According to previous studies [319], the adoption of Continuous Integration (CI) practices help to maintain the code quality. This is so because, when a project adopts CI practises it uses automated scripts to run build and testing. Thus, the developer responsible for the integration might be detecting and resolving most part of the conflicts locally, before pushing changes to the shared repository. Such perception seems to be aligned with previous empirical evidence [212] that broken builds occur more frequently in regular commits than in merge commits.

Silva et al. [265], on the other hand, justify that the high conflict rate of broken merge builds is due to four main causes: (i) build environment (Travis timeouts, unavailability of external services such as package manager servers, bugs in build scripts, and configuration problems) issues; (ii) integration conflicts; (iii) defects in post-integration changes; and (iv) defects, and consequently breakages, inherited from the parents. They reported that most breakages are due to the first cause. In addition, they suggested that build conflicts occur during software development, but many conflicts do not reach the remote repositories as developers fix them before sending their contributions. They also reported that more than one build conflict may occur in a merge scenario as the causes for these conflicts are independent. Finally, they argue that their conflict rate should be interpreted as conflict occurrence rates that reach public repositories and, therefore, are more problematic.

Test failures. Basically the same studies [1, 49, 50, 163, 257] that investigated conflicts due to build failures evaluated conflict due to test failures. Silva et al. [265] work is the only exception since their focus is on the understanding of build failures. Before showing the conflict rate, we highlight that conflicts due to test failure are highly dependent on the type of tests and test coverage of the subject projects. As the studies did not report the test coverage of their subject project it is even harder to compare their results.

Similar to conflicts due to build failures the conflict rate due to test failures is sparse. It varies from 0.008% to 35.00%. Acciloy et al. [1] obtained a conflict rate due to test failures of 0.008%. Their justifications for the low conflict rate is the same presented to build failures (see topic above). Similarly, Seibt et al. [257] also obtained low conflict rates due to test failure varying from 0.47% (using an unstructured strategy) to 0.71% (using the sequence of merging with the unstructured, semi-structured, and structured merging strategies). The other studies got higher conflict rates. For instance, Brun et al. [49, 50] got conflict rates of 3%, 4%, and 28% for Voldemort, Git, and Perl5, respectively. Their average conflict rate is of 6.43%. Kasi and Sarma [163] found conflict rates varying from 5.60% to 35.00%. Their average conflict rate is 19.86%. Note that contrary to all other studies, Kasi and Sarma [163] found that conflicts due to test failures were found more often than conflicts due to build failures. The authors do not discuss the high conflict rate, but highlight the importance of identifying test failures as soon as possible.

Other conflict percentages. The three studies presented in Section 2.4.2 reported the conflict rate for their types. Nguyen et al. [216] presented the conflict rate for each analysed project. For short, most of the conflicts are content conflicts, followed by naming conflicts

and remove/update conflicts. For instance, 89.68% of the conflicts in Rails are content conflicts, 7.35% are naming conflicts, and 2.97% are remove/update conflicts. Santos et al. [249] showed that the conflict rate that has only direct conflicts is 5.6%. The conflict rate of issues holds no direct conflicts but presents at least one indirect conflict is of 18.83%. Finally, they found that 10,34% of issues have only pseudo conflicts. Brindescu et al. [41] reported the percentage of each conflict type they proposed. Semantic conflicts are the one dominating the conflicts (59.46%) followed by formatting conflicts (23.21%), disjoint conflicts (14.53%), delete conflicts (1.24%), other conflicts not classified (0.96), and comment conflicts (0.60%).

2.4.4 Conflict Size

We found two studies [110, 228] investigating the merge conflict size in terms of chunks and lines of code. Other studies presented some size measures when investigating other merge conflict characteristics. For instance, Mahmood et al. [192] found that most merge conflicts were due to minor changes in terms of size, e.g., the addition of parameter values. In their analysis, 28 out of 40 conflicts only consisted of one-line changes. Only twelve of the conflicts were associated with changes involving two or more lines.

Looking at the number of chunks of merge conflicts, Ghiotto et al. [110] found investigating 2731 Java projects that 40% of the failed merges have a single conflicting chunk and 90% have 10 or fewer. The remaining 10% had 11 or more chunks, with the maximum an astounding 10315 chunks (merge 7a9c34 of project Jnario, which was the result of a conflict between a set of feature enhancements and refactorings on one branch and a major framework upgrade on the other branch). They found that 62% of the conflicting files have just one conflicting chunk and 95% of the conflicting files have five or fewer conflicting chunks. Just in rare cases (less than 0.4%), an individual file has more than 20 conflicting chunks.

Regarding the number of lines of code involved in merge conflicts, Pan et al. [228] found investigating the Microsoft Edge project that 35.17% and 57.74% of the conflicts in C++ are 1-2 lines of change by the main and forked branches, respectively. Overall, 28% of the conflicts are of 1-2 lines for both main and forked branch. Looking at a huge number of conflicts, Ghiotto et al. [110] found that 94% of the conflicting chunks have up to 50 Lines Of Code (LOC) in each version, 68% have up to 10 LOC in each, and slightly over half (50%) 5 or fewer. At the other end of the spectrum, 0.05% of the chunks have more than 2000 LOC in each version, with the extreme case involving 13035 and 14074 LOC, respectively (merge 310dbe of project ThingML). Across all 175805 conflicting chunks, 4147 (2%) involve more than 50 LOC in both versions, while 6042 (3%) have more than 50 LOC in one version and less than 50 LOC in the other. Further examination indicates that a subset of projects is responsible for these excesses. While 700 projects (26%) have at least one chunk with both versions involving more than 50 LOC, only 95 projects (3%) have more than ten such chunks and just four projects have over a hundred such chunks. The average size across all projects of the source version is 19.5 LOC and 27.6 LOC for the target branch, with average standard deviations of 20.6 and 28.6 LOC, respectively, and median size of 2.0 and 2.5 LOC. Overall, the numbers show that some large chunks drag the mean size upwards and increase the

standard deviation, despite the majority of chunks having less than three lines of code in one or both versions.

For short, we see that merge conflicts are often small with an average of up to 28 LOC and median size of up to 3 LOC. There are some outliers with thousands of LOC, but 94% of the conflicting chunks have up to 50 LOC in each version.

2.4.5 How Long Conflicts Last

We found three studies investigating the time conflicts last. Brun et al. [49] investigated nine projects (mentioned in Section 2.4.3) and found that merge conflicts resulted from textual changes persisted for 9.8 days (with median values of 1.6 days) and conflicts from a higher order (e.g., break build or tests) lasts for 11 days on average (with median values of 1.9 days). Kasi and Sarma [163] analysed four popular OSS projects and found that textual conflicts lasted from 6 to 26.51 days (median in the projects varying from 2 to 10 days). Build conflicts last from 0.75 to 5 days (median from 0.75 to 8). Test Conflicts last from 6.01 to 30.5 days (median from 2 to 14 days). Brindescu et al. [43] found by an in-site observation of 7 developers that conflict resolution took from 40 to 2 190 seconds (36.5 minutes).

2.4.6 Language Constructs in Merge Conflicts

We found eight studies [1, 2, 110, 192, 261, 265, 311, 317] investigating language constructs in merge conflicts, as described next.

Yuzuki et al. [317] analysed the characteristics of conflicts at the method-level. Using the version history of 10 Java projects, they found that 44% of conflicts were due to concurrent changes (edits in the same part of the method made by two or more developers), 48% to removing methods in their entirety, and 8% to renames.

Accioly et al. [2] study conflict characteristics to understand patterns of conflicts, their causes, and extents in real-world scenarios. Specifically, they analyse merge conflicts in 123 OSS Java projects by replaying merge scenarios using the semi-structured merge tool FSTMerge [11], leading to nine conflict patterns: (i) edit the same method or constructor (*EditSameMC*), (ii) edit the same field declaration (*EditSameFd*), (iii) same signature and different bodies for methods or constructors (*SameSignatureMC*), (iv) add the same field declarations (*AddSameFd*), (v) different edits to the modifier list (*ModifierList*), (vi) different edits to the same implements declaration (*ImplementList*), (vii) different edits to the same extends declaration (*ExtendsList*), (viii) different edits to the same enum constant declaration (*EditSameEnumConst*), and (ix) different edits to the same annotation method default value declaration (*DefaultValueA*). *EditSameMC* was, by far, the most frequent conflict pattern, representing 84.57% of the collected conflicts. The second most frequent pattern was *EditSameFd* (5.46%), followed by *SameSignatureMC* (5.21%), *ModifierList* (3.53%), *AddSameFd* (0.47%), *ExtendsList* (0.35%), and *ImplementList* (0.28%), *EditSameEnumConst* (0.12%), and *DafaultValueA* ($\approx$ 0.00%). A percentage of 28.97% of the collected conflicts were classified in one of these categories. More specifically, 48% of the false positives were due to different spacing, 37.69% due to consecutive line edit, and the remaining 14.31% were due to both

reasons. Regarding the kinds of code changes most likely related to conflicts, *EditSameMC* is by far the one more likely of leading to merge conflicts, the other measures have probabilities lower than 0.1%.

In a follow up study, Accioly et al. [1] present 2 predictors (*EditSameMC* and *EditDepMC* - editions to directly dependent methods) that can be used to identify potential merge conflict ahead of time. Their results indicate that, considering both conflict predictors together, they achieved a precision of 57.99%. In particular, *EditSameMC* individual precision is 56.71%, and *EditDepMC* precision is 8.85%. Moreover, they achieved a recall of 82.67% if considering both predictors together, while *EditSameMC* individual recall is 80.85% , and *EditDepMC* recall is 13.15%. They conclude that with their current form, high numbers of erroneous predictions make these techniques unsuitable for industry adoption.

Ghiotto et al. [110] analyse merge conflicts in 2731 OSS Java projects (see Section 2.4.4). Regarding the language constructs related to merge conflicts, the top-10 language constructs found in their automated analysis are (ordered by frequency): (i) *method invocation* (20%), (ii) *variable* (17%), (iii) *commentary* (14%), (iv) *if statement* (8%), (v) *import* (6%), (vi) *method signature* (6%), (vii) *method declaration* (5%), (viii) *annotation* (3%), (ix) *return statement* (3%), and (x) *for statement* (1%). Half the conflicting chunks have a single language construct, 72% have one or two constructs, and 90% have up to four constructs. Deeper investigating patterns between the language constructs of conflicting chunks (e.g., method invocation) and developer decisions (e.g., source version), Ghiotto et al. [110] presented the top 10 association rules ordered by the lift: (i) method invocation, method signature, return statement, and try statement → new code (2.39), (ii) method declaration, method invocation, method signature, try statement, and variable → new code (2.36), (iii) method declaration, method signature, try statement, and variable → new code (2.36), (iv) method declaration, method invocation, try statement, and variable → new code (2.32), (v) method declaration, try statement, and variable → new code (2.32), (vi) method signature, return statement, and try statement → new code (2.31), (vii) comment, do statement, and if statement → source version (1.89), (viii) comment, do statement, if statement, and method invocation → source version (1.89), (ix) comment, do statement, if statement, and variable → source version (1.89), (x) comment, do statement, if statement, method invocation, and variable → source version (1.89). Their results show a prominence of method-related constructs, as co-occurring with comments, control flow statements, and references to variables. Furthermore, they noticed that all rules are resolved either through the addition of new code or by picking the source version. The persistence of method-related rules points to the fact that it could be beneficial to focus the design of new merge techniques on such constructs.

Wuensche et al. [311] propose an approach to detect higher-order conflicts in large software projects. Their approach searches for potential conflicts between changes that are undergoing pre-submit testing. To this end, it constructs a static call graph (not considering inheritance and dynamic call dispatching) and searches for dependencies between changed code entities. To evaluate their approach, they created a prototype and used it on SAP HANA, a large industrial product developed in C++, during a 22 month time-span. To identify the cause of merge conflicts, they consider four cases: *signature changed*, *include statements*, *duplicate definition*, and *unknown* (e.g., incomplete documentation or missing the conflicting merges or fix). As a result, changes in the signature were responsible for 50% of the build

conflicts, including statements for 7%, duplicated definitions for 6% and the remaining 37% of the build conflicts were due to unknown causes.

Mahmood et al. [192] investigate changes that led to merge conflicts using ElasticSearch, a large OSS project, as case study (see Section 2.4.4). They identified 534 merge conflicts that cannot be resolved automatically and selected 40 to deeply look at them. As a result, they found six merge conflict category: (i) *change of method call or object creation* (MC_OC), (ii) *change of an assert statement expression* (AS_EXP), (iii) *addition of statements in the same area* (ADD_STMT), (iv) *modification and removal of statements* (MOD/RMV_ STMT), (v) *changes in different statements in the same area* (D_STMT), and (vi) *changes of if statement condition* (IF_C). They found that most merge conflicts were due to minor changes in terms of size. For instance, the addition of parameter values. 28 out of 40 conflicts only consisted of one-line changes. Only twelve of the conflicts were associated with changes involving two or more lines. Note that the size of a change leading to a merge conflict does not imply that the conflicts is minor. The resolution of such "small" merge conflicts can require larger changes as well. 25 out of 40 conflicts are a result of changes made in MC_OC cause for conflicts. The second most common cause for conflicts is ADD_STMT. Furthermore, all code-level changes leading to conflicts occurred within method bodies. This confirms a finding by Accioly et al. [2], who found that most merge conflicts (85%) are due to changes that are made in method bodies.

Shen et al. [261] investigated 204 merge conflicts from 15 OSS repositories (100 textual, 100 build (syntactic) conflicts, 4 dynamic (semantic) conflicts) aiming at better understanding how conflicts were introduced. As a result, they found that 51% of textual conflicts were caused by the contradictory statement updates between branches. 93% of compiling conflicts occurred when one branch adds one or more references to a program entity that is updated, removed, or replaced in another branch. 75% of dynamic (semantic) conflicts happened because the test oracle added by one branch does not correspond to the code implementation updated by the other branch. The observed dynamic conflicts were introduced for two reasons: (i) the inconsistent changes between code implementation and test oracle, and (ii) the inconsistent maintenance of the same class hierarchy.

Silva et al. [265] investigated the frequency, structure, and adopted resolution patterns of 239 build conflicts from 451 OSS Java projects. Their catalogue of build conflicts has six causes: (i) *unavailable symbol* (reference for a missing symbol), (ii) *incompatible method signature* (unmatched method reference), (iii) *project rules* (unfollowed project guidelines), (iv) *incompatible types* (type mismatch between expected and received parameters), (v) *unimplemented method* (method from super type or interface), and (vi) *duplicated declaration* (elements with the same identifier). The most recurrent cause is unavailable symbols, which can be split into sub-categories, like unavailable symbol class, method and variable. These build conflicts are caused not only by static semantic problems but also static analysis performed after the compilation phase during the build process.

2.4.7 Tools to Build Merge Scenarios and Identify Merge Conflicts

There are several studies [2, 110, 114, 180, 225, 276] that automatically rebuild merge scenarios aiming at identifying merge scenario- and merge conflict-characteristics. However, not all of them propose a tool or framework, they just share the source code [2, 110]. In this section, we focus on studies that indeed give a name to their tool/framework.

GHTorrent [114] is an offline mirror of GitHub that allows users to either download the data as a SQL or MongoDB database, or run their queries online. It is not that hard to link *GHTorrent* data with contribution data (i.e., Git historical data), however, one might relate contribution data taking the three-way merge development pattern into account. It requires much more effort since historical data might be stored taking the timeline of the events.

PyDriller [276] is a recent tool for analysing Git history to extract data such as commits, developers, source code, etc. GitMiner[1] is an open-source tool that stores data extracted from Git and GitHub in a database. Although both PyDriller and GitMiner can be used to detect merge commits by selecting commits with more than one parent, neither provides any additional option to analyse merge scenarios or merge conflicts. GrimoireLab[2] is an industrial-level tool that is capable of gathering data from version control systems, issue trackers, mailing lists, wikis, and among other sources. However, it does not contain any tooling for analysing merge scenarios.

MERGANSER is a scalable tool that extracts merge scenarios and merge conflicts data from Git repositories [225]. For a given a list of repositories, the tool: (i) clones the repositories, (ii) extracts repository meta-data such as the number of stars and forks, (iii) detects merge scenarios, (iv) extracts the information about the merge scenarios such as the number of developers involved or development duration, (v) replays the merge scenarios to detect conflicting files and regions, and (vi) stores all the above information in a SQL database.

As seen, we have found tools that retrieve communication/social perspective, such as GHTorrent, and tools that mines the contribution perspective, such as MERGANSER. To the best of our knowledge, there is no tool to integrate technical and social perspectives. This integration might support studies investigating the influence of social factors on technical factors and vice-versa. This kind of study provides a broader view of software development evolution of OSS repositories and might bring new findings and directions to studies on software coordination, especially when investigating the life-cycle of merge conflicts.

2.5 Merge Strategies

In this section, we discuss merging techniques (Section 2.5.1), algorithms (Section 2.5.2), and tools (Section 2.5.3) proposed to deal with software repositories. Later, in Section 2.5.4, we present studies that compare merge strategies and do not propose any strategy.

Before investigating merging techniques, algorithms, and tools, it is important to understand what is going to be compared. Merging techniques vary between two-way and

1 https://github.com/UnkL4b/GitMiner
2 https://chaoss.github.io/grimoirelab/

three-way merge techniques. *Two-way merging* attempts to merge two versions of a software artefact without relying on the common ancestor from which both versions originated. With *three-way merging*, the information in the common ancestor is also used during the merge process. This makes three-way merging more powerful than two-way merging. Almost all current available merging techniques, algorithms, and tools make use of three-way merging.

2.5.1 Merging Techniques

Merge strategies often give rise to an abundance of conflicts being reported. There are a number of ways to reduce the number of reported conflicts to a more manageable number. Mens [203] mentioned that researchers and practitioners should take 6 criteria into account when proposing or choosing a merging technique. The criteria are: (i) granularity (characters, words, phrases, sentence, paragraph, lines, intra-procedural merging), (ii) degree of formality (*ad hoc* vs fully-formal), (iii) domain independence versus customizability, (iv) accuracy (false positives), (v) scalability and efficiency (performance and memory usage), and (vi) degree of automation (manual, interactive, fully automatic).

When thinking about merging techniques, researchers normally define them as *text-based*, *syntactic*, and *semantic* merging techniques. There is also a similar classification taking unstructured merge, semi-structured merge, and structured merge techniques into account. Even though the link between these two classification does not perfectly match, we structured this section according to these three merging techniques.

Text-based or Unstructured Merging Techniques

Text-based merging techniques consider software artefacts merely as text files (or binary files). The most common approach is to use line-based merging, where lines of text are taken as indivisible units. Example of this are the RCSMERGE in the Revision Control System (RCS) [293], FILEMERGE [4], Domain Software Engineering Environment (DSEE) [177, 178, 189], the CVS [28], and merge tools that can be found in commercial configuration management tools. A few studies before the 2000s [146–148, 292] propose text-based techniques. Line-based merging has the disadvantage that it cannot handle two parallel modifications to the same line very well. Only one of the two modifications can be selected, but the two modifications cannot be combined. According to Perry et al. [232] and similarly seen in Section 2.4.3, about 90% of the changed files could be merged without problems. However, the degree of parallel changes is high – merge conflicts involved between 2 to up to 16 parallel changes. The more challenging task lies in providing automated ways to deal with those 10 percent of the situations that cannot be merged automatically. The reason text-based merging fails in those cases has to do with the fact that text-based merge techniques do not consider any syntactic or semantic information.

After the 2000s, text-based merging techniques started to be often called unstructured merging. It remains until nowadays the most widely used in software merge tools. Unstructured merging is widely used UNIX tools such as *Diff* and *Merge* and is used in the most popular version control systems, chiefly GIT and SUBVERSION (SVN). While the implementation is complex, the idea is straightforward: the algorithm considers software artefacts simply as a sequence of text lines. An algorithm inspired by *the longest common subsequence problem* [27] is used to identify changed blocks in the lines that make up the

versions to be merged. The algorithm walks through the changed blocks and applies a set of merge rules to either accept changes into the merge output or flag conflicts where the correct change to accept can not be decided.

As such, the algorithm is applicable to any file that can be interpreted as a sequence of text lines. This property and the very high runtime efficiency are the major selling points for unstructured merge. There is, however, a downside. The granularity of the algorithm is at the level of text lines; the fact that it is source code that is being merged is not taken into account. This leads to weaknesses in conflict resolution since the structure of the source code is not considered.

Syntactic or Semi-structured Merging Techniques

Syntactic merging techniques are more powerful than text-based merging because it also takes the syntax of software artefacts into account. A text-based often gives rise to unimportant conflicts such as code comment that has been modified in parallel by different developers or conflicts that arise because of the introduction of some line breaks and tabs that make the code easier to read. A syntactic merging tool can ignore all these conflicts. There are several studies proposing syntactic merging techniques before the 2000s [20, 38, 51, 118, 201, 232, 240, 306, 312, 315]. Westfechtel [306] and Buffenbarger [51] have pioneered in proposing merge algorithms which incorporate context-free and context-sensitive structures. Their syntactic-oriented approaches are language-independent, since language features such as alternatives, lists, and structures are represented at an abstract level. Later, a variety of approaches on syntactic diff and merge have been proposed.

Asklund [20] proposes to minimise the number of reported merge conflicts by using fine-grained revision control, where the software changes should be as small as possible. By evolving the software with small increments, the number of merge conflicts will remain small in each step. This idea is supported by Perry et al. [232], who have noticed on a statistical basis in a large-scale industrial case that making large changes tends to lead to parallel versions that cannot be merged without some very costly overhead and coordinated effort. In the end, these large changes are often related to merge conflicts.

Once thinking about syntactic and semi-structure merging techniques they might differ depending on how the technique works, since a semi-structure technique might consider part of the structure when merging. After the 2000s semi-structured merging techniques were proposed. Semi-structured merge targets the middle ground between unstructured (often text-based) and structured (often considering the semantic of the changes) merge algorithms. It is aimed at resolving as many of the conflicts that structured merging would resolve as possible while speeding up the merge procedure by using unstructured merging for parts of the source code. The key idea is that the parser producing the Abstract Syntax Tree (AST) is interrupted when a certain level (e.g., method declarations) is reached. The parser is extended with a new kind of AST node that represents the remaining source code (e.g., the method bodies) as blocks of text.

Semi-structured merges could be configured to stop at any level (e.g., class bodies, method bodies, bodies of loops or conditional expressions). In JDime, semi-structured merge uses structured merge for everything above and including the level of method declarations and unstructured merge for the bodies of methods and constructors, which has been successful in practical settings [11]. This leads to a significant speed-up compared to fully structured

merge, while retaining the ability to recognize and correctly match reordered methods and fields. In other words, the usual structured algorithms are used to merge the ASTs, but when leaves representing text blocks (e.g., method bodies) are matched or merged, the algorithm delegates to an unstructured strategy. The AST resulting from the merge will then contain text block nodes representing the results of unstructured merging between blocks of text. These sections of code will be included in the code resulting from the transformation of the surrounding AST. We describe merge tools in Section 2.5.3.

Semantic or Structured Merging Techniques

There are cases that syntactic merging techniques are unable to detect some frequently occurring conflicts. For example, in the merge that a variable “n" was used, but it was renamed to “arg" in the target branch. As it is not declared in the program, we have a semantic conflict in the system. A syntactic merging technique is unable to detect this conflict even though the program is still syntactically correct. More specifically, it is static semantic conflict since most compilers will detect the problem as an “undeclared variable" error. To identify semantic conflicts, deeper analyses might be performed, such as graph-based merge approaches (such as [33, 201]) and context-sensitive merge approaches (such as [306]). Indeed, one has no guarantees about how the execution behaviour of the merged program relates to the behaviour of the programs being merged. The resulting behavioural conflicts can only be countered by resorting to even more sophisticated semantic merge techniques that rely on the runtime semantics of the code. Most approaches for detecting behavioural conflicts [30, 33, 141, 150, 316] rely on complex mathematical formalism, such as denotational semantics, program dependence graphs, and program slicing.

As we could see, there are different ways to investigate semantic merging. In this section, we group studies into seven different groups as described below.

Merge matrices. All pairs of operations that lead to an inconsistency are summarised in a so-called conflict table or merge matrix. This makes it possible to detect merge conflicts by performing a simple table lookup. Munson and Dewan [210] propose a merging framework that is entirely based on the idea of merge matrices. In this framework, the matrices do not only specify the conflicts that can occur, but also the action that should be performed to resolve the conflicts. The main problem with merge matrices arises when we want to merge more than two versions (i.e., octopus merges). For example, if there are M kinds of operations and there are N versions that need to be merged, we would get an N-dimensional merge matrix that requires polynomial space complexity of O(M^N).

Conflict Sets. Edwards [93] used conflict sets in the context of collaborative applications to group together potentially conflicting combinations of operations based on the application-supplied semantics. Conflict sets correspond to the types of conflicts that may exist in an application. As such, they are statically defined, in the sense that they remain fixed as long as the application semantics does not change.

Operational-based Merge. Lippe and van Oosterom [187] introduce the operation-based merging, whose core idea can be summarised by the following slogans invented by the authors: “merging = composition of operations from each development line and merge-conflict = commutation conflict”. Lippe and van Oosterom [187] claim that an operation-based approach offers more support for conflict resolution by giving the example that a single global operation can be applied to a number of different points of the program.

Operation-based approaches have been applied to conflict detection and resolution of Unified Modeling Language (UML) models [169]. Dig et al. [86] use this operation-based merging technique and show empirically that many more merge conflicts could be solved by a tool that understood the semantics of change operations. Perry and Kaiser [231] also use the idea of operation-based merging, however, their technique supports a hierarchy of graph partitioning, which makes it possible to limit the number of potential conflicts at each phase and to resolve all conflicts in an incremental fashion. Because graph partitioning is intractable [130], efficient heuristics are needed to obtain partitions that approximate the ideal case. Another problem is that each modification that is being made can alter the dependencies, so that a new partitioning might be needed after each iteration.

Graph-based techniques. Horwitz et al. [141] were the first to develop a powerful algorithm for merging program versions without semantic conflicts, based on the semantics of a very simple assignment-based programming language. The merge algorithm uses the underlying representation of program dependence graphs [142] and uses the notion of program slicing [143] to find potential merge conflicts. Yang et al. [316] proposed a technique to overcome some limitations by introducing semantic-preserving transformations. Brinkley et al. [33] propose another extension to handle programs that contain procedures (which may be mutually recursive). While all these approaches allow us to detect behavioural conflicts, they have the important disadvantage that they remain restricted to a particular implementation language. Berzins [30] and Dampier et al. [76] propose solutions for a language-independent definition, however, it cannot be used to diagnose and locate conflicts between changes in the concrete syntactic representation of a program. This makes the approach impractical. Berzins [29] addresses this problem by restricting the general formalism to only special cases. Hunt and Tichy [145] proposed an extensible, language-aware technique that can deal with renaming and non-local conflicts, though it too has limitations in not identifying behavioural differences caused by dynamic binding, as one example. While structured merge techniques have improved precision, the cost of being language specific (less general) and typically more computationally expensive (less performant) seems to have prohibited widespread use in practice to date.

Semantic Diff with Graph-based techniques. Semantic Diff [150] seems more practical, however, it is a two-way merge tool and checks for local dependencies only. Hence, it cannot detect more global inter-procedural semantic conflicts. While the approach has been illustrated for C programs, it seems to be generalised to other programming languages as well. However, the presence of late binding and polymorphism in object-oriented programs makes it far from trivial to apply the approach in an object-oriented setting. Mens [202] presented an alternative lightweight approach to detect semantic merge conflicts. The idea is to use graphs to represent software artefacts, conflict situations can be detected as an instance of a more general graph pattern. As disadvantages of this approach, complex behavioural conflicts cannot always be detected and looking for a sub-graph in a graph can be a very expensive operation in large software systems. The most obvious way to localise the effect of changes and, consequently, to reduce the likelihood of merge conflicts is by resorting to information hiding techniques. This is not always sufficient since change often has effects beyond the imposed encapsulation boundaries. Nevertheless, approaches like Semantic Diff [150], that only detect local changes within each procedure of a program, but ignore inter-procedural effects, seem to perform fairly well in practice.

Multi-version Models Based on Typed Graphs. Barkowsky and Giese [23] consider monitoring merging and related consistency problems permanently at the level of models and abstract syntax. Their approach introduces multi-version models based on typed graphs, which permit to store changes and multiple versions in one graph in a compact form and allow to study the different versions and their merge combinations. The following capabilities are considered: (i) study well-formedness for all versions at once without the need to extract and explicitly consider each version individually; (ii) report all possible merge conflicts that may result for merges of any two versions without the need to extract and explicitly merge all pairs of versions; (iii) report all violations of well-formedness conditions that will result for merges of any two versions independent of any merge decisions without the need to extract and explicitly merge all pairs of versions. Their approach promises to support early conflict detection and collaboration for managing conflicts and their risks, while not having to decide how to later merge conflicting versions. The approach also aims for better scalability in case there are many versions that are considered in parallel.

Versions as first-class citizens. Carvalho and Seco [55] propose a language-based approach to software versioning. Unlike the traditional approach of mainstream version control systems, where each evolution step is represented by a textual diff, they treat versions as first-class citizens. Each evolution step, merge operation, and version relationship, is represented as code in the program. Hence, their type system and operational semantics for Versioned Featherweight Java (VFJ) includes the ability to merge branches, and to detect and solve conflicts. By lifting the versioning aspect, usually represented by text diffs, to the language level, they pave the way for tools that interact with software repositories (e.g. CI/Continuous Delivery (CD), GitHub Actions) to have more insight regarding the evolution of the software semantics. In the end, the authors design a code slicing algorithm that produces a snapshot with the features of the version and the corrections that are introduced by the required version. When version contexts are crossed, either implicitly or explicitly, lenses are statically put in place to transform the state of an object from one version to another at runtime. Thus, one version can adapt seamlessly to previous ones, without the need for explicit adaptation code.

As we can see, in most of the approaches mentioned above, the source code is represented as an AST instead of a sequence of lines. Merging ASTs consists of two steps: First, the nodes of the trees are matched to establish equality relationships between pairs of nodes from two different revisions. In three-way merge, the matching algorithm would match the revisions *source* and *target* with the *base* revision as well as the *source* and *target* revisions with each other. Second, the revisions *source* and *target* are merged to form the AST representing the merge result. To construct the merged AST, the algorithm walks the two ASTs to be merged in lockstep and applies three-way merge rules [203] to decide whether to include nodes, delete them, or flag conflicts.

As this process exploits a variety of properties of the language being merged, a range of situations that are undecidable for an unstructured algorithm become trivial to resolve [10]. Another example in which knowledge about language structure is useful and could be exploited by a structured merge tool is the merging of loops: A *for loop* in Java consists of a head and the associated body, with the head being made up of three distinct parts. These parts are usually located on the same line but are represented by distinct sub-trees in an

AST. A further benefit of structured merge is that formatting is not present in an AST and as such has no influence on the merge process [257].

The increased ability to resolve conflicts comes at a cost. Structured merge relies on being able to transform source code into a structured representation. In practice, this means that one has to implement (or use) an appropriate parser for every language to be supported. For widely used languages such as Java, these parsers already exist and require only minor modification to be usable in a merge tool. Any parser being considered for structured merge will, at least, have to support programmatic construction of ASTs and be extensible enough to add the concept of a conflict into the AST structure and the pretty-printing algorithm [257].

A more severe problem of structured merge is the runtime complexity of the underlying matching and merging algorithms, which work on trees making them inherently more complex than their counterparts in unstructured merging, which consider only collections of lines. For example, to maximise precision, the matching step of the algorithm would need to handle both shifted sub-trees as well as renamed nodes in the ASTs [182]. This corresponds to solving the *tree amalgamation problem* and the *maximum common embedded sub-tree problem*, both of which are known to be NP hard [53]. Although structured algorithms can be adjusted to reduce their complexity (e.g., by decreasing the granularity of the data structures) even these compromises result in polynomial or even exponential algorithms. This necessitates that matching trees are constrained. For example, in JDime, only nodes at the same level in the respective ASTs can match (see Section 2.5.3).

2.5.2 Merging Algorithms

The distinction between an unstructured diff algorithm and a structured one is that the former operates on raw text, while the latter operates on some form of structure that the text encodes [203]. Most often, that entails some form of tree structure, ranging from ordered trees to represent structured text documents [61] to fully resolved ASTs. More generalised graph representations can also be utilised [12, 203]. Below we present the identified merging algorithms that we used in the merging techniques presented above. If an algorithm is presented in a tool, we opted to present the algorithm together with the tool in Section 2.5.3.

Diff3. Diff3 is simple and captures the essence of unstructured merge [166]. Diff3 consists of two steps: First, three versions (sequences of text lines in the case of source code files), one of which is considered the base version, are passed to the algorithm. The diff algorithm is called twice to identify the longest common subsequences between the changed versions and the base version. The matching between versions and their base are then overlaid to form a sequence of chunks, either stable (all versions agree) or unstable (at least one of the versions differs from the base). Finally, the changes made in each chunk are examined and, if possible, merged. Chunks in which only one version applies changes to the base are merged. However, if two versions make inconsistent changes to the base, the algorithm reports a conflict [257].

LADiff. LADiff represents one of the earliest structured diff algorithms that can deal with insertions, deletions, updates and moves [61]. It targets structured text documents, such as LaTeX and HTML. The algorithm relies heavily on an assumption that each leaf

node in a tree T1 has at most one highly similar leaf node in another tree T2. This makes it unsuitable for source code differencing [174].

ChangeDistiller. ChangeDistiller improves upon LADiff by removing the assumption of unique matching options for leaf nodes [107], making it more suitable for source code differencing. Leaf nodes are however represented as text, meaning that there is still room for increased granularity.

GumTree. GumTree is a structured diff algorithm that like LADiff and ChangeDistiller can operate on insertions, deletions, updates and moves. However, it operates on a fully resolved AST, making it more granular [102].

CalcDiff. CalcDiff is another structured diff algorithm that operates on a control flow graph instead of an AST [12]. It is specifically designed to target object-oriented languages, and in particular with static code analysis in mind, such as being able to predict test coverage changes based on changes to the production source code [174].

OT. The Operational Transformation (OT) approach [94, 283] was introduced for synchronous collaborative editing systems. OT approaches can be decomposed into a concurrency control algorithm and transformation functions. A transformation function changes an operation to take into account the effects of the concurrent executions. Examples of OT algorithms are SOCT2 [304] and MOT2 [54].

RGA. RGA [246] is a Commutative Replicated Data Types (CRDT) algorithm designed for concurrent operations to be natively commutative. CRDT algorithms assign a unique identifier to each element. The identifiers are totally ordered and remain constant for the whole lifetime of the document. RGA [246] specifies on the identifiers (aka *s4vector*) the last previous element visible during its generation. Thus, the tombstones also remain after the deletions. RGA algorithm is based on the hash table. During the integration of the remote operation, RGA iterates over the hash table and compares the *s4vector* until retrieving the correct target position.

Logoot. Similar to RGA, Logoot is a CRDT [305]. Logoot [305] generates identifiers composed of a list of positions. The identifiers are ordered with a lexicographic order. A position is a 3-tuple containing a digit in a specific numeric base, a replica identifier and a clock value. Identifiers are unbounded to allow for arbitrary insertion between two consecutive elements. Unlike OT and RGA algorithms, Logoot does not need to store the tombstones since elements are not linked through insertions operations.

Datatype Generic. Miraldo et al. [204] present an alternative datatype generic algorithm for computing the difference between two values of any algebraic datatype. This algorithm maximises sharing between the source and target trees, while still running in linear time.

2.5.3 Merging Tools

Before describing merging tools, we present 7 evaluation dimensions and 6 design dimensions of tools for detecting conflicts considering conflict-management tools proposed by Dewan [80]. The evaluation dimensions are: (i) false positives and negatives; (ii) effort required to find potential conflicts; (iii) effort required to fix a conflicts, (iv) change to the traditional software process, (v) privacy invasion, (vi) computation and communication cost, (vii) conflict-specific screen real-estate during software development. The design dimensions

are: (i) conflict identification stage, (ii) tools extended/created, (iii) granularity of conflicting constructs, (iv) conflict criteria (e.g., concurrent edit status, conflict-count, component-count, dependency-based), (v) communicating conflict information (e.g., push, pull, awareness), and (vi) individual vs. collaborative conflict inspection.

Structured merge tools typically make use of a structured diff algorithm to identify changes across revisions, and based on that information use varying strategies for computing a merge. The topic was first studied in the early 1990s [306]. Below we describe 16 tools to support software merging identified in the literature. We sorted the tools from the oldest to the newest publication year. Tools that focus on supporting merge conflict resolution are presented in Section 2.8.4.

3DM. 3DM is a move-enabled three-way merge tool designed for Extensible Markup Language (XML) documents, with a novel merge algorithm that is applicable to any form of ordered tree [186]. It operates on units of small node contexts of three nodes; a parent node, and two of its children in the order they appear in its child list. This makes the merge granular, and it is also efficient with a time complexity of $O(n * log(n))$.

Roennau. Similar to *3DM*, Roennau [247] presented a tool for merging XML documents to apply diffs computed on one version of a document to another version of it. It has the benefit of not requiring all three revisions to be present on the same machine, which may prove useful in situations where bandwidth is highly limited. It is however by nature less precise than a traditional three-way merge, such as the one implemented by 3DM.

MolhadoRef. Dig et al. [86, 87] present MolhadoRef to merge software in the presence of object-oriented refactoring at the Application Programming Interface (API) level. It is also semantic-based and is concerned with operations which have well-defined semantics: inserting and deletion of packages, classes, method declarations and field declarations. Dotzler and Philippsen [92] propose 5 general optimizations that can be integrated into tree differencing algorithms and decrease the number of resulting editing actions.

FSTMerge. Apel et al. [11] noted that structured and unstructured merge each has strengths and weaknesses. They developed FSTMerge, a semi-structured merge, that alternates between structured and unstructured approaches. FSTMerge is the earliest example of semi-structured merge [11]. Merge tools built on FSTMerge have been shown to improve upon unstructured merge for Java [11], Python [58] and C# [59].

JDime. JDime is a three-way structured merge tool for Java that implements its own tree differencing and merging algorithms [179]. It is an evolution of FSTMerge. The matching step is simplistic and can only detect insertions and deletions. A heuristic look ahead mechanism built on top of the matching does however allow for limited move and update detection [182]. Leßenich et al. [182] attempt to improve JDime by employing a syntax specific look-ahead to detect renaming of declarations and shifted code. They demonstrate that their solution can significantly improve matching precision in 28% while maintaining recall.

EnvisionIDE. EnvisionIDE is an approach to structured merge using a generic, textual representation of ASTs, and then merge with a standard line-based merge algorithm [19]. The proposed algorithm in EnvisionIDE can work either with unique identifiers stored across revisions to avoid the need for tree differencing, or use a differencing algorithm such as GumTree to compute matching options.

SafeMerge. Sousa et al. [269] proposed SafeMerge, a semantic tool that checks whether a merged program does not introduce new unwanted behaviour. They achieve that by combining lightweight dependence analysis for shared program fragments and precise relational reasoning for the modifications. They found that the SafeMerge can identify behavioural issues in problematic merges that are generated by unstructured tools. In other words, SafeMerge takes a step towards eliminating bugs that arise due to 3-way program merges by automatically verifying semantic conflict-freedom, a notion inspired by earlier work on program integration [141, 316].

AutoMerge. Zhu et al. [320, 321] proposed a version space algebra-based tool, named AutoMerge. Their goal is to find a set of matching nodes that maximises the quality function, preventing the matching of logically unrelated nodes, and, as consequence, unnecessary conflicts. They found that AutoMerge was able to reduce the number of reported conflicts by 63% when compared to original JDime, being only 17% slower. Besides, they found that about 99% of the results yielded by AutoMerge exactly correspond to the original developers' result, compared to 93% from JDime [257].

jsFSTMerge. By adapting the FSTMerge, Tavares et al. [288] build competitive semi-structured merge tools for JavaScript, called jsFSTMerge. Their results show that the expected smaller effort for creating semi-structured merge tools [11], one of the main advantages of semi-structured merge over structured merge [12, 51, 118, 306], might not be as small as expected, at least for scripting languages. They found evidence that jsFSTMerge reports fewer spurious conflicts than unstructured merge, without compromising the correctness of the merging process. The gains, however, are much smaller than the ones observed for Java-like languages, suggesting that semi-structured merge advantages might be limited for languages that support both commutative and non-commutative declarations at the same syntactic level. The authors concluded that, to be effectively adopted in practice, semi-structured JavaScript merge tools require substantial adaptation, improvements, and tuning.

IntelliMerge. Shen et al. [260] present a graph-based refactoring-aware merging algorithm for object-oriented programs developed in Java, named IntelliMerge. The authors explicitly enhance its ability in avoiding and resolving refactoring related conflicts, without sacrificing its precision and performance, compared with state-of-the-art merging approaches. IntelliMerge works in 4 sequential steps: code-to-graph, matching, merging, and graph-to-code [260]. While graph-based merging techniques typically suffer from excessive running times [260], IntelliMerge is shown to be even faster than a comparable specialisation of FSTMerge.

InCline Lillack et al. [184] propose and define integration intentions and implement a prototype Integrated Development Environment (IDE) tool InCline (intention-based clone integration). InCline works with C pre-processor, but it is otherwise language-independent. It offers five editable views on the variant code. Unlike the views of diff tools, designed for code merging, our views take integration of variants with configuration options as a first class concept.

TOM. Ji et al. [153] propose general test oracles for merges inspired from the contract of semantic conflict freedom and make the oracles applicable for all real-world merge scenarios (i.e., 2-way, 3-way and octopus merges). After that, based on their proposed test oracles, the authors address how to find the Unit Under Testings (UUTs) from the whole

project and which variant involved in the merge scenario should be used to generate test cases. They implement a tool, named Testing on Merges (TOM), to automatically find the impacted methods due to changes and then generate test cases to reveal conflicts. Moreover, the authors constructed a benchmark named MCon4J (Merge Conflicts for Java). MCon4J contains a total of 389 three-way merges and 389 octopus merges respectively, in each of which merge conflicts exist. Experimental results show that TOM found 45 conflict 3-way merges and 87 octopus merges.

RefMerge. Ellis et al. [95] propose RefMerge, which is a re-imagined design and implementation of Dig et al.'s work [88]. RefMerge follows the same approach of reverting and replaying refactors. The authors compare RefMerge with IntelliMerge. IntelliMerge completely resolves 78 (4%) of the merge scenarios while RefMerge completely resolves 143 (7%). For scenarios the tools cannot completely resolve, IntelliMerge reduces the overall conflicting LOC in 414 scenarios (21%) by a median 51% reduction while it increases it in 584 scenarios (29%) by a median 164% increase. RefMerge reduces the conflicting LOC in 287 scenarios (14%) by a median 13% reduction and increases it for 274 scenarios (14%) by only 23% increase. RefMerge reduces the number of false positives and false negatives by 5% and 83% respectively, while IntelliMerge increases them by 279% and 1 383%. RefMerge struggles most with *move method* refactoring whereas IntelliMerge struggles most with *add parameter* and *rename parameter* rafactorings.

DeepMerge. Dinella et al. [89] propose DeepMerge, a tool that uses a combination of (i) an edit-aware embedding of merge inputs (*base*, *source*, and *target* versions) and (ii) a variation of pointer networks, to construct resolutions from input segments. DeepMerge is evaluated on JavaScript due to its importance and growing popularity and the fact that analysis of JavaScript is challenging due at least in part to its weak, dynamic type system and permissive nature [269]. At the end, DeepMerge uses a deep learning algorithm to merge code that an unstructured merge technique (Diff3) failed to merge. As a result DeepMerge has an accuracy of 36.5% which is 9 times greater than the accuracy of jsFSTMerge (at that point the state of the art to merge JavaScript code) for the same dataset.

MergeBERT. MergeBERT is a neural program merge framework based on token-level three-way differencing and a multi-input variant of the bidirectional transformer encoder (BERT) model [286]. Svyatkovskiy et al. [286] evaluate MergeBERT against structured and semi-structured program merge tools like jsFSTMerge and JDime, as well as neural program merge models. The authors show evidence that MergeBERT outperforms the state-of-the-art, achieving 2–3× higher accuracy on merge resolution. Comparing MergeBERT with the language model baseline and DeepMerge, MergeBERT achieved better accuracy (68.2% against 48.1% and 42.7%, respectively). Comparing with JDime (in Java) and jsFSTMerge (in JavaScript), MergeBERT also performed better than both tools (63.2% vs 21.6% and 65.6% vs. 3.6%). For the language specific models, performance is fairly consistent across all four languages with top-one accuracy ranging from 63.2% to 68.2%. For the multilingual models, performance is fairly consistent across all four languages with top-one accuracy ranging from 62.9% to 67.6%. Thus, from a pragmatic perspective, if one chooses to simplify their use of MergeBERT by training just one model instead of one model per language, then the performance takes only a negligible hit. Furthermore, the authors performed a user study with 25 developers from large OSS projects. They asked participants to evaluate whether MergeBERT resolution suggestions are acceptable on a set of 122 of their own real-world

conflicts. Their results suggest, in practice, MergeBERT resolutions would likely be accepted at a higher rate than estimated by the performance metrics chosen. The authors conclude that the local view of a conflict is sufficient to merge a majority of conflicts. Around 16% of the conflicts require external information to correctly resolve. One direction to improve MergeBERT is to consider external context as an additional information source for resolving conflicts.

Spork. Larsen et al. [174] propose Spork a structured merge tool. Spork is tailored to the Java programming language, leveraging both syntax and semantics of important language constructs to avoid or resolve conflicts. A key technical novelty of Spork is that it builds upon the merge algorithm of the 3DM merge tool for XML documents [186]. In Spork, the authors augment the 3DM algorithm, and demonstrate that the core principles are applicable to the Java programming language. As the authors show in their evaluation, Spork improves upon the state of the art with respect to the aforementioned problems. First, Spork reuses source code from the input files when printing. This improves formatting preservation over the state of the art in more than 90% of merged files, with 4 times better preservation in the median case. Second, Spork's running time performance slightly improves upon the competition in the median case, but more importantly it significantly reduces the quantities and magnitudes of the largest running times. Regarding conflicts, Spork performs better than JDime and on par with AutoMerge. Regarding running times, Spork is slightly faster in the median case, but more importantly reduces both amounts and magnitudes of excessive running times. Regarding formatting preservation, Spork decreases the formatting changes by an order of magnitude. According to the authors, Spork can be considered to be pushing the state of the art of software merging.

2.5.4 Studies Comparing Merge Tools and Strategies

Six studies comparing merge techniques called our attention [58–60, 179, 197, 257]. Mehdi et al. [197] compared four merge algorithms (*git merge* - diff3 algorithm, *one operational transformation approach* - SOCT2 algorithm, RGA, and Logoot) to measure programmer effort required to use various textual merge tools [197]. They investigated six GitHub projects (git, bootstrap, node, html5-boilerplate, d3, and gitorious) and compared scenarios where one merge technique successfully generates a candidate merge while another one reports a conflict. As a result, SOCT2, RGA, and Logoot reported a reduction in the merge blocks (i.e., the number of modifications in the difference) of 33%, 31%, and 5% when compared to *git merge*, respectively. Regarding the merge lines (i.e., the number of lines manipulated by these modifications), the average gain is between 32% and 35% for all operation-based algorithms when compared to *git merge*.

With a somewhat different focus, Leßenich et al. [179] and Cavalcante et al. [58] examined 50 and 60 projects, respectively, to compare semi-structured and unstructured merge techniques in terms of how many conflicts they report. Both studies found that semi-structured merge techniques can reduce the number of conflicts by approximately half, but not eliminate them.

Cavalcanti et al. [59] compared merge results of several semi-structured merge tools on more than 30 000 merges from 50 OSS projects, identifying conflicts incorrectly reported by

only one of the tools (false positives), and conflicts correctly reported by only one of the tools (false negatives). Cavalcanti et al. [59] used the findings from their previous study [58] to improve their semi-structured merge technique to address conflicts involving import declarations and initialization blocks. Their results show that the number of false positives is significantly reduced by semi-structured merge, and they are easier to analyse and resolve than those reported by unstructured merge.

In a third moment, Cavalcanti et al. applied both semi-structured and structured merge strategies to more than 40 000 merge scenarios (triples of base commit, and its two variants parent commits associated with a non-octopus merge commit) from more than 500 GitHub OSS Java projects. In particular, they assessed how often the two strategies report different results, and identify false positives (conflicts incorrectly reported by one strategy but not by the other) and false negatives (conflicts correctly reported by one strategy but missed by the other). Their results show that: (i) Semi-structured and structured merge report similar numbers of conflicts, but the number of merge scenarios with conflicts is reduced by about 19% using structured merge; (ii) Semi-structured and structured merge differ in about 24% in terms of reported number of conflicts when applied only to conflicting scenarios of their sample; (iii) Semi-structured and structured merge differ when changes occur in overlapping text areas that correspond to different AST nodes and semi-structured and structured merge differ when changes occur in non-overlapping text areas that correspond to (a) different but incorrectly matched nodes and to (b) the same node; (iv) Semi-structured merge reports false positives (spurious conflicts) in more merge scenarios (36) than structured merge (4); (v) Semi-structured merge reports more false positives (9 times more scenarios with false positives), and structured merge misses more conflicts (has more false negatives; 8 times more scenarios with missed conflicts). (vi) A semi-structured merge tool that can resolve consecutive lines conflicts would present an even closer number of scenarios with conflicts to structured merge, and fewer scenarios in which the two strategies differ.

Seibt et al. [257] analysed 7 727 merge scenarios from 10 GitHub OSS Java projects. They integrated merge scenarios using three merging strategies (unstructured, semi-structured, and structured) and possible combinations of them (e.g., first merging with an unstructured strategy, then a semi-structured strategy, and finally a structured strategy). They investigated (i) the runtime cost, (ii) the number of conflicts, and (iii) the size of conflicts when using progressively more complex merge strategies. They called *US*, *SS*, and *S* unstructured, semi-structured, and structured techniques, respectively. Furthermore, when a technique is followed by another one, they represent it by an arrow ($\rightarrow$). For instance, a scenario where the unstructured merge is performed followed by an semi-structure merge is represented by $US \rightarrow SS$. Their results show that: (i) Combining the rankings, *US* is the fastest, followed by the combined strategies having *US* as their first component, then followed by *SS*, $SS \rightarrow S$, and *S* (fully structured merging); (ii) Using more complex merge strategies leads to a statistically significant decrease (up to 30%) in the number of conflicting merge commits, however, they observed a small increase in build and test failures amongst the strategies when the combining complexity of the strategy increases, and; (iii) Examining the size of the conflict, they found that the reduction in the number of conflicts reported by structured merge outweights its potential to produce very large conflicts. The main take away message is that combined strategies are an attractive compromise between runtime and conflict resolution potential, while their increase in test failures remains small enough to make them viable in

practice. Hence, combined strategies retain the low runtime of unstructured merge for most scenarios while resolving, at least, as many conflicts as structured merge in cases where unstructured merge fails.

2.6 Factors Related to Merge Conflicts

In this section, we present studies that show factors (Section 2.6.1) and technical debts and organisational structure (Section 2.6.2) related to merge conflicts.

2.6.1 Measures Related to Merge Conflicts

In Table 2.1, we present 21 measures related to merge conflicts extracted from studies using different approaches to come up with these measures. We present the measure name, the impact/relation with merge conflict, and the studies a given measure was used. We keep only studies understanding the relation among measures (factors) and merge conflicts. Hence, we ignore studies that (i) purely investigate the source code and provide assumptions on measures related to merge conflicts or (ii) use a mentioned measure in another context than predicting merge conflicts. Note that measures characterising the conflict itself were presented in Section 2.4.6.

Regarding the impact, $\uparrow$ stands for a positive impact and $\leftrightarrow$ stands for a neutral impact. A positive impact means that increasing the value of the measure increases the chance of merge conflicts arise. A neutral impact means that increasing or decreasing the value of a measure does not influence the merge conflict occurrence. Note that some measures have a positive and neutral impact. It means that studies found a different impact for a given measure. We describe each reference presented in Table 2.1 below. References are ordered from the oldest to the newest publication year.

Leßenich et al. [180] surveyed 41 developers and extracted a set of seven indicators (the number of commits, commit density, number of files changed by both branches, larger changes, fragmentation of changes, scattered changes across classes or methods, and the granularity of changes above or within class declarations) for predicting the number of conflicts in merge scenarios. They also checked additional indicators mentioned in the survey, i.e., whether the more developers contribute to a merge scenario, the more likely conflicts happen and whether branches that are developed over a long time without a merge are more likely to lead to merge conflicts. After determining the respective value for each branch, they compute the geometric mean of these values. To evaluate the indicators, the authors performed an empirical study on 163 OSS Java projects, involving 21 488 merge scenarios. They found that none of the indicators can predict the number of merge conflicts, as suggested by the developer survey. Hence, they assumed that these indicators cannot predict the frequency of merge conflicts. However, none of the seven indicators have a strong correlation with the number of failed merges. They found a conflict rate of 11% when considering all files and 6% when considering only Java code. It shows that not all conflicts are in programming language files and these conflicts (e.g., in documentation, configuration, or generated files) might be simpler to resolve.

Table 2.1: Measures Related to Merge Conflicts

Measure Name	Impact	Reference
Branch duration	↑ ↔	[68, 199, 200, 226]
Commit density	↔	[180, 199, 226]
Lack of communication	↑	[68]
Length of commit messages in a branch	↑	[294]
Number of AST changes above class	↔	[180]
Number of AST changes inside class	↔	[180]
Number of chunks	↔	[180]
Number of commits	↑ ↔	[68, 82, 180, 199, 200, 226, 294]
Number of committers/developers	↑ ↔	[68, 82, 180, 199, 200, 226]
Number of developers in both branches	↑	[199, 200]
Number of files	↑ ↔	[68, 82, 180, 199, 200, 226]
Number of files changed in both branches	↑ ↔	[180, 199, 200, 226]
Number of lines of code	↑ ↔	[68, 180, 199, 226]
Number of non-modular contributions	↑	[82]
Number of parallel lines changed	↑	[294]
Number of self-conflict merges	↑	[199]
Programming language	↑	[199]
Scattering Degree methods	↔	[180]
Scattering Degree classes	↔	[180]
Time PR was opened	↑	[294]
Total duration	↑	[82, 199, 200, 294]

↑ and ↔ stand for a positive and neutral impact of a given measure on the number of merge conflicts, respectively.

Owhadi-Kareshk et al. [226] investigated if conflict prediction is feasible. They verified nine indicators (the number of changed files in both branches, number of changed lines, number of commits and developers, commit density, keywords in the commit messages, modifications, and the duration of the development of the branch) for predicting whether a merge scenario is safe or conflicting. They adopted norm-1 as the combination operator to combine the indicators extracted for each branch into a single value. To evaluate the predictor, they performed an empirical study on 744 GitHub repositories in seven programming languages, involving 267 657 merge scenarios. Similar to Leßenich et al. [180], they did not find a strong correlation between the chosen indicators and conflicts, but using the same indicators, they designed a classifier that was able to detect safe merge scenarios (without conflicts) with high precision (0.97 to 0.98) using the random forest classifier. The number of simultaneously changed files is the most important feature for predicting merge conflicts. The number of commits in each branch shows a weak correlation, but a much lower importance level by the decision tree classifier. Remaining feature sets show low correlation coefficients and importance.

Dias et al. [82] conducted a study to understand better how conflict occurrence is affected by technical and organisational factors. They investigated seven factors related to modularity, size, and timing of developers' contributions. They computed the geometric mean of the branch values for each factor. The authors analysed 125 projects, involving 73 504 merge scenarios in GitHub repositories using Ruby (100) and Python (25) as main programming languages that use the Model-View-Controller (MVC) pattern. Conflict rate is 13.4% for Ruby projects, with project rates ranging from 0.9% up to 54.5%. For the Python projects conflict rate is 10.0%, with project rates ranging from 2.1% up to 37.5%. They found that merge conflict occurrence significantly increases when contributions to be merged are not modular (i.e., files are from a different MVC slice). So, aligning slices and task structure by defining tasks that focus on specific slices, and only executing in parallel tasks that focus on disjoint slices, gives no guarantees of conflict avoidance. Furthermore, contribution modularity is not associated with the extent of merge damage. The absence of strong correlation is also observed with a corresponding per project analysis. This suggests contribution non-modularity has no predictive power concerning the number of merge conflicts or the number of files with conflicts. So, whereas non modularity can be used to predict conflict occurrence and the associated damage, it cannot predict the extent of the damage. Contradictorily with Leßenich et al. [180] and Owhadi-Kareshk et al. [226], Dias et al. [82] found that more developers, commits, changed files, and contributions developed over long periods are more likely associated with merge conflicts. However, no evaluated attributes showed predictive power concerning the number of merge conflicts.

Menezes et al. [200] and a follow up study [199] investigated measures related to merge conflicts. They analysed 182 273 merge scenarios from 80 projects using 8 different main programming languages. They investigated a set with 17 measures where 5 of them represent the merge conflicts itself and other 2 are used to calculate other measures. Hence, their analyses contributed to 10 out of the 21 measures present in Table 2.1. They performed a set of statistical analyses involving, for instance, the normality of measures, the Mann-Whitney test, Cliff's Delta to calculate the effect size, and association rules of two disjoint entity sets (pairs) to identify correlations among investigated pairs. They investigated 8 Research Questions (RQs), we summarised them as follows. RQ_1 - the branch-duration and

total duration have a small impact on the occurrence of merge conflicts (effect sizes of -0.3 and -0.27, respectively). Despite the small impact, the association rules indicate that the occurrence of conflict increases when time increases (lift close to 1 for durations of 1-7 days and lift $>$ 2.4 for durations bigger than 30 days). RQ_2 - the number of commits has a small impact for the source branch (effect size of -0.24) and a large impact for the target branch (effect size of -0.53) on the occurrence of merge conflicts. The association rules indicate that the chances of conflict increase when the number of commits increases (according to the ranges of commits, lifts in source branch range from 0.61 to 1.54 and lifts in the target branch range from 0.35 to 3.78). RQ_3 - the number of committers has no impact for the source branch (p-value is 0.32) and a large impact for the target branch (effect-size of -0.48) on the occurrence of merge conflicts. The association rules indicate that the chances of conflict increase when the number of committers increases, especially for the target branch (lift goes from 0.67 for 1–3 committers to 6.05 for $>$30 committers). RQ_4 - the number of changed files has a small impact for source branch (effect-size of -0.29) and a large impact for the target branch (effect-size of -0.57) on the occurrence of merge conflicts. The association rules indicate that the chances of conflict increase when the number of files increases (more than 30 files in the source branch has lift 1.53; more than 6 files in B2 has lift 1.39; more than 30 files in the target branch has lift 3.43). RQ_5 - the number of changed lines has a small impact for source branch (effect-size of -0.33) and a large impact for the target branch (effect-size of -0.51) on the occurrence of merge conflicts. The association rules indicate that the chances of conflict increase when the number of changed lines increases (lift goes from 0.31 for 0–10 LOC to 2.28 for $>$10 000 LOC in source branch and 0.31 for 0–10 LOC to 4.73 for $>$10 000 LOC in the target branch). RQ_6 - the association rules indicate that the chances of conflict increase when the project is written in PHP (53%), JavaScript (23%), and Java (9%). RQ_7 - the association rules indicate that having some intersection increases the chances of conflict (67-99% in 265%, 1-33% in 83%, and 34-66% in 22%). For instance, with an intersection of 67% to 99% of developers working in both branches the occurrence of conflicts is 265% greater than when there is no intersection. RQ_8 - the percentage of self-conflicts range from 5.46% (of 3 152 conflicting chunks) in Yii2 project to 66.23% (of 835 conflicting chunks) in Vert.x project. Looking at each programming language, the conflict rate are: 4.47% for C, 9.47% for C#, 9.48% for C++, 10.44% for Java, 11.98% for JavaScript, 15.09% for PHP, 8.51% for Python, and 8.75% for Ruby projects.

Trif et al. [294] predicted conflicts using machine learning (Support Vector Machine (SVM), naive bayes, random forest) and deep learning (neural networks). They used 10 features: (i) whether a PR was created before merging the branches, (ii) the number of added and deleted lines, (iii) the number of developers on each branch, (iv) the duration of parallel development, (v) the number of simultaneously changed files, (vi) the number of added files, deleted files, renamed files, copied files, modified files, (vii) the number of commits on each branch, (viii) the density of commits in the last week of development, (ix) the frequency of selected keywords in the commit messages on both branches, and (x) commit message length. Their random forest analysis show 5 top factors that cause conflicts: (i) the number of parallel lines changed, (ii) whether a PR was opened before the merge, (iii) the number of commits on a branch, (iv) the active time of development, and (v) the minimum length of commit messages in a branch. Their best results were for random forest and neural networks with 0.75 and 0.77 of precision and 0.69 and 0.93 of recall, respectively.

Costa et al. [68] conducted a survey research with 109 software developers to understand the way they use branches, the occurrence of conflicts, the resolution process, and factors that can lead or avoid merge conflicts. Here, we focus on the factors related to merge conflicts. The top-7 factors are: Branching duration (76), lack of communication (64), number of files (53), number of developers (42), number of lines of code (31), same developers in many branches (28), number of commits (24). The survey participants also mentioned in low frequency factors related to organisational structure, technical debt and merge conflict characteristics like do not keep repositories up to date (5), code formatting (3), parallel editing in the same code (2), tasks not mapped correctly (2), and many features in development (1). We do not add these measures either because the number of mentions were very low, or we present them in another section. For instance, in Section 2.4.6 we present merge conflict characterization and in Section 2.6.2 we present technical debt and organisational factors related to merge conflicts.

2.6.2 Technical Debt and Organizational Structure Related to Conflicting Code

In this section, we present studies that found technical debt and organisational structure related to conflicting code. Studies are sorted from the oldest to the newest publication year.

Purushothaman and Perry [236] investigated small source code changes (i.e., one-line changes) during the development process in a popular and large system composed of more than 200 million lines of code and 50 subsystems. Their results show that: (i) nearly 40% of changes that were made to fix defects introduced one or more other defects in the software; (ii) changes are often small (nearly 10% of changes involved changing only a single line of code. Nearly 50% of all changes involved changing 10 or fewer lines of code. And, nearly 95% of all changes were those that changed fewer than 50 lines of code); (iii) less than 4% of one-line changes result in error; (iv) the probability that the insertion of a single line might introduce a defect is 2%; (v) nearly a 50% chance of at least one defect being introduced if more than 500 lines of code are changed.

Dig et al. [87] previously argued that since refactorings cut across module boundaries and affect many parts of the system, they make it harder for VCSs to merge the changed code. They proposed refactoring-aware merging, with the argument that if a merging tool understands the refactorings that took place, it may be able to automatically resolve the conflict and save the developer's time. This study was already mentioned in Section 2.5.3.

Fauzi et al. [104] performed a systematic mapping on Software Configuration Management (SCM) in global software development. They found 6 issues developers face in this context: (i) dispersed software teams do not get information on what other teams are doing, so it is difficult to know the traceability of each module and the definition of modifications or problems to be handled is unclear; (ii) dependency, delay, and increased time required to complete Modification Requests (MRs); (iii) working in different SCM environments, which leads to MRs being handled at various levels in the project; (iv) lack of a planned baseline; (v) SCM process management including lack of coding standards, code ownership, unclear flow of development, and tool selection, and; (vi) artefacts with different versions and content at each site.

Cataldo and Herbsleb [56] presented an empirical analysis of a large-scale project where they examined the impact that software architecture characteristics, and organisational factors have on the number of conflicts. They concluded that architecture related factors such as the nature and the quantity of component dependencies, as well as organisational factors, such as the geographic dispersion of development teams, can lead to a higher integration failure rate.

Shihab et al. [262] presented an empirical study that evaluated and quantified the relationship between software quality and various aspects of the branch structure used in software projects. They examined Windows Vista and Windows 7 and compared components that have different branch characteristics of two branching strategies (branching according to the software architecture versus branching according to organisational structure). Their results show that (i) software components with high branching activity have more failures; (ii) software components spread across many branch families have more failures; (iii) branches with higher architectural mismatch (e.g., subsystems, areas, components) have more failures; (iv) branches with higher organisational mismatch (e.g., number of managers, development leads, engineers) have more failures; (v) organisational structure has a stronger relationship with failures than architectural mismatch.

Estler et al. [97] investigated the relationship between merge conflicts and developers' awareness, analysing data from 105 student developers. They investigated the frequency and significance of merge conflicts and awareness of the work of team members, the effects of merge conflicts and of insufficient awareness on project development, and the frequency and detail level with which awareness information should be provided. Their results show that (i) the likelihood of incurring into merge conflicts is not significantly affected by the location of developers within the same team; (ii) interruption due to insufficient awareness occur frequently for teams of non-trivial size; (iii) interruptions impact more negatively than merge conflicts measures and other measures, such as productivity, motivation, and keeping to the schedule; and, (iv) developers appreciate having access to awareness information frequently but not in real time.

Ahmed et al. [5] analysed 163 projects and 6 979 failed merges investigating (i) whether program elements involved in merge conflicts contains code smells, (ii) which code smells are more associated with merge conflicts, and (iii) whether the code smells associated with merge conflicts affect the quality of the resulting code. As a result, they found that program elements that are involved in merge conflicts are, on average, 3 times more smelly than program elements that are not involved in a merge conflict. Regarding the associated code smells, the top 5 smells in terms of their (mean) numbers per conflict are: *God Class*, *Data Clump*, *Sibling Duplication*, *Data Class* and *Distorted Hierarchy*. The three code smells with the strongest correlation are: *God Class*, *Internal Duplication* and *Distorted Hierarchy*. These smells all relate to cases where object-oriented design principles of encapsulation and structuring are not well used, leading to problems with developers making conflicting parallel changes. Semantic conflicts are more common (76.12% in the manually labelled data and 75.23% in the automated classified data), as compared to the non-semantic conflicts (23.88% in manually labelled and 24.77% in automated classified data). Semantic merge conflicts are associated with smells at the method level. Regarding the quality analysis, code smells have a significant impact on the final quality of the code. Since McFadden's adjusted R^2 penalises a model for including too many predictors, had the code smells not mattered, removing

it could have increased the adjusted-R^2 instead of reducing it. The authors found that the presence of code smells on the lines of code involved in a merge conflict has a significant impact on its bugginess. Tool builders can use the information of incidence of code smells to support distributed work – either in predicting likelihood of conflicts or their difficulty. Code smells can also be used as a factor to schedule tasks (e.g., program elements that have code smells should not be edited in parallel) or assign tasks (e.g., developers with higher experience should work on smelly program elements).

Mahmoudi et al. [193] analysed 2925 GitHub repositories and uses the state-of-the-art refactoring detection tool, RefactoringMiner [295], which is able to precisely detect 15 types of refactorings. They wanted to know (i) how often merge conflicts involve refactored code, (ii) whether conflicts that involve refactoring are more difficult to resolve, and (iii) the types of refactoring more commonly involved in conflicts. As a result, they found that 22% of merge conflicts involve refactoring, which is remarkable taking into account that we investigated only 15 refactoring types while refactoring books describe more than 70 different types. More precisely, 11% of conflicting regions have at least one involved refactoring. Regarding the conflict resolution, they found that conflicting regions that involve refactorings tend to be larger (i.e., more complex) than those without refactorings. Furthermore, conflicting merge scenarios with involved refactorings include more evolutionary changes (i.e., changes leading to conflict) than conflicting merge scenarios without involved refactorings. Regarding the types of refactoring, *extract method* is more involved in conflicts than its typical overall frequency, with a small effect size. *Extract interface* and *extract superclass* are also more involved in conflicts, but with negligible effect sizes.

By investigating ElasticSearch, Mahmood et al. [192] (see Section 2.4.4) found that refactoring, feature introduction, feature enhancement, bug fix, and test improvement are the top-five causes of changes at project-level related to conflicts.

Amaral et al. [8] investigated the source code history of 29 popular Java Apache projects aiming at investigating the bug-introduction changes with merge conflicts and co-change dependencies. Regarding the extent conflicting merge scenarios induce bugs, the authors found that conflicting merge scenarios are not more prone to introduce bugs than usual commits. That is, 9.43% of the conflicting merge scenarios introduce bugs, while RA-SZZ linked 9.53% of all commits as bug introducing changes. Besides, conflict resolution represents 0.51% of all bug-introducing commits. Nonetheless, this finding suggests that merge conflict resolution is an important source of bugs, and 150 bugs have been introduced due to conflicting merge resolution.

Brindescu et al. [41] investigated 143 OSS projects to identify effects the merge conflicts have on the quality of the code in two ways. First, they investigated how likely code resulting from a merge conflict to contain bugs. Commits that are involved in a merge conflict are 2.38 times more likely to contain a (future) bug fix. They found that *non-trivial* merge conflicts (no textual-conflicts) are 26.81 times more likely to need a bug fixing commit compared to lines involved in *trivial* (textual) merge conflicts. In summary, the authors found that code that was involved in a merge conflict has a higher likelihood of being involved with a future bug. While some bugs are injected in the merge resolution, others were likely already there (e.g., changes in *disjoint* merge conflicts). In either case, a closer scrutiny of code involved in a merge conflict is warranted. Second, they investigated factors that affect the quality of the

code resulting from a merge conflict resolution. They found that the code in merge conflicts were twice as likely to contain bugs as other changes. Further, if the changes included semantically interacting changes, the likelihood of a defect is 26 times that of non-conflicting changes. They found eight main factors: (i) number of references to other files (0.084), (ii) number of references to the file involved in the merge (-0.035), (iii) number of non-core contributor authors involved in the merge (-1.898), (iv) number of authors involved in the merge (-0.5634), (v) number of AST nodes changed (0.0007), (vi) number of classes involved in the merge (-0.1636), (vii) number of methods involved in the merge (0.3756), and (viii) the number of LOC changed (0.00003). The effect of outward dependencies (number of external references from the file involved in a merge conflict) is positive, therefore changes that refer to external file (class) elements are more likely to contain bugs. The authors found a negative effect of inward dependencies (number of references to the file involved in merge). They also found a negative coefficient for the size of the change (number of classes involved in the merge conflict), the number of authors involved in the change and the number of non-core contributors. The correlation between the number of authors with changes involved in a merge conflict and the number of future bug fixes is also negative. In line with previous research ([168, 206, 307]), Brindescu et al. [41] found a positive effect of number of methods involved in the merge conflict, number of LOC changes and number of AST nodes changed on future bug fixing commit counts.

Ji et al. [154] investigated merge conflicts and resolutions in *git rebases* of 82 Java repositories from GitHub with a total of 51 183 rebase scenarios. Their results show that (i) 7.6% of PRs have rebases, (ii) conflict rate of rebases is between 24.3% and 26.2%, (iii) the likelihood of conflicts from (Git) rebases is not significantly different from three-way merge conflicts (*git merge* command) comparing to the results of existing studies [110, 317, 322].

Pan et al. [228] (already mentioned in Section 2.4.4) found that a majority (∼47%) of the files with conflicts were written in C++ (the core functionality of both Chromium and Microsoft Edge are written in C++). Among these C++ files, the authors found that 12.34% are related to *headers* and *macros*. Furthermore, they found that 31.49% of 1-2 lines of conflicts in C++ files are due to addition, deletion, or updating the *include* section. For short, we found two studies investigating the merge conflict location. While Dias et al. [82] show evidence that changes across MVC slices influence positively the occurrence of merge conflicts, Pan et al. [228] found that changes on programming language files are conflict-prone when compared to changes in non-programming language files.

Maddila et al. [190] investigated concurrent edits in the same code areas in the PR based software development model. Their analyses include a six months history of changes made in six large repositories in Microsoft. As a result, they found that files concurrently edited in different PRs are more likely to introduce bugs.

Dias et al. [84] suggest using change impact analysis to find dependencies between code artefacts. They implement a prototype named DeltaImpactFinder to find such dependencies. DeltaImpactFinder analyses and compares the impact of a change in its origin and destination branches. They call the difference between these two impacts the *delta-impact*. If the delta-impact is empty, it means that there is no semantic merge conflict and the merge can continue automatically. Otherwise, the delta-impact contains what are the sources of possible conflicts. As a result, they found that patches that introduce a different set of dependencies to the different versions are likely to result in semantic merge conflicts.

2.7 Avoiding Merge Conflicts

The best way to deal with merge conflicts is creating mechanisms to avoid them. In this section, we investigate strategies and heuristics (Section 2.7.1) and tools and frameworks (Section 2.7.2) to avoid merge conflicts.

2.7.1 Strategies and Heuristics to Avoid Conflicts

In this section, we present studies, grouped into 5 topics, that show evidence of strategies and heuristics useful to avoid merge conflicts. We ignore studies that present strategies and heuristics to somehow improve software development but they do not take merge conflicts into account.

Increasing coordination activities . Begole et al. [26] investigate the work rhythms of developers. They use minute-by-minute records of computer activity coupled with locality of the activity, calendar appointments, and e-mail activities to provide meaningful visualisations for group coordination. The passive nature of developers' interaction with these visualisations requires users to engage and coordinate with each other, which differs from version control systems that actively support the software development process.

Contribution rules or commit policy. A few studies [2, 49, 308] have found that the process of committing changes to a shared repository is typically governed by a commit policy that aims to minimise merge conflicts, eliminate build problems and avoid test outcome degradations. A policy imposed by the project management usually consists of a small number of informally stated guidelines that developers are encouraged to follow. For example, many development teams follow the "*Commit early and commit often*", "*never commit code that does not compile*", and "*test your changes before committing*". Related to contribution rules, Brun et al. [49] found that keeping its local repository updated and pushing developers changes is an efficient way to be aware of other changes and avoid merge conflicts.

Reducing the number of branches. There are a few studies [14, 37, 46, 234] investigating software branching. Bird et al. [37] surveyed 124 experienced Microsoft engineers to determine the difficulty and time associated with integrating changes from multiple branches as well as tools and practices used to verify such work. As a result, they found that removing high-cost-low-benefit branches would have saved 8.9 days of delay and only introduced 0.04 additional conflicts on average. In a similar direction, Bruegge and Dutoit [46] give two heuristics for minimising merge conflicts. First, *changes to the main development line should be made in separate development branches*. Second, *the number of branches should be as small as possible*. Only use branches when parallel development is required. Creating a branch is a significant event that should be carefully planned and approved by management. In other words, reducing the number of branches made developers more productive but slightly increased the number of merge conflicts. Our assumption is that tightening contribution rules would support merge conflict reduction. By surveying 140 version control users, Phillips et al. [234] came up with four main observations: (i) continuous integrations are typically not done in practice; (ii) successful branching and merging strategies focus on reducing the frequency and complexity of merge conflicts, as well as preventing product

quality regressions during merge operations; (iii) branching satisfaction is influenced by the types of branches created, with the use of feature, release, and experimental branches having the most impact. The choice of version control system has little correlation with branching satisfaction; and, (iv) merging satisfaction is influenced by the frequency of upstream merges, with the use of upstream periodic and event-driven merges having the most impact.

Using patterns to manage branches. Appleton et al. [14] present 32 patterns (best practices) for managing branching in parallel development projects. Their branch patterns were divided into four groups: basic elements, creation patterns, policy patterns, branch structuring patterns. Despite their work being published more than 25 years, most of them remain up to date. For instance, *codeline policy* and *merge early and often*. They further present 13 common traps and pitfalls in branching that they call anti-patterns. These anti-patterns are also presented by Bird et al. [37]. Here, we just mention them: (i) merge paranoia; (ii) merge mania; (iii) big bang merge; (iv) never-ending merge; (v) wrong-way merge; (vi) branch mania; (vii) cascading branches; (viii) mysterious branches; (ix) runaway branches; (x) volatile branches; (xi) development freeze; (xii) integration wall; and, (xiii) spaghetti branching. For instance, they define that a big bang merge is when developers defer branch merging and attempt to merge all branches simultaneously.

Use feature toggles to manage code evolution. CI practices recommend avoiding branching, and working directly on a shared master branch (a.k.a. trunk-based development), where developers would frequently synchronise their enhancements [209]. Benefits include unexpected merge conflicts being revealed earlier and hence, being easier to deal with. But trunk-based development is thought for daily integration. If upgrades span beyond the day, then incomplete upgrades might need to be integrated. For these incomplete developments, "feature toggles" are proposed [238]. Rather than keeping a feature branch open, incomplete features are integrated but annotated with a toggle. In this way, toggles behave as feature annotations in software product line code. Feature toggles prevent code from being considered unless the feature toggle is on [209]. During development, a developer can enable the feature for testing and disable it for other users. Feature toggles are a time-honoured way to keep latent capabilities of an application in a CI setting where applications are deployed continuously.

Breaking commits into small/atomic changes. Breaking commits into smaller changes is not a new idea. In fact, tools that "untangle" commits, often containing a bundle of unrelated changes, into smaller commits containing few logical units of changes, together with a more descriptive message, have been proposed [2, 22, 83].

Keep others aware of your changes. Souza et al. [270] have drawn results from empirical data from three software development teams that were observed and interviewed. Their results suggest that the awareness network of a software developer is fluid (it changes during the course of software development work) and is influenced by three main factors: *the organisational setting* (e.g., the reuse program in the BSC corporation), *the software architecture*, and, finally, *the recency of the project*. Finally, they observed that software developers try to manage their awareness networks to be able to handle the impact of interdependent actions. Cataldo et al. [57] show the importance of timely and efficient recommendations and the implications for the design of collaboration awareness tools. Studies like this form a basis for building solutions that are scalable and responsive.

Choosing the right developers to merge. Costa et al. proposed methods to recommend experts for integrating changes across branches [65] and characterised the problem of developers' assignment for merging branches [67]. They analysed merge profiles of eight software projects and checked if the development history is an appropriate source of information for identifying the key participants for collaborative merge. They also presented a survey on developers about what actions they take when they need to merge branches, and especially when a conflict arises during the merge. Their studies report that the majority of the developers (75%) prefer collaborative merging (as opposed to merging and taking decisions alone). This reiterates the fact that tools that facilitate collaboration, by providing early warnings, are important in handling merge conflict situations.

2.7.2 Tools and Frameworks

In this section, we briefly describe 53 tools and frameworks proposed to avoid merge conflicts. We classified them into 8 types which includes tools to recommend an order to perform development tasks, systematically write commit messages and untangling changes, and keep developers aware of other developers' changes.

Version Control Systems

Costa et al. [69] present a list with version control systems including, for instance, Git, Mercurial, and Darcs. They also present some frameworks to support version control systems in distributed software development such as RepoGuard and OSCAR. Furthermore, They present 13 challenges related to version control in distributed software development. For instance, dispersed software teams do not get information about what other teams are doing and dependency and delay due to physical distance to the server, mainly in centralised topology. We believe that some of these challenges have been minimised over the last 10 years. As Costa et al. [69] present a great overview on this topic and it is not closely related to merge conflict avoidance, we leave details to be found in their study.

Tools to Build Systematic Commit History

Aware that developers often write code incompatible with a systematic strategy or bundle unrelated changes in a single commit, a few researchers have proposed tools to support developers on these tasks. In what follows, we present three tools related to it.

Commit Bubbles. Barik et al. [22] propose Commit Bubbles to support developers blending coding and commit activities through fragments to minimise context switching and treating history revision as a routine, rather than exceptional process.

EpiceaUntangler. Dias et al. [83] propose EpiceaUntangler to help developers share untangled commits by using fine-grained code change information.

Codebase Manipulation. Muslu et al. [211] propose Codebase Manipulation a tool that automatically records a fine-grained history and manages its granularity by applying granularity transformations.

Refactoring Awareness

As seen in Section 2.6, refactoring is related to merge conflicts. Hence, it is important for developers to be aware when others are refactoring code.

CatchUp!. Henkel and Diwan [134] propose CatchUp! a tool that captures and replays refactoring actions within an integrated development environment semi-automatically. The tool uses descriptions of refactorings to help application developers migrate their applications to a new version of a component.

Task Order Recommendation

By wisely choosing which tasks to work on in parallel, a development team could likely reduce conflict occurrence [245]. In particular, we should expect lower integration conflict risk from parallel tasks that focus on unrelated features and affect disjoint and independent file sets. Perry et al. [233] found that increased parallel work, in addition to causing conflicts, can also lead to an increase in software defects. Developers are known to follow informal processes (e.g., check in partial code, email the team about impending changes) to avoid having to resolve conflicts when committing changes, or rush to commit their work in an effort to avoid being the developer who has to resolve the conflicts [275]. Aiming at supporting the order development tasks should be done, a few tools were proposed. Next, we describe these tools ordered from the oldest to the newest publication year.

Hipikat. Cubranic and Murphy [71] propose Hipikat to predict task interfaces based on the project's version history. They assume that similar tasks are likely to change or use the same code elements. Hence, Hipikat generates recommendations by performing a textual similarity analysis of CVS repositories, issue-tracking systems (e.g., Bugzilla), newsgroups, and web sites associated with the project.

Mylyn. Kersten and Murphy [165] propose Mylyn to monitor developers' workspace to track relevant resources (e.g., selected or edited files) and to update the IDE accordingly. In this sense, by using a prioritising policy for resources based on user interaction, Mylyn delineates a task context during task execution.

TopicXP. Savage et al. [253] propose TopicXP, a search tool to assist developers while defining task interfaces manually. TopicXP receives as input a query using keywords and outputs relevant files for a task based on their vocabulary and static dependencies from code. The search effectiveness depends on the quality of the task descriptions, which must clarify the task purpose.

Cassandra. Kasi et al. [163] propose Cassandra, a tool that analyses a set of constraints to recommend an optimum order of task execution per developer. The constraints relate to the files each task is supposed to edit according to the developers, their dependent files that are identified by call-graph analysis, and the developer's preferred sequence for executing tasks.

Taiti. Rocha et al. [245] propose Taiti a tool that, for a given task, computes its test-based task interface (TestI). The tool works for tasks associated with Cucumber acceptance test scenarios, and approximates the set of files having code that could be executed by running the scenarios. Their results bring evidence that, in the specific context of behaviour-driven development, Cucumber tests associated with a task might help to predict application files changed by developers responsible for the task. They also found that the better the test coverage of a task, the better the TestI predictive power.

Collaboration Tools

Software development is a collaborative activity in which business analysts, customers, system engineers, architects, and developers interact. The concurrent edition of models and

processes requires synchronous collaboration between practitioners who often cannot be physically present at a common location. Software modelling requires concurrency control in real time, thus enabling geographically dispersed developers to edit and discuss the same diagrams, and improving productivity by providing a means through which to easily capture and model difficult concepts through virtual workspaces and the collaborative edition of artefacts by means of tools which permit synchronised interactions [155]. In this section, we present tools, ordered by publication year, that support software development collaboration.

SoftDock. Suzuki and Yamamoto [285] propose SoftDock, a framework which solves the issues related to software component modelling and their relationships, describing and sharing component models information, and ensuring the integrity of these models. With SoftDock, developers can therefore work by analysing, designing, and developing software from component models and transfer them by using an exchange format, thus permitting communication between team members.

SYSIPHUS. Bruegge et al. [45] propose SYSIPHUS, a distributed environment which provides a uniform framework for system models, collaboration artefacts, and organisational models, with services for exploring, searching, filtering, and analysing the models.

Galaxy Wiki. Xiao et al. [313] present Galaxy Wiki, an online collaborative tool based on the wiki concept which permits the existence of a collaborative authoring system for documentation and coordination purposes, thus allowing developers to compile, execute, and debug programs in wiki pages.

CollabDev. Sarkar et al. [250] propose CollabDev, a human assisted collaborative knowledge tool to analyse applications in multiple languages and render various structural, architectural, and functional insights to the members involved in maintenance.

IMPROMPTU. Biehl et al. [31] propose IMPROMPTU, a framework for collaboration in multiple display environments. It allows users to share task information through displays via off-the-shelf applications.

CoRED. Lautamäki et al. [175, 219] proposed CoRED – Collaborative Real-time Editor – a web-browser-based collaborative real-time code editor for Java applications with error checking and automatic code generation capabilities.

Workspace Awareness

Workspace awareness tools monitor the developers' workspace and notify them about ongoing changes that are potentially conflicting. The goal of workspace awareness tools is to make developers aware of each other's changes before these are committed to a central repository, so that they can take proactive steps to prevent or minimise unforeseen interferences and/or duplicative development [308]. Such steps may include talking to other developers, reassigning tasks, and postponing changes until the other developer has made a commit [308]. In what follows, we present these workspace awareness tools ordered from the oldest to the newest publication year.

EPOS. Lie et al. [183] provide sophisticated mechanisms to coordinate artefact sharing among workspaces. EPOS supports four different policies that can be instituted among different pairs of workspaces: (i) all artefacts are shared immediately, (ii) artefacts are pushed to other workspaces, (iii) artefacts are pulled from other workspaces, and (iv) artefacts are implicitly propagated through the central repository.

CoVer. Haake and Haake [126] propose CoVer (Contextual version server). The concepts offered by CoVer can be used by hypertext applications to define application-dependent version support.

Adele. Estublier and Casalles [98] propose Adele. Adele extends EPOS with programmable process support for specifying sharing policies on a per-artefact (type) basis. In both cases, the primary objective is more related to workspace integration than workspace awareness.

COOP/Orm. Magnusson and Asklund [191] propose COOP/Orm. In COOP/Orm, active diffs instantly communicate changes to other developers who can see those changes both in the version tree and the actual artefact.

BSCW. Appelt [13] proposes BSCW, a web-based, shared, centralized workspace with integrated versioning facilities that allow it to be used as a configuration management system. Awareness is provided statically, via web-based icons that enrich the web page for each artefact with information concerning its state, and dynamically, via a monitor that continuously informs authors of what activities are taking place in the central workspace.

CHIME. Dossick and Kaiser [91] propose CHIME, an Internet- and Intra-netbased application which allows users to be placed in a 3D virtual world representing the software system. Hence, users interact with project artefacts by "walking around" the virtual world, in which they collaborate with other users through a feasible architecture.

Coven. Carrol and Sprenkle [63] propose Coven, a tool that supports awareness through soft locks. Before changing any artefacts, developers place soft locks with associated messages on those artefacts. When another developer attempts to place a soft lock on an artefact that already has a soft lock, that developer is presented with the message attached to the lock.

iScent. Anderson and Bouvin [9] propose iScent (intersubjective collaborative event environment), an awareness mechanism that can scale to meet the project awareness needs of large organisations.

TUKAN. Schuemmer and Haake [255] created TUKAN a distributed, cooperative software development environment that supports fine-grained editing and versioning of artefacts. In TUKAN, developers can select different collaboration modes and, upon accessing or modifying an artefact, are informed via "weather" icons whether potential conflicts exist with other developers.

State Treemap. Molli et al. [208] propose State Treemap, an awareness widget for multi-synchronous groupware. Its visualisation component shows which artefacts are being modified (both locally and remotely) and which artefacts have already been committed to the configuration management repository.

Elvin. Fitzpatrick et al. [106] propose Elvin, a content-based pure notification service used to support awareness with a virtual work environment and perceptual resources for awareness.

Palantír. Sarma et al. [242, 251, 252] propose Palantír. Palantír monitors other developer's workspaces, and notifies the developer, in a non-obtrusive manner, if a conflict has happened. Palantír is a tool that addresses some of the performance issues from previous work by leveraging a cache for doing dependency analysis. Palantír uses dependency graphs in order to generate dependency information which is used to split the processing

load between distributed developers. The dependency graphs are created using Dependency Finder [194].

Celine. Estublier and Garcia [99] propose Celine and point out the importance of considering semantic dependencies between artefacts in different files in the context of workspace awareness tools. Celine explores the relationships between awareness, process and system models, and shows how the knowledge of these models can be used to improve the relevance of an awareness system. Celine also takes into account factors such as the workspace topology and the cooperative engineering policy that is being used.

Lighthouse. Silva et al. [264] propose Lighthouse to show the changes being made at the design level. Their tool presents all changes (awareness) from the perspective of changes to the model (in the form of UML diagrams) of all of the developers' projects.

SYSIPHUS. Bruegge et al.[47] propose SYSIPHUS, a distributed environment providing a uniform framework for system models, collaboration artefacts, and organisational models. In SYSIPHUS, system models, collaboration artefacts, and organisational models are given equal emphasis and live in a single, shared repository.

ADAMS. De Lucia et al. [77] present ADAMS (ADvanced Artefact Management System). It is an artefact-based process support system, supporting permissions definition, quality management and storing traceability links between artefacts that enables software engineers to create and store traceability links between artefacts. ADAMS supports the branching and merging of artefacts as well.

STEVE. De Lucia et al. [78] propose STEVE (Synchronous collaborative modelling Tool Enhanced with VErsioning management). It is a collaborative tool supporting distributed UML modelling of software systems. STEVE has been integrated in ADAMS to provide synchronous and asynchronous collaborative modelling functionalities. In particular, it allows developers to access and modify the same UML diagram at the same time, thus allowing distributed team members to discuss and model the system directly within ADAMS.

CollabVS. Dewan and Hegde [81] developed CollabVS, which looks for dependency conflicts in classes, methods and files. This is achieved by recording their cursor position and activity using the IDE. It also considers dependencies such as classes depending on super classes. In a different paper [133] with more focus on the implementation specifics reveals that dependencies are found using binary analysis.

FastDASH. Biehl et al. [32] propose FastDASH, a tool that fosters awareness between members of a team. FastDASH provides a dashboard that shows the files that are checked out, modified, and staged by other members of the team. FastDASH is a tool that scans every single file that is edited/opened in every developer's local workspace and communicates about their changes back and forth through a central server.

Ariadne. Souza et al. [273] present the Ariadne tool which analyses software projects for dependencies and helps to find coordination problems through a visual environment. Ariadne retrieves updated code from the project codebase, only when a developer commits to the repository. As a result, they fail to employ any conflict prediction process before the code enters the repository; thereby failing to prevent inconsistencies from entering the codebase.

YooHoo. Holmes and Walker [139, 140] present YooHoo, an awareness system to help developers to keep apprised of code changes, providing notifications in a flexible manner. It works similarly to Ariadne.

WAV. Al-Ani et al. [6] present WAV (Worspace Activity Viewer). WAV visualises the developers and artefacts in a project using a 3D metaphor and gives managers an overview of ongoing activities in the project using information extracted directly from developers' workspaces.

SafeCommit. Wloka et al. [308] introduce SafeCommit. It performs deeper program dependence analysis by identifying changes that pass a given set of test cases. The proposed technique identifies changes that are covered by original and edited test suites (either pass or fail) as well as changes that do not have coverage, to identify changes that may fail a given test suite.

Syde. Hattori and Lanza [131] propose Syde. It creates an AST of a program for each developer to determine conflicts by comparing changed AST nodes. It informs the team about who is changing what parts of the system in real time. The AST of two developers is compared to detect the conflicts. This conflicting information is also given to the concerned developers. Syde uses different colours to indicate severity of conflict warnings. Conflicts in which one entity is already checked-in is considered severe and shown in red colour. Conflicts in which both the entities are not checked in are less severe and shown in yellow colour. Indirect conflicts are not supported by Syde.

CASI. Servant et al. [258] propose CASI. It provides a real-time visualisation of concurrent activities of collaborative developers on shared artefacts, to the granularity of individual methods to help developers detect conflicts early. CASI shows all the program elements that are influenced by the changes made in the team, so that developers can coordinate more efficiently.

CoDesign. Bang et al. [21] proposed CoDesign, a collaborative software modelling environment that supports system design in geographically distributed work settings. CoDesign's main contribution is an extensible conflict detection framework for collaborative modelling.

Crystal. Brun et al. [48–50] propose Crystal, which monitors selected branches in the repository. Crystal preemptively merges the branches in the background and will notify the developers of any conflicts that arise. It detects both direct conflicts (changes to the same line), and indirect conflicts (changes to a different line that cause build or test failures). The tool merges changes in a shadow repository as they are committed in order to catch these types of conflict as early as possible.

CSM. Huyen and Ochimizu [149] propose CSM (from Change Support Model) to construct the artefact-related part of the information repository. CSM is a combination of model-based approach, process support approach, and awareness support approach. CSM can provide change workers with very comprehensive views of shared artefacts.

Conflicts. Hattoti et al. [132] propose Conflicts which is built over Syde [131]. It extracts structural information from Java files using the AST structure and uses this to capture conflict information. It only captures conflicts caused due to syntactic changes and is implemented for two collaborative developers.

WeCode. Guimaraes and Silva [123] propose WeCode, which also merges in uncommitted code, in order to improve the time to detection of a merge conflict. Guimaraes and Silva

introduced a technique to continuously merge changes in the IDE in order to detect merge conflicts as soon as possible. To avoid sending notifications about easily resolved textual merges, they use structural merging. Test conflicts are identified by two different methods: *conflicts covered* by automated test cases are found by running the tests. If no appropriate test case is available, an *abstract semantic graph* is used to identify dependencies and highlight potential conflicts.

CloudStudio. Estler et al. [96] propose CloudStudio [96], a web based IDE integrating configuration management and real time awareness. The configuration management part implements the basic notions of SCM such as repositories, branches and push/pull operations. Each project is maintained as a master repository on the cloud and personal repository in each developer's workspace. Awareness system of CloudStudio shows lines with different colors in the editor which indicate changes done by other developers to those lines. This is possible because it maintains a database for each line which keeps information regarding all the users changing the line. Direct conflicts are shown in IDE when two developers modify the same line. Indirect conflicts can be determined by compiling the changes in IDE. CloudStudio also provides communication mediums like Skype and Chatbox.

Manhattan. Lanza et al. [173] propose Manhattan a tool that generates visualisations about team activity whenever a developer edits a class and notifies developers through a client program, in real time. While this shows useful 3D visualisations about merge conflicts in the IDE itself (thus being non-intrusive and natural to use), it is not adaptive (it does not automatically reflect any changes to the code in the visualisation, unless the user decides to re-import the code base).

Bellevue. Guzzi et al. [125] propose Bellevue. It is an IDE extension to make committed changes always visible, code history accessible inside developers' workspaces, and displays the historical changes in a non-obtrusive way. Bellevue offers an interactive view that shows detailed historical information for files and specific chunks with respect to a previous version. It also allows developers editing code while reviewing the history.

Semex. Nguyen et al. [218] propose Semex, a tool for detecting which combination of merged changes causes a test conflict based on a technique called variability-aware execution. First, the tool separates the changes done by each parent commit in the merge scenario and encodes each one using conditionals around them (if statements) to integrate all these changes in a single program. Semex then uses variability-aware execution to detect semantic conflicts by running existing project tests, if available, on this single program, exploring all possible combinations of the encoded changes. The tool then knows which combinations of commits lead to test failure and reports the set of commits that, if integrated, would cause a test conflict.

BDCI. Pastore et al. [230] propose Behaviour Driven Conflict Identification (BDCI) which focuses on a program's behaviour. BDCI identifies all methods that have been modified and monitors their entry and exit points during test executions to derive possible input and output values. In the case of conflicting changes to the observed values, higher-order conflicts are likely the cause.

CCPF. Arora et al. [18] propose the Continuous Conflict Predication Framework (CCPF). This framework describes a conflict prediction process with four steps: (i) collaborative software development activation, (ii) state information sharing, (iii) conflict prediction,

(iv) awareness visualisation which is executed continuously during the development process. During this process, the current activities of collaborative developers are monitored, within their Eclipse IDEs, and modifications made by them to various artefacts are used to predict potential direct and indirect conflicts. This process creates an inspection layer, well before the commit process, in order to identify arising conflicts and require developers to resolve these conflicts before moving ahead with the development process. The framework is realised through the implementation of collaboration over GitHub (CoG) tool [16, 17].

Wuensche prototype. Wuensche et al. [311] propose a prototype to find potential higher-order merge conflicts. Their approach uses a statically constructed call graph which reuses data from previous runs to scale well with very large source code repositories. They evaluate their prototype on known build and test conflicts in SAP HANA, a very large software project in C++, and examine the properties and root causes of those conflicts.

ConE. Maddila et al. [190] designed a method to help developers discover changes made on other branches that might conflict with their own changes. They propose *ConE*, a novel technique to (i) calculate the between two PRs that are active at the same time frame, and (ii) determine the existence of Rarely Concurrently Edited (RCE) files.

Support CI and CD

Development practices such as CI and CD also support early conflict detection, as the code is frequently integrated, verified by build and test scripts, and released to production [3]. We found only one study proposing an infrastructure to support continuous integration and delivery as we describe below.

SoftFab. Spanjers et al. [277] present SoftFab, an infrastructure which enables projects to automate the building and test process, and which manages all the tasks remotely through a control centre.

2.8 Conflict Resolution

Resolving merge conflicts is non-trivial, especially when the changes diverge significantly [49]. The resolution process can be tedious and can cause delays as developers figure out how to approach and resolve conflicts [163]. Poorly performed merge conflict resolutions have been known to cause integration errors [37], workflow disruptions, and jeopardise project efficiency and introduce delays [97].

Once merge conflicts have been detected, techniques are needed to resolve them. These can range from a manual - and often time-consuming - process, over an interactive with the software developer, to a fully automated conflict resolution tool. A lot depends on the kinds of conflicts the tool intends to solve and the level of accuracy that needs to be reached. Depending on the kind of merge conflict, different resolution strategies may be necessary. One particular kind of conflict occurs when two parallel changes can only be merged if they are applied in a certain order because the inverse order gives rise to an inconsistency. This is typically the case when a renaming is involved. Next, we present studies on different stages of merge conflict resolution.

We organised this section, presenting studies that measure the conflict resolution difficulty (Section 2.8.1), barriers and challenges to resolve conflicts (Section 2.8.2), conflict resolution

strategies (Section 2.8.3) and support the conflict resolution (Section 2.8.4). Then, we present evaluations to check whether the conflict resolution worked (Section 2.8.5) and backup strategies when the resolution did not work (Section 2.8.6).

2.8.1 Conflict Resolution Difficulty

We found four studies investigating the merge conflict resolution difficulty, we present them below from the oldest to the newest publication year.

Nelson et al. [214] presented an extension of the work of McKee et al. [196]. They investigated the whole life-cycle of conflict resolution taking practitioners' perspective into account with two surveys. One investigating barriers developers face when dealing with merge conflicts and the other survey investigating the processes developers follow to monitor, plan, and evaluate merge conflicts resolution. As a result regarding merge conflict resolution difficulty, developers reported 6 main factors: (i) complexity of the conflicting code, (ii) number of conflicting code locations, (iii) ownership of the conflicting code, (iv) size of the conflicting code, (v) approaching deadlines, and (vi) work schedule constraints. These concerns cause developers to alter their resolution strategy, and in some cases delay the resolution, which can have negative consequences.

Mahmoudi et al. [193] investigated the impact of refactorings in merge conflicts (see study description in Section 2.6.2). Regarding merge conflict resolution difficulty, they investigated whether conflicts that involve refactoring are more difficult to resolve. As a result, they found that conflicting regions that involve refactorings tend to be larger (i.e., more complex) than those without refactorings. Furthermore, conflicting merge scenarios with involved refactorings include more evolutionary changes (i.e., changes leading to conflict) than conflicting merge scenarios without involved refactorings.

Ghiotto et al. [110] investigated the nature of merge conflicts (see the study description in Section 2.4.6). Regarding merge conflict resolution difficulty, they created a Difficulty Ratio (DR). Their DR is equal to the number of complex conflicting chunks divided by the sum of the number of complex conflicting chunks and the number of straightforward conflicting chunks. For them, complex chunks involve the addition of new code when resolving the merge conflicts and, the resolution of straightforward conflicting chunks needs just one of the versions or the concatenation or combination of the conflicting code. The top 10 programming language constructs or combination of language constructs with larger DR of resolving conflicts are: (i) comment, method invocation, and variable, (ii) method invocation and variable, (iii) if statement, (iv) if statement, method invocation, and variable, (v) import, (vi) if statement and method invocation, (vii) variable, (viii) comment and variable, (ix) annotation and method declaration, and (x) comment and method declaration. Hence, they empirically showed that conflicts that involve method invocation and variables are more difficult to resolve than conflicts involving method declarations, for instance.

Brindescu et al. [42] predicted the difficulty of merge conflicts using five machine learning algorithms (Bagging (Bootstrap aggregating), Bayes Network (BayesNet), Multi-Layer Perceptron (Perceptron), Logistic Regression (LogReg), and SVM) in a sample of 128 Java projects. As a result, these machine learning algorithms achieved an AUC of 0.85, 0.78, 0.75, 0.73, 0.56 for Bootstrap aggregating, BayesNet, Perceptron, LogRegr, and SVM, respectively.

Their model identifies 21% of the conflicts as *Severe*, and the rest 79% are classified as *Trivial*. Investigating what makes a merge conflict difficult, they identified a subset of ten factors which are: (i) total (sum) cyclomatic complexity of all files modified in both branches, (ii) the maximum dependencies of all files modified in both branches, (iii) average dependencies of all files modified in both branches, (iv) average cyclomatic complexity of all files modified in both branches, (v) number of differences of the AST trees between all modified files in both branches, (vi) LOC differences between commits of source and target branches, (vii) lines of code changed, (viii) number of authors, (ix) length of commit pattern between branches, and (x) length of author pattern between branches. Interestingly, they identified differences between-human perceived features and machine-learned features for predicting the difficulty of merge conflicts, for example diffusion, which is perceived as unimportant by developers, but picked up as important by their model.

2.8.2 Barriers and Challenges to Resolve Conflicts

A few studies [110, 214] have found that when conflicts are complex, and generally require new code to be written, developers avoid resolving them. For instance, Ghiotto et al. [110] found that in these cases, developers just commented on the conflicting code in a way that does not break any tests.

Surveying developers, Nelson et al. [214] found 8 key barriers developers face when resolving conflicts: (i) understanding the conflicting code, (ii) having expertise in the area of conflicting code, (iii) having enough meta information about the conflicting code (who made the change, why, and when), (iv) having tools presenting understandable info, (v) changing assumptions within code, (vi) understanding the project structure, (vii) having trustworthiness of tools, and (viii) having informative of commit messages. Developers rely heavily on their knowledge of the conflicting code when implementing their merge resolutions.

2.8.3 Conflict Resolution Strategies

We divide this section into the process developers follow to resolve merge conflicts and strategies followed to resolve the merge conflict. Note that in the first case, we present studies showing the reasoning behind the merge conflict resolution. In the second case, we present studies investigating the solution based on the code to be merged.

Developer´s Process to Revolve Conflicts

We present studies from the oldest to the newest publication year.

Munson and Dewan [210] propose a number of different resolution strategies (which they refer to as merge policies) for three-way merging. All these resolution strategies are implemented in a uniform and customizable way using merge matrices that can be fine-tuned by the user. We only discuss two of these strategies: consolidation and reconciliation. *Consolidation* is used when two revisions of a common base version need to be merged and it is expected or known that most of the parallel changes are complementary (e.g., when the changes are made to different program modules). In case of deletions, the merge

proceeds automatically. In case of overlapping changes, one of both changes is chosen interactively. *Reconciliation* is used when the revisions to be merged are likely to introduce merge conflicts. In that case, simply selecting the appropriate revision in case of an overlap is not sufficient since one may interactively want to combine the changes that lead to the conflict. Reconciliation is typically needed if different software developers make independent changes to the same part of the code. If they make changes to different weakly coupled parts of the code, however, consolidation is the more appropriate strategy.

Edwards [93] proposes a number of interesting resolution strategies. The explosion strategy calculates all possible paths that may lead to a valid solution. Afterwards, the user can choose the most appropriate solution. The main disadvantage of this strategy is that it can lead to a combinatorial explosion of potential solutions. The promotion strategy tries to avoid conflicts by reducing the dependencies in a sequence of modification operations. Some operations that depend on earlier operations can be promoted into new operations that do not cause a dependency anymore. The recursive acceptance strategy represents a “what if" scenario in which the user can iteratively resolve cascading conflicts. If an operation conflicts with earlier operations, one can either remove the current operation or remove one of the earlier conflicting operations. This process is applied recursively until all conflicts have been resolved.

Nelson et al. [214] identified that developers normally have a strategy to resolve conflicts. Their strategy involves one or more of the following 5 steps: (i) examining the merge, (ii) analysis/manipulation of the code, (iii) examining the code, (iv) focus on design concerns, or (v) examine project organisation. They also found that when developers feel that their experience is not sufficient to resolve the merge conflict, they generally seek help from other developers to resolve the conflicts.

Brindescu et al. [43] showed that developers normally follow 6 steps on the conflict resolution: (i) look at external data sources, (ii) open a particular file to work on, (iii) read or scroll through the source code, (iv) edit source code, (v) read a chunk on either side, and (vi) run the build or perform test. In addition, Brindescu et al. [43] observed that when developers wanted a quick solution, developers skip directly to step v). Regarding the first step, which is look at external data sources, developers search for information on seven sources: (i) diff between merged versions, (ii) commit history, (iii) source code, (iv) output running the application, (v) build and tests output, (vi) documentation, and (vii) colleagues. Once developers got stuck, they follow two patterns: *StuckForaging* and *HuntingForEvidence*. *StuckForaging* shows that finding the right information is nontrivial. This is because of: (i) the way conflicts are presented in, at times, fragmented across multiple disconnected sources, and (ii) false leads that developers have to backtrack from. This can be distracting, as the information initially presented by the tools may not identify the root cause of the conflict. The information is often split across different types of tools and data archives (e.g., source code, version history, documentation, requirements etc.). Synthesizing the information to see if it is actually useful is not easy, which is highlighted by the *HuntingForEvidence* pattern. A corner case that made this especially difficult was when parallel changes had conflicting requirements, meaning the conflict resolution required a deeper solution since it involved design or fundamental codebase/behaviour decisions. Such decisions are crucial, as the wrong choice can lead to further problems later in a project’s evolution.

Developer ´s Solution to Resolve Conflicts

Yuzuki et al. [317] study the conflict resolutions at method level on 779 conflict commits from 10 Java projects. They argue that when developers are resolving conflicts they usually use merge tools such as KDIFF3, or DIFFMERGE. Hence, most developers visualise the parts where conflicts are detected and display them. Hence, developers fix code using that information. That is the reason why there is no case that new code which is not written in both are not used. Therefore, their classification consists of looking whether a developer chose the code from the source or target branch. They also argue that in the case that conflicts are detected in multiple methods, it is one of the most significant problems that developers must select which method should be resolved at first. It is impossible to obtain a satisfactory resolution with only such information [150, 221]. Like that, because existing merging tools do not give enough information for conflict resolution, it is considered that they have no choice but to adopt code from one of the merged branches. With all, their results show that 99% (771/779) of conflicts are resolved by adopting the source or the target branch code.

Nguyen et al. [216] investigated merge conflicts and resolutions of four OSS repositories. They report the rollback rate which means the percentage of times developers decided to not merge versions. The rollback rate varies from 0.70% to 6.98%. Most of the rollbacks happened without creating a new commit such as using the commands "`git checkout SHA-1-B1; git reset -soft HEAD; git commit -amend`". They considered four types of resolutions: (i) applying both changes, (ii) applying a change from one site only (either from source or target branch) , (iii) applying no changes, and (iv) other than these. They investigated cases that two continuous lines are in conflict (called adjacent-line conflicts) and looked at whether the conflict resolution applied to both site changes. In these cases the rate varied from 24.39% (in Samba) to 85.01% (in Linux Kernel). They found that across the four investigated repositories 75.41% of the conflicts were fixed by applying both changes. The authors considered these adjacent-line conflicts false positives arguing that developers did some unneeded extra work for solving the conflicts. For 20.93% of cases of adjacent-line conflicts, developers chose a change from one site only and, for 3.66%, developers chose applying no changes or other than these (e.g., adding new code).

Brindescu et al. [41] investigated 143 OSS projects aiming at better understanding the impact of merge conflicts and their effect on software quality (see study description in Section 2.6.2). Regarding the resolution strategies, they observed that developers follow three strategies: (i) *select one* means that the solution is the same as one of the branches; (ii) *interleave* means that solutions contains lines from both the branches and none of the lines were changed and no new lines were added; and, (iii) *adapted* means that existing lines were changed or/and new lines were added. As a result, they found that *adapted* was the most commonly used resolution strategy (60.82%), compared to *interleave* (26.38%) and *select one* (12.80%). When dealing with higher-order merge conflicts the percentage is even greater.

Ghiotto et al. [110] investigated the nature of merge conflicts (see the study description in Section 2.4.6). Their conflict resolution classification includes when developers choose: (i) the source branch version (e.g., often their version), (ii) the target version (current mainline version), (iii) a concatenation of both versions, (iv) a combination of both versions, and v) new code is necessary. Their results show that developers normally choose: (i) the source version (50%), (ii) target version (25%), (iii) new code (13%), (iv) combination (9%), and (v) concatenation (3%). Just 13% of the conflicting chunks involve new code, meaning that

in 87% of the cases, all of the code that is necessary to resolve a conflict already exists and is present in the two versions in conflict.

Shen et al. [261] investigated detection and resolution of merge conflicts (see study description in Section 2.4.6). They investigated, textual, build (compiling), and test (dynamic) conflicts. As a result related to how developers resolve conflicts, 69% of textual conflicts were handled by giving up the edits in one branch and the remaining 31% of conflicts by somehow integrating the edits from both sides. 95% of build conflicts were handled by applying extra edits to the integrated version. These extra edits were usually similar to some of the edits in one branch. Developers handled all test conflicts by applying extra edits to the integrated versions; 75% of these extra edits were similar to those from one branch.

Pan et al. [228] used program synthesis to learn merge conflict resolutions (see study description in Section 2.4.4). Regarding resolution patterns, they found that 39.5% of the resolution strategies involved concatenating the changes from source and target branches.

2.8.4 Conflict Resolution Support

In this section, we present approaches and tools to support merge conflict resolution. Note that we already presented merging algorithms (Section 2.5.2), merging tools (Section 2.5.3), and collaboration tools (Section 2.7.2) that might support merge conflict resolution. The difference between the tools and approaches presented in this section is that they were proposed to support conflict resolution, while the others were proposed to merge code or support multiple developers contributing to the source code. Studies are ordered by the oldest to the newest publication year.

Koegel et al. [170] propose an approach to collaborative merging to facilitate discussion on conflicts and collaborative conflict resolution. The approach is based on the application and integration of Rationale Management into Model Merging and they called their approach Issue-based Model Merging. In this approach conflicts are aggregated into an issue (similar to a GitHub issue). Alternative choices for resolving the conflicts are proposals. Their goal with the approach is to address? invisible design decisions, isolated design decisions, forced design decisions, and interactive merge.

Brosch et al. [44] propose to use a model checker to detect semantic merge conflicts in the context of model versioning. This technique is used to check the semantic consistency of an evolving UML sequence diagram with respect to state machine diagrams that remain unchanged. In other words, they use the overlapping parts of the diagrams as glueing points to construct a coherent picture of the system.

Niu et al. [222] develop scoreRec, a tool that recommends the conflict resolutions ordered by estimating the cost and benefit of resolving conflicts. The contributions of scoreRec lie in the leverage of both structural and semantic information of the source code to generate conflict resolution recommendations, as well as the hierarchical presentation of the recommendations with detailed explanations.

Costa et al [65] reported that developers usually have a hard time while branching merges because it might hold numerous contributions from different developers and they need to understand changes in order to integrate them. Based on this problem they propose a tool called TipMerge, which recommends expert developers that are best suited to resolve

conflicts in a particular area of code. TipMerge checks the developer's knowledge coverage of changed files and methods and shows which set of developers have the highest joint coverage (the Team Recommendation). As the number of developers in the ranking can be high, we use an optimization algorithm to find which developers make the best team to deal with a specific merge case. They evaluated it on 2040 merges across 25 OSS projects and found that TipMerge can improve joint knowledge coverage by an average of 49% in merge scenarios [66].

Nishimura and Maruyama [220] proposed MergeHelper, a tool that reduces the manual effort necessary to resolve merge conflicts by replaying fine-grained code changes related to conflicting class members. MergeHelper captures code changes as sequences of fine-grained atomic operations. This way, developers can replay all changes involved in a conflict, which can help in resolving them. It follows a similar approach than the one present in the first versions of MolhadoRef. Their approach only considers edits and has problems with long edit histories and finer granularity of operations [88].

Nelson et al. [214] obtained 10 overlapping tool-sets to support merge conflict resolution by surveying developers (see study description in Section 2.8.1). These tool-sets consists of: Git, Vim/vi, Text Editor (unspecified), GitDiff, GitHub, Eclipse, KDiff3, Meld, SourceTree, Sublime Text. Their results indicate that developers are concerned about tool-set fragmentation, and therefore adding an additional tool might be counterproductive to the workflow of most developers.

Xing and Maruyama [314] propose an approach to (semi-)automatically repair behavioral merge conflicts once they are found. For this purpose, they leverage Automated Program Repair (APR) techniques, which fully-automatically fix faults (or bugs) exposed by tests. APR is worth exploring to produce a behavioural-correct, executable merged program although it is subject to high degree of overfitting as pointed out by several studies [176, 267]. They assume that programmers should decide the behaviour of a merged program and usually prepare test cases to check if it behaves correctly in the early stage, when they address the resolution of merge conflicts. Using APR techniques based on a generate-and-validate strategy, each of the candidates generated from a faulty program will be validated with the given test cases. If a program that passes all the test cases is found, it can be considered a fixed one. In merging programs independently modified by multiple programmers, their test cases not only help both the programmers reach consensus on the behaviour of a merged program but also guarantee its behavioural correctness.

Sung et al. [284] develop MrgBldBrkFixer to evaluate the feasibility of automated fixes of merge induced build breaks. In other words, MrgBldBrkFixer resolves semantic merge conflict for a divergent fork by analysing the AST diffs for changes in the upstream to construct a patch for merge conflicts. MrgBldBrkFixer requires developers' manual work to classify the build breaks, and the tool heavily relies on the AST analysis for C++ code only. The types of resolution/fixes of conflicts are: (i) include statement update, (ii) entire function definition/call update (e.g., function body move/add/removal), (iii) function name update, (iv) function type/specifier update, (v) function param/argument update (e.g., param/arg add/rem/reorder), (vi) function param/arg's type update, (vii) class/namespace/enum reference update (e.g., field type update). Using real development data of Microsoft Edge collected in a three-month period, they performed a feasibility study and the result shows that 40% of the build breaks targeted by MrgBldBrkFixer can be repaired automatically.

Shen et al. [259] introduce SoManyConflicts, a tool to help the developer handle multiple merge conflicts in a more systematic way. Specifically, SoManyConflicts follows three steps: (i) analyses the interrelation between conflicts in terms of syntactic dependency, similarity, and hierarchy, and (ii) constructs a conflict-relation graph then (iii) applies classical graph algorithms to cluster and order related conflicts. Furthermore, SoManyConflicts can also suggest resolution strategies for those unresolved conflicts interactively based on those already resolved conflicts. The authors hope that SoManyConflicts can reduce the burden and error-proneness of conflict resolution, thus improving the productivity as well as the code quality in collaborative software development. Recruiting 5 developers from the OSS community and the industry the authors found that: (i) the grouped and ordered conflicts by SoManyConflicts make sense in most cases, and (ii) as more conflicts in a group are resolved, the precision of the suggested strategy for the rest of conflicts in the same group increases. However, they also found that: (i) some actually-related conflicts are not reported by SoManyConflicts, which means that it does not identify some other features indicating the relevance of conflicts, and (ii) some conflicts are false positives or mismatching reported by the merging algorithm in Git, which are expected to be identified to save the manual effort.

Zhang et al. [318] proposed GMerge, that automatically suggests merge conflict resolutions. GMerge takes as input a merge conflict and merge histories from both upstream and downstream. GMerge returns a conflict resolution which indicates which lines of code need to change, and how. To empirically evaluate GMerge, They selected real-world Microsoft Edge semantic merge conflicts. Their evaluation shows that GMerge learns the correct resolutions at the state-of-the-art 64.6% of accuracy. It demonstrates the effectiveness of k-shot learning, which provides a cost-effective and language-agnostic solution for real-world semantic merge conflicts. The closest work to GMerge is MrgBldBrkFixer [284]. However, while MrgBldBrkFixer requires developers' manual work to classify the build breaks, and the tool heavily relies on the AST analysis for C++ code only, GMerge is scalable, fully automated and language-agnostic by leveraging large scale language models.

2.8.5 Evaluating Whether the Resolution Worked

Surveying developers, Nelson et al. [214] found that developers normally use 6 strategies to check whether their conflict resolution was effective: (i) all tests pass, (ii) code successfully compiles, (iii) code looks correct (i.e., visual passes), (iv) version control system warnings are gone, (v) merged code is approved during code review, and (vi) merged code accepted into production code-base. In this process they mentioned 5 tool-sets [214]: (i) version control systems (e.g., Git, SVN, and CVS), (ii) continuous integration tools (e.g., Travis CI, Jenkins, and TFS), (iii) program analysis tools (e.g., Coverity and CodeSonar), (iv) devOps tools (e.g. Nagios, Monit, and Kabana), and (v) release management tools (e.g., Chef, Puppet, and Salt).

2.8.6 Backup Strategies

Surveying developers, Nelson et al. [214] found that when the conflict resolution did not work, developers mentioned 4 main backup strategies: (i) take it off-line - this strategy involves moving conflicting code away from shared branches or code repositories, and working locally to resolve the conflict without disrupting other developers, (ii) collaborating - where developers seek out other developers that are more knowledgeable about the area of conflicting code, (iii) try again (i.e., merge the same code together and hope that their tools are able to succeed with a second attempt, and (iv) redoing changes - by way of reverting and manually recreating the changes found in conflicting commits when their initial attempt failed. Surprisingly, 13.33% of the developers mentioned that they do not have a backup strategy.

2.9 Human Factor Investigations

There are several studies showing that human factors play an important role in software quality. These studies include investigations on developers productivity when they learn from experience of other developers individually, from groups, and from organisational-unit level [39] and the influence of the number of developers [198, 307], organisational structure [213], and code ownership [35, 52, 109, 119, 235, 237, 291] on the number of failures.

To mention a few of them, Bird et al. [35] investigated whether ownership influences the number of pre-release faults and post-release failures in the context of two commercial systems: Windows Vista and Windows 7. As a results, they found that: (i) developers who owns less than 5% of lines of code of components (named minor contributors) is more likely to introduce pre- and post release failures, (ii) higher levels of ownerships are related to fewer failures, (iii) the number of minor contributors negatively affects software quality, and (iv) without minor contributors, the ability to predict failure-prone components is greatly diminished, supporting the hypothesis that minor contributors are related to software quality. Similarly, Businge et al. [52] investigated the influence of ownership on the number of failures in the context of small-sized Android applications. As a result, concurring with Bird et al. [35], they found that minor contributors are related to more failures and applications with few major contributors are more reliable than applications with larger number of minor contributors. At the end, studies investigating the relation between code ownership and the number of failures found similar results and recommend that (i) changes made by minor contributors should be reviewed with more scrutiny, (ii) potential minor contributors should communicate desired changes to developers experienced with the respective file/binary, and (iii) components with low ownership should be given priority by quality assurance resources.

2.10 Conclusion and Perspectives

In this chapter, we saw that the social perspective is taken into account when proposing strategies, approaches, and tools for increasing coordination in software development (Section 2.1). We also saw that there are studies investigating human factors (Section 2.9) and the communication flow developers follow when contributing to the source code (Section 2.3). However, note that in none of these topics, merge conflicts are investigated. On the side, we saw in our extensive background on the merge conflict life-cycle that the social perspective is often ignored. We summarise the merge conflict investigations below.

Once **characterising merge conflicts** (Section 2.4), we saw that they are a solid problem in software development. Several researchers named merge conflicts as they are identified and not how they indeed are. Our merge conflict taxonomy might minimise such confusion and provide a standard nomenclature for future research. The conflict rate varies from 0.00% to 87.84% depending on aspects like the conflict type, version control system, merging strategy, number of projects, projects domain, programming language, coordination practices, contribution rules, and etc. Regarding the size and duration of merge conflicts, we saw that merge conflicts are often small with a median size up to 3 LOC and last for a few minutes or days, depending on project structure. An empirical study with several projects investigating the merge conflict duration might be useful to better understand the time merge conflicts last. Finally, we saw that changes in the same part of methods and constructors or in their signature are the most common language constructs related to merge conflicts. With all, we saw several studies investigating merge conflict characteristics and highlighting opportunities for further studies that are still an open gap in the literature.

Once **investigating merge strategies** (Section 2.5), we presented merge techniques that started ignoring the structure of the code and, later, started to look at the syntactic and semantic of the changes. On one hand, strategies looking at the semantic of the changes, remove conflicts due to ordering and formatting. On the other hand, it increases the runtime complexity of the underlying merging algorithms making them harder to be used in practice. We could see a similar evolution in the algorithms and tools related to software merging. In addition, we saw that merge tools are not only concerned with integrating, but also resolve some conflicts. Studies comparing merge tools and strategies are important to, not only deeply understand these tools and strategies, but also to see how they perform in practice. With all, we saw that merge strategies are evolving over the years and semantic/structured merge strategies are not far to be used in practice.

When **investigating factors related to merge conflicts** (Section 2.6), we identified eleven factors or related topics, which we present below.

1. *Why studies present different results related to measures related to merge conflicts?* The simple answer regards the used data and approach. For instance, while Leßenich et al. [180] investigated factors strongly correlated to merge conflicts. In some cases, they found a moderate correlation, but preferred to report that such factors show no influence on the occurrence of merge conflicts. A study investigating social factors with machine learning classifiers might provide different results.

2. *Work remotely is not related to the occurrence of merge conflicts*. Estler et al. [97] found that the location of developers does not influence the occurrence of merge conflicts.

3. *Changes across MVC slices are conflict-prone*. Dias et al. [82] show evidence that changes across MVC slices influence positively the occurrence of merge conflicts,

4. *Some programming languages are conflict-prone*. The results from Dias et al. [82] and Menezes et al. [199] show that some programming languages are more related to merge conflicts than others. However, as the difference is not thumping, other factors like team knowledge, supporting tools will have a great impact on the choice of a programming language for a project.

5. *Type of file and extension might influence the merge conflict rate*. Leßenich et al. [180] found a conflict rate of 11% when considering all files and 6% when considering only Java code. Contradictorily, Pan et al. [228] found that changes on programming language files are conflict-prone when compared to changes in non-programming language files. It shows that not all conflicts are in programming language files and these conflicts (e.g., in documentation, configuration, or generated files) might be simpler to resolve.

6. *Concurrent edit to files are buggy*. Maddila et al. [190] found that across all investigated repositories, the percentage of bug inducing edits is consistently higher for concurrently edited files than for non-concurrently edited ones. They observed that concurrent edits consistently are correlated with bug fixes, more so than non-concurrent edits and all edits.

7. *Conflict code is bug-prone*. Amaral et al. [8] found that conflicting merge scenarios are not prone to introduce bugs. However, bugs might be introduced on the merge conflict resolution. In the same direction, Brindescu et al. [41] found that code involved in merge conflicts has a higher likelihood of being involved with a future bug. They found that the code in merge conflicts were twice as likely to contain bugs as other changes. Further, if the changes included semantically interacting changes, the likelihood of a defect is 26 times that of non-conflicting changes. Hence, we can conclude that code related to conflicts is often buggy. One way to minimise the chances of introducing bugs is to change small pieces of code [236].

8. *High branch activity is related to more failures*. Shihab et al. [262] found that architectural and organisational mismatch is related to merge conflicts. For instance branches where more areas or components (architectural) or more managers and engineers (organisational) are involved are failure-prone. At the end, they conclude that high branch activity is associated with a higher number of failures. A way to minimise the number of failures and conflicts is to divide tasks and integrate branches more often.

9. *Conflict rate is similar to git merge and git rebase scenarios*. Ji et al. [154] found that practitioners will face similar conflict rates integrating code with rebasing scenarios (e.g., using *git rebase* command) than integrating code with common three-way merge scenarios (e.g., using *git merge* command).

10. *Conflict code is often related to low quality code*. Ahmed et al. [5] found that program elements involved in merge conflicts are smelly. More specifically, code classified as

God Class, *Data Clump*, *Sibling Duplication*, *Data Class* and *Distorted Hierarchy* is often a conflicting code. Note that these code smells are often related to agglomeration of functionality/responsibility of a component. The authors also found that the presence of code smells on the lines of code involved in a merge conflict has a significant impact on its bugginess which is also inline with the results from Brindescu et al. [41]. A way to improve the software quality and minimise conflicts is to periodically search for smelly code and remove them as soon as the symptom is identified.

11. *Refactoring is associated with merge conflicts*. Mahmoudi et al. [193] found that one in each five conflicts involve refactoring. They also found that conflict regions that involve refactorings tend to be larger. The top refactorings related to conflicts involve extracting code (*extract method*, *extract interface*, or *extract superclass*) from a component. For instance, a class or method is concentrating on multiple behaviours. Hence, developers modularize these behaviours in multiple methods. Note that this kind of refactoring is often related to ways to minimise code smells found by Ahmed et al. [5]. In other words, these refactorings support the reduction/removal of code smells often related to merge conflicts. Considering that merge conflicts in refactored code are often harder to resolve [87], a way to avoid concurrent changes is to schedule a period for developers to focus only on refactorings.

Looking at the measures from Table 2.1 and all factors presented in Section 2.6, we noted that simply looking at the size of changes does not help to predict merge conflicts. It is often related to (i) the parts of code changed (e.g., changes involving multiple layers), (ii) the quality of the code (e.g., smelly code), (iii) the duration of the merge scenario (e.g., branches forgotten by developers or that were very complex), and (iv) the type of the changes (e.g., refactoring). In the end, factors related to merge conflicts are often related to bad development and coordination practices. Creating processes and better coordinating social and technical assets may reduce the number of merge conflicts.

Once presenting studies investigating **ways to avoid merge conflicts** (Section 2.7). We saw that strategies and tools are often complementary. For instance, creating contribution rules and reducing the number of branches might reduce the number of merge conflicts. However, using a tool like Cassandra to better coordinate ongoing tasks and Crystal to keep developers aware of ongoing changes might be more effective.

Nelson et al. [214] surveyed developers and found that merge conflict resolution is normally (73.68%) reactive (i.e., after it occurs developers fix it). Only 14.71% of the developers monitor conflicts periodically. The surveyed developers mentioned 10 tools to monitor conflicts: Git, GitHub, Email, SVN, VCSs (in general), VisualStudio (IDE), PagerDuty, GitLab, Jenkins, and TFS (a VCS). The low percentage for monitoring conflicts used by the surveyed developers is due to the fact that these developers still do not trust existing tools. As evidence, Guzzi et al. [125] conducted an exploratory investigation and tool evaluation for supporting collaboration in teamwork. They found that, while automatic merge tools were used, developers did not trust them, and manually checked the end result.

Therefore, there is no silver bullet to avoid merge conflicts and its reduction depends on several factors. For instance, developer skill, task requirements, project´s restrictions (e.g., duration and costs), and project and company size. This last factor is related to how

a solution scales. For instance, speculative merging does not work for companies like Microsoft, Google, and Meta where tens of thousands of PRs are created every week.

Tools and strategies presented in Section 2.7, help us see that software development practices are evolving over the years and studies comparing strategies and tools are welcome to highlight limitations and challenges to improve the state-of-the-art of merge conflict avoidance. For instance, Kaur et al. [164] compared six collaboration and awareness tools (CollabVS, Crystal, Palantír, Syde, WeCode, CloudStudio). As a result of this comparison, they found 4 limitations: (i) increased false positives; (ii) overloading developers with notifications and warnings; (iii) privacy invasion; and, (iv) communication cost. Based on these limitations, they present requirements for this kind of tool. Regarding overloading developers with notifications and warnings, Estler et al. [97] found that real-time awareness can be distracting rather than helpful. An important task is then to find the correct trade-off for notifying developers.

When presenting studies related to **merge conflict resolution** (Section 2.8), we saw that some merge conflicts are harder than others, we saw that there are barriers when resolving conflicts often related to the knowledge and understanding of the whole change, including for instance, the project structure. We also saw how developers reason when thinking about conflict resolution and that the integrated code is often in the changeset. We saw that there are tools to support conflict resolution able to automatically resolve most merge conflicts and developers have in mind procedures to follow to check whether the conflict resolution worked and revert changes in case the resolution fails.

With all, we envision that: (i) merge conflict resolution might be part of the development environment, (ii) merging tools might be very conservative to resolve conflicts to gain developers trust, (iii) supporting tools should provide visualisations, meta-information, and recommendations of developers to resolve conflicts, and (iv) developers might create policies to resolve conflicts and have a wide test-suite to support identifying semantic and behavioural conflicts. Supporting some of our points, Nelson et al. [214] surveyed developers mentioned 4 main factors that need to be improved in the tool-sets they use: (i) usability, (ii) exploration of project history, (iii) less-relevant information filtering, and (iv) graphical presentation of information.

In the next chapter, we present an empirical study investigating the relation between the communication activity and the occurrence of merge conflicts.

3

On the Relation Between Communication Activity and Merge Conflicts

This chapter shares material with a prior publication: "On the Relation Between GitHub *Communication Activity and Merge Conflicts" [302]*

In this chapter, we empirically investigate the relation between communication activities and merge conflicts. In Section 3.1, we introduce this chapter presenting the context, problem, goals, and discussions of our study. In Section 3.2, we present the communication networks which are essential to understand how we investigate the communication activity. In Section 3.3, we present the study setting of this empirical study. In Section 3.4, we present our results. In Section 3.5, we discuss and show the value of our results. Finally, in Section 3.6, we conclude this chapter and present perspectives of this study.

3.1 Introduction

Software development is a collaborative and distributed activity in which success depends on the ability to coordinate social and technical assets [157]. In this collaborative process, developers are often supported by version control systems when solving tasks (e.g., bug fixing and adding new features). Version control systems help them to manage changes to a common code base by tracking all code contributions over time. This allows a group of developers to address different tasks simultaneously without losing changes. After fulfilling their tasks, developers merge the proposed changes to the main repository.

Simultaneous contributions to a common code base may introduce problems of their own during integration, often manifesting as merge conflicts (see Section 2.4). As merge conflicts are unexpected events, they have a negative effect on project's objectives compromising the project success, especially when arising often [123] [252] [163]. On the other hand, researchers found that proper communication among contributors is fundamental for the project success [34] [122] [256][271]. For instance, Liu et al. [188] found that GitHub communication supports a more coordinated development activity. Despite the number of studies exploring merge conflicts [2][11][49][123][180] and communication activity [75] [115]

[158] [229] [266] [296], the role of communication activity for the occurrence or avoidance of merge conflicts in practice has not been thoroughly investigated.

Our goal is to investigate and understand the relation between GitHub communication activity and merge conflicts. One of the reasons why communication is related to project success may be that keeping contributors aware of what others are doing may avoid the emergence of merge conflicts. Hence, to get a more precise understanding about what kind of communication may be helpful for avoiding merge conflicts, we use different measures of communication. For instance, the communication related to the merge scenario's code changes may be more efficient for avoiding merge conflicts than the general communication in the merge scenario. Or, the communication among *developers* only may be more important for avoiding merge conflicts than the communication among all *contributors*. Contributors are all GitHub users who have contributed to the project (e.g., communicating or changing the source code). A developer is a contributor who has changed the source code.

To achieve our goal, we have conducted a large empirical study analysing the history of 30 repositories of popular software projects. In total, we considered 19 thousand merge scenarios, 325 thousand files, and 1.5 million chunks. For this purpose, we mined and linked contributions (Git) and communication (GitHub) data. Regarding code contributions, we reconstructed the merge scenarios that are present in the subject projects' histories. Regarding communication activity, we quantified the amount of GitHub communication in merge scenarios by means of three alternative approaches with distinct granularity: one considering the communication of all active contributors (*awareness-based*), the second linking communication related to the merge scenario's contributions by means of pull requests and related issues (*pull-request-based*), and the third considering communication mapped to artefacts that has been changed in the merge scenario (*changed-artefact-based*). To obtain a deep understanding of the communication activity, we also distinguished between the communication related to contributors (*contributors' communication*) and developers (*developers' communication*), for each approach.

To understand the association between GitHub communication activity and the occurrence of merge conflicts (i.e., the two covariates), we performed three analyses. First, we analysed the *bivariate* relationship between the two covariates, as is common in empirical software engineering studies. Second, we analysed the *multivariate* relationship between the two covariates taking confounding factors into account (e.g., the number of files changed and developers involved in the merge scenario). Third, we analysed the *moderating influence* of individual merge scenario characteristics on the strength of the relation of the two covariates (e.g., the relation may be stronger in larger merge scenarios). For these analyses, we use Spearman's rank correlation, principal component analysis, partial correlation, and moderation effects.

Summarising our results, the bivariate analysis indicates a weak (< 0.3) but highly significant positive correlation between the number of merge conflicts and the amount of communication. The multivariate analysis indicates no relation between the two covariates which suggests that the positive relation between the two covariates found in the bivariate analysis is spurious under the assumption that both covariates are confounded by merge scenario characteristics. In the moderation effect analysis, we investigated the influence of three measures on the strength of the relation of the two covariates: the number of lines of code, the number of developers involved, and the number of days a merge scenario lasts.

Regarding the number of lines of code, we found a significant moderating influence on the strength of the relation between the two covariates for the awareness- and changed-artefact-based approaches and for both communication measures. Regarding the number of developers, we also found a significant moderating influence on such relations, however, the influence lasts only for the contributors' communication. Regarding the number of days, we found no significant moderating influence on the strength of the investigated relation. Note that the moderating effect analysis does not invalidate the multivariate analysis, it only presents results for the subset of "larger" merge scenarios. In practical terms, our results contradict the popular belief which suggests that a great communication activity helps to avoid merge conflicts, since we did not find a relation between GitHub communication activity and the occurrence of merge conflicts in the multivariate analysis.

Puzzled by our unexpected and negative results and aiming to get deeper into all our covariables, we analysed each of them separately, which supports the robustness and reliability of our methodology. From this further analysis, several topics for discussions arose. Most notably, (i) bivariate analysis is not enough to investigate the complex interplay of project success, communication, merge conflicts, and contextual factors, (ii) contributors and developers normally communicate independently of the emergence of merge conflicts, (iii) The size of the merge scenario's code change is not related to the number of developers involved, and (iv) speculative merge strategies (e.g., GitHub pull requests) drastically reduce the number of merge conflicts.

Overall, in this chapter, we make the following contributions:

- We present evidence that communication activity and merge conflicts are not related in the general case when controlling for confounding factors.
- We provide evidence that the developers' communication has a negative influence on the emergence of merge conflicts for the 10% largest merge scenarios in terms of lines of code. On the other hand, contributor's communication for the same setting has a positive influence on the emergence of merge conflicts. Therefore, developers' communication is more efficient for avoiding merge conflicts in the 10% largest merge scenarios than contributors' communication.
- We offer a rigorous methodological approach to multivariate analysis of correlation structures in socio-technical repository data analysis.
- We provide evidence by a manual analysis that merge scenarios with few developers and large changes are often related to bug fixing while merge scenarios with many developers and small changes are often related to the introduction of new features to the project.
- We provide evidence of the benefits of using speculative merge strategies (e.g., GitHub pull requests) by showing that the percentage of conflicting merge scenarios without using pull requests is 139 times greater than when using pull requests.
- We make our infrastructure publicly available to mine fine-grained information from software repositories.
- We make all data publicly available for replication and follow-up studies on a supplementary Web site [303].

3.2 Building Communication Networks

Considering experience from previous work presented in Section 2.3 and the benefits, comprehensiveness, and popularity of GitHub when compared to other communication tools and channels, we chose to rely in our study on the GitHub platform. Another benefit is that by using GitHub, projects should follow the three-way development model, hence, we can use the time a merge scenario lasts to define the analysis time span (i.e., not a predefined one). Aware of the drawbacks of using only one communication channel and considering that contributors may talk about topics not related to the code changes (e.g., usability or configuration problems), we pursue three approaches to capture communication amongst contributors (awareness-, pull-request-, and changed-artefact-based), and differentiate the communication of among all contributors (*contributors' communication*) and among developers only (*developers' communication*).

In Chapter 2, we presented how previous work has explored merge conflicts, communication activity, as well as the three-way development model and the communication flow in the three-way development model. In addition, we presented how communication activity may be useful for avoiding merge conflicts given the popular belief that communication and collaboration activities are mutually dependent for project success. Aiming to provide a clearer understanding of which GitHub communication activity may be more useful for avoiding merge conflicts, we create communication networks for each merge scenario using three approaches: *awareness-based*, *pull-request-based*, and *changed-artefact-based*, which vary in terms of granularity and coverage.

Communication networks are built from operational data from GitHub. Specifically, we queried the GitHub API retrieving all issue events (e.g., labelling, commenting, and opening) from each issue of each subject project. A network can be formalised as a graph $G = (V, E)$, where V is a set of vertices (contributors) and E is a set of edges (communication edges), denoted by *V(G)* and *E(G)*, respectively. An edge $e \in E$ between $u \in V$ and $v \in V$ is denoted by $e = \{u, v\}$. The three communication approaches as well as their purposes are described as follows.

Awareness-based approach. This approach links communication and contribution data by means of active contributors during a merge scenario. It tries to minimise the threat of using only one channel to capture a project's communication by building a graph of all contributors that communicate during a merge scenario. There are six steps to build communication networks using this approach (see Algorithm 1). For each merge scenario, we get all GitHub events during the merge scenario time range (step 1.1) and retrieve the issues these events belong to (step 1.2). Then, we determine all events related to these issues (step 1.3) and exclude events that happened after the merge, because these events are out of scope since the issue has already been addressed (step 1.4). Finally, we retrieve the set of developers who created the events (step 1.5) and build a full graph with them (step 1.6).

Pull-request-based approach. This approach links communication and contribution data by means of pull requests and their related issues. It is motivated by the flow of communication in a three-way development model (see Section 2.3). It is meant to retrieve a refined view on communication compared to the awareness-based approach, since it considers only communication of some issues that are related to the merge scenario and not

Algorithm 1 Awareness-based approach

input : MS, I ▷ *Sets of merge scenarios and issues*
output : NET ▷ *A tuple with a graph for each approach and merge scenario*
$E \leftarrow \{\ e \mid e \in \bigcup_{i \in I} i.events\ \}$ ▷ *Get all events of all issues*
for each ms in MS **do**
  $E_{ms} \leftarrow \{e \mid e \in E \wedge e.time \in [ms.bTime, ms.mTime]\}$ ▷ *Step 1.1*
  $I_{ms} \leftarrow \{e.issue \mid e \in E_{ms}\}$ ▷ *Step 1.2*
  $E_{ms} \leftarrow E_{ms} \cup (\bigcup_{i \in I_{ms}} i.events)$ ▷ *Step 1.3*
  $E_{ms} \leftarrow E_{ms} \setminus \{\ e \mid e \in E_{ms} \wedge e.time > ms.mTime\}$ ▷ *Step 1.4*
  $Contrib_{ms} \leftarrow \{e.contributor \mid e \in E_{ms}\}$ ▷ *Step 1.5*
  $Ed_{ms} \leftarrow \{\ \{c_1, c_2\ \} \mid c_1, c_2 \in Contrib_{ms} \wedge c_1 \neq c_2\}$ ▷ *Step 1.6*
  $NET.add(ms, G(Contrib_{ms}, Ed_{ms}))$
end for

all issues opened during the merge scenario. There are six steps to build communication networks using the pull-request-based approach (see Algorithm 2). For each merge scenario, we look for a pull request with the same hash as the merge commit (step 2.1) and mine the pull request body and comments to find related issues (step 2.2). This mining process consists of cleaning the text (removing blocks of code and external URLs because they may refer to issues of other projects) and looking for the pattern referring to other issues (`i.e., #([0-9]+), e.g., #1 or #123`). As blocks of code are between three quotation marks (‴) or indented by four spaces and, as URLs follow the pattern "char, slash (/), and char", we remove them. Then, we look at each remaining word of the text and check if it contains the pattern #([0-9]+). If so, we retrieve the GitHub issue. Our mining process is reliable since we followed instructions from the GitHub API documentation[1], tested, and we also checked whether the related issue exists in the repository before adding it into our analysis. After that, we get all events that happened in related issues (step 2.3), exclude the ones that happened after the merge commit (step 2.4), retrieve the set of developers that contribute to them (step 2.5), and build a full graph with these developers (step 2.6).

Algorithm 2 Pull-request-based approach

input : MS, I ▷ *Sets of merge scenarios and issues*
output : NET ▷ *A tuple with a graph for each approach and merge scenario*
for each ms in MS **do**
  $pr \leftarrow \{\ i \mid i \in I \wedge i.hash = ms.mergeCommitHash\ \}$ ▷ *Step 2.1*
  $RI_{ms} \leftarrow \{\ i \mid i \in I \wedge (\ i \in pr.relatedIssues \vee i = pr\ )\ \}$ ▷ *Step 2.2*
  $E_{ms} \leftarrow \{\ e \mid e \in (\bigcup_{i \in RI_{ms}} i.events)$ ▷ *Step 2.3*
  $E_{ms} \leftarrow E_{ms} \setminus \{\ e \mid e \in E_{ms} \wedge e.time > ms.mTime\}$ ▷ *Step 2.4*
  $Contrib_{ms} \leftarrow \{e.contributor \mid e \in E_{ms}\}$ ▷ *Step 2.5*
  $Ed_{ms} \leftarrow \{\ \{c_1, c_2\ \} \mid c_1, c_2 \in Contrib_{ms} \wedge c_1 \neq c_2\}$ ▷ *Step 2.6*
  $NET.add(ms, G(Contrib_m, Ed_{ms}))$
end for

1 https://developer.github.com/v3/

Changed-artefact-based approach. The main motivation for this approach is to obtain a finer-grained communication than the awareness-based approach with greater coverage than the pull-request-based approach. To achieve this, it links communication and contribution data by means of changed artefacts (files) referred through commits in opened issues in a merge scenario. In other words, we retrieve only communication via issues that discuss files changed in the merge scenario. So, like the pull-request-based approach, we are able to retrieve communication related to the merge scenario code changes, however, it is not pull request dependent. There are ten steps to build networks using the changed-artefact-based approach (see Algorithm 3). For each merge scenario, we determine the set of files changed in the merge scenario (step 3.1), all events that happened during the merge scenario (step 3.2), the issues each of these events belongs to (step 3.3), and all commits related to these issues (step 3.4). Commits can be related to issues in two ways: (i) contributors link the issue Identifier (ID) in the commit message, hence, they will be referred to in the GitHub API or (ii) contributors mention commit hashes in the issue's body or comments. After the first four steps, we refine the set of commits by keeping only the ones that have changed files modified in the merge scenario or in the merge commit (step 3.5). Then, we refine our set of issues related to the merge scenario by keeping only the ones that refer to the commits that changed files modified in the merge scenario (step 3.6). Next, we get a set of events that belong to these issues (step 3.7), exclude events after the merge (step 3.8), and get a set of contributors who created events in the remaining issues (step 3.9). Finally, we build a full graph with the remaining contributors (step 3.10).

Algorithm 3 Changed-artefact-based approach

$input : MS, I$ ▷ *Sets of merge scenarios and issues*
$output : NET$ ▷ *A tuple with a graph for each approach and merge scenario*
$E \leftarrow \{\ e \mid e \in \bigcup_{i \in I} i.events\ \}$ ▷ *Get all events of all issues*
for each ms in MS **do**
  $F_{ms} \leftarrow \{\ f \mid f \in ms.files\ \}$ ▷ *Step 3.1*
  $E_{ms} \leftarrow \{e \mid e \in E \wedge e.time \in [ms.bTime, ms.mTime]\}$ ▷ *Step 3.2*
  $I_{ms} \leftarrow \{e.issue \mid e \in E_{ms}\}$ ▷ *Step 3.3*
  $C_{ms} \leftarrow \{c \mid c \in \bigcup_{i \in I_{ms}} i.commits\ \}$ ▷ *Step 3.4*
  $C_{ms} \leftarrow \{c \mid c \in C_{ms} \wedge (c.files \in F_{ms} \vee c = ms.mC)\ \}$ ▷ *Step 3.5*
  $I_{ms} \leftarrow \{i \mid i \in I_{ms} \wedge i.commits \in C_{ms}\ \}$ ▷ *Step 3.6*
  $E_{ms} \leftarrow E_{ms} \cup (\bigcup_{i \in I_{ms}} i.events)$ ▷ *Step 3.7*
  $E_{ms} \leftarrow E_{ms} \setminus \{\ e \mid e \in E_{ms} \wedge e.time > ms.mTime\}$ ▷ *Step 3.8*
  $Contrib_{ms} \leftarrow \{e.contributor \mid e \in E_{ms}\}$ ▷ *Step 3.9*
  $Ed_{ms} \leftarrow \{\ \{c_1, c_2\ \} \mid c_1, c_2 \in Contrib_{ms} \wedge c_1 \neq c_2\}$ ▷ *Step 3.10*
  $NET.add(ms, G(Contrib_{ms}, Ed_{ms}))$
end for

Note that our setup for the second and third approach ensures that communication is related to the merge scenario code changes. We know all GitHub communication related to a given merge scenario. This does not mean that the developers talked about the conflict or a potential conflict, but that they communicated to make others aware of their code changes. As we consider making developers aware of a key to avoid merge conflicts, we would like to know whether communication on a merge scenario (not on something else) leads to

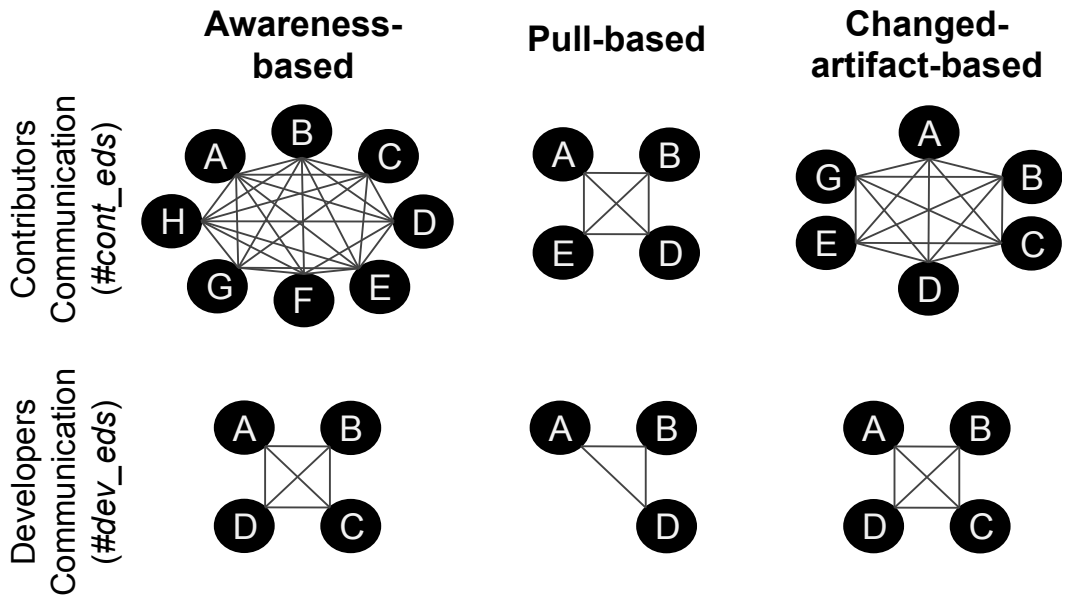

Figure 3.1: Communication Networks, based on the Example of Figure 2.1

less/more merge conflicts. Looking at the amount of communication (e.g., the number of GitHub commentaries) when the conflict happened hints at the merge conflicts *resolution strategies*, which is a different story.

Contributors' communication versus developers' communication. Contributors in the communication networks are people that communicate during a merge scenario, but not necessarily coded. Developers are people that communicate and coded during a merge scenario. Hence, developers are a subset of contributors.

Example of communication networks. The top of Figure 3.1 shows the three networks created from the example of Figure 2.1. As we can see, the *awareness-based network* contains more contributors and edges than the other networks, since it includes the communication of all four opened issues during the merge scenario. The *pull-request-based network* contains only contributors from issues #1 and #2, since issue #2 is the pull request (merge commit: 718ec42), and contributors of this issue indicate that issue #1 is related to the problem description. The *changed-artefact-based network* is between the two other networks in terms of size, since it contains developers from issues #1, #2, and #4. Even though issue #4 is not directly related to the merge scenario, two developers who contribute to the merge scenario (DevB and DevC) refer to commits present there. Hence, this communication may have been important to make developers aware of the merge scenario code changes.

Note that, in this example, for all communication network approaches, there is communication among non-developers. Therefore, to have an understanding about the communication among developers only, in addition to the three approaches, we distinguish between the communication of all contributors (#*cont_eds*) – *contributors' communication* – from the communication among developers only (#*dev_eds*) – *developers' communication*. The bottom of Figure 3.1 shows the networks for each approach containing only edges among developers. As we can see, the number of edges among developers is smaller for each approach since the total developers' communication is a subset of the contributors' communication.

3.3 Study Setting

In this section, we present the study setting of this empirical study, whose overall goal is to *investigate and provide an understanding on the role of* GitHub *communication activity for the occurrence of merge conflicts in the context of the pull-based development model*. First, we describe research questions and hypotheses (Section 3.3.1) followed by explanations of how we selected subject projects (Section 3.3.2). Then, we present our approach to retrieve contribution and communication data (Section 3.3.3). Finally, we explain how we answered the research questions (Section 3.3.4).

3.3.1 Overview of the Explored Relation in Each Research Question

Our discussion of the literature has shown how painful merge conflicts are for the project objectives and how essential communication activity is for the project success (see Chapter 2). Nevertheless, despite the plausible connection between communication activity and merge conflicts, the role of communication activity for merge conflicts to occur or to be avoided has not been thoroughly investigated. This motivated our first RQ:

> **RQ$_1$**: *Is there a correlation between* GitHub *communication activity and the occurrence of merge conflicts?*

This RQ addresses the *direct* relation between communication activity and merge conflicts in our subject projects. This direct correlation between merge conflicts and communication activity may be influenced due to the presence of confounding factors (i.e., merge scenarios' characteristics, such as size, number of developers, and duration), causing a spurious correlation. Only if this correlation still holds true after accounting for confounding factors, an interpretation is legitimate and we may interpret the results. This leads us to our second RQ.

> **RQ$_2$**: *How does the correlation between* GitHub *communication activity and merge conflicts change when taking confounding factors into account?*

RQ$_1$ and RQ$_2$ concentrate on the correlation between GitHub communication activity and merge conflicts for all subject merge scenarios. However, it is likely that the strength of this correlation depends on the characteristics of the merge scenario in question. For instance, it seems reasonable to expect that the correlation is different for small and large merge scenarios. *Moderation effects* arise when a situation where a third variable determines how strong a relationship between two variables is [289]. If we provide evidence that additional covariates influence the strength of the relation of communication activity and the occurrence of merge conflicts, we may find, for instance, that an intensive communication activity becomes fundamental to avoid merge conflicts only in very large merge scenarios. This motivates our third RQ:

Table 3.1: Statistics Captured for each Merge Scenario

Measure	Description
	Merge conflict measure
#conflicts	Number of merge (chunk) conflicts present in the merge scenario
	Communication measures
#cont_eds	Number of pairs of contributors who communicate in a merge scenario
#dev_eds	Number of pairs of developers who modified the code and communicate in a merge scenario
	Context variables
#lines	Number of modified lines of code in the merge scenario
#chunks	Number of chunks modified in the merge scenario
#files	Number of files modified in the merge scenario
#devs	Number of distinct developers who contributed to the merge scenario
#commits	Number of commits in the merge scenario
#days	Number of days a merge scenario lasts

> **RQ$_3$**: *What is the influence of merge scenario characteristics on the strength of the relation between* GitHub *communication activity and the occurrence of merge conflicts?*

To answer RQ$_3$, we formulate the following hypotheses:

> **H$_1$**: *The larger a merge scenario is, the stronger is the relation between* GitHub *communication activity and merge conflicts.*
>
> **H$_2$**: *The more developers are involved in a merge scenario, the stronger is the relation between* GitHub *communication activity and merge conflicts.*
>
> **H$_3$**: *The longer the merge scenario is, the stronger is the relation between* GitHub *communication activity and merge conflicts.*

To measure these relations, we compute a total of 8 variables. In Table 3.1, we describe all variables explored in this study. In Figure 3.2, we illustrate the relationships we investigate representing RQs by means of the measures presented in Table 3.1. So, as we use two measures of communication for each out of the three communication network approaches (see Section 3.2), we analyse the relation between GitHub communication activity and merge conflicts six times for each RQ.

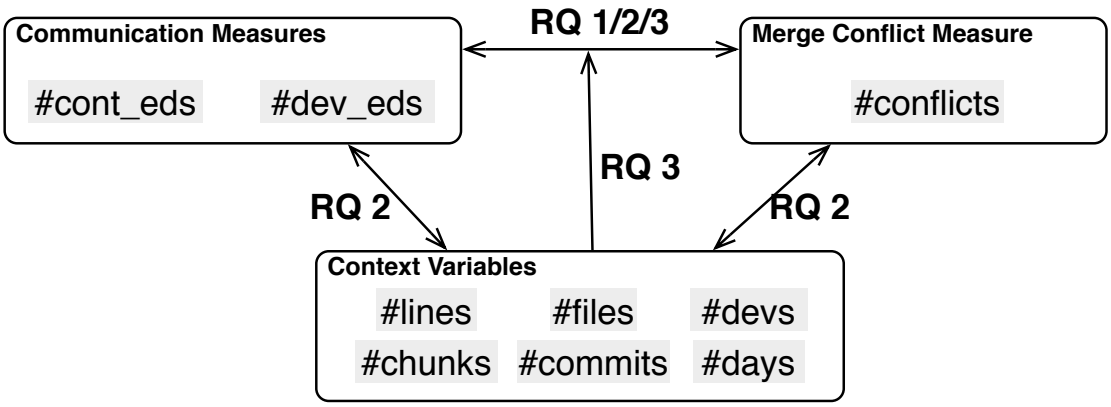

Figure 3.2: Research Questions and Hypotheses Model

3.3.2 Subject Projects and Experiment Setup

Overall, we selected 30 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. The reasons why we chose GitHub were described in Section 2.3 and 3.2. We selected the corpus as follows. First, we retrieved the 150 most popular projects on GitHub, as determined by the number of stars [40]. Then, we applied the 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 50 issues and 50 pull requests, (iii) projects with less than two commits and two GitHub events per month in the last six months, (iv) projects in which it was not possible to reconstruct most of the merge scenarios, and (v) the balance of the main programming language of the subject projects.

We created these filters inspired by Kalliamvakou et al. [162]. 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 and third filters capture active community projects on GitHub and not just mirror projects, such as Linux's mirror on GitHub. The fourth filter excludes projects such as *kubernetes*[2] and *moby*[3] because we considered that these projects do not mostly use the pull-based model (i.e., do not follow the three-step merge [115]) and they could bias our analyses. Details of how we rebuild merge scenarios are described in Section A.3.1. As most of the popular projects are developed in JavaScript, in the fifth filter, we excluded 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 arrived at 30 projects developed in 16 programming languages (i.e., a project can be developed using more than one programming language), such as JavaScript, CSS, and C++, containing around 19 thousand merge scenarios that involve 325 thousand files changed, 1.5 million chunks, 14 thousand contributors, and 134 thousand commits. In Table 3.2, we provide

2 https://github.com/kubernetes/kubernetes
3 https://github.com/moby/moby

information and statistics of each subject project. More details, such as the subject project's URLs, are available on the supplementary Web site [303].

We rebuilt the merge scenarios from the whole history subject projects history. Aiming at a fairer analysis, in Table 3.3, we present four refinements that we did in our merge scenario dataset: (i) keep only scenarios created after opening the first GitHub issue of the project, because it is not possible to recover communication from before, (ii) keep only merge scenarios from which more than one branch has been touched, because only in these cases merge conflicts may arise, (iii) keep only merge scenarios to which multiple developers were contributing, because only these scenarios need to keep developers aware of the change of others, and (iv) keep only merge scenarios that have been integrated using pull requests. The last refinement is applied only for the analysis using the pull-request-based approach (see Section 3.2). We discuss insights from this table in Section 3.5 since they are important to understand contribution activity, but not fundamental to answer our RQs.

3.3.3 Data Acquisition

Given that software development is social in nature, we build socio-technical relationships to obtain an authentic representation of developers' contribution and communication.

Code Contributions. Our strategy for contribution data acquisition is presented in Appendix A.3.1. To illustrate the code contribution data acquisition, for the example presented in Figure 2.1, we obtain 3 merge conflicts (#*conflicts*), 19 lines of code (#*lines*), 2 files (#files) and 4 chunks (#*chunks*) changed, 4 commits (#*commits*), 4 developers (#*devs*), and the merge scenario lasts 2 days (#*days*).

Communication activity. Considering experience from related work presented in Section 3 and the benefits, comprehensiveness, and popularity of GitHub when compared to other communication tools and channels, we chose to rely in our study on the GitHub platform. Another benefit is that by using GitHub, projects should follow the pull-based development model, hence, we can use the time a merge scenario lasts to define the analysis time span (i.e., not a predefined one). Aware of the drawbacks of using only one communication channel and considering that contributors may talk about topics not related to the code changes (e.g., usability or configuration problems), we select only projects that extensively use GitHub communication mechanism (see filters (ii) and (iii) of Section 3.3.2), pursue three approaches to capture communication amongst contributors, and differentiate the communication of among all contributors (*contributors' communication*) and among developers only (*developers' communication*), as we presented in Section 3.2.

Analysis Scripts and Data Availability. Our analysis framework is presented in Chapter A, analysis scripts (R) 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 [303].

Table 3.2: Overview of the Subject Projects

Subject Project	Main Prog. Language	#Stars	#MS	#Files	#Cont.
animate.css	CSS	34 290	151	1 392	186
javascript	JavaScript	73 792	548	1 871	727
jquery	JavaScript	49 498	248	4 322	632
vue	JavaScript	108 362	160	2 258	523
html5-boilerplate	JavaScript	41 001	229	1 274	378
electron	C++	62 713	2 845	40 540	1 013
awesome-python	Python	52 886	434	469	661
reveal.js	JavaScript	41 519	612	6 617	469
Semantic-UI	JavaScript	42 171	719	51 856	421
socket.io	JavaScript	42 871	422	3 785	346
express	JavaScript	39 339	284	2 608	455
redux	JavaScript	42 784	469	6 199	883
moment	JavaScript	37 973	991	24 621	831
create-react-app	JavaScript	52 636	524	14 568	963
nw.js	C++	34 057	539	13 598	161
impress.js	JavaScript	33 747	158	584	184
Chart.js	JavaScript	33 359	669	6 603	562
flask	Python	37 517	667	7 202	730
material-design-lite	HTML	30 411	674	6 841	258
httpie	Python	36 137	76	412	132
material-design-icons	CSS	35 462	22	3 707	48
jekyll	Ruby	34 899	1 464	16 703	1 068
AFNetworking	Objective-C	31 328	701	4 213	657
thefuck	Python	36 360	354	3 705	153
normalize.css	CSS	31 763	84	353	138
requests	Python	33 652	1 144	7 365	788
RxJava	Java	34 395	1470	36 642	360
public-apis	Python	40 317	560	887	472
lantern	Go	36 312	1 616	53 472	94
awesome-machine-learning	Python	34 290	408	415	343

#Stars, #MS, #Files, and #Cont. denote the number of GitHub *stars, the number of merge scenarios, the number of changed files, and the number of contributors.*

Table 3.3: Overview of the Refinements Applied to our Dataset of Merge Scenarios

Refinement	#MS	#MS by IN	#CMS	Conflict Rate
Initial number (IN)	19 232	100.00%	1 079	5.61%
Possible to communicate	18 607	96.75%	1 041	5.59%
Both branches touched	7 769	40.40%	1 041	13.40%
Multiple developers	6 487	33.73%	858	13.23%
Use pull requests	3 436	17.87%	7	0.20%

#MS and #CMS denote the number of merge scenarios and the number of conflicting merge scenarios. Conflict Rate stands for the #CMS divided by #MS

3.3.4 Operationalization

We operationalize our research questions and hypotheses through several variables. In what follows, we explain how we will answer each research question.

Answering RQ$_1$. To answer RQ$_1$, we check whether there is a correlation between the number of conflicts and the amount of GitHub communication. To quantify this correlation, we compute the bivariate correlation of the number of conflicts (*#conflicts*) and communication measures (*#cont_eds* and *#dev_eds*) for each communication approach. As we have two measures for communication and three approaches, we compute the correlation six times. We use Spearman's rank correlation because it is reliable when the observed covariates are count data and highly skewed. Spearman's rank correlation is +1 in the case of a perfect monotonic correlation, -1 in the case of a perfect reverse monotonic correlation. Values around 0 imply no monotonic relation between the variables [152].

Answering RQ$_2$. To answer RQ$_2$, we perform a multivariate analysis involving principal component analysis [159] and partial correlation [167] (both based on Spearman's rank correlation). Using a principal component analysis, we reduce the number of dimensions we have (i.e., one dimension for each variable of Table 3.1) to the first two principal components that retain a maximum share of common variance, which simplifies the discussion of the correlation structure. We choose to use partial correlation coefficients for three main reasons: (i) it is simple and straightforward, (ii) it does not require assumptions on the distribution, as parametric models, like regression models and structural equation models would, and (iii) it does not introduce any form of assumed causality between X and Y, like a regression model would. The partial correlation of X (i.e., the number of conflicts) and Y (i.e., the communication measure), taking into account Z (i.e., a confounding factor) is defined by

$$\rho_{XY|Z} = \frac{\rho_{XY} - \rho_{XZ} \cdot \rho_{ZY}}{\sqrt{1-\rho_{XZ}^2} \cdot \sqrt{1-\rho_{ZY}^2}}. \tag{3.1}$$

When there is more than one variable Z like in our case (i.e., the context variables of Table 3.1), the partial correlation is computed based on the residual variance of X and Y

after partialling out the correlations with all the confounding factors. That is, $\rho_{XY|Z_1,...Z_k}$ is equal to the correlation of the residuals of a regression of X and Y on $Z_1, \ldots, Z_k$ [167].

Answering RQ$_3$. To answer RQ$_3$, we use partial correlations, as we did in RQ$_2$, and measure the strength of the respective moderation effects by splitting the sample according to the number of code of lines changed (H$_1$), the number of developers involved (H$_2$), and the time range (H$_3$) of a merge scenario. As our data are right skewed, most merge scenarios are small, short, and have few developers involved. Hence, to get merge scenarios that are in fact large, long, and with many developers involved, we split the sample according to the 90% rule, as a previous study suggests [299]. The correlation significance test is based on the maximum likelihood algorithm-based estimation, which converges only when there is enough variation in every covariate. As result of this analysis, no matter where we split the sample (e.g., 50%, 70%, or 90%), whenever the algorithm converged, it led to the same general conclusions with similar levels of significance. Specifically, the 90% rule assures a relatively equal coverage of all projects (i.e., there is a relative homogeneity across projects) and chooses only "large" merge scenarios (with respect to the measure for each hypothesis) in all projects. The results of the splitting rule analysis, further details, and data to test it can be found on our supplementary Web site [303].

P-value correction for multiple hypothesis testing. To answer our RQs, we conduct various significance tests, asking whether the observed effects are statistically significant. When offering multiple potential covariates to the number of merge conflicts, we augment the chance of a type-one-error, that is, we augment the chance of finding at least one significant relationship. For answering RQ$_1$ and RQ$_2$, we test three potential covariates per hypothesis (that is one for each communication approach). For RQ$_3$, we test three potential partners for four different hypotheses: correlation in upper and lower quantile, and with *#cont_eds* or *#dev_eds*. Therefore, for all three RQs, the chance of at least one type-one-error when using $\alpha = 0.05$ is $1 - 0.95^3 \approx 14.3\%$. To be significant at a 5% level, we require the test p-values to be smaller than $1 - \sqrt[3]{0.95} \approx 1.7\%$ [310].

3.4 Results

In this section, we present the results of our empirical study, structured according to our RQs. For short, in Section 3.4.1, we present the results of the correlation analysis between GitHub communication activity and the occurrence of merge conflicts. In Section 3.4.2, we present the results of the correlation analysis between GitHub communication activity and the occurrence of merge conflicts taking confounding factors into account. In Section 3.4.3, the present the results of the analysis investigating the influence of merge scenario characteristics on the strength of the relation between GitHub communication activity and the occurrence of merge conflicts.

Table 3.4: Spearman's Correlation Between the Subject Measures

	Awareness-based		**Pull-request-based**		**Changed-artefact-based**	
	#cont_eds	*#dev_eds*	*#cont_eds*	*#dev_eds*	*#cont_eds*	*#dev_eds*
RQ_1	*0.192***	*0.221***	*−0.004*	*0.003*	*0.237***	*0.222***
RQ_2	*0.012***	*−0.006***	*−0.011*	*0.001*	*−0.010***	*−0.003***

** $p < 0.003 \cong \alpha = 0.01$ *(Correction for multiple hypothesis testing).*

3.4.1 RQ_1: Is There a Correlation Between GitHub Communication Activity and the Occurrence of Merge Conflicts?

In Table 3.4, we present the results of Spearman's rank correlation analysis for merge conflicts (*#conflicts*) and communication measures (*#cont_eds* and *#dev_eds*) for each communication approach proposed. As we can see, the estimated correlation of the number of conflicts with the contributors' communication is rather weak *0.192* (awareness-based), *−0.004* (pull-request-based), and *0.237* (changed-artefact-based). Regarding the correlation between the number of conflicts and the developers' communication, the coefficients are *0.221* (awareness-based), *0.003* (pull-request-based), and *0.222* (changed-artefact-based). Despite being weak (smaller than 0.3), the correlation coefficients for the awareness-based and changed-artefact-based approach measures are significant at a 99% confidence level, whereas in the pull-request-based approach the correlation coefficients are not significant.

Comparing the correlations for the different communication measures (*#cont_eds* and *#dev_eds*), the changed-artefact-based approach coefficients are greater than the awareness-based approach coefficients. Another point to note is that, in the case of awareness-based approach measures, the coefficient for the developers' communication is greater than contributors' communication while the opposite is true for the changed-artefact-based approach measures.

RQ_1 Summary: Overall, the bivariate correlation analysis shows a significant weak positive correlation for awareness-based and changed-artefact-based communication approaches with the number of merge conflicts. In practical terms, more GitHub communication can be observed in merge scenarios with more merge conflicts.

3.4.2 RQ_2: How Does the Correlation Between GitHub Communication Activity and Merge Conflicts Change When Taking Confounding Factors into Account?

In Figure 3.3, we show the two dimensional output from the principal component analysis for each communication approach, which covers *71.9% (57.2% + 14.7%)*, *57.2% (44% + 13.2%)*, and *73.5% (58.4% + 15.1%)* of the total variance for the awareness-based, pull-request-based, and changed-artefact-based communication approaches, respectively. The arrows represent the weights of each variable in the respective principal component and its

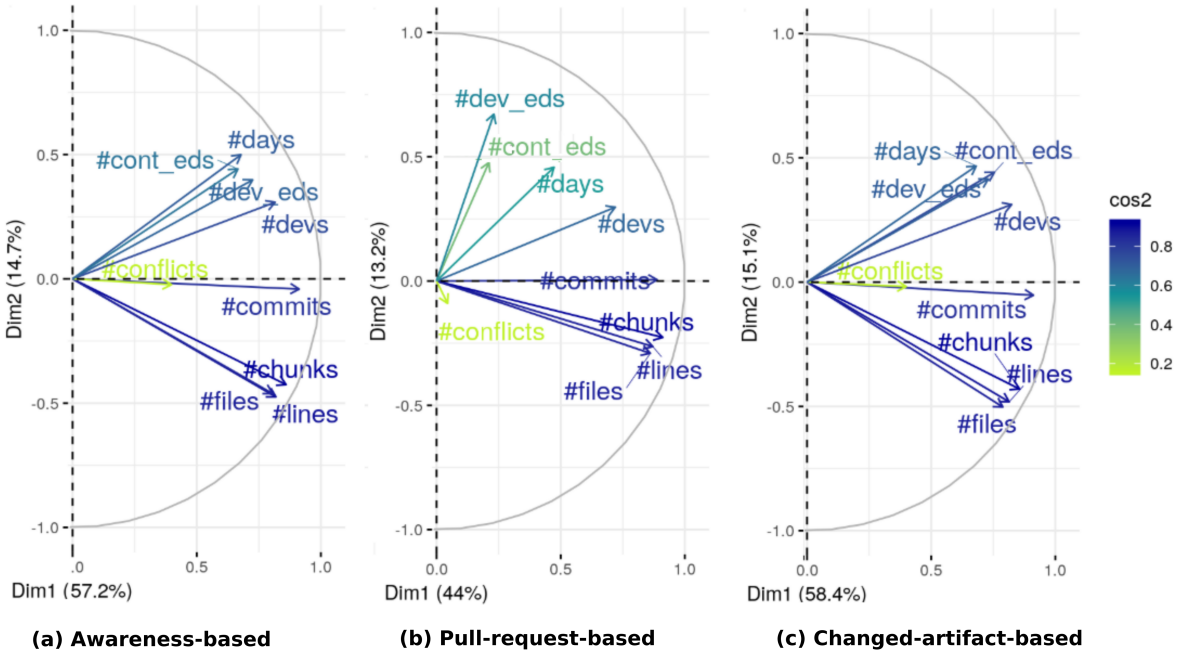

Figure 3.3: Principal Component Analysis of our Covariables

colour 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.

The data from Figure 3.3 suggest classifying the confounding variables into three groups (size, social dimension, and commit activity). The arrows representing *#chunks*, *#files*, and *#lines* point into the same direction; they represent the *size* of a merge scenario. Pointing to another direction, *#devs* and *#days* correlate strongly, and it is therefore legitimate to say that usually, merge scenarios with more developers take longer until they come to their end. We call this group the *social dimension*. The variable *#commits* is "undecided" between the two groups. This is not surprising because a large number of commits can either result from the participation of many developers or from larger merge scenarios as we will discuss in Section 3.5. Therefore, we keep this variable separate, in a group named *commit activity*.

In Table 3.4, we present the results of our multivariate analysis using partial correlation for the three proposed approaches (below the answer of RQ_1). When considering the contributors' communication, the correlation coefficients are 0.012 (awareness-based), -0.011 (pull-request-based), -0.010 (changed-artefact-based). When considering the developers' communication, in the same order, the correlation coefficients are -0.006, 0.001, and -0.003. None of the six values are not significantly different from zero.

Table 3.5: Median Splits and Correlations Between Number of Conflicts and Communication Measures

Hyp.	Mod.	Comm.	Awareness-based		Changed-artefact-based	
			$\hat{\rho}$ lower	$\hat{\rho}$ upper	$\hat{\rho}$ lower	$\hat{\rho}$ upper
H_1	*#lines*	*#cont_eds*	*0.008*	*0.113**	*0.016*	*0.139***
		#dev_eds	*0.003*	*−0.097**	*0.010*	*−0.097**
H_2	*#devs*	*#cont_eds*	*−0.019*	*0.130***	*−0.013*	*0.216***
		#dev_eds	*−0.038**	*−0.070*	*−0.035**	*−0.025*
H_3	*#days*	*#cont_eds*	*−0.008*	*0.017*	*0.007*	*0.015*
		#dev_eds	*−0.005*	*−0.054*	*0.003*	*−0.068*

Hyp., Mod., Comm., $\hat{\rho}$ lower and $\hat{\rho}$ upper stand for hypotheses, the name of the moderator variable, the communication measure, the estimated rank correlation for the lower and upper split sample, respectively. * $p < 0.017 \cong \alpha = 0.05$, ** $p < 0.003 \cong \alpha = 0.01$.

RQ_2 Summary: Accounting for confounding factors via partial correlations, the positive correlations found in RQ_1 disappear. In other words, the multivariate analysis reveals that there is no relation between the communication measures and number of merge conflicts when taking confounding factors into account. In practical terms, GitHub communication activity does not correlate with the occurrence or avoidance of merge conflicts.

3.4.3 RQ_3: What Is the Influence of Merge Scenario Characteristics on the Strength of the Relation Between GitHub Communication Activity and the Occurrence of Merge Conflicts?

As illustrated in Figure 3.3, the size measures (i.e., *#lines*, *#chunks*, and *#files*) are highly correlated, so we choose the *#lines* measure to represent merge scenario size when answering RQ_3. In Table 3.5, we show the results of the estimates of the partial correlation for the awareness-based and changed-artefact-based networks. We do not present the results for the pull-request-based approach, because, similar to RQ_1 and RQ_2, they are not significant for any hypothesis. We explain the reasons for these not significant values for the pull-request-based approach in Section 3.5.

As our hypotheses make assumptions on the upper split (i.e., about the larger, longer, and with many developers involved in the merge scenarios), we focus our answers only on significant values of the upper split. Overall, we found significant values only for H_1 and H_2. For both approaches and hypotheses, the correlation coefficients are significant and positive when considering the contributors' communication. Hence, we accept H_1 and H_2 when considering the *#cont_eds* measure. For both approaches, the correlation regarding developers' communication is significant for H_1. Therefore, we accept H_1 as there is a

stronger negative correlation between *#dev_eds* and the number of conflicts in the larger merge scenarios. Even though it may seem at first sight, our results for RQ_3 do not contradict the results for RQ_2. There is no global monotonic relationship (RQ_2), however, there is a relation for "larger" merge scenarios (RQ_3) in terms of changed lines of code (H_1) and number of developers (H_2).

RQ_3 Summary: Regarding H_1, for awareness-based and changed-artefact-based approaches, we found a weak but statistically significant positive correlation *#cont_eds* measure. For the awareness-based approach, we further found a weak but significant negative correlation when using *#dev_eds* measure. Therefore, for these cases, we accept H_1. Regarding H_2, for both approaches, we found a weak significant positive correlation for *#cont_eds* measure. Therefore, for this measure, we accept H_2. Regarding H_3, we did not find any significant correlation, so we reject H_3 for all cases. We can conclude that merge scenarios' size and the number of developers involved influence the strength of the relation between GitHub communication activity and the occurrence of merge conflicts when investigating the "larger" merge scenarios. In practical terms, (i) in the 10% larger merge scenarios more communication activity among all contributors using the awareness- and changed-artefact-based approaches are associated with more merge conflicts, (ii) in the 10% larger merge scenarios more communication activity among among developers only using the awareness- and changed-artefact-based approaches are associated with less merge conflicts, and (iii) in the 10% merge scenarios with more developers involved, more communication among all contributors using the awareness- and changed-artefact-based approaches correlate with more merge conflicts. In all these cases, we found a weak but significant correlation.

3.5 Discussion

Given the popular belief that communication and collaboration success are mutually dependent (see Chapter 2), our results can be seen as a *negative result*, finding no indication for a global monotonic relationship between the amount of GitHub communication and the occurrence of merge conflicts for the majority of merge scenarios (answer of RQ_2). Even when considering the "larger" merge scenarios, with the moderation effects analysis (answer of RQ_3), we found only a weak correlation. As negative results are often suspected to be due to the failure of the research design [85] [180] [223] [239], we start with a discussion of potential threats to validity of our study, before we present the implications of our study for researchers and practitioners.

3.5.1 Threats to Validity

External validity. External validity is threatened mainly by three 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 models is limited. This limitation of the sample was necessary to reduce the influence of confounds, increasing internal validity, though [263].

While more research is needed to generalise to other version control systems, development models, and communication platforms, we are confident that we selected and analysed 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 3.3.2).

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 minimise this threat, we manually looked at 50 issues randomly selected from 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) 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 3.3.2–3.3.3, 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, the need for triangulation through interview data. Interviewing developers could make our analyses and findings more reliable, however, as 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 will discuss in Sections 3.5.2–3.5.3.

Internal validity. There are two major threats to the internal validity of our study. First, we may miss important communication since developers may use different communication channels (e.g., e-mail lists, IRCs, Gitter, Slack). This fact may also influence how developers communicate on the GitHub platform. We mitigate this threat by using the awareness-based approach. This communication approach gets the communication of all active contributors in the merge scenario even if they mostly use other channels or do not talk about merge scenario code changes in GitHub. This is still a limitation if contributors completely ignore GitHub to communicate. However, as discussed in Section 2.3, GitHub is one of the widely-used channels [229] and together with the filters presented in Section 3.3.2, we assume that we selected only projects that extensively use GitHub to communicate.

Second, developers may talk about other topics not related to the merge scenario code changes [15]. To mitigate this threat, we have considered two communication approaches (pull-request-based and changed-artefact-based) able to capture a focused communication (i.e., related to the merge scenario code changes). The pull-request-based approach considers only the communication of the merge scenario pull request and related issues. The changed-artefact-based approach, on the other hand, considers only the communication of opened issues during the merge scenario that contain commits that touch files changed in the merge scenario (see Section 3.2 for more details).

3.5.2 Insights and Implications for Researchers

Longer merge scenarios with more developers involve more GitHub communication, but not necessarily more merge conflicts. In Figure 3.3, we illustrate that the more time and developers are involved in a merge scenario, the more communication activity we observe. However, this does not mean more merge conflicts. To better understand it in practice we did a manual and random analysis. This analysis consisted of looking at 25 issues and 25 pull-requests per subject project to understand *how* contributors interact and *what* they are talking *about*. As result, we identified two main patterns: (i) using pull requests, developers present their questions about their on-going code changes, and many contributors (including non-developers) help them with their questions, and (ii) for issues (excluding pull requests), contributors normally describe bugs, new issues, or problems (e.g., problems when configuring the tool or showing that the issue is duplicated). Therefore, we may say that most GitHub communication in issues (excluding pull requests) happens even before developers create or change a branch to implement the issue's solution. Therefore, despite being different both issue and pull-request communication are important and related to the merge scenario code changes.

The size of the merge scenario's code change is not related to the number of developers involved. It seems obvious that the more time or developers are involved in a merge scenario, the larger the code changes will be. However, our principal component analysis reveals that this relationship is not so trivial. In other words, more developers and time do not mean larger changes. To understand the context of the changes, we manually analysed 100 merge scenarios, split into two patterns that contradicts our first thoughts: (i) short merge scenarios, with few developers involved, and with many changes and (ii) long merge scenarios, with many developers involved, and few changes. Here, we present one example for each pattern. To represent the former pattern we chose a merge scenario from project *lantern*[4] that lasted only 4 days, with only two developers involved. However, 156 thousand lines of code of 3 380 chunks distributed in 446 files were changed. This merge scenario is related to solving a critical bug that came from the upgrade of a third tool that does not work in the Safari browser. To represent the second pattern we chose a merge scenario from project *public-apis*[5] that lasted 96 days, with 65 developers involved, however, only 130 lines of code of 39 chunks of one file were changed. This merge scenario consisted of adding new code. For short, they only updated the README.md file. Regarding GitHub communication activity, in the first example (pattern i), very few communication either among contributors or among developers only was found. In the second example (pattern ii), a high level of contributors' communication activity was found, however, few developers' communication was found. This communication behaviour applies to all communication approaches. These two examples represent well the analysed set since many merge scenarios with few developers and large changes (pattern i) were related to bug fixing while many merge scenarios with many developers and small changes (pattern ii) were related to the introduction of new features to the project. In a nutshell, the type and criticality of the change may influence the characteristics of the code changes (size, time, and developers

4 GitHub.com/getlantern/lantern; commit 86be2a8
5 github.com/toddmotto/public-apis; commit 0870841

involved), as well as how contributors communicate in general. We leave the discussion regarding merge conflicts to the next paragraph.

Are large merge scenarios conflict-prone? Against our expectations, we did not find a strong correlation between the size of merge scenario code changes (in terms of number of files, chunks and lines of code involved) and the occurrence of merge conflicts. The results for RQ_3 The reason may be the location of the changes (e.g., which files, branches, and architectural layers the changes happened to). For instance, a merge scenario of the project *RxJava*[6] changed 642 files, 10 011 chunks, involving 44 developers and took 260 days. However, it has no merge conflicts. It is important to highlight that both branches involved complex changes, such as removal of files and semantic changes. One thing that contributed to the absence of merge conflicts is that in this example no file was changed in both branches and most of the changes happened in one branch (44 developers changed 612 files in the target branch while 4 developers changed 30 files in the source branch). Developers that changed the source branch also changed the target branch. Therefore, the size of the changes of a merge scenario is not sufficient to predict merge conflicts. In this vein, Leßenich et al. [180] tried to predict merge conflicts, but even though they have used factors that practitioners indicated to be related to the emergence of merge conflicts (e.g., scattering degree among classes, commit density, number of files), none of these factors had a strong correlation with the occurrence of merge conflicts. This shows together with our results that predicting merge conflicts is not trivial and further investigation is still necessary.

Contributors and developers communicate regularly to keep others informed. By differentiating the communication of contributors and developers, we also learn whether there is a difference in the effect of communication of either of the two groups on the occurrence of merge conflicts. Unexpectedly neither of the two communication groups correlates with the number of merge conflicts when taking confounding factors into account (RQ_2). In Figure 3.4, we compare contributors (*#cont_eds*) and developers (*#dev_eds*) communication for each merge scenario and approach. The difference in the scale of the x-axis and y-axis, in which y-axis is much greater, means that most communication happens between contributors that do not change the source code. On average, *#cont_eds* is equal to 33 019 (awareness-based), 3.25 (pull-request-based) and 2 439 (changed-artefact-based), whereas *#dev_eds* is 11.22 (awareness-based), 0.15 (pull-request-based), and 9.37 (changed-artefact-based). In addition, the average of #devs is 5.85 for the dataset used to the awareness-based and change-artefact-based approach analyses and 4.23 for the dataset used for the pull-request-based approach analysis. So, most of the GitHub communication takes place among contributors. Given the average of *#devs_eds* for awareness-based and changed-artefact-based, we can assume that developers also communicate, although less than non-developers. This amount of communication among contributors and also among developers only brings us to the conclusion that in general developers and non-developers communicate to keep others aware of their contribution in a merge scenario.

Developers' communication is more efficient than contributors' communication for avoiding merge conflicts. As stated by the last paragraph, contributors and developers communicate among them and among each other. Therefore, contributors normally keep others aware of their code changes. For the general case, there is no relation between the

6 github.com/ReactiveX/RxJava; commit 25ebda

Figure 3.4: Contributors' Versus Developers' Communication

GitHub communication and the occurrence of merge conflicts. However, when considering the 10% larger merge scenarios in terms of lines of code contributors' and developers' communication have a different behaviour. While contributors' communication is related to an increase in the number of merge conflicts, the developers' communication is related to a decrease in the number of merge conflicts. Therefore, for the larger merge scenarios, developers' communication is helpful for avoiding merge conflicts being also more efficient than the contributors' communication.

Communication approach does not change the understanding on the occurrence of merge conflicts. We consider the three communication approaches used in this study (awareness-, pull-request-, and changed-artefact-based) valid to measure the communication granularity they were proposed to measure. Yet, independent of the communication approach, we did not find different relations on the occurrence of merge conflicts in the general case (RQ_2). Therefore, we cannot point out what type of communication approach is best to avoid merge conflicts. In any event, there are three points that we can discuss: (i) as the pull-request-based approach is dependent of GitHub pull-requests and most of the merge scenarios are not integrated by means of pull-requests, this communication approach has a limited applicability; (ii) developers normally talk about the changed artefacts since the average of the amount of communication of the awareness-based approach (general GitHub communication) and the changed-artefact-based approach (GitHub communication related to artefact changed) are similar (11.22 edges against 9.37); and, (iii) regarding contributors' communication the awareness-based approach and the changed-artefact-based approach have a very different average (33 019 edges against 2 439).

Limitations of bivariate analysis. The bivariate analysis is simple and intuitive, as it directly quantifies the correlation of interest. On the other hand, the multivariate analysis takes other factors that may influence the correlation of interest into account. Looking at the answers to RQ_1 and RQ2, our results for these two analyses are contradictory. Even though we are aware that the bivariate analysis is not enough to investigate the complex interplay of project success, communication, and contextual factors, our intention by presenting both results is to highlight the limitations of bivariate analysis. In other words, a simple and intuitive analysis may provide wrong results, hence it is necessary to reflect on the *big-picture* before determining which variables should be analysed and modelled.

A causal perspective. Our results for RQ_2 reveal a no significant relation between GitHub communication activity and merge conflicts. Therefore, more communication activity does not imply fewer or more merge conflicts, and more or fewer merge conflicts does not result from a lack of communication. Such causal analysis of the relation between GitHub communication and merge conflicts can only be achieved with a more detailed understanding of the timely order of events (e.g., communication before and after commits), which is out of scope of this study. For that reason, our results are limited to determining whether there is a correlation among these two covariables or not.

Why do not researchers investigate the relation between merge conflicts and communication activities? As seen in Chapter 2, there are dozens of studies investigating merge conflicts and communication activities individually. However, none of them in fact investigate the relation and influences of communication on the occurrence of merge conflicts. We have the assumption that this is not a trivial topic and requires a great infrastructure to do such an investigation. We hope our study and open-source infrastructure foment research in this topic.

3.5.3 Insights and Implications for Practitioners

Number of commits should be used with care. The number of commits is a metric often used by practitioners to obtain a feeling of the contributions of others as well as to predict conflicts [115] [180]. However, even though this metric correlates most with the number of conflicts in our study (compare arrow of *#commits* and *#conflicts* in Figure 3.3 (a) and (c)), the correlation is weak (≈ 0.2). Hence, this metric should be used with care, as it depends on how developers commit their code (e.g., for each function or for each feature implemented). For instance, in the project *lantern*[7] there are two commits from the same developer, but following completely different patterns. The former has changed 527 files, 3 379 chunks, 175 458 lines of code, and it is not involved in a merge conflict. The latter has changed only 1 file, chunk, and line of code, but it was involved in a merge conflict. Two points to highlight are that (i) these commits were part of merge scenarios in which the two merged branches were changed and (ii) the target developer made substantial semantic changes (i.e., not only formatting or ordering changes).

Pull requests lead to fewer merge conflicts. As can be seen in the fourth row of Table 3.3, 6 487 merge scenarios have both branches touched and have multiple developers contributing to it. In addition, we see in the last row of Table 3.3 that of 3 436 merge scenarios using pull

7 github.com/getlantern/lantern; commits:9d0bbbb and 6b6b534

requests only 7 have 9 merge conflicts (2 merge scenarios have 2 conflicts) of which 0.2% are conflicting merge scenarios. Subtracting the merge scenarios that use pull requests from the ones with both branches touched and with multiple developers, we get 3051 merge scenarios that were merged without pull requests (e.g., by the git merge command) of which 851 conflicting merge scenarios can be found. Hence, these merge scenarios present a share of 27.89% of conflicting merge scenarios. In other words, the share of conflicting merge scenarios without using pull requests is 139 times greater than when using pull requests.

The low number of merge conflicts when integrating merge scenarios by using pull requests is likely the reason for why we did not find significant correlation values when answering all RQs for the pull-request-based approach. The reason for the low number of merge conflicts comes from the fact the pull requests simulate the merge. With the simulation, developers have the chance of changing the source code before merging the branches (i.e., avoiding merge conflicts). This raises two questions: *Why are pull requests not used in most of the merge scenarios?* And, *why some merge scenarios still have merge conflicts when using pull requests?* For the first question, it is necessary to conduct interviews with developers, but we understand that, for some merge scenarios, it is simpler to merge branches locally. Regarding the second question, we manually checked each of these conflicting merge scenarios aiming at discovering why developers did not remove the conflicts before merging branches. We observed two things. First, the conflicts are in files that do not "break" the code and are normally in files not tested, such as *README.md* or *.gitignore*. Second, a commit resolving the conflict is added little time after the merge by another developer. Our assumption is that, whoever merged the code, merged because it did not fail in the test suit and another developer, maybe more experienced, chose the "best" option later on. Given the low number of merge conflicts when using pull requests, developers should adhere to such practice as well as to continuous integrations, and awareness tools in their workday tasks.

3.6 Final Remarks and Perspectives

Software development is a collaborative activity where success depends on the ability to coordinate social and technical assets. Software merging is a challenging and tedious task in the practice of collaborative and concurrent software development, mainly when merge conflicts arise. Merge conflicts are unexpected events that have a negative effect on a project's objectives, compromising the project budget and schedule, especially when they arise often. On the other hand, empirical research has found evidence for a beneficial effect of communication on project coordination and success. So, it is believed that great communication activity helps to avoid merge conflicts. However, in spite of such belief the role of communication activity for merge conflicts to occur or to be avoided had not been thoroughly investigated.

Aiming to investigate the relation between GitHub communication and the occurrence of merge conflicts (i.e., the two covariates), we rebuilt merge scenarios' contributions and communications of 30 subject projects. To get a deep understanding of communication work practices in these projects, we used three communication approaches (awareness-, pull-request-, and changed-artefact-based) and differentiate the communication among all

contributors and the communication among developers only. Our investigation comprises three three analyses. First, we started investigating the direct correlation among these two covariates as is common in empirical software engineering studies. As result, we found a weak but significant positive correlation when using the awareness- and changed-artefact-based approaches. Despite being simple and intuitive, this bivariate analysis does not consider the complex interplay of project success, communication, merge conflicts, and contextual factors, which may have led us to a misleading conclusion. Aiming to properly explore this interplay, we performed a multivariate analysis. In this analysis, we found no significant relation between communication measures and number of merge conflicts. We conclude that the bivariate analysis is spurious. Considering that it is likely that the strength of the relation between the two covariates depends on merge scenario characteristics, we performed a moderation effect analysis. As a result of the moderation effect analysis, we found that the number of lines of code and the number of developers involved in the merge scenario influences the strength of such relation. Thus, there is no overall monotonic relation, but there is a relation for "larger" merge scenarios.

The seemingly contradictory results from our three analyses should alert the reader that overly simplistic bivariate analysis can lead to wrong conclusions. To avoid such mistakes, it is necessary to reflect on the *big-picture* to determine which variables should be analysed and modelled when investigating complex environments such as collaborative software development.

Our results contradict the common belief that communication is beneficial for avoiding merge conflicts. If this was the case, we would expect a strong negative correlation between the communication activity and the number of merge conflicts. Puzzled by our results and to ensure that they are robust, reliable, and straightforward, we provided triangulation through a manual investigation of the merge scenarios' contribution and communication activity separately.

Regarding the merge scenarios' contribution, our results suggest that:

- The number of developers do not influence the size of the code changes in a merge scenario,
- More time and developers are not accompanied by more merge conflicts,
- The type of the change influences the size of merge scenarios. Bug fixing normally represents short time-life scenarios, with few developers, and large code changes. The introduction of new code (features), on the other hand, represents long time-life scenarios, with many developers, and few code changes,
- Larger changes are not more conflicting-prone than smaller changes. It suggests that the location of the changes are more related to the emergence of merge conflicts than the size of the change,
- The number of commits should be used with care since it depends on how developers commit to the project. As exemplified, the same developer may follow different committing patterns, and
- The use of pull requests reduces the number of merge conflicts compared to merge scenarios integrated without pull requests 139 times.

Regarding the communication activity, our results suggest that:

- Indeed contributors and developers normally communicate. Hence, our unexpected results did not come from a lack of communication or because we were not able to retrieve such communication
- GitHub issue and pull-request communication are both important to understand merge scenario code changes. However, the former is more related to problem clarification and the latter is more related to the ongoing code changes,
- The communication approach choice (awareness-, pull-request-, and changed-artefact-based) does not change the results of the multivariate analysis which means that one is not better than the others for avoiding merge conflicts, and
- For the moderation effect analysis, the developers' communication was shown to be more efficient than the contributors' communication for avoiding merge conflicts.

In the next chapter, we investigate whether it is possible to predict merge conflicts using only social measures and the predictions of merge conflicts taking social and technical assets into account.

4

Predicting Merge Conflicts Considering Social and Technical Assets

This chapter shares material with a prior publication: "Predicting Merge Conflicts Considering Social and Technical Assets" [298]

In this chapter, we investigate merge conflict prediction taking social and technical assets into account. In Section 4.1, we introduce this chapter presenting the context, problem, and goals of our study. In Section 4.2, we present our classification of developers. In Section 4.3, we present the study setting of this empirical study. In Section 4.4, we present our results followed by our discussion on the topic (Section 4.5) and threat to validity of our study (Section 4.6). Finally, in Section 4.7, we conclude this chapter.

4.1 Introduction

Successful collaborative software development depends on the ability to coordinate technical and social assets [156]. Version control systems help developers to manage concurrent contributions across a project's evolution [323]. Although typically a large number of commits cleanly merge, concurrent changes can overlap, leading to *merge conflicts* (see Chapter 2). When merge strategies are inefficient in reducing the number of merge conflicts, developers should continuously integrate their changes and keep aware of what others are doing. To support awareness, researchers have developed tools to alert developers about potential merge conflicts before they become too complex [32, 81, 163, 252]. Awareness tools speculatively pull and merge all combinations of available branches. The downside is that, constantly pulling and merging a large number of branch combinations, can quickly get prohibitively expensive [49]. One opportunity for decreasing this cost is to reduce the number of speculative merging operations in merge scenarios concentrating only the ones that are prone to conflict. To achieve this, researchers use machine learning techniques for predicting merge conflicts [226].

As we saw in Section 2.6, previous work concentrated on the prediction of merge conflicts considering technical assets and often ignored the social perspective (i.e., developers and

their relationship). Thus, as current merge conflict predictions, in terms of recall, are low, we hypothesise that information on social aspects might increase recall when predicting merge conflicts. Since coding is a social task, it might be simple for developers to know their role and relationship with other developers in a merge scenario. Hence, in addition to reducing the costs of speculative merging techniques, an understanding of the influence of the social dimension (e.g., developer role) on the emergence of merge conflicts might be useful to guide the coordination of developers aiming at reducing the number of merge conflicts. To illustrate how useful knowing the developer's role who caused the merge conflict can be for project coordination, we selected a merge scenario of project *create-react-app*[1]. In this merge scenario, 62 developers changed 651 chunks distributed into 121 files. Despite the high number of developers involved, the top contributors of the two merged branches introduced all conflicting code. Therefore, by making these developers aware of the other code changes, they could have communicated to understand the changes avoiding the merge conflicts or, at least, simplify the conflict resolution since they could explain their changes to each other and decide together what should remain in the target branch.

Our overall goal is to predict merge conflicts taking the social dimension into account. To achieve our goal, we have conducted a large empirical study analysing the history of 66 repositories of popular software projects with a total of 78740 merge scenarios. We classified developers as top and occasional based on their code contributions with distinct granularity (project and merge-scenario level). Aiming at increasing our knowledge on developer roles, we first look at the relation of each role separately, then we combine project- and merge-scenario-level information. Later, after getting this initial understanding of the relation between developer roles and merge conflicts, we devised three models to predict merge conflicts. We used three classifiers (decision tree, random forest, and K-Nearest Neighbours (KNN)), and seven balancing techniques (e.g., Synthetic Minority Oversampling TEchnique (SMOTE), Adasyn, and over-sampling). The first model is composed of only social measures, the second is composed of only technical measures, and the third model is composed of all (social and technical) measures. Creating these three models enables us to pin down how different measures influence the predictions and if social measures are useful in practice.

We found that top contributors slightly contribute to more merge conflicts at project level, and occasional contributors contribute to more merge conflicts than top contributors at merge-scenario level. When combining the granularity, we found that top contributors at project level that are occasional contributors at merge-scenario level are more related to merge conflicts than all other combinations of developer roles. When these developers touch (i.e., add, delete, change) the source branch, the chances of merge conflicts are 32.31%. Regarding predictions, random forest performed better in most cases and our models can correctly predict all conflicting scenarios (i.e., it achieved 100% of recall). Looking at other performance measures (e.g., precision, f1-score, accuracy, and AUC), the models with all and only technical measures performed better than the model composed by only social measures.

Albeit technical assets have proven essential to predict merge conflicts, our findings shall call the attention of researchers and practitioners to focus on social assets and the branches developers are touching in their analyses.

1 https://github.com/facebookincubator/create-react-app/commit/1e83e8

Overall, we make the following contributions:

- We provide evidence that it is possible to predict merge conflicts by looking only at social measures (e.g., developer roles, the number of developers involved, and the branch the developers touch);
- We analyse the relation between developer roles and merge conflicts from three different perspectives: (i) with developer roles investigated individually, (ii) with developer roles at project and branch level combined, and (iii) using machine learning classifiers with three models (i.e., social measures vs. technical measures vs. all measures).
- We show that code changes in the *source* branch are more conflict-prone than code changes in the *target* branch. For instance, when top and occasional contributors at merge-scenario level touch the source branch, 4.36% and 24.60% of the merge scenarios lead to conflicts, respectively. On the other hand, only 4.88% and 8.32% of the merge scenarios lead to conflicts when top and occasional contributors touch the target branch.

4.2 Developer Roles

Previous work [36, 70, 90, 156, 158, 205, 243, 290] had classified developers into core and peripheral roles aiming at understanding the organisational structure of open source projects. Mockus et al. [205] found empirical evidence for the *Mozilla browser* and the *Apache Web server* that a small number of developers are responsible for approximately 80% of the code modifications. Their approach consists of counting the number of commits made by each developer and then computing a threshold at the 80% percentile. Developers with a commit count above the threshold are considered core and, developers below the threshold are considered peripheral. They rationalised this threshold by observing that the number of commits made by developers typically follows a *Zipf* distribution (which implies that the top 20% of contributors are responsible for 80% of the contributions) [70]. The *Zipf* distribution was also observed in other studies [90, 243, 290]. Other researchers used network metrics and analysed core and peripheral developers over the project evolution [36, 156, 158]. For instance, Joblin et al. [156] empirically classified developers into core and peripheral to model the organisational structure using network metrics (e.g., degree- and eigenvector-centrality) and analysed how the set of core developers changed over time.

Despite several studies classifying developers into roles, none of them analyse the influence of the developer roles on the emergence of merge conflicts. We use *top contributors* and *occasional contributors* classification instead of core and peripheral developers because, as suggested by a previous study [156], we consider that these terms better represent high- or low-frequency contributors, respectively. Similar to previous work [70, 205, 243, 290], we use the 80% percentile to classify top contributors (core). Furthermore, as suggested by previous work [156, 158], we recompute the developer roles for each merge scenario. Differently from them, we classify developers with distinct granularity: *project* and *merge-scenario level*.

Top and occasional contributors at project level classification. Top and occasional developers at project level are classified based on their code contributions on the whole

project at the end of each merge scenario (i.e., at the merge commit). Practically, we follow 5 steps. First, for each merge commit we checked it out using the GIT CHECKOUT SHA command, where SHA is the identifier for the merge commit. Hence, for each merge commit, we run the GIT BLAME command to compute the authorship of each line of code in the whole project. Second, we sum up the lines of code each developer contributed creating a map where each developer has an unique identifier (key) and an object with the developer information as a value. This object includes an attribute informing the number of lines of code this developer changed in the whole project at the moment of the merge commit. Third, get the total lines of code in the project by summing all developer contributions. Fourth, we create a list of developers in descending order based on their code contributions (i.e., developers that contribute most are at the top of the list). Fifth, we get the top developers from the list until the sum of their contributions makes up 80% of the total contributions at merge commit time. These developers are classified as top contributors. All other developers are considered occasional contributors.

Top and occasional contributors at merge-scenario level classification. Top and occasional developers at merge-scenario level are classified based on their code contributions in a merge scenario. The classification at merge-scenario is similar to the project level, the only difference is in the first step. Instead of measuring the authorship of each developer in the whole project, we measure the code contribution of each developer in the merge scenario. In other words, for each merge commit, we measure only the lines of code changed between the base and merge commit. Hence, top contributors at merge scenario level are the developers that contribute to 80% of the changed lines of code in the merge scenario and all other developers are occasional contributors.

The distinction of project and merge-scenario level is essential because, while the developer roles at project level give a more global view of the code contributions, developer roles at merge-scenario level give a more focused view on merge scenario code changes and on merge conflicts. In Section 4.3.3, we describe the investigated measures as well as exemplify how the developer roles are computed in practice.

4.3 Study Setting

In Figure 4.1, we illustrate our four steps, which consist of (i) defining our goals and research questions, (ii) selecting subject projects, (iii) acquiring data, (iv) operationalizing and analysing data. We describe these processes in the following four sections.

4.3.1 Goals and Research Questions

Our overall goals are threefold:

- **To understand which developer roles cause proportionally more merge conflicts**. Knowing which developer roles are more often involved in merge conflicts can: (i) avoid or minimise conflicting merge scenarios since project coordinators and developer themselves can increase the coordination and communication where conflict-prone developer roles are working on. Hence, they can be aware sooner of other

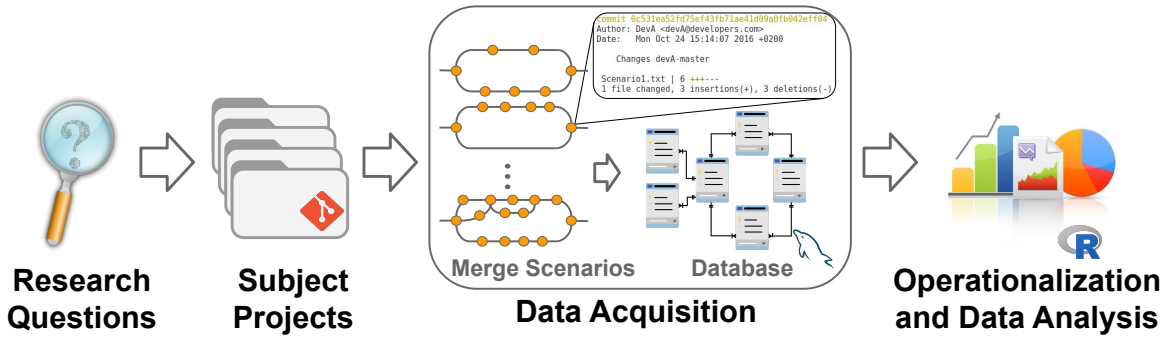

Figure 4.1: Study Setting Overview

changes and fix conflicts in its earlier stages or even avoid them. For instance, as seen in the example merge scenario presented in the beginning of this chapter, properly coordinating specific developer roles (i.e., making them aware or other changes and communicate with each other) can be enough for avoiding merge conflicts and (ii) support on the conflict resolution since project coordinators and developer themselves can increase the communication of developer roles often involved in conflict to support the merge conflicts resolution.

- **To find whether it is feasible to predict merge conflicts using only social measures**. Showing that it is possible to predict merge conflicts using social measures can not only minimise the number of speculative merging as motivated in the beginning of this chapter, but also highlight the importance of social measures in software analysis. Hence, we show evidence of why researchers should consider social measures more often in their analyses.

- **To find whether combining social and technical assets improve the state-of-the-art of predicting merge conflicts**. Previous work has predicted merge conflicts using technical measures, adding the social perspective might improve previous results improving the state-of-the-art of merge conflict predictions.

We investigate the relationship between the developer role and the emergence or avoidance of merge conflicts in four ways, represented by the following RQs:

> **RQ$_1$**: *Which developer role is more often related to merge conflicts considering project and merge-scenario level separately?*
>
> **RQ$_{1.1}$**: *Are top contributors at project level proportionally related to more merge conflicts than occasional contributors?*
>
> **RQ$_{1.2}$**: *Are top contributors at merge-scenario level proportionally related to more merge conflicts than occasional contributors?*

> **RQ$_2$**: *Which combination of developer roles is related to merge conflicts combining project and merge-scenario level classification?*

> **RQ$_3$**: *Are merge conflicts predictable using only social measures?*

> **RQ$_4$**: *Is a model combining social and technical measures better than a model composed of only social measures to predict merge conflicts?*

Note that the first RQ is simple such that developers can identify developer roles without tool support. In the second RQ, we increase the complexity, but developers with a comprehensive understanding of the project can still identify developer roles without tool support. In the third and fourth RQs, we use more information and a more sophisticated approach. It makes manual identification difficult. Answering these four RQs, we expect an actionable insights overview of the influence of the subject developer roles on the occurrence of merge conflicts, especially when triangulating social and technical measures.

4.3.2 Subject Projects

We selected the corpus of subject projects by retrieving the 100 most popular projects on GitHub, as determined by the number of stars [40] and, then, we applied the following four filters created based on Kalliamvakou et al. work [162]: (i) projects that do not have a classified programming language as the main file extension since we are interested in programming language projects; (ii) projects with less than two commits per month in the last six months, since we are interested in active community projects on GitHub; (iii) projects in which it was not possible to reconstruct at least 50% of the merge scenarios, since we are interested in projects that use the three-way merge pattern in the majority of integrations (see Appendix A.3.1). The inclusion of projects that follow other development patterns could bias our analysis. In Section 4.3.3, we detail how we rebuilt merge scenarios; and, (iv) balancing the programming language of projects consists of excluding less popular JavaScript projects until they are not the majority of subject projects. Including most projects of a programming language could bias our analysis, as we explain in Section 4.6.

We restricted our selection to GitHub because it is one of the most popular platforms to host repositories, and it has been investigated and used in prior work [75, 116, 266, 281, 296, 301, 302]. We limited our analysis to Git repositories because it simplifies the identification of merge scenarios in retrospect and is a popular practice as well.

After applying all filters, we obtained 66 projects, developed in 12 programming languages (e.g., JavaScript, Python, Java, Go, and C++), containing 78 740 merge scenarios that involve more than 1.5 million files changed, 10.4 million chunks, and 3 950 conflicting merge scenarios. bootstrap[2], react[3], TypeScript[4], redis[5], and lantern[6] are examples of selected projects. In Figure 4.2, we show the distribution of merge scenarios (ms), conflicting merge scenarios (cms), number of files, number of chunks, number of commits, and number of developers by each subject project. In other words, each project represents a dot in the

2 https://github.com/twbs/bootstrap
3 https://github.com/facebook/react
4 https://github.com/microsoft/TypeScript
5 https://github.com/antirez/redis
6 https://github.com/getlantern/lantern

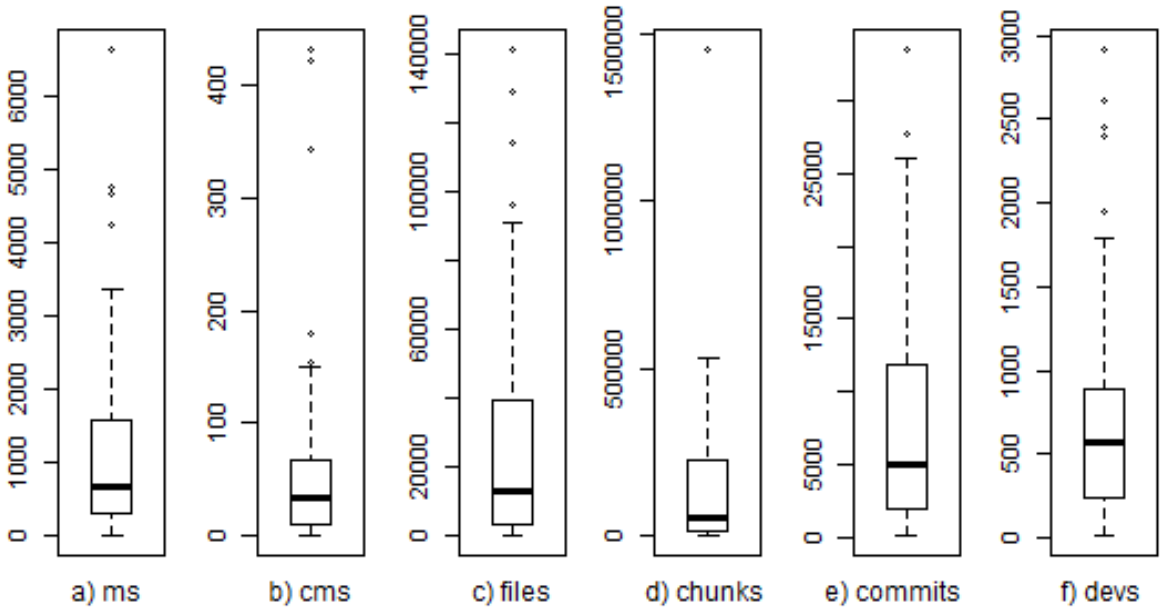

Figure 4.2: Descriptive Statistics by Subject Project

graphs and the number of files, for instance, is the sum of all files of a project. The complete list of projects with URL, programming language, and descriptive statistics is in Appendix B.

4.3.3 Data Acquisition

In this section we show: (i) how we acquire data for each merge scenario, (ii) details about the developer roles classification, (iii) the investigated measures, (iv) how we computed the investigated measures, (v) an example of how the investigated measures are computed, and (vi) where our data and framework is available.

Acquiring data. Data acquisition follows the steps presented in Section A.3.1. At the end, for this study we are interested in a *set* of developers for each merge scenario. For each developer, we retrieved: 1) a unique identifier, 2) the merge scenario identifier, 3) a boolean flag demonstrating if it is or not a conflicting merge scenario, 4) the list of files touched in the target branch, 5) the list of files touched in the source branch, 6) the number of chunks changed in the target branch, 7) the number of chunks changed in the source branch, 8) the number of lines of code changed in the target branch, 9) the number of lines of code changed in the source branch, 10) the number of commits in the target branch, and 11) the number of commits in the source branch.

Classifying developers. To classify developers into top and occasional, we followed the approach described in Section 4.2. Note that at project level we consider developers contribution in the whole project for each merge commit. Hence, top and occasional contributors at project level might not be active developers in a given merge scenario. By active developers, we mean developers that touched (i.e., created, edited, or deleted) one of the integrated branches of a merge scenario. At the merge-scenario level, we consider

only code contributions in a merge scenario. Hence, all top and occasional contributors at merge-scenario level are active developers.

Investigated measures. In Tables 4.1 and 4.2, we present the investigated measures. The reasoning behind our choice is related to three factors: i) fine-grained measurement, ii) already used in the literature, and iii) inexpensive computation.

Fine-grained measurement. Considering that the code contributions are normally different on the merged branches which may influence either the occurrence of merge conflicts as well as the developers' role that contribute to the branch [67, 110], we differentiate contributions from *target* and *source* branches for all investigated measures.

Already used in the literature. We selected the measures by surveying the literature on merge conflicts and related areas, such as code evolution or software maintenance (see the Reference column of Tables 4.1 and 4.2). Furthermore, developers reported that most of the selected measures are useful to identify merge conflicts [180]. Note that measures found in the literature are often coarse-grained (i.e., ignore the branch contributions that happened). As our measures are fine-grained, not all rows of Tables 4.1 and 4.2 have a reference linked to it.

Inexpensive computation. We selected measures which extraction is computationally inexpensive aiming at making the prediction used in practice.

Computing measures. To get the measures from a merge scenario, we basically aggregate measures from active developers (i.e., the set mentioned before). For instance, to come up with the value of loc_t, we aggregate the number of source lines of code (i.e., excluding blanks and comments) in the target branch of all developers for a given merge scenario. The counting of *loc* is part of our framework and follows a similar implementation of CLOC TOOL[7]. As another example, to come up with the value of *devs* from a merge scenario, we got a set of all active developers (represented by the unique identifier) of a merge scenario (represented by the merge scenario identifier).

Exemplifying computation of measures. In the example of Figure 4.3, DEV C faced merge conflicts in File 1. These conflicts appeared given the concurrent changes of DEV A in the *target* branch with the changes of DEV C and DEV B in the *source* branch. We see that three files changed in this merge scenario (*files* - File 1, File 2, and File 3) where two changed in the target branch ($files_t$ - File 1 and File 2) and two files changed in the source branch ($files_s$ - File 1 and File 3). The number of chunks is six (*chunks*) where four chunks are in the target branch ($chunks_t$ - two chunks in File 1 from DEV A and two in File 2 from DEV A and DEV D) and four chunks are in the source branch ($chunks_s$ - two chunks in File 1 from DEV C and DEV B and two in File 3 from DEV A and DEV B). The number of lines of code is twelve (*loc*) where seven are in the target branch (loc_t) and five are in the source branch (loc_s). The number of commits is five (*commits*) where two are in the target branch ($commits_t$ - hashes: 923E4D5 and 20BBDF7) and three are in the source branch ($commits_s$ - hashes: A562FA6, 35DBC8F, and 0E8F458).

In Table 4.3, we illustrate the number of lines of code each developer contributed at the moment of the merge commit (hash: C2ECB2C) at project and merge-scenario level. Despite of in the beginning of the merge scenario, Dev X committed 26 lines of code (hash FF1E147 - 5 *loc* in File 1, 5 *loc* in File 2, 4 *loc* in File 3, and 12 *loc* in File 4), until the merge commit DEV A, DEV B, DEV C, and DEV D changed 8, 2, 1, and 1 lines of code, respectively. Hence, at the

7 https://cloc.sourceforge.net/

Table 4.1: Variables of Our Study (Part 1)

Variable	Description	References
	Dependent variable	
has_conflict	Boolean informing if the merge scenario has conflicts	[180, 302]
	Independent (Technical) variables	
files	Number of files touched in the merge scenario	[301, 302]
$files_t$	Number of files touched in the target branch	[103, 171, 180] [226, 252]
$files_s$	Number of files touched in the source branch	[103, 171, 180] [226, 252]
$files_{t\&s}$	Number of files touched in the target and source branches	[180]
chunks	Number of chunks touched in the merge scenario	[180, 301, 302]
$chunks_t$	Number of chunks touched in the target branch	
$chunks_s$	Number of chunks touched in the source branch	
loc	Number of source lines of code touched in the merge scenario (i.e., code churn)	[301, 302]
loc_t	Number of source lines of code touched in the target branch	[103, 180, 226]
loc_s	Number of source lines of code touched in the source branch	[103, 180, 226]
commits	Number of commits created in the merge scenario	[103, 171, 180] [226, 302]
$commits_t$	Number of commits created in the target branch	
$commits_s$	Number of commits created in the source branch	
	Independent (Social) variables	
top_p	Number of top contributors at project level	[70, 205] [243, 290]
$top_{p\&t}$	Number of top contributors at project level contributing to the target branch	
$top_{p\&s}$	Number of top contributors at project level contributing to the source branch	
occ_p	Number of occasional contributors at project level	[70, 205] [243, 290]
$occ_{p\&t}$	Number of occasional contributors at project level contributing to the target branch	
$occ_{p\&s}$	Number of occasional contributors at project level contributing to the source branch	
top_{ms}	Number of top contributors at merge-scenario level	
$top_{ms\&t}$	Number of top contributors at merge-scenario level contributing to the target branch	
$top_{ms\&s}$	Number of top contributors at merge-scenario level contributing to the source branch	

Table 4.2: Variables of Our Study (Part 2)

Variable	Description	References
	Independent (Social) variables	
occ_{ms}	Number of occasional contributors at merge-scenario level	
$occ_{ms\&t}$	Number of occasional contributors at merge-scenario level contributing to the target branch	
$occ_{ms\&s}$	Number of occasional contributors at merge-scenario level contributing to the source branch	
$devs$	Number of active developers in a merge scenario	[301, 302]
$devs_t$	Number of active developers in the target branch	[97, 103] [171, 226]
$devs_s$	Number of active developers in the source branch	[97, 103] [171, 226]
$devs_{t\&s}$	Number of active developers in target and source branches	

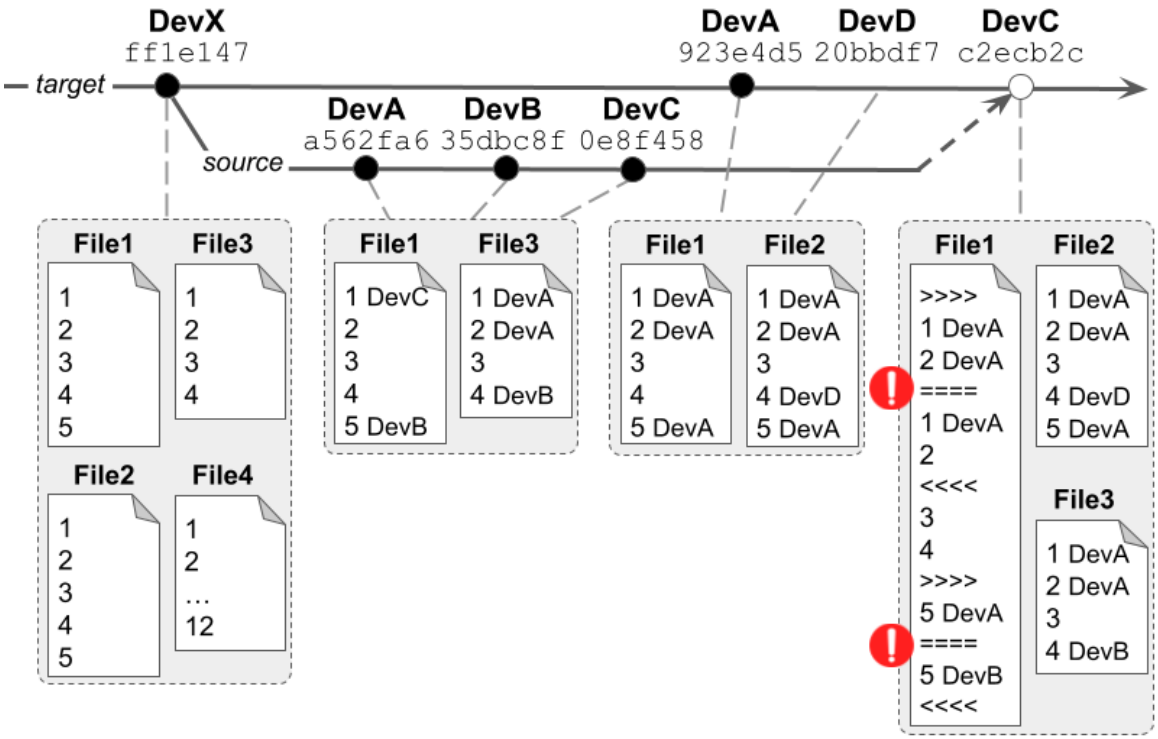

Figure 4.3: Illustrative Merge Scenario

Table 4.3: Developer Code Contributions at Project and Merge-scenario Level

Developer	Project	Merge-Scenario
Dev X	17	-
Dev A	8	8
Dev B	2	2
Dev C	1	1
Dev D	1	1
Total loc	**29**	**12**

merge commit, Dev X authored 17 lines of code (3 *loc* in File 1, 1 *loc* in File 2, 1 *loc* in File 3, and 12 *loc* in File 4). Note that changes from DevA, Dev B, and Dev C are concurrently causing merge conflicts. Note that Dev X is not an active developer in the exemplified merge scenario since she does not commit between the base and merge commit.

At the merge commit the project had 29 lines of code. Hence, the first developers with the large number of lines of code that touched 23 lines of code are classified as top contributors. Hence, Dev X and Dev A are top contributors and Dev B, Dev C, and Dev D are occasional contributors at project level. We see that 12 lines of code changed in the exemplified merge scenario. Hence, developers that touched 10 lines of code are classified as top contributors. Hence, Dev A and Dev B are top contributors and Dev C and Dev D are occasional contributors at the merge scenario level.

Looking at the social measures we see that 4 developers are active in this merge scenario (*devs* - Dev A, Dev B, Dev C, and Dev D) where two touched the target branch ($devs_t$ - Dev A and Dev D) and three touched the source branch ($devs_s$ - Dev A, Dev B, and Dev C). The only developer that touched both target and source branches is Dev A ($devs_{t\&s}$). The number of top contributors at project level is one (top_p - Dev A) since despite of Dev X was classified as top contributor, she is not active in the illustrated merge scenario. As Dev A contributed to the target and source branch, $top_{p\&t}$ and $top_{p\&s}$ is one. The number of occ_p is three (Dev B, Dev C, and Dev D) where Dev D contributed to the target branch (i.e., $occ_{p\&t}$ is one) and Dev B and Dev C contributed to the source branch (i.e., $occ_{p\&t}$ is two). Looking at measures at merge-scenario level, the number of top developers is two (top_{ms} - Dev A and Dev B) where Dev A contributed to the target branch ($top_{ms\&t}$) and Dev A and Dev B contributed to the source branch ($top_{ms\&s}$). The number of occasional contributors is two (occ_{ms} - Dev C and Dev D) where Dev D contributed to the target branch ($occ_{ms\&t}$) and Dev C contributed to the source branch ($occ_{ms\&s}$).

4.3.4 Operationalization

The operationalization of RQ_1 and RQ_2 consists of getting the set of merge scenarios that a given developer role participated in (#*MS*), a subset of these merge scenarios which have merge conflicts (#Conf. MS), and share of conflicting merge scenarios (i.e., #MS $\times$ #Conf. MS) investigated in each RQ (see Section 4.3.1). For instance, in $RQ_{1.1}$ we want to find

a subset of merge scenarios from all 78 740 investigated merge scenarios that have top contributors contributing to both *target* and *source* branches. From this subset, we get the number of merge scenarios that have merge conflicts and compute the share of conflicting merge scenarios. We performed a chi-square test to verify whether the developer role (top and occasional) differs significantly. The chi-square test is adequate because we have large and unpaired data (i.e., the number of merge scenarios varies depending on the developer role), variables under analysis are categorical (e.g., top- or occasional-contributors), and the outcome is binomial (i.e., conflicting or safe merge scenarios). The null and alternative hypotheses for RQ_1 and RQ_2 are:

> ***H***$_0$*: Developers' role and emergence of merge conflicts are independent.*
>
> ***H***$_a$*: Developers' role and emergence of merge conflicts are not independent.*

where the p-value is below 0.01 (i.e., 99% significance level), we reject the null hypothesis (H_0) and accept the alternative hypothesis (H_a). Accepting the alternative hypothesis suggests that the variables are related, but the relationship is not necessarily causal. As we measured several attributes for each merge scenario, we grouped them using set operations (e.g., union) and treated each influencing factor separately. Aiming at getting a baseline for comparison of our results, we compared the results of each developer role with the overall average of conflicting merge scenarios for all merge scenarios in analysis. Thus, we increased the knowledge over our data and internal validity.

The operationalization of RQ_3 and RQ_4 consists of using data acquired as described in Section 4.3.3 and follows three steps: (i) to balance our data since merge conflicts happen in only the minority of merge scenarios, (ii) to select the target measures (i.e., features), and (iii) to predict conflicting scenarios using three classifiers. We used multiple balancing techniques, sets of measures, and classifiers to show practitioners which configurations perform better on our data. For data balancing, we chose seven techniques (under, over, both, SMOTE, BorderlineSmote, SVMSmote, Adasyn). For feature selection, in RQ_3, we created a model using only the social measures presented in Tables 4.1 4.2 as we want to investigate the prediction of merge conflicts using only social assets. In RQ_4, we created two models, one using only the technical measures and the other one all measures presented in Tables 4.1 and 4.2. We build these two models to be able to compare our results with the model created for RQ_3 and with a previous study [226]. To predict conflicts, we chose three classifiers (decision tree, random forest, and KNN), because they are simple yet achieve good results for binomial classification. Due to the importance of hyper-parameters, we used grid-search with 10-fold cross-validation to find the right hyper-parameters to use. For each classifier, we tuned it using all possible hyper-parameters. For instance, for *decision tree*, we set the hyper-parameters: *max_depth* (10, 50, 80, 100, 150, 200), *max_feature* (auto, sqrt, log2), *min_samples_split* (2, 3, 5, 10), *min_samples_leaf* (1, 2, 3, 5, 10), *criterion* (gini, entropy) and, *splitter* (best, random). The complete list of hyper-parameters and tuning values, as well as a description of each balancing technique and classifier are available at Appendix C. **Performance Measures**. We showed *precision*, *recall*, and *f1-score* for conflicting and safe merge scenarios. Furthermore, even though the previous work [226] mentioned that *accuracy* is not a good performance measure when dealing with a discrepant difference between the majority and minority classes, we also showed *accuracy* and *AUC* for our general

predictions. Presenting results for conflicting and safe scenarios provides a complete view of how a detector would perform in practice than only presenting general measures. In our case, it is desirable to have higher recall than higher precision for conflicting scenarios since it is better to predict all conflicting scenarios and some false-positives (i.e., reported as conflicting scenarios, but they are safe in practice) than miss some conflicting scenarios (i.e., true-negatives). In other words, it is better to suggest speculative merges for some safe-scenarios than ignore some real conflicting scenarios. We considered f1-score the second most relevant performance measure since its computation combines precision and recall. In cases where we found the same value for the performance measures, we present the results for the model with lower values for the hyper-parameters.

4.4 Results

In this section, we present the results structured according to our RQs. Overall, we investigated 78 740 merge scenarios of which 3 950 of them have merge conflicts. It corresponds to an average of 5.02% conflicting merge scenarios. We use this percentage in RQ_1 and RQ_2 to compare if a developer role is above or below the general average.

4.4.1 RQ_1: Which Developer Role is More Often Related to Merge Conflicts Considering Project and Merge-scenario Level Separately?

We answer this question by looking at data from project and merge-scenario level separately. In Table 4.4, we present the general result for $RQ_{1.1}$ and the results for each branch. Top contributors at project level contributed to the target and source branches in 45 297 merge scenarios and 3 290 of them have merge conflicts. It represents a share of 7.26% of conflicting merge scenarios. Occasional contributors at project level contributed to 60 609 merge scenarios, 3 409 of them have merge conflicts. It represents a share of 5.62% of conflicting merge scenarios. Note that it does not need to be the same developer. It just needs to have at least one given developer role contributing to the target branch and at least one developer contributing to the source branch. With the chi-square test (*X-squared=103.01*, *df=1*, *p-value*$< 2.2^{e-16}$), we reject the null hypothesis and accept the alternative hypothesis. Thus, we conclude that there is a relationship between the developer role and the emergence of merge conflicts. We found a similar result for the target and source branches.

In Table 4.5, we present the general result for $RQ_{1.2}$ and the results for each branch. Top contributors at merge-scenario level contributed to 75 142 analysed merge scenarios and 3 623 conflicting merge scenarios. It represents a share of 4.82% of conflicting merge scenarios. Occasional contributors at merge-scenario level contributed to 21 751 merge scenarios and to 2 880 conflicting merge scenarios. It represents a share of 13.24% of conflicting merge scenarios. With the chi-square test (*X-squared=1600.4*, *df=1*, *p-value*$< 2.2^{e-16}$), we reject the null hypothesis and accept the alternative hypothesis. Thus, we conclude that there is a relationship between the developer role and the emergence of merge conflicts. We also found a similar result for the target and source branches.

Table 4.4: Top and Occasional Contributors at Project Level Contributions Overview

Branch	Dev. Role	#MS	#Conf. MS	%Conf.MS
Target & Source	Top	45 297	3 290	**7.26%**
	Occ.	60 609	3 409	5.62 %
Target	Top	33 872	2 301	**6.79%**
	Occ.	46 826	2 745	5.86 %
Source	Top	20 925	2 590	**12.38%**
	Occ.	35 086	2 970	8.46 %

***Dev.**: developer, **#MS**: number of merge scenarios and **#Conf.MS**: number of conflicting merge scenarios. The light-gray background highlights the main discussed developer roles*

Table 4.5: Top and Occasional Contributors at Merge-scenario Level Contributions Overview

Branch	Dev. Role	#MS	#Conf. MS	%Conf.MS
Target & Source	Top	75 142	3 623	4.82 %
	Occ.	21 751	2 880	**13.24%**
Target	Top	62 214	3 039	4.88 %
	Occ.	17 100	1 424	**8.32%**
Source	Top	53 812	2 344	4.36 %
	Occ.	8 023	1 974	**24.60%**

***Dev.**: developer, **#MS**: number of merge scenarios and **#Conf.MS**: number of conflicting merge scenarios. The light-gray background highlights the main discussed developer roles*

Comparing the results with the general average, we observed that contributors at project level have a greater percentage for all the cases. For instance, top and occasional contributors have a share of 7.26% and 5.62% conflicting scenarios, respectively. For developer roles at merge-scenario level, we see that occasional contributors have a share of conflicting merge scenarios above the general average (between 8.32%–24.60%) while top contributors do not (between 4.36%–4.88%).

We expected that top contributors are related to more merge conflicts than occasional contributors since the more code a developer changes, the greater the chance of happening conflicting merge scenarios. However, our results of $RQ_{1.2}$ shows that, at merge-scenario level, occasional contributors are more often involved in conflicting merge scenarios than top contributors. To illustrate, let us consider a merge scenario with 4 developers changing 100 lines of code. Dev A changed 50 lines of code (50% chance to be related to conflicts), Dev B changed 40 lines of code (40%), Dev C and Dev D changed 5 lines of code each (5% each). In this merge scenario, Dev A and Dev B are top contributors and Dev C and Dev D are occasional contributors. Note that the chance of Dev C and Dev D being in merge conflict is only 10%. Nevertheless, even despite this small chance, these occasional contributors were responsible for all conflicting changes.

RQ_1 Summary: At project level, top contributors are related proportionally more to conflicting merge scenarios than occasional contributors, and occasional contributors collaborate to more merge scenarios than top contributors. At the merge-scenario level, the share of conflicting merge scenarios is greater for occasional contributors than for top contributors. Around one quarter of the contributions of occasional contributors at merge-scenario level in the source branch are related to merge conflicts.

4.4.2 RQ_2: Which Combination of Developer Roles is Related to Merge Conflicts Combining Project and Merge-scenario Level Classification?

In Table 4.6, we present the general result for RQ_2 and the results for each branch. As expected, merge scenarios with top contributors at project touching the target and the source branches that are also top contributors at merge-scenario level occurred more often than merge scenarios with top contributors at project level touching the target and source branches and occasional contributors at merge-scenario level (44 497 against 15 834). However, when looking at the proportion of conflicting merge scenarios, top contributors at project level that are occasional contributors at merge scenario level have a higher percentage (15.76%) than all other developer roles. With the chi-square test ($X\text{-}squared = 1229.6$, $df=3$, $p\text{-}value < 2.2^{e-16}$), we reject the null hypothesis and accept the alternative hypothesis. Thus, we conclude that there is a relationship between the developer role and the emergence of merge conflicts. We found a similar result for the target and source branches.

Table 4.6: Top and Occasional Contributors Combining Project and Merge-scenario Level Contributions Overview

Branch	Dev. Role	#MS	#Conf. MS	%Conf.MS
Target & Source	$\text{Top}_p \circ \text{top}_{ms}$	44 497	3 070	6.90 %
	$\text{Top}_p \circ \text{occ}_{ms}$	15 834	2 496	**15.76%**
	$\text{Occ}_p \circ \text{top}_{ms}$	57 053	3 084	5.41 %
	$\text{Occ}_p \circ \text{occ}_{ms}$	21 195	2 800	13.21 %
Target	$\text{Top}_p \circ \text{top}_{ms}$	23 728	1 579	6.65 %
	$\text{Top}_p \circ \text{occ}_{ms}$	11 268	1 164	**10.33%**
	$\text{Occ}_p \circ \text{top}_{ms}$	31 009	1 880	6.06 %
	$\text{Occ}_p \circ \text{occ}_{ms}$	16 188	1 329	8.21 %
Source	$\text{Top}_p \circ \text{top}_{ms}$	11 184	1 678	15.00 %
	$\text{Top}_p \circ \text{occ}_{ms}$	4 943	1 597	**32.31%**
	$\text{Occ}_p \circ \text{top}_{ms}$	18 793	1 933	10.29 %
	$\text{Occ}_p \circ \text{occ}_{ms}$	1 914	1 974	24.93 %

***Dev.**: developer, **#MS**: number of merge scenarios and **#Conf.MS**: number of conflicting merge scenarios. The light-gray background highlights the main discussed developer roles*

RQ_2 Summary: Looking at developer roles at project and merge-scenario levels together, we found that merge scenarios with top contributors at project level and occasional contributors at merge-scenario level touching the target and source branches have the greatest share of conflicting merge scenarios in general and for the analysis of both branches. Surprisingly, around one-third of the contributions of these developer roles in the source branch are related to merge conflicts. It is surprising because it represents six times more than the general average (i.e., without considering developer roles) and three times more the contributions of the same developer roles in the target branch.

4.4.3 RQ_3: Are Merge Conflicts Predictable Using Only Social Measures?

When answering RQ_3 and RQ_4, we present only the best performance results according to the criteria described in Section 4.3.4. Presenting only a few results is necessary since we have results for a combination of three models (social vs. technical vs. social and technical measures), seven balancing techniques (i.e., under-, over-, both-, SMOTE-, BorderlineSmote-, SVMSmote-, Adasyn-sampling), and three classifiers (i.e., decision tree, random forest, and KNN). The complete results can be seen in our Appendix C.

In Table 4.7, we present the results of our predictions using social measures for each classifier highlighting the best balancing techniques. By best balancing techniques, we mean the balancing techniques that balanced our data in a way that made our classifiers perform better. Reinforcing, we use the recall and f1-score of conflicting scenarios as our main performance measures (see Section 4.3.4). Hence, once we got the best setup (i.e., the combination of classifier and balancing technique) for conflicting scenarios, we highlight their results.

For the predictions of conflicting scenarios using only social measures, the setup with random forest performed better when using balanced data from under, SMOTE, or Adasyn-sampling technique. With this setup, we achieve a recall, f1-score, and precision of 1.00, 0.26, and 0.15, respectively. Regarding safe scenarios, we found a recall, f1-score, and precision of 0.72, 0.83, and 1.00, respectively. In terms of accuracy and AUC this setup achieved 0.60 and 0.79.

Note that the setup using KNN classifier with data from Adasyn-sampling technique achieved better accuracy and AUC are 0.73 and 0.83 than the setup with better recall. Furthermore, note that none of the setups achieved high f1-score and precision for conflicting scenarios (all values below 0.3).

RQ_3 Summary: Using only social measures we created a model in which the random forest classifier achieved 1.00 of recall. Therefore, we conclude that it is possible to predict conflicting merge scenarios using only social measures. Classifying all conflicting scenarios correctly means that we can reduce speculative merging considerably without missing any real conflicting scenarios. In any event, we highlight the low precision of our model which leads to an open challenge of increasing the precision of models using only social measures.

Table 4.7: Performance Overview for Social Measures

Classifier	Bal. Tech.	Scenario	R	F1	P	Acc.	AUC
Decision Tree	Over	Safe	0.60	0.75	**1.00**	0.62	0.80
		Conflicting	0.99	0.21	0.12		
Random Forest	Under SMOTE Adasyn	Safe	0.58	0.74	**1.00**	0.60	0.79
		Conflicting	**1.00**	0.20	0.11		
KNN	Adasyn	Safe	0.72	**0.83**	**1.00**	**0.73**	**0.83**
		Conflicting	0.94	0.26	0.15		

***Bal. Tech.**: balancing technique, **R**: recall, **F1**: f1-score, **P**: precision, and **Acc.**: accuracy*

4.4.4 RQ$_4$: Is a Model Combining Social and Technical Measures Better than a Model Composed of Only Social Measures to Predict Merge Conflicts?

Before looking at the results combining social and technical measures, we present the results of a model using only technical measures. As mentioned, we created this model aiming at increasing our understanding on the models as well as fomenting discussions. In Table 4.8, we present the results of our predictions, similar to what we did when answering RQ$_3$. For the predictions of conflicting scenarios, the setup using random forest classifier and balanced data from SMOTE- or Adasyn-sampling techniques performed better. With this setup, we achieved a recall, f1-score, and precision of 1.00, 0.56, and 0.39, respectively. Regarding safe scenarios, we found a recall, f1-score, and precision of 0.92, 0.96, and 1.00, respectively. In terms of accuracy and AUC, we found 0.92 and 0.95.

Note that we found the maximum value for the model using only social measures in terms of recall for conflicting scenarios. However, the value for other performance measures increases in the model using technical measures. For instance, f1-score and precision for conflicting scenarios increase from 0.26 to 0.92 and from 0.15 to 0.39, respectively. We also see an increase for safe scenarios and general measures. For instance, the accuracy for the social and technical models are 0.73 and 0.92, respectively.

In Table 4.9, we present the predictions of our model using social and technical measures similar to Table 4.7 and Table 4.8. For the predictions of conflicting scenarios, the setup using random forest classifier and balanced data from under- or over-sampling techniques performed better. With this setup, we found a recall, f1-score, and precision of 1.00, 0.56, and 0.39, respectively. Regarding safe scenarios, we found a recall, f1-score, and precision of 0.92, 0.96, and 1.00 for the same setup, respectively. In terms of accuracy and AUC, we found 0.92 and 0.96, respectively.

As seen, the results of a model using only technical measures and the other using all (social and technical) measures are basically the same. Only the AUC increased from 0.95 to 0.96. For the technical and all measures models, the random forest classifier performed slightly

Table 4.8: Performance Overview for Technical Measures

Classifier	Bal. Tech.	Scenario	R	F1	P	Acc.	AUC
Decision Tree	Over Both SMOTE BorderlineSmote Adasyn	Safe	0.88	0.93	**1.00**	0.88	0.94
		Conflicting	**1.00**	0.46	0.30		
Random Forest	SMOTE Adasyn	Safe	0.92	**0.96**	**1.00**	**0.92**	**0.95**
		Conflicting	**1.00**	0.56	0.39		
KNN	Adasyn	Safe	0.75	0.85	**1.00**	0.76	0.84
		Conflicting	0.93	0.28	0.16		

***Bal. Tech.**: balancing technique, **R**: recall, **F1**: f1-score, **P**: precision, and **Acc.**: accuracy*

Table 4.9: Performance Overview for All (Technical and Social) Measures

Classifier	Bal. Tech.	Scenario	R	F1	P	Acc.	AUC
Decision Tree	Over	Safe	0.87	0.93	**1.00**	0.87	0.93
		Conflicting	**1.00**	0.44	0.29		
Random Forest	Under Over	Safe	0.92	**0.96**	**1.00**	**0.92**	**0.96**
		Conflicting	**1.00**	0.56	0.39		
KNN	Adasyn	Safe	0.73	0.84	0.99	0.74	0.82
		Conflicting	0.92	0.27	0.16		

***Bal. Tech.**: balancing technique, **R**: recall, **F1**: f1-score, **P**: precision, and **Acc.**: accuracy*

better than the other classifiers and the data from under- and over-sampling presented better results than the data from other balancing techniques.

Observing that no real improvements were obtained adding the technical and social measures, in Figure 4.4 we show the correlation-matrix to identify whether the investigated measures correlate with each other. Be aware that correlating pairs of investigated variables provide a limited and simpler viewpoint compared to the machine learning classifiers predictions. As we can see in Figure 4.4, some social measures are correlated with each other and also with some technical measures. For instance, occ_p has a high positive correlation with *devs* (0.78) and $occ_{p\&t}$ (0.73) and a moderate positive correlation with $occ_{p\&s}$ (0.65), occ_{ms} (0.64), *commits* (0.60), $occ_{ms\&t}$ (0.55). All correlations were computed using Spearman-rank based correlation with 95% of confidence level. Spearman-rank based correlation is invariant for linear transformations of covariates and is simple and useful to understand the relation among our covariables [152]. Having social measures correlated with each other might have provided similar information to the social model not improving its performance. Having social measures related to technical measures made the addition of social measures to

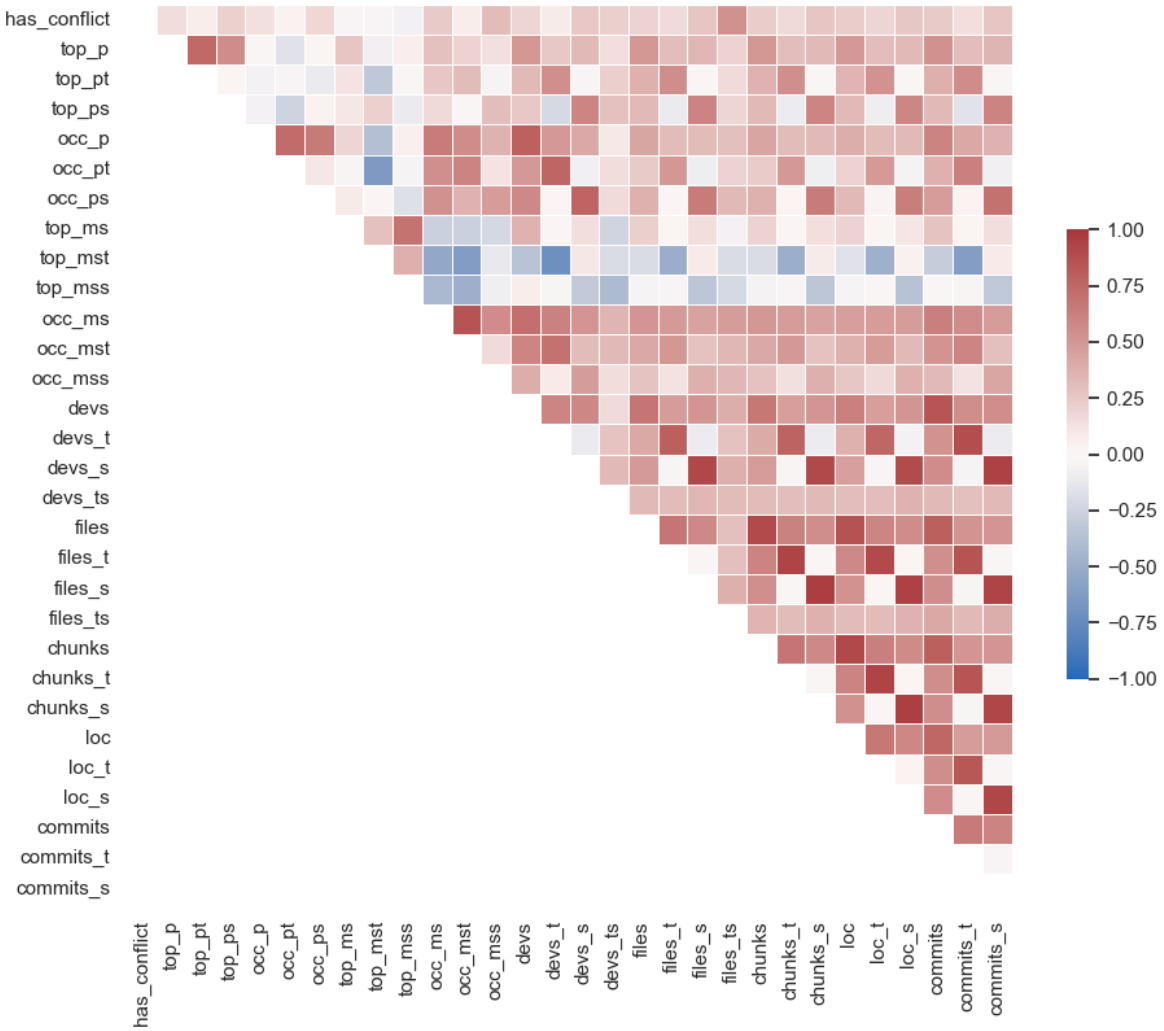

Figure 4.4: Correlation Matrix of Investigated Variables

the technical model, introducing only information that technical measures had already provided. We come back with a discussion on this topic in Section 4.5.2.

RQ$_4$ Summary: In terms of recall, the three models we built (only social vs. only technical vs. all measures) found 1.00 of recall for conflicting scenarios (i.e., they were able to retrieve all real conflicting scenarios). Considering accuracy, AUC, f1-score, and precision for conflicting and safe scenarios, the models using only technical and all social and technical measures performed better than those using only social measures.

4.5 Discussion

We divide this section into three parts. First, we compare our results with previous work predicting merge conflicts (Section 4.5.1). Second, we present a reflection upon our results

Table 4.10: Comparison of our Results with the Results of Owhadi-Kareshk et al. [226]

Study	Scenario	Recall	F1-score	Precision
Owhadi-Kareshk et al. [226]	Safe	**[0.93,0.96]**	**[0.95,0.97]**	[0.97,0.98]
Ours		0.92	**0.96**	**1.00**
Owhadi-Kareshk et al. [226]	Conflicting	[0.68,0.83]	**[0.57,0.68]**	**[0.48,0.63]**
Ours		**1.00**	0.56	0.39

(Section 4.5.2). Finally, we present implications of our results and findings to practitioners, researchers, and tool builders (Section 4.5.3).

4.5.1 Comparing Results

As mentioned in Section 2.6, there are six studies predicting merge conflicts. As the approach and results from Accioly et al. [1], Leßenich et al. [180], Rocha et al. [245], and Dias et al. [82] differ significantly from ours, it is not fair comparing our results. For instance, while we compare developer roles and the used machine learning classifiers to predict conflicts, Accioly et al. [1] computed recall and precision to identify merge conflicts related to two types of code changes. Hence, even though they also compute recall and precision, our results are not comparable. Despite Trif et al. [294] used machine learning like us, they present just a general recall and precision, i.e., they do not differ from safe and conflicting scenarios. Furthermore, they do not show f1-score and AUC. Hence, we opted to not compare their results with ours. Owhadi-Kareshk et al. [226], on the other hand, used machine learning classifiers like us and present recall, precision, and f1-score for safe and conflicting scenarios making our results comparable. Even though we use a different set of measures/variables, subject projects, and they present the results by programming language, we consider our results comparable.

In Table 4.10, we present the results for the performance measures presented in their study (i.e., recall, f1-score, and precision) which is a subset of our performance measures. Aiming at providing a fair comparison, we show the interval of their results by programming languages for the random forest classifier. Similar to our study, the random forest classifier was the classifier that performed better. Looking at safe scenarios, they presented better recall, similar f1-score, and lower precision. Looking at conflicting scenarios, we presented higher recall and lower f1-score and precision. It is important to mention that we focus on increasing recall of conflicting merge scenarios since missing real conflicting scenarios might damage speculative merge tools and hurt users' confidence on the predictions making them stop using tool support [128]. Hence, we consider it essential to retrieve all real conflicting scenarios. This choice made us decrease precision. In other words, we ensure that all real conflicting scenarios were correctly classified, but we classified some safe scenarios as conflicting scenarios (see Section 4.3.4).

4.5.2 Reflecting on Results

Occasional contributors are more related to conflicting scenarios than top contributors. As mentioned, we expected that top contributors are related to more merge conflicts than occasional contributors since the more code a developer changes, the greater the chance of happening conflicting merge scenarios. However, our results when answering RQ_1 (see Table 4.5) show the opposite. In other words, at merge-scenario level, occasional contributors are conflict-prone when compared to top contributors. For instance, when looking at the source branch, the percentage of conflicting merge scenarios for occasional contributors is 24.60%, while for top contributors the percentage of conflicting merge scenarios is only 4.36%. We speculate that there may be two reasons for this phenomenon: i) occasional contributors normally change more code than necessary to address a task, such as fix a bug and ii) occasional contributors take more time than necessary to complete a task. We plan an *in-situ* investigation in future work to draw a conclusion about it.

One-third of scenarios have merge conflicts when top contributors at project level and occasional contributors at merge-scenario level touch the source branch. Once we know the conflict-prone developer roles, project coordinators or developers themselves should increase awareness when these developer roles are touching the source code. Considering that it is easy for practitioners collecting the required information (i.e., developer roles at project and merge-scenario level and the touched branch), they can use this information in practice without tool support. Looking at our data, we saw that merge conflicts are rare when there is none or one occasional contributor at both project and merge-scenario level touching the source branch. However, when there are two or more occasional contributors, the chances of conflicting merge scenarios increase considerably. Looking at Table 4.6, we saw that one-third of the scenarios led to conflicts when there is at least a top contributor at project level and an occasional contributor at merge-scenario level touching the source branch.

Random forest performed better than decision tree and KNN classifiers. Looking at the answers of RQ_3 and RQ_4, we can see that the random forest classifier performed better than the other classifiers. For instance, in the technical- and all measure models, random forest performed better or equally for all performance measures presented in Table 4.8 and Table 4.9. Owhadi-Kareshk et al. [226] found similar results, as we discussed in Section 4.5.1. With all, we suggest this classifier for further analysis and research on conflict predictions. Be aware that all classifiers used the same data to predict the conflicts. So, the performance is indeed related to the competence of a classifier that retrieves better recall, f1-score, precision, and AUC.

Adasyn-sampling is a reasonable balancing technique to use for merge scenario data. Adasyn-sampling is one of the newest balancing techniques and performed better in six out of the nine cases we explored. Over- and SMOTE-sampling also performed well, appearing in four and three cases we investigated, respectively. We suggest Adasyn-sampling balancing technique for further analysis and research on merge conflicts.

The touched branch might be insightful for different kinds of analyses. Following our study, we see that the answers of RQs are complementary. Answering RQ_1, we took a simple viewpoint. Answering RQ_2, we combined developer roles at both project and merge-scenario level. This information was fundamental to achieve reasonable performance

measures because we increased our knowledge over our data, especially for the touched branch confounding factor. In fact, we are not the first ones exploring the touched branch factor. However, while previous work [67, 110] reports different contribution patterns on the target and source branches, we are the first ones to show that the touched branch influences the emergence of merge conflicts. In some cases, only the fact of considering the touched branch triples the share of conflicting merge scenarios. For instance, while occasional contributors at merge-scenario level touching the target branch have a share of 8.32%, these contributors touching the source branch have a share of 24.60% (see Table 4.5). Therefore, as the touched branch played an important role in our analyses and on previous work [67, 110], We speculate that the touched branch might be useful for studies mining repositories, investigating project quality criteria, predicting bugs, and other anomalies. For instance, it might provide a new perspective and increase performance when predicting bugs on software systems.

Computing social versus technical measures. As mentioned, developers deeper into the project have a great knowledge of what is going on. Hence, they are able to classify developers at project and merge scenario level without formal measurement. Considering the results of RQ_1 and RQ_2, they are able to identify conflict-prone scenarios with informal measurement and without tool support. In the case of top contributors at project level and occasional contributors at merge-scenario level, around one third of the merge scenarios have merge conflicts (see Table 4.6). On the other hand, when a formal measurement is preferred, computing technical measures is simpler because social measures are computed based on the lines of code (a technical measure). Therefore, to compute the developer role related measures, we first need to compute technical measures and then, compute social measures.

Why did the model with social and technical measures not perform better than the model with only technical measures? We see two factors influencing the performance of the model with all measures: i) inserting confounds and ii) increasing the complexity of the investigated phenomenon.

Inserting confounds. Confounds are variables related to each other, but which are not positively impacting the predictions. We already showed a discussion on this topic when answering RQ_4. Hence, in Section 4.4.4, we saw that some social measures are correlated with each other (e.g., Spearman-rank of 0.78 between occasional contributors at project-level (occ_p) and the number of developers (*devs*)) and also with some technical measures (Spearman-rank of 0.60 between occ_p and the number of commits (*commits*)). Having variables correlated with each other in our model is not necessarily bad, however, it does not help improving the performance of our model.

Increasing the complexity of the investigated phenomenon. The model using only technical measures is composed of 13 independent variables. The model with only social measures is composed of 16 independent variables. Hence, the model with all measures is composed of 29 independent variables which increases the complexity of the investigated phenomenon. Machine learning classifiers are able to identify which variables are more relevant to predict the dependent variable (i.e., minimising over-fitting). However, the more complex the phenomenon, the more difficult it will be to find a function that describes that behaviour. Considering that some variables do not add useful information to the model and the great complexity of the investigated phenomenon with all variables, social measures were not

able to improve the performance on the predictions of the technical model. At least adding the social measures did not confuse the technical measures decreasing the performance of the all measures model compared with the technical model performance.

4.5.3 Implications for Practitioners, Researchers, and Tool Builders

Researchers should focus more on the social perspective and on the branch developers touch. The social perspective and the touched branch factor are often ignored when dealing with merge conflicts. Even though using only social measures does not perform optimally, our study reinforces that social information and the touched branch influence on the emergence of merge conflicts. We suggest researchers using developer roles and the touched branch information when investigating merge conflicts, as well as, on other kinds of analysis mining software repositories.

Tool builders should use developer roles for building tools that reduce speculative merging. We show evidence that some developer roles are more often related to conflicting scenarios than others. So, we suggest tool builders using this information to reduce speculative merging. Hence, before performing speculative merging, their tools filter merge scenarios that have a chance of having merge conflicts. Developer roles can also be useful to construct awareness tools. For instance, merge scenarios that have top contributors at project level and occasional contributors at merge-scenario level touching the source branch, might be closely coordinated/monitored since one third of them end with merge conflicts.

Social measures are a good alternative to retrieve conflicting scenarios. As seen in Section 2.9, human factors play an important role in software development. In our study, we were able to retrieve all real conflicting merge scenarios using developer roles. As discussed in Section 4.5.2, developers with a deep understanding of the project collaboration are able to manually classify developers into top and occasional contributors without formal measurement. Hence, they can avoid merge conflicts by coordinating conflict-prone developer roles more closely. Once automated classification is desired, they can use speculative or awareness tools as previously discussed.

Practitioners should worry more about the order of development tasks. Once it is clear that a conflict will arise when developers touch the same piece of code in different branches, practitioners might find ways to define an order to perform their tasks in a way that they are not going to touch the same parts of code in different branches (i.e., excluding the chances of merge conflicts arise). Researchers have been investigating and creating tools to support this [103, 110, 163, 252, 301]. So, practitioners can already use the proposed tools.

4.6 Threats to Validity

In this section, we discuss potential threats to the validity of our study to help further research and replications of this study. In the following, we detail the main internal and external threats to validity.

Internal validity. We discuss three main internal threats to validity. First, we used simple and common metrics to classify developers. This poses the threat that the metrics do not ac-

curately capture reality. This threat is minor, as existing evidence indicates that those metrics accurately reflect the developers' perception [70, 90, 243, 290]. Second, we used a single alias instead of looking at developers' contributions across multiple information sources (i.e., mailing list, social networks, and version-control system). Although contributors in general are interested in the relevance/recognition of their contributions, maintaining multiple aliases would not be productive. For this reason, we think this threat has limited influence on developer classifications. Third, we selected subject projects from different programming languages; hence, one language could have dominated our dataset. To minimise this threat, we checked and excluded less popular JavaScript projects until they do not dominate our dataset, as presented in filter *iv* of Section 4.3.2.

External validity. Three factors can contribute to external threats to validity. First, we used Git and GitHub as platforms, the three-way merge pattern, and the set of metrics. Generalizability to other platforms, projects, development patterns, and set of metrics is limited. This sample limitation was necessary to reduce the influence of confounds, increasing internal validity, through [263]. While more research is needed to generalise to other version control systems and development patterns, we are confident that we select and analyse 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 sample real and active projects (see Section 4.3.2). Second, we could not retrieve information from binary files; hence, we may miss information from some merge scenarios. Unfortunately, we could not do anything about that, however, the number of binary files is normally small in software projects. Third, performing only automated analyses. Interviewing or surveying developers could make our analyses more trivial; however, considering that developers think they are doing the right thing, their answers could not point to their faults.

4.7 Conclusion

In this study, we investigated the relation of top and occasional contributors on the emergence of merge conflicts and merge conflict predictions using social and technical assets. To achieve our goal, we mined 66 repositories of popular software projects with a total of 78 740 merge scenarios.

As a result of our initial analysis to understand the influence of developer roles on merge conflicts, we saw that those roles are practical and statistically related to the emergence of merge conflicts. When looking at project level, top contributors are more related to merge conflicts than occasional contributors. On the other hand, when looking at merge-scenario level, occasional contributors are more related to merge conflicts than top contributors. Joining the analysis of project and merge-scenario level, we saw those scenarios, where top contributors at project level and occasional contributors at merge-scenario level contribute, are more related to merge conflicts than the other combination of developer roles. We also found that contributions on the source branch are more conflict-prone than contributions on the target branch. For instance, 24.60% of the contributions of occasional contributors in

the source branch resulted in merge conflicts, while only 8.32% of these contributors on the target branch resulted in merge conflicts.

Our predictions achieved 100% of recall for the three models we built (social measures vs. technical measures vs. all measures). Predicting merge conflicts using social and technical assets is useful in practice and these models retrieved all real conflicting scenarios. At the end, we reinforce the importance of using the information of the touched branch and the social perspective in analyses of software repositories. These pieces of information are important since coding is a social task and they played an important role in our analyses.

In the next chapter, we investigate how developers contribute to the source code looking deeper to the contributions of five developers of different projects.

5

Behind Developer Contributions on Conflicting Merge Scenarios

This chapter shares material with a prior publication: "Behind Developer Contributions on Conflicting Merge Scenarios" [300]

In this chapter, we investigate how developers contribute to software projects and the rate of contributions related to merge conflicts. This is a follow up study suggested by the end of Chapter 4. In Section 5.1, we introduce this chapter presenting the context, problem, goals, and discussion of our study. In Section 5.2, we present the setting of this empirical study. In Sections 5.3 to 5.5, we present our study results for each of our three research questions. In Section 5.6, we discuss related work. In Section 5.7, we discuss threats to the study validity while, in Section 5.8, we conclude this empirical study.

5.1 Introduction

Contemporary software development greatly depends on simultaneous contributions of several developers [185, 227, 286]. Such observation particularly stands in the OSS) development [116, 205], which has GitHub as a prominent enabling platform. We refer to all GitHub users who have contributed to an OSS project (by either communicating with other users or changing source files) as contributors [302]. Users whose contribution is highly frequent (in this work, we consider users who are responsible for 80% of all contributions to an OSS project) are called *top contributors*. The remaining users are called *occasional contributors*.

Contributions may occur at two distinct levels of the OSS project development: *merge-scenario level* and *project level* [301, 302]. The former refers to contributions in a merge scenario, while the latter refers to contributions on the whole project at the end of each merge scenario (i.e., at the merge commit). Given the distributed nature of contemporary software development, contributions often affect a specific changing source file simultaneously and may lead to merge conflicts [2, 154]. Whenever the integration of simultaneous contributions leads to merge conflicts, we have a *conflicting merge scenario*. Contributors who were involved in a conflicting merge scenario are called *conflicting contributors*.

As seen in Section 2.4, several empirical studies investigated conflicting merge scenarios, however, little has been empirically done in terms of investigating the involvement of OSS

contributors in conflicting merge scenarios. We advocate that understanding social aspects of conflicting contributors (and their activity on changing source files) can help decide which OSS contributors to instruct (and when to do it) with the purpose of avoiding conflicting merge scenarios.

In this chapter, we present a large-scale quantitative study aiming at investigating what is *behind developer contributions on conflicting merge scenarios*. We are particularly interested in the three following aspects of developer contributions. First, we assess the extent in which OSS contributors are involved in conflicting merge scenarios. Second, we look at the top contributors to understand their involvement with conflicting merge scenarios. Third, we characterise changing source files typically involved in conflicting merge scenarios.

To achieve our goals, we systematically collect both contributor data and contribution data from 66 popular GitHub projects. From a total of 25 397 distinct contributors, we analysed 2 972 (11.70%) contributors who are involved in at least one conflicting merge scenario. We rely on both descriptive and inferential statistics to address our research questions. We summarise below our main study findings and their potential implications.

- **80% of conflicting contributors were involved in only one or two conflicting merge scenarios.** Thus, a small group of conflicting contributors (only 20%) is involved in the majority conflicting merge scenarios. We advocate that training this specific small group could significantly reduce the number of merge conflicts.
- **64% of the 66 OSS projects under analysis had their top contributor as the one involved in more than 50% of the conflicting merge scenarios.** This result emphasises the role of top contributors in the success of OSS projects with respect to the overall project quality.
- **A small set of changing source files were involved in conflicting merge scenarios.** Additionally, our results suggest that the most changed source files are affected by trivial, minor changes. Thus, we believe that contribution rules defined for each OSS project could avoid merge conflicts caused by these trivial, minor source file changes.

5.2 Study Design

In this section, we describe our study design as follows. In Section 5.2.1, we introduce our study goal and RQs and in Section 5.2.2, we describe the data extraction and analysis procedures.

5.2.1 Study Goal and Research Questions

We rely on the Goal-Question-Metric template [24] to systematically define our study goal as follows: *analyse* OSS project contributors involved in conflicting merge scenarios; *for the purpose of* acquiring empirical evidence on characteristics and activities performed by the contributors; *with respect to* 1) how often contributors are involved in conflicting merge scenarios and the extent of such involvement, 2) key characteristics of the contributors

involved in conflicting merge scenarios, and 3) characteristics of the contributions in terms of changed source files; *in the context of* popular GitHub projects.

We defined the three RQs with the purpose of driving our study:

> **RQ_1:** *To what extent open source project contributors get involved in conflicting merge scenarios?*

We are particularly interested in the distribution of each developer role rather than a general analysis. Thus, RQ1 is decomposed into the two sub-questions below:

> **$RQ_{1.1}$:** *How often do contributors get involved in conflicting merge scenarios?*
>
> **$RQ_{1.2}$:** *What is the proportion of involvement in conflicting merge scenarios by conflicting contributors?*

> **RQ_2:** *How often are top contributors or top conflicting contributors involved in conflicting merge scenarios?*

> **RQ_3:** *What are the main characteristics of the changed source files in conflicting merge scenarios?*

While answering RQ_1 we are able to quantify the number of developers often involved in conflicting scenarios, the answer of RQ_2 provides us characteristics of these contributors. Finally, answering RQ_3 we are able to know the type of files conflicting contributors often change. Hence, researchers and practitioners themselves can better know which developers might better get synchronised and provide guidelines to avoid future merge conflicts.

5.2.2 Data Extraction and Analysis Procedures

To perform this study, we follow the procedures presented in Appendix A.3.1 to rebuild merge scenarios and similar to the procedures presented in Section 4.3.2 to select subject projects. To be honest, this is a follow up study suggested by the previous study presented in Chapter 4. Next, we present the analysis procedures to answer our RQs.

RQ_1 Analysis: To answer $RQ_{1.1}$, we create a table with all OSS contributors who are involved in at least one merge conflict. After, we group the contributors into five categories according to the number of conflicting merge scenarios they are involved in: one, two, three to five, six to ten, and more than ten conflicting merge scenarios. These values were arbitrarily chosen to support the data visualisation given its power law distribution. In our case, power law distribution means that several merge scenarios have few contributors and few merge scenarios have several contributors. As we know the developer roles in each merge scenario, we normalise each developer role to compare the number of developers who contribute in each group. To answer $RQ_{1.2}$, we also group contributors and analyse the distribution over the subject developer roles. However, we are interested in comparing

their total of contributions with respect to their conflicting contributions. For instance, we investigate how often the contributions of a certain contributor lead to merge conflicts.

RQ_2 Analysis: Our RQ_2 analysis is two-fold. First, we analyse the share of conflicting merge scenarios of the top contributors by project. Top contributors are those who contribute to majority merge scenarios. Second, we analyse the share of conflicting merge scenarios of top conflicting contributors. Top conflicting contributors are those who are involved in the highest number of conflicting merge scenarios in the subject projects.

RQ_3 Analysis: We rely on manual and qualitative procedures to address RQ_3. We start by obtaining the list of top conflicting contributors from the five subject projects with the highest number of conflicting merge scenarios. After, we manually inspect each conflicting merge scenario. Our goal is to capture factors that may have led to the merge conflicts. We provide details on which factors were explored in Section 5.5. For short, we perform three analyses: 1) we explore the numbers of merge scenarios and conflicting merge scenarios in these projects; 2) we discuss the impact of project rules on merge conflicts; and, 3) we analyse some changed files related to conflicts.

5.3 RQ_1: To What Extent Open Source Project Contributors Get Involved in Conflicting Merge Scenarios?

In this section, we answer RQ_1 by discussing the extent in which OSS contributors are involved with conflicting merge scenarios. In Section 5.3.1, we provide the results regarding how often contributors are involved in conflicting scenarios, while in Section 5.3.2, we present the results on the proportion of involvement in conflicting merge scenarios in contrast to all merge scenario contributions.

5.3.1 $RQ_{1.1}$: How Often Do Contributors Get Involved in Conflicting Merge Scenarios?

Overall Results. In Figure 5.1, we depict the distribution of contributors by the number of conflicting merge scenarios they were involved in. Such distribution is presented in terms of five groups: one, two, three to five, six to ten, and more than ten conflicting merge scenarios. We present both the absolute number of contributors and the percentages with respect to the total of 2972 contributors involved in at least one conflicting merge scenario. We discuss below our main findings.

Our data suggests that about 80% of contributors are involved in only one or two conflicting merge scenarios. Indeed, we found that majority contributors are involved in either one conflicting merge scenario (62.6%) or two conflicting merge scenarios (17.3%). On one hand, this result can be expected if we consider the nature of globally distributed development [195] and the inherent complexity of modern OSS projects involving several contributors [192]. On the other hand, it contrasts with past work assumptions on the relationship between merge conflicts and the little inexperience of new contributors with a

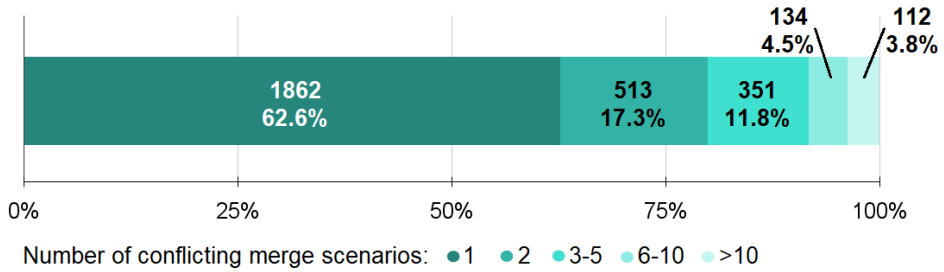

Figure 5.1: Distribution of Contributors by the Number of Conflicting Merge Scenarios They Were Involved in

specific OSS project [117]. Interestingly, however, we see in data of Figure 5.1 that approximately 20% of contributors are involved in more than three conflicting merge scenarios. We also highlight that only 3.8% of the contributors are involved in more than ten conflicting merge scenarios. This result can be explained because, even in large OSS projects with hundreds of contributors, a very small group of contributors are responsible for most tasks on maintaining or evolving a project.

Results by Developer Role. In Figure 5.2, we show the distribution of the investigated developer roles by the number of conflicting merge scenarios. That is, we compute the distribution of contributors by their respective top or occasional roles (see Section 4.2). We expect this new perspective helps us understand the nature of those contributors who are more often involved in conflicting merge scenarios. We discuss below our main observations.

Project-level contributions. In Figure 5.2(a), we show that 74.61% of top contributors are involved in more than ten conflicting merge scenarios, while only about 20% of occasional contributors are involved in such scenarios. This result is explainable since top contributors make the majority of key decisions on the project maintenance and evolution. On the other hand, approximately 50% of occasional contributors are involved in one or two conflicting merge scenarios. It is interesting given the occasional nature of many contributions [241, 244], which may be fine-grained and less likely to generate conflicts.

Merge scenario-level contributions. In Figure 5.2(b), we show similar trends especially for occasional contributors. Slightly more than 50% of top contributors are involved in more than ten conflicting merge scenarios, against less than 25% for occasional contributors. This is an expressive percentage, although less expressive than the 75% of top contributors at project level. We speculate that such drop in percentages may be because we are looking at a fine-grained level and the changes of top contributors are more specific, like fixing a bug or introducing a new feature. On the other hand, about 40% of occasional merge scenario-level contributors are involved in one or two conflicting merge scenarios.

$RQ_{1.1}$ Summary: Overall, approximately 80% of OSS contributors are involved in a very small number of conflicting merge scenarios, i.e., one or two scenarios. With the analysis by developer roles, we see that top contributors are often involved in more than 10 conflicting merge scenarios while occasional contributors are often involved in less than 5 conflicting merge scenarios.

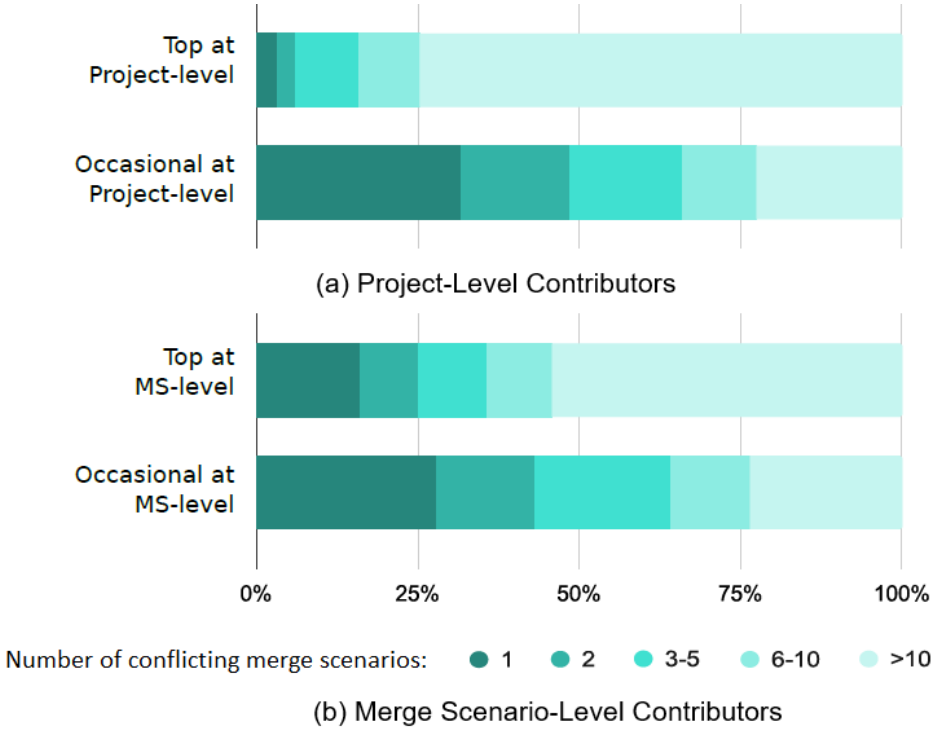

Figure 5.2: Distribution of Contributors by the Number of Conflicting Merge Scenarios They Participate

5.3.2 $RQ_{1.2}$: What Is the Proportion of Involvement in Conflicting Merge Scenarios by Conflicting Contributors?

In this section, we investigate the rate among all contributions and the conflicting contributions for the 2972 contributors that are involved with merge conflicts.

In Figure 5.3, we show this rate divided into four groups: up to 25%, between 25% and 50%, between 50% and 75%, and greater than 75%. The first (<= 25%) and last (> 75%) groups are the ones with more contributors. To illustrate the first group, a developer of project *netdata*[1] contributed to 1085 merge scenarios and only two of them resulted in merge conflicts, i.e., this developer has a conflicting rate of 0.18%.

Regarding the first group, we find that 484 contributors have a very small conflicting rate (i.e., up to 5%). These are probably top contributors (at project- and merge-scenario-level) that skip conflicts. Regarding the last group (> 75%), we found that 1019 contributors introduce merge conflicts in all of their contributions. It represents 34.3% of the contributors involved with merge conflicts and 4% of all subject contributors.

In Figure 5.4, we present the share of conflicting contributions by the amount of merge scenarios contributions at project- and merge-scenario-level. As we observe in this figure, the majority of the contributors of all developer roles are in the group that has a conflicting rate smaller than 25%. On the other hand, we see that occasional (at project- and merge-scenario-level) developers have more than 25% of developers in the group with a rate of

1 https://github.com/netdata/netdata

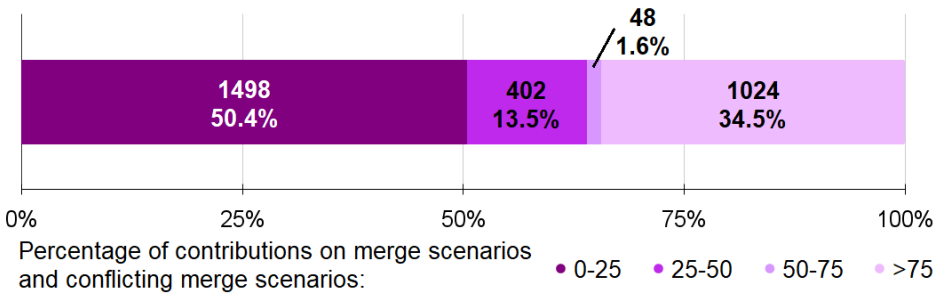

Figure 5.3: Number of Developers Related to Conflicting Merge Scenarios

conflicting merge scenarios above 75%. Hence, we assume that most of the contributors that have a very high conflicting rate do not contribute often to the project. Looking at our data, we observed that the code contributions of these occasional developers who rate is 100% of conflicting contributions vary from 1 to 25 merge scenarios. In fact, 997 of them contributed to up to 5 merge scenarios.

$RQ_{1.2}$ Summary: About 50% of the contributors involved with merge conflicts have a rate of 25% of their contributions into conflicts. The majority of these developers are top contributors at project- and merge-scenario levels. Surprisingly, about 34% of conflicting contributors have merge conflicts in all of their contributions. Given the low number of contributions and supported by our data, we see that most of them are occasional contributors.

5.4 RQ_2: How Often Do Top Contributors or Top Conflicting Contributors are Involved in Conflicting Merge Scenarios?

In this section, we give an overview of the top contributors (Section 5.4.1) as well as of the top conflicting contributors of each project (Section 5.4.2).

5.4.1 Most Active Contributors

In Figure 5.5, we present the percentage of conflicting merge scenarios for top contributors of each subject project. We can see in this figure that 37.9% of the top contributors participated in up to 25% of the conflicting merge scenarios. We also see that 21.2% of the top contributors were involved with 25-50% of conflicting merge scenarios, 27.3% of the top contributors participated in 50-75% of the conflicting merge scenarios, and 13.6.% of these top contributors participated in more than 75% of the conflicting merge scenarios. In other words, in 27 out of 66 projects the top contributors were involved with more than 50% of the conflicting merge scenarios.

In 12 projects, the top contributors participate in more than a thousand merge scenarios. However, in only 4 projects, these developers are involved with the most conflicting merge

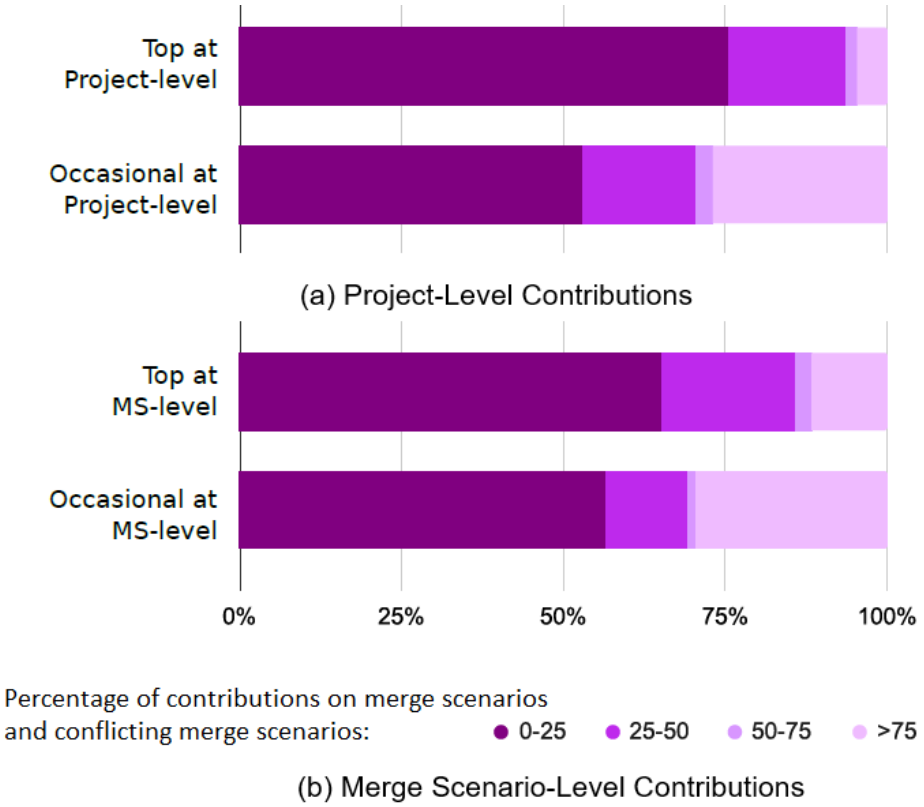

Figure 5.4: Share of Conflicting Contributions by the Amount of Merge Scenario Contributions at Project- and Merge-scenario Levels

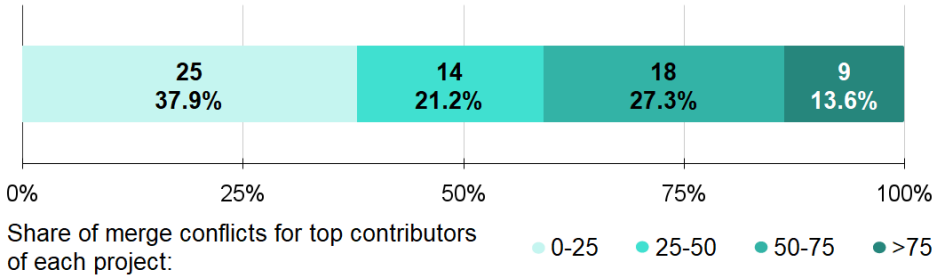

Figure 5.5: Share of Conflicting Merge Scenarios for the Top Contributor of each Subject Project

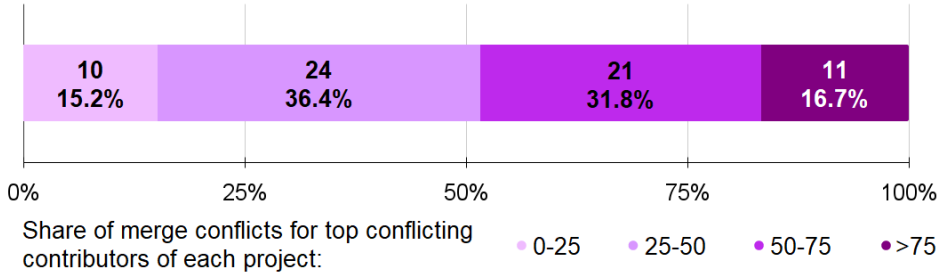

Figure 5.6: Share of Conflicting Merge Scenarios for the Top Conflicting Contributor of each Subject Project

scenarios in their project. One example is a contributor to the BOOTSTRAP[2] project. She contributed to 3 167 merge scenarios and was involved with 306 conflicting merge scenarios. It represents participation in 72.5% of conflicting merge scenarios of this project. We give details of this case in Section 5.5 when answering RQ_3.

5.4.2 Most Active Conflicting Contributors

In Figure 5.6, we depict the percentage of conflicting merge scenarios for top conflicting contributors of each subject project. For short, we observe in this figure that 15.2% of the contributors participated in 25% of the conflicting merge scenarios while 36.4%, 31.8%, and 16.7% of these contributors were involved with 25-50%, 50-75%, and 75% of the conflicting merge scenarios, respectively. In other words, in 32 out of 66 projects the top conflicting contributor is involved with more than 50% of the conflicting merge scenarios.

In a few cases, top conflicting contributors participated in more than 100 conflicting merge scenarios. For instance, in the case of project D3[3] the same developer is involved with 364 conflicting scenarios. It represents 84.26% of the conflicting merge scenarios of that project.

Based on a manual verification, we compare the contributors that represent both charts of Figures 5.5 and 5.6. We see that they are the same developers in 42 projects; i.e., the developer that contributes to more merge scenarios in the project is also the one involved with the majority of conflicting merge scenarios. Interestingly, in two projects, the top contributor is involved with the same number of conflicting merge scenarios than other contributors. As the participation of top contributors and the top conflicting contributors in the majority of conflicting merge scenarios often happens, it is interesting to look deeper at their contributions aiming at identifying coding practices that lead to merge conflicts. We perform this analysis in Section 5.5.

2 https://github.com/twbs/bootstrap
3 https://github.com/d3/d3

RQ_2 Summary: In 42 out of 66 projects, the top contributor is also the top conflicting contributor. In 39.4% of the projects, the top contributors participate in the majority of conflicting merge scenarios in their project. Similarly, in 48.49% of the projects, the top conflicting merge scenario contributors are involved with the majority of the conflicting merge scenarios that happened in their respective projects. In other words, in 27 and 32 projects, the top contributor and top conflicting contributors are involved with more than 50% of the conflicting scenarios, respectively. It may be an indicator that these contributors follow bad practices which make them participate in the majority of the merge conflicts of their projects.

5.5 RQ_3: What Are the Main Characteristics of the Changed Source Files in Conflicting Merge Scenarios?

Looking at our data, we found three projects (CS-Notes, markdown-here, systems-design-primer) that the top conflicting contributors participated in all conflicting merge scenarios. However, since the number of conflicting merge scenarios is small (i.e., up to 37) in these cases, we decided to focus on the five projects with more than 1 thousand conflicting merge scenarios: d3, bootstrap, meteor, webpack, FreeCodeCamp. To preserve the identity of the top conflicting contributors of these projects, we anonymously nickname them C1, C2, C3, C4, and C5.

We organise this section according to the three analyses described in Section 5.2.2 to answer RQ_3 with data of these 5 contributors.

5.5.1 Contributions of Top Conflicting Contributors

In Table 5.1, we present the overall contributions of the top five conflicting contributors considering their absolute and relative numbers of conflicting merge scenarios. We organise Table 5.1 into three major parts. In the first one (Project), we show the overall number of merge scenarios (#MS) and conflicting merge scenarios (#CMS) in the five selected projects. In the second part (Contributor), we depict the #MS and #CMS that the contributor participated in. That is, she changed at least one line of code in these merge scenarios. Finally, in the third part (Conflict), we indicate #CMS that the contributor was actually involved in conflicting code. That is, their committed code was responsible for triggering a merge conflict in the respective merge scenario.

For instance, the project of C1 has 432 conflicting merge scenarios, but this contributor only participated in 397 of them. From these, C1 committed conflicting code into 364 merge scenarios. Moreover, we see 1076 merge scenarios in her project and C1 contributed to 866 them. Hence, she contributed to 80.48% of the merge scenarios of this project. C2, C3, C4, and C5 participate in 47.52%, 30.73%, 62.87%, and 21.39% of the merge scenarios of their projects, respectively. Only C1 and C4 participated in the majority of the merge scenarios in their projects. Therefore, the high participation of these contributors on conflicting merge

Table 5.1: Overview contributions of the top-five conflicting contributors

	Project		Contributor		Conflict	
Id	**#MS**	**#CMS**	**#MS**	**#CMS**	**#CMS**	**%CMS**
C1	1076	432 (40.15%)	866 (80.48%)	397 (91.90%)	364 (84.26%)	42.03
C2	6665	422 (6.33%)	3167 (47.52%)	358 (84.83%)	306 (72.51%)	9.66
C3	2737	345 (12.60%)	841 (30.73%)	178 (51.59%)	134 (38.84%)	15.93
C4	2486	132 (5.31%)	1563 (62.87%)	116 (87.88%)	89 (67.42%)	5.69
C5	4665	108 (2.31%)	998 (21.39%)	100 (92.59%)	90 (93.33%)	9.02

***#MS**: number of merge scenarios,*

***#CMS**: number of conflicting merge scenarios,*

***%CMS**: percentage of CMS by MS of the contributor.*

scenarios does not necessarily come only from high contributions to the majority of merge scenarios in the project.

Focusing on the percentages of columns #CMS (Contributor), we note that all contributors changed more than 50% of conflicting merge scenarios. Similarly, all contributors, except C3, committed code causing conflicts in the majority of the conflicting scenarios, as we see in the #CMS (Conflict) column of Table 5.1. This is an indicator that these developers indeed influence the number of merge conflicts.

Aiming at finding out whether these contributors are isolated cases or recurrent practices of several developers in these projects, we compare the rate of conflicting merge scenarios in the project (3rd column) and the rate of conflicting merge scenarios of the contributor (last column in Table 5.1). For instance, we see that the rate of conflicting merge scenarios of C1's project is 40.15%. However, C1 has a higher rate of conflicting merge scenarios (42.03%) than its project. In fact, all contributors presented in Table 5.1 have a higher rate of conflicting merge scenarios than the general average data of their respective projects. In the case of C5, while the project rate of conflicting merge scenarios is 2.31%, she has a rate of 9.02%; i.e., four times higher. These results suggest that these five contributors have relevant impact on the high percentages of merge conflicts in their projects. Therefore, a deeper analysis in their code changes is needed to uncover what these specific developers do.

RQ_3 Summary 1: Top conflicting contributors are not always involved with most merge scenarios in their respective projects, although they participated in most conflicting merge scenarios. In fact, their committed code is responsible for more merge conflicts that the average rate of conflicting merge scenarios in their projects which suggests that their coordination is crucial to the project success.

Table 5.2: Comparing Conflicting Merge Scenarios Before and After the Creation of Contribution Rules

Project Name	#MS before	#MS after	%CMS before	%CMS after
bootstrap (C2)	1 544 (231)	5 121 (191)	14.96%	3.73%
meteor (C3)	526 (51)	2 211 (294)	9.70%	13.30%
webpack (C4)	99 (5)	2 387 (127)	5.05%	5.32%
freeCodeCamp (C5)	1 391 (81)	3 274 (27)	5.82%	0.83%

***#MS**: number of merge scenarios, **%CMS**: percentage of conflicting merge scenarios. The numbers in parenthesis stand for the number of conflicting merge scenarios.*

5.5.2 Project Contribution Rules

Previous work [2, 10, 174] has shown that several merge conflicts arise from formatting or from the location of code changes in a file. An easy way to minimise merge conflicts due to formatting and the location of the changes is through the definition of contribution rules. Contribution rules normally define the contribution process as well as the code style. Aiming at identifying whether these five projects have defined contribution rules, we looked at their *README.md* file searching for links to other files or definitions of contribution rules. Except for project D3, we found contribution rules for the other four projects. Furthermore, we observe that developers often define contribution rules in a file named *CONTRIBUTING.md*. Aiming at finding out if the rate of conflicting merge scenarios increased or reduced after creating this file, we get the date this file was merged to the main branch for the first time and compare the number of merge scenarios with the number conflicting merge scenarios before and after the creation of this file.

In Table 5.2, we present a summary of this analysis for the four projects with contribution rules defined. As we see in this table, most merge scenarios were created after the creation of the contribution rules file for all projects. For two of them, the creation of contribution rules dramatically reduced the share of conflicting merge scenarios from 14.96% to 3.73% and from 5.82% to 0.83%. For the other two cases, it does not seem to have an impact on the conflicting merge scenarios rate. Note that we evaluate neither the quality of the contribution rules nor the number of contributors in these two time frames; we only check if the file with contribution rules exists or not. Hence, other factors may influence the emergence of merge conflicts (limitations of this approach are discussed in Section 5.7). Anyway, we believe that adding contribution rules helps to avoid the emergence of simple merge conflicts and, consequently, the general number of merge conflicts.

RQ_3 Summary 2: The analysis of contribution rules diverges among projects. However, for most projects, we observed that the contribution rules may reduce the emergence of merge conflicts.

Table 5.3: Overview of Changed Files in Projects of Top Five Conflicting Contributors

Cont.	#Dist. Files	#Dist Files with Confl.	#File versions	#File versions with Confl.
C1	1 824	79 (4.33%)	39 202	603
C2	5 930	224 (3.78%)	142 533	1 717
C3	7 020	313 (4.46%)	114 757	1 129
C4	4 636	153 (3.30%)	92 266	275
C5	2 876	72 (2.50%)	52 875	138

***Cont.**: Contributor, **#Dist. Files**: number of distinct files, **#Dist. Files with Confl.**: number of distinct files with merge conflicts, **#File versions**: number of file versions, **#File versions with Confl.**: number of file versions with merge conflicts*

5.5.3 Changed Files

We have the feeling that some files are conflict-prone, for instance, because they change more often than other files or because they are somehow more central to the project. We investigated these assumptions in 4 ways.

First, in Table 5.3, we present an overview of distinct files (i.e., files with different paths and names), all file versions (i.e., all versions of distinct files) for the five projects that C1-C5 contribute to as well as the number of distinct files and file versions with merge conflicts. A new version of a file arises when one or more developers change it in a merge scenario. To illustrate the data of Table 5.3, look at the project of C1. In this project, there are 1 824 distinct files of which 79 have merge conflicts. From these distinct files, we find 39 202 versions of which 603 have merge conflicts. The rate of merge conflicts in distinct files is 4.33%, 3.78%, 4.46%, 3.30%, 2.50% for the projects that C1, C2, C3, C4, and C5 contribute to, respectively. Hence, if all files change equally and considering an average of 4% conflict rate, the chance of emerging merge conflicts would be of around 1 in each 25 files changed. Looking at the number of file versions (*#File versions* column), we see that files often change. For instance, the 5 930 distinct files of C2's project changed 142 533 times. If all files change equally (i.e., each file changed 24.90 times), the chance would be of at least one merge conflict in each file. However, as the 1 717 merge conflicts were distributed over only 224 files, if they change equally, each of these files would have around 8 merge conflicts. Therefore, the main conclusion we can draw from this table is that files change often and merge conflicts are concentrated in a few files.

Second, in Table 5.4, we present the top-three files changed for the projects of C1-C5 with the number of merge scenarios emerging conflicts in these files and the percentage it represents. To illustrate, the two most changed files for the project that C1 contributed to (*d3.js* and *d3.min.js*) changed in 732 and 715 merge scenarios and had merge conflicts in

Table 5.4: Top-three Files Changed Over Time for Projects of C1-C5

Cont.	Top-3 files	#MS	#CMS	%CMS
C1	d3.js	732	70	9.56
	d3.min.js	715	291	40.70
	package.json	425	9	2.12
C2	docs/index.html	2 384	66	2.77
	dist/css/bootstrap.min.css	2 008	52	2.59
	dist/css/bootsatrap.css	1 928	137	7.11
C3	packages/meteor-tool/package.js	595	69	11.60
	.../meteor-release-experimental.json	544	54	9.93
	packages/webapp/package.js	497	24	4.83
C4	package.json	1 030	39	3.79
	lib/Compilation.js	651	7	1.08
	lib/Parser.js	435	4	0.92
C5	package.json	826	9	1.09
	seed/.../basic-javascript.json	677	0	0.00
	server/boot/user.js	479	0	0.00

***Cont.**: Contributor, **#MS**: number of merge scenarios,*
***#CMS**: number of conflicting merge scenarios, and*
***%CMS**: percentage of conflicting merge scenarios over all merge scenarios.*
The light-gray background just groups files from the same project

70 and 291 of them, respectively. It means that these files changed in 68.03% and 66.45% (732 and 715 out of 1 076) of the merge scenarios of this project and in 9.56% and 40.70% of the times they changed, merge conflicts emerged. Therefore, we see that these files have a greater chance to change and to arise merge conflicts than the average of files of this project (9.56% and 40.70% against 4.33%). On the other hand, the third most changed file for the same project (*package.json*) had only 9 conflicts in the 425 merge scenarios that it changed (2.12%). From the top three files most change for the project of C5, only the first one have merge conflicts. This table supports our understanding that files that often change usually have more merge conflicts than files that change less frequently. However, only the number of changes may lead to wrong conclusions and the outcome may change from project to project.

Third, in Table 5.5, we present the top-three conflicting files for the projects of C1-C5 with the number of merge scenarios emerging conflicts in these files and the percentage it represents. To illustrate, the top three conflicting files for the project that C2 contribute to (*bootstrap-1.2.0.css*, *bootstrap.css*, and *bootstrap-1.0.0.css*) have merge conflicts in 77.78%, 21.71%, and 25.93% of the times they changed, respectively. On the other hand, the top-three conflicting files for the C5 project cause merge conflicts in less than 2.00% of the times they changed. Note that only 5 files appear in both Table 5.4 and Table 5.5 and that despite

Table 5.5: Top-three Conflicting Files Over Time for Projects of C1-C5

Cont.	Top-3 files	#MS	#CMS	%CMS
C1	d3.min.js	715	291	40.70
	d3.v2.min.js	203	88	43.35
	d3.js	732	70	9.56
C2	bootsatrap-1.2.0.css	216	168	77.78
	bootsatrap.css	631	137	21.71
	bootsatrap-1.0.0.css	324	84	25.93
C3	packages/meteor-tool/package.js	595	69	11.60
	.../meteor-release-experimental.json	544	54	9.93
	packages/babel-compiler/packages.js	396	41	10.35
C4	package.json	1 030	39	3.79
	test/.../StatsTestcases.tests.js.snap	260	14	5.38
	yarn.lock	379	8	2.11
C5	seed/challenges/basic-javascript.json	471	9	1.91
	package.json	826	9	1.09
	README.md	444	7	1.58

***Cont.**: Contributor, **#MS**: number of merge scenarios,*

***#CMS**: number of conflicting merge scenarios, and*

***%CMS**: percentage of conflicting merge scenarios over all merge scenarios.*

The light-gray background just groups files from the same project

the majority of files in these five projects are JavaScript files, only 5 out of 15 files of Table 5.5 are JavaScript files. With these analyses, we conclude that frequently changed files are more conflict-prone than others, but that the results may vary from project to project. Hence, merge conflict prediction strategies may have better performance when learning with previous merge scenarios of a project, i.e., there is no silver bullet strategy across projects. However, we also argue that the performance will depend on the specific characteristics of a project (e.g., life-cycle, domain, type of project, and programming language).

Fourth, with the knowledge acquired from previous discussion, we compute the Spearman's rank correlation between the number of times each file changed and the number of times code changes caused merge conflicts for all subject projects. Spearman's rank correlation is more adequate for our analysis than Pearson's correlation because our data do not have a normal distribution [152]. We find a Spearman's rank correlation of 0.26 with 99% significance level (p-value $< 2.2^{e-16}$). Since some files are more conflict-prone than others (see discussion of the 2nd and 3rd investigations above), we refine our analysis including only files with merge conflicts. In this analysis, we find a Spearman's rank correlation of 0.55 with 99% significance level (p-value $< 2.2^{e-16}$). Moreover, we believe that other data refinements, artificial intelligence techniques, and analysis considering characteristics of each project may increase even more this correlation.

RQ_3 Summary 3: The analysis on changed files reveals that files changing more often are conflict-prone. In addition, we also found that merge conflicts are concentrated in a few files and argue that better predictions of merge conflicts can be achieved by considering historical and characteristics of each project.

5.6 Related Work

In this section, we discuss studies related to merge conflicts and developer classification to support the organisational structure of OSS.

Merge conflicts have a negative effect on project's objectives compromising the project success, especially when arising frequently [123, 252]. Hence, researchers have investigated, for instance, merge strategies [10, 11], prediction strategies [49, 123, 163], awareness tools [32, 163, 252], tried to understand types of code changes related to conflicts [2, 110, 180], and strategies to efficiently resolve merge conflicts [111, 154, 214, 301]. For instance, Ji et al. [154] empirically investigated merge conflicts and resolutions in Git rebase scenarios. Their results suggest that rebases are often performed (about 8% of pull requests have rebases and 41% have two or more rebases). Several rebases are performed for the purpose of reducing reviewing changes in pull requests. About 25% of rebases involve textual conflicts, and approximately 29% of conflict rebases involve the introduction of new tokens. This study is particularly interesting because it emphasises how complex the conflicting merge scenarios can be in practice, and how important it is to assist developers in resolving merge conflicts. As another example, Gonzalez and Fraternali [111] investigated merge conflict resolution, but now through the proposal and evaluation of a strategy that extends Git Rerere (REuse REcorded REsolution) with novel features. Git Rerere was designed to automatically resolve conflicts that are similar to previously solved conflicts. Their results suggest that the tool can resolve about 49% of the conflicts generated during the merge process, with most solutions being similar or the same as solutions manually performed by developers. This study is particularly interesting because it suggests that several merge conflicts have limited size, i.e., they involve one or two lines of code, and their solutions are significantly simple. Such finding is inline with our discussion regarding the importance of project contribution rules to resolve simple merge conflicts (see Section 5.5.2).

On the other hand, researchers have classified developers aiming at understanding the organisational structure of OSS [36, 70, 90, 156, 205, 243, 290]. For instance, Mockus et al. [205] found empirical evidence for the Mozilla browser and the Apache Web server that a small number of developers are responsible for approximately 80% of the code modifications. Their approach consists of counting the number of commits made by each developer and then computing a threshold at the 80% percentile. Although using a different approach, their result is inline with our results and also inline with previous work [90, 243, 290]. As another example, Joblin et al. [156] empirically classified developers into core and peripheral to model the organisational structure using network metrics (e.g., degree- and eigenvector-centrality) and analysed how the set of core developers changed over time.

Despite the number of studies exploring merge conflicts and developer roles separately, we lack studies specifically focused on the contribution degree of different types of OSS developers to the occurrence of conflicting merge scenarios. We fill this gap performing

three analyses first getting an overview on the topic and later looking deeper at the impact of code contribution patterns of developers of 5 projects as well as to their source code changes and the influence of project contribution rules on the emergence of merge conflicts. Our results bring an understanding of which developer roles and source code changes are related to merge conflicts. Furthermore, we provide implications and directions to researchers and practitioners avoiding conflicting merge scenarios.

5.7 Threats to Validity

We discuss possible construct, internal, conclusion, and external threats to the study validity [309].

Construct Validity. Construct validity concerns inferring the result of the experiments to the concept or theory [309]. For instance, we used only metrics based on frequency of changes to classify developers between top and occasional contributors. This poses a threat that the metrics do not accurately capture actual scenarios of distributed software development. However, we believe this threat does not invalidate our main findings since existing evidence indicates that these metrics accurately reflect the developers' perception. Additionally, to answer RQ_3, we focus only on the existence of contribution rules in a specific file (*CONTRIBUTING.md*). For instance, we did not evaluate the quality of these contribution rules. A further study could investigate other characteristics of the contribution rules, such as its extension and clarity.

Internal Validity. Threats to internal validity are influences that can affect the independent variable to causality [309]. In our case, this threat may refer to the chosen dataset. For instance, since we selected projects from different programming languages, one or a few languages could have dominated our dataset. To minimise this threat, we excluded less popular JavaScript projects until they do not represent more than 50% our dataset. We excluded 6 projects with this filter. Another threat is the choice of only 5 subject projects to answer RQ_3. We selected them because they are large (i.e., 22.39% of all investigated merge scenarios and 36.43% of all conflicting merge scenarios). Hence, we investigated 1 439 out of 3 950 conflicting merge scenarios. It brings a confidence level of 99% within $\pm 3\%$ margin of error that we are measuring a significant amount of conflicting merge scenarios.

Conclusion Validity. Threats to the conclusion validity are concerned with issues that affect the ability to draw the correct conclusion between the treatment and the outcome [309]. In our study, a potential threat to conclusion validity is the reliability of the data extraction, since not all information was clear to answer the RQs. As a result, some data had to be inferred and sometimes cross-discussions among the paper authors took place to reach a common agreement.

External Validity. Threats to external validity are conditions that limit our ability to generalise the results of our study [309]. Our data come from Git and GitHub platforms and we restricted our analyses to projects following the three-way merge pattern. Therefore, we cannot generalise our results to other platforms, projects, development patterns, and developers. While more research is needed to allow generalisation, we select and analyse a largely used platform and a high number of software projects from various domains, programming languages, sizes, and coordination practices.

5.8 Conclusion

In this chapter, we presented a large-scale quantitative study which investigated what is behind developer contributions on conflicting merge scenarios. We were interested in the three aspects of developer contributions: (i) the extent in which OSS contributors are involved in conflicting merge scenarios; (ii) the involvement of top contributors with conflicting merge scenarios; and, (iii) the source files typically involved in conflicting merge scenarios. To achieve our goals, we systematically collected both contributor data and contribution data from 66 popular GitHub projects. From a total of 25 397 distinct contributors, we analysed 2 972 (11.70%) contributors who are involved in at least one conflicting merge scenario. We rely on both descriptive and inferential statistics to address our research questions.

Our results suggest that 62.6% of the contributors are involved only once with merge conflicts and only 3.8% are involved with more than 10 merge conflicts. Regarding the second analysis, the top contributor is also the top conflicting contributor in 42 out of 66 projects. Regarding the third analysis, we found that files changing more often are more conflict-prone than files that rarely change. In addition, we also found that merge conflicts are concentrated in a few files. Based on these results, we argue that training this small group of contributors could significantly reduce the number of merge conflicts. We also argue that merge conflict predictions might benefit by considering historical and characteristics of each project.

In the next chapter, we provide analyses at the end of the merge conflict life-cycle investigating challenges on resolving merge conflicts.

6

Challenges of Resolving Merge Conflicts

This chapter shares material with a prior publication: "Challenges of Resolving Merge Conflicts: A Mining and Survey Study" [301]

In this chapter, we empirically investigate the challenges of resolving merge conflicts mining software repositories and surveying software developers. In Section 6.1, we introduce this chapter presenting the context, problem, goals, and discussions of our study. In Section 6.2, we present the mining study. In Section 6.3, we present the survey study. After, in Section 6.4, we provide a broad discussion about our analyses, non-correlated variables, previous work results, and the merge conflict life-cycle. In Section 6.5, we present threats to validity of our study. Finally, in Section 6.6, we present our final remarks.

6.1 Introduction

Version control systems help developers to manage code changes over time by tracking all code contributions, especially when involving collaborations of multiple developers [323]. This allows developers to address different programming tasks (e.g., bug fixing and adding new features) simultaneously without losing changes. After fulfilling their tasks, developers can merge their changes to the main repository. Simultaneous code changes may introduce problems of their own during integration, often manifesting as merge conflicts (see Chapter 2).

Despite the number of studies investigating merge conflicts, the understanding of challenges and strategies on the *resolution* of merge conflicts is limited [214]. Previous studies have asked developers about barriers and strategies when resolving merge conflicts [43, 196, 214], empirically investigated developers' choice when resolving merge conflicts [110], or empirically investigated merge conflict resolutions in Git rebases [154]. A study showing which factors make merge conflicts longer to resolve is still missing. Such a study can guide practitioners to avoid the creation of time-consuming conflicts, to better coordinate their tasks, and to avoid delaying core-tasks on the project life-cycle (e.g., developing new features and fixing bugs). So far, there is no reliable knowledge on the factors that make merge conflict resolution longer in practice. While previous studies provide an initial

understanding of merge conflict resolution, an empirical study investigating factors that increase merge conflict resolution time in practice may not only confirm and add nuances to previous findings but also pin down the most impacting and recurring factors. These factors together with the knowledge acquired from previous studies may either serve as best practices for developers saving time on merge conflict resolution or as guidelines for tool builders to better support practitioners. At the same time, our results outline opportunities for researchers to improve the state of the art of merge conflicts. Our study is guided on the overarching RQ:

> **RQ**: *Which factors do make merge conflicts longer to resolve in practice?*

To answer this RQ, we conduct a two-phase study. First, we rebuild and extract information from all merge scenarios of 66 GitHub projects, selected based on their popularity. Inspired by previous work [110, 180, 214], we extracted 11 variables for each subject merge scenario. These variables include the *time* (in seconds) to resolve the merge conflict, *measures directly related to merge conflicts*, such as the number of conflicting chunks (*#ConfChunks*), the number of conflicting files (*#ConfFiles*), and the complexity of code in conflict (*CodeComplexity*), and *measures indirectly related to merge conflicts*, such as the number of developers involved (*#Devs*), the number of lines of code (*#LoC*), and the number of chunks (*#Chunks*) of the merge scenario. We group the independent variables into: *directly* and *indirectly* related to merge conflicts, since we suspect that merge conflict resolution depends not only on conflicting code but also on changes not in conflict. In this phase, we performed three main analyses: (1) a *correlation analysis* for each pair of the investigated variables (using correlation matrix and principal component analysis), (2) a *multiple regression model* analysis, and (3) an *effect-size analysis* using Cohen's f^2 measuring the impact of independent variables on our dependent variable.

Second, we conducted a survey with 140 developers from subject projects to triangulate our results and provide a broader understanding of the challenges on merge conflict resolution. For short, we asked developers: (1) to describe how they estimate how hard/time-consuming a merge conflict is to be resolved. Therewith, we checked if our independent variables are in line with measures used in practice, (2) a few statements to understand their processes on merging their contributions and resolving conflicts. The main goals of this analysis were to minimise threats to validity of our dependent variable and to cross-validate our results, and (3) to share their experiences when dealing with merge conflicts to help us understand their challenges and needs.

Summarising our results, the correlation analysis indicates that measures *indirectly* related to merge conflicts (i.e., measures related to the merge scenario changes) are stronger correlated with merge conflict resolution time than measures *directly* related to merge conflicts (i.e., merge conflict characteristics). The regression model analysis reveals that *#LoC*, *#ConfChunks*, *#Devs*, *#ConfFiles*, *#ConfLoC*, and *#Files* have a positive correlation with merge conflict resolution time. Surprisingly, *#Chunks* and *CodeComplexity* show a negative correlation with merge conflict resolution time. In the effect-size analysis, we found that *#Chunks* has a medium effect on merge conflict resolution time, whereas *#Devs*, *#LoC*, *#ConfChunks*, and *CodeComplexity* only have a small effect. Cross-validating our results, survey participants mentioned 25 measures used to quantify how hard/time-consuming is

the resolution of merge conflicts. Measures indirectly related to merge conflicts are among the most cited by them. In addition, they reported that they often merge their changes right after finishing addressing an issue, resolve conflicts right after they occur, and usually look at changes not in conflict to resolve merge conflicts. These results increase internal validity on the variables used in our empirical study. Related to developers' experience dealing with merge conflicts, survey participants pointed out four major challenges on merge conflict resolution: *lack of coordination*, *lack of tool support*, *flaws in the system architecture*, and *lack of testing suite or pipeline for CI*. Indirectly, these challenges bring explanations of why some relatively simple merge conflicts (in terms of the subject variables in our empirical study) took a while to be resolved.

Aiming at deeper explanations for our results, we analysed specific factors individually and triangulated our data with manual analyses. These manual analyses include a comparison of the 100 shortest and 100 longest conflicting scenarios and observations of how developers resolved the merge conflicts, for instance. As a major result of these analyses, we found: (i) a dependency among conflicting and non-conflicting code, which normally increases the time necessary for developers resolving merge conflicts, (ii) reasons for why it is better to commit many small chunks of code instead of few large chunks, and (iii) why characteristics of merge scenarios influence the merge conflict resolution time more than characteristics of the merge conflicts themselves.

Overall, we make the following contributions:

- We propose a taxonomy of challenges on merge conflict resolution acquired by quantitative empirical data and by surveying developers of subject projects.
- We provide evidence that *#Chunks*, *#LoC*, *#ConfChunks*, *#Devs*, and *CodeComplexity* have an effect on merge conflict resolution time.
- We found that variables indirectly related to merge conflicts (e.g., number of chunks changed in a merge scenario) have a higher influence on the merge conflict resolution time than variables directly related to merge conflicts (e.g., the number of conflicting chunks).
- We found a positive correlation between *#LoC*, *#Devs*, *#Files*, *#ConfChunks*, *#ConfFiles*, and *#ConfLoC* and merge conflict resolution time and a negative correlation between *#Chunks* and *CodeComplexity* and the merge conflict resolution time.
- By a manual analysis of the 100 shortest and the 100 longest merge scenarios, we observed that file extension and dependencies among conflicting and non-conflicting code make developers take longer to resolve merge conflicts.
- We found that, in more than 50 % of the cases, developers have changed the files that are in conflict before they resolve the merge conflicts.
- We found that despite 30 out of the 66 projects having at least one conflicting merge scenario, due to formatting changes, formatting changes result in merge conflicts in only 2.42 % of the merge scenarios.

6.2 Mining Study

In this section, we present our empirical study setup and results. Our overall goal in this study is to *investigate which factors make merge conflicts resolution longer in the practice of collaborative software development*.

6.2.1 Study Setting

In what follows, we describe the experiment variables, subject selection process, and statistics we use.

6.2.1.1 *Experiment Variables*

To quantify the time of resolving merge conflicts, that is, our *dependent variable*, we measure the time difference between the merge commit and the latest commit of the merged branches (parent commits). To learn which factors may influence the time for resolving merge conflicts, we defined a set of ten *independent variables*, inspired by the literature [180, 196, 214] and described in Table 6.1. Note that all variables have been suggested by developers for the investigation of merge conflicts, although some of them were suggested for other phases of the merge conflict life-cycle [180, 196, 214]. Our survey confirms that developers use these variables for estimating the time/difficulty of merge conflict resolution (Section 6.3).

We explain below the rationale of choosing each variable. For a better overview, we classify the variables into three groups: *time*, *variables directly related to merge conflicts*, and *variables indirectly related to merge conflicts*.

Time. With *#SecondsToMerge*, we aim at capturing how much time (in seconds) passed for resolving a merge conflict (Table 6.1). Note that this is our operationalization. In our eyes, it represents the sweet spot of accuracy that is achievable in a post-hoc analysis. Considering that this variable is central to our study and might not be very precise, we surveyed 140 developers (Section 6.3) and provided a broader discussion about this variable in Sections 6.4 and 6.5.

Variables directly related to merge conflicts. This group contains the majority of variables investigated in this study. As our goal is to analyse factors that influence merge conflict resolution time, it is reasonable to choose measures that directly quantify the size, complexity, and the knowledge of the integrator (i.e., who solved the merge conflict) on the conflicting code: *CodeComplexity* (via Lizard[1]), *#ConfChunks*, *#ConfFiles*, *#ConfLoC*, *%FormattingChanges*, and *%IntegratorKnowledge*. The last two variables are important because they can control for other variables. For instance, we may observe very large conflicting chunks, although these chunks occurred largely because of formatting changes (i.e., adding/removing line breakers and changing code spacing). Hence, these conflicting chunks would be easier and faster to resolve than other conflicting chunks (see Section 6.4).

Variables indirectly related to merge conflicts. As merge conflict resolution may depend on code changes not involved in a conflict, we considered also properties not directly related

1 https://pypi.org/project/lizard/

Table 6.1: Variables of our Study, Along with Their Descriptions

Variable	Description
Dependent variable	
#SecondsToMerge	The shortest time difference between the parent commits and the merge commit
Independent variables, directly related to merge conflicts	
CodeComplexity	Sum of the cyclomatic complexity of conflicting chunks
#ConfChunks	Number of conflicting chunks
#ConfFiles	Number of conflicting files
#ConfLoC	Number of conflicting lines of code changed
%FormattingChanges	Percentage of formatting changes of conflicting chunks among all chunks
%IntegratorKnowledge	Percentage of the sum of conflicting chunks in files that the integrator had committed before the merge commit among all chunks
Independent variables, indirectly related to merge conflicts	
#Chunks	Number of chunks
#Devs	Number of developers changing code
#Files	Number of files
#LoC	Number of lines of code changed

to merge conflicts (i.e., all the code changes in the merge scenario): *#Chunks*, *#Devs*, *#Files*, and *#LoC*.

Example. In the merge scenario of Figure 6.1, four developers (*#Devs*), namely DevA, DevB, DevC, and DevD, changed ten lines of code (*#LoC*), of which eight are in conflict (*#ConfLoC*). These lines of code changed four chunks (*#Chunks*), of which three are in conflict (*#ConfChunks*). These chunks belong to two files (*#Files*) and, as there are conflicts in both files, the number of conflicting files (*#ConfFiles*) is two. As DevD is the developer who solved the merge conflict (i.e., the integrator), and she committed to both files before the merge commit, we reason that DevD had knowledge of all changed files (*%IntegratorKnowledge*). Regarding the time to resolve the merge conflict (*#SecondsToMerge*), we compute the time difference between the source branch's parent commit (hash: 20BBDF7) and the merge commit (hash: C2ECB2C). That is, *#SecondsToMerge* is equal to T6 minus T5 in seconds. As Figure 6.1 is only a simple and abstract example, it is not possible (or meaningful) to calculate *%FormattingChanges* and *CodeComplexity*.

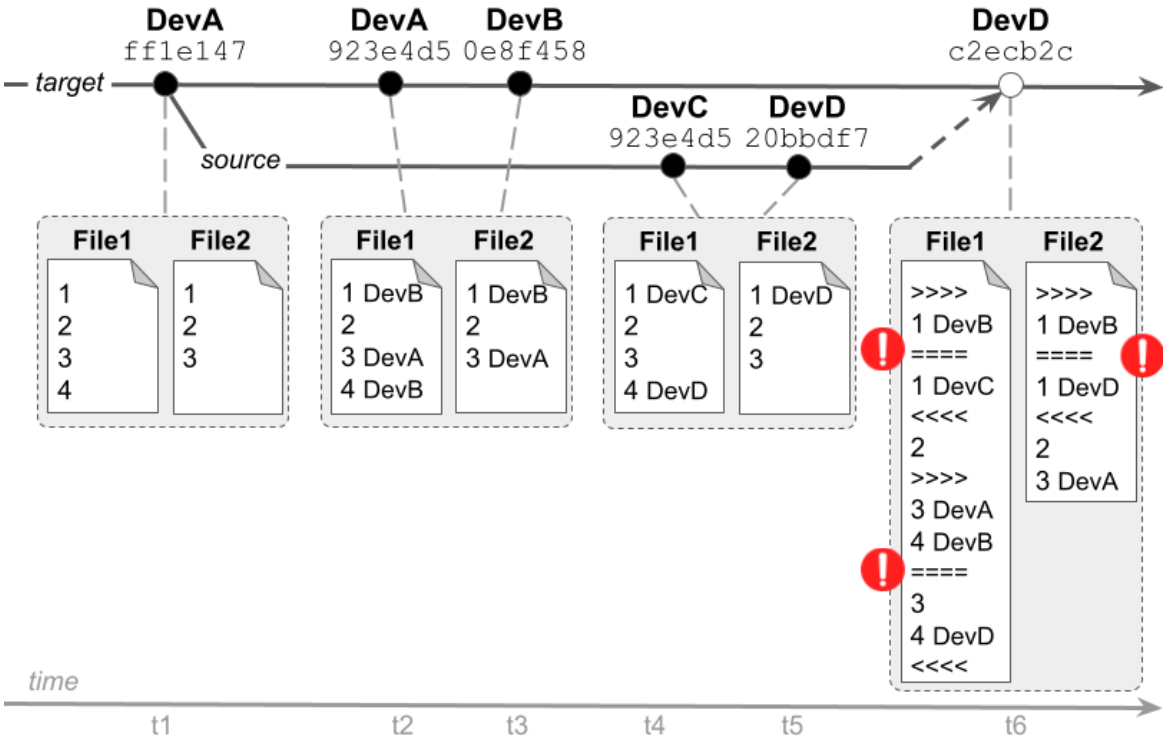

Figure 6.1: Example for a Merge Scenario with Conflicts. Four Developers Contributed to Two Files on the Branches *target* and *source*, Resulting in Three Merge Conflicts

6.2.1.2 *Subject Projects*

We selected the corpus of subject projects as follows. First, we retrieved the 100 most popular projects on GitHub, as determined by the number of stars [40]. Then, we applied the following four filters which we created based on the work of Kalliamvakou et al. [162]: (i) keep only programming projects (i.e., projects that have a programming language classified as the main file extension), (ii) keep only active projects (i.e., at least two commits per month in the last six months), (iii) keep only projects in which we were able to reconstruct more than 50% of the merge scenarios (see Section A.3.1), and (iv) keep only projects with merge conflicts.

In Figure 6.2, we show the number of projects after each filter. These filtering steps aim at selecting active projects in terms of code contributions with an active community and at increasing internal validity. The first filter captures only software development projects, excluding projects that are, for example, repositories of books and interview tips. The third filter excludes projects such as kubernetes[2] and moby[3] because these projects do not mostly use the three-way merge [115] which could bias our analyses. Details on how we rebuild merge scenarios are in Section A.3.1.

We restricted our selection to GitHub because it is one of the most popular platforms to host repositories and it has been investigated and used in prior work [75, 116, 266, 281,

2 https://github.com/kubernetes/kubernetes
3 https://github.com/moby/moby

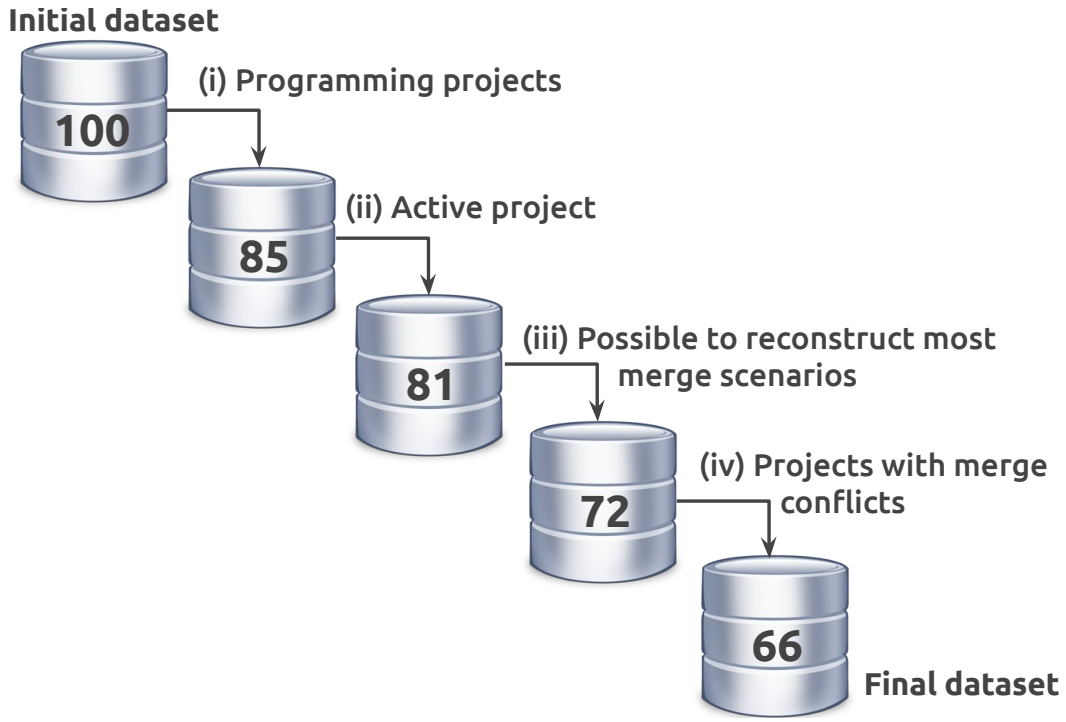

Figure 6.2: Number of Subject Projects after each Filter

296]. We limited our analysis to Git repositories because it simplifies the identification of merge scenarios in retrospect.

After applying all filters, we obtained 66 projects developed in 12 programming languages (e.g., JavaScript, Java, C++, and Python), containing 81 005 merge scenarios that involve more than 2 million files changed, 10.8 million chunks, and 2 608 conflicting merge scenarios.

6.2.1.3 *Statistical Analysis*

The statistical analysis of our study is threefold. First, we perform a ***correlation analysis*** of all covariables, using the Spearman rank-based correlation, which is invariant for linear transformations of covariates. This analysis is simple and useful to understand the relation among our covariables, to build a consistent regression model, and to support the discussions in Section 6.4. Spearman rank-based correlation is -1 in the case of a perfect negative correlation, +1 in the case of a perfect positive correlation, and values around 0 imply that there is no correlation between the investigated variables [152]. Note that the purpose of this first correlation analysis is data exploration, we are not going to draw conclusions on such correlation coefficients. So, a correction on the p-values to account for the multiple comparisons is not necessary. In any event, when looking at the correlation between dependent and independent variables, we performed a Bonferroni p-value adjustment for each independent variable (i.e., p-value < 0.005). Still in the first analysis, to better understand the correlations among variables and reduce the number of dimensions (i.e., variables) in the regression model, we perform a Principal Component Analysis (PCA). PCA is important because, by removing correlated variables from the regression model, we avoid common pitfalls on modelling data [287].

With the insights of the first analysis, we build a ***multiple linear regression model*** in our second analysis for understanding the relation between our dependent and our independent variables. Multiple linear regression models are relatively simple yet powerful to achieve our goal and significantly easier to explain and interpret than other models such as neural networks and deep learning models. Coefficients in the regression model are interpreted similarly to the Spearman rank-based correlation coefficients from the first analysis. The multiple linear regression model of a y dependent variable on the $x_{1...n}$ independent variables is represented by

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n + \varepsilon_0. \tag{6.1}$$

The β coefficients measure the association between the independent variables and the dependent variable. β_j can be interpreted as the average effect on y of one unit increase in x_j, holding all other independent variables fixed [151]. To define the used model, we compare the variance of different models, such as the model with all independent variables and the model with a simplified number of independent variables and choose the model with greater variance, as suggested by previous work [100, 113, 151] (details in Section 6.2.2.3).

Finally, we performed an ***effect-size analysis*** in the context of an analysis of variance. For short, the effect-size analysis is necessary because independent variables may change differently and, even with the results of the regression model, we are not able to classify the most influencing factors. The Cohens's f^2 for sequential multiple regression is:

$$f^2 = \frac{R^2_{AB} - R^2_A}{1 - R^2_{AB}} \tag{6.2}$$

where B is the variable of interest, A is the set of all other variables, R^2_{AB} is the proportion of variance accounted for A and B together, and R^2_A is the proportion of variance accounted only for A. By default, Cohen's f^2 effect size values from 0.02 to 0.15 are small, from 0.15 to 0.35 are medium and greater than 0.35 are termed large values [64]. In addition to Cohens's f^2, we also report the η^2 and ω^2 values to increase the confidence of our effect-size analysis. η^2 is the proportion of the total variability in the dependent variable that is accounted for by the variation in the independent variable [181]. It is the ratio of the sum of squares for each group level to the total sum of squares. It can be interpreted as the percentage of variance accounted for by a variable. ω^2 is widely viewed as a less biased alternative to η^2, especially when sample sizes are small [181]. η^2 and ω^2 effect size values smaller than 0.01 are very small, from 0.01 to 0.06 are small, from 0.06 to 0.14 are medium, and greater than 0.14 are termed large values [105].

6.2.2 Results

In this section, we present the results of our empirical study. First, we present an analysis in the distribution of our dependent variable. Then, the rest of the section is structured according to the three analyses presented at the end of Section 6.2.1.3.

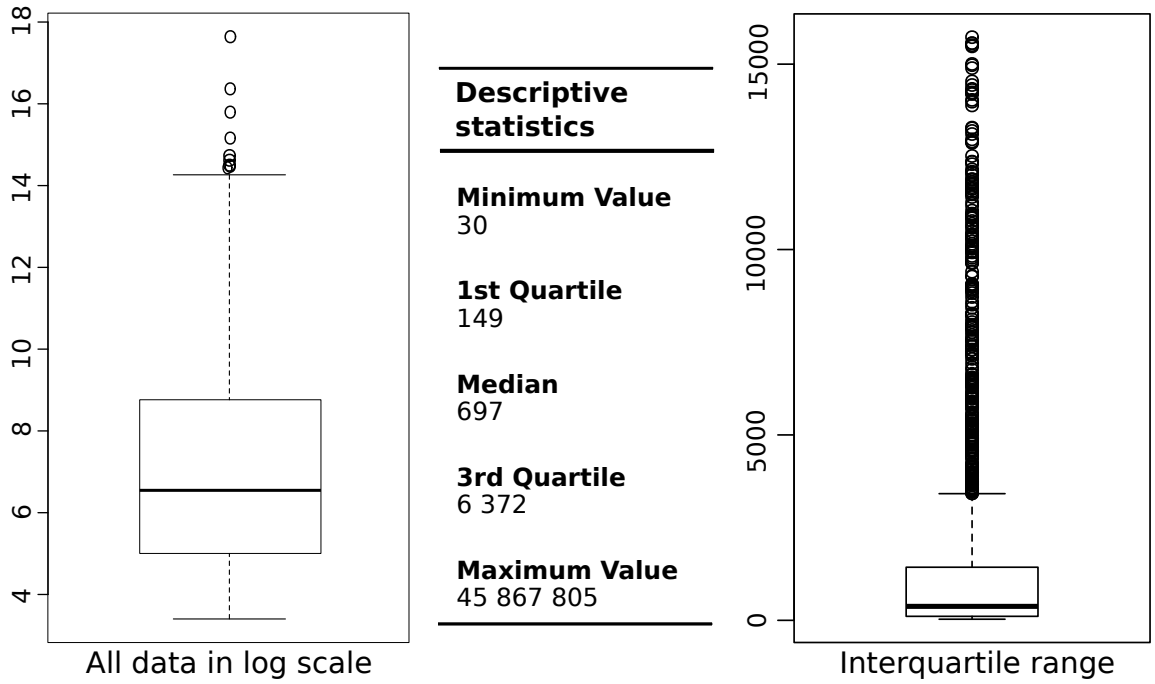

Figure 6.3: Dependent Variable Distribution in Seconds

6.2.2.1 *Dependent Variable Distribution*

In Figure 6.3, we show two boxplots with descriptive statistics of our dependent variable (*#SecondsToMerge*). The statistics include the minimum value, first quartile, median, third quartile, and maximum value in seconds. As seen, the fastest merge conflict resolution took 30 seconds and the maximum took 45 857 805 seconds (around 530 days). This is definitely an outlier scenario that has been forgotten by developers for one and half years and integrated later. Median is a reasonable measure to analyse since outlier scenarios would distort the mean. Looking at the median, we see that half of the merge conflicts took up to 11 minutes to be resolved (697 seconds). Looking at the first and third quartile, we see that 25% of the conflicting merge scenarios took 149 seconds (≈2.5 minutes) and 75% of the conflicting merge scenarios took up to 6 372 seconds (≈1.77 hours). In the right-most boxplot, we show data in the IQR. We show this boxplot since the box-plot with all data does not give the real idea of the time to resolve merge conflicts. In the IQR the median is 377 seconds (≈6 minutes).

Comparing a recent study [43] that recorded the merge conflict resolution time from seven developers resolving 10 merge conflicts, they found that these developers took from 40 to 2 190 seconds (≈36.5 minutes). Even though, the number of conflicting scenarios in this previous study is quite limited and our variable might not be precise on measuring the time developers really spent resolving merge conflicts, which makes difficult to compare these variables, it is worth to mention that *#SecondsToMerge* are not far from their records. In addition to this comparison, we provide follow-up analyses and discussions to increase construct validity of the choice of our dependent variable (see Sections 6.3.2, 6.4, and 6.5).

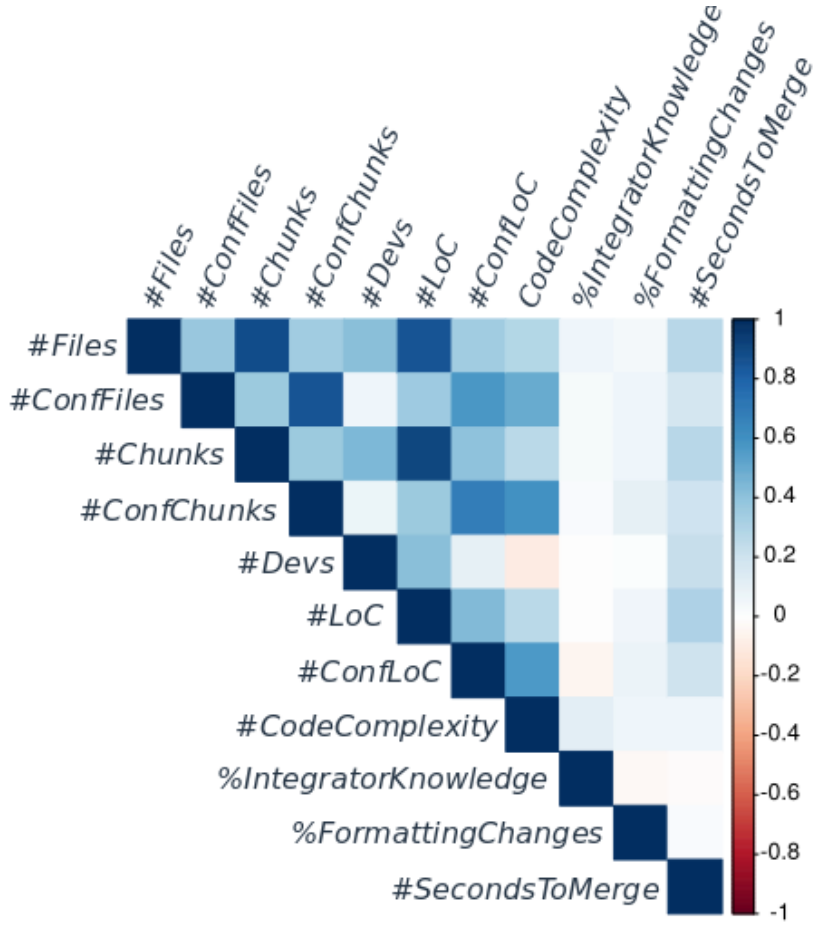

Figure 6.4: Correlation Matrix for all Pairs of Variables

6.2.2.2 *Correlation Analysis*

In Figure 6.4, we present a correlation matrix among all covariables of our analysis. As expected, merge scenario size measures (i.e., *#Chunks*, *#Files*, and *#LoC*) have a high correlation among themselves (above 0.8). Merge conflict size measures (i.e., *#ConfChunks*, *#ConfFiles*, and *#ConfLoC*) show a moderate to high correlation among themselves (above 0.5). The other merge conflict related measures do not have a strong correlation. For instance, *%IntegratorKnowledge* and *%FormattingChanges* have a correlation coefficient smaller than 0.1 with most of the merge conflict related measures. The only exception is *%IntegratorKnowledge* with a positive correlation coefficient with *CodeComplexity* (0.115).

Next, we pay more attention to the correlation between the dependent variable and each independent variable. For short, the correlation is significant with a confidence interval of 99.5% for all independent variables, except *%IntegratorKnowledge* and *%FormattingChanges*. In Table 6.2, we present the correlation coefficients for the significant ones. Note that the correlation coefficients of these eight variables are rather small, but significant. Also note that the top three variables with highest correlation coefficient are variables that measure the merge scenario size (*#LoC*, *#Chunks*, and *#Files*) and not merge conflicts.

Table 6.2: Correlation Coefficients for Independent Variables with the Dependent Variable

Measure	Coefficient	Measure	Coefficient
#LoC	0.308	*#ConfLoC*	0.206
#Chunks	0.279	*#ConfChunks*	0.194
#Files	0.270	*#ConfFiles*	0.180
#Devs	0.228	*CodeComplexity*	0.061

Aiming at reducing the number of dimensions and grouping similar variables, we performed a PCA. It reduces the number of dimensions to the first two principal components that retain a maximum share of common variance, which simplifies the discussion of the correlation structure. In Figure 6.5, we show the two-dimensional output from the principal component analysis, which covers *56.1 %* (*38.9 %* + *17.2 %*) of the total variance of our data. The arrows represent the weights of each variable in the respective principal component and their colours represent the square cosine (cos2). The square cosine represents the share of original variation in the variable 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.

The data visualised in Figure 6.5 suggest to classify the independent variables into four groups: *merge scenario size*, *merge conflict size*, *social activity*, and *integrator's prior knowledge/type of change*. The arrows representing *#Chunks*, *#Files*, and *#LoC* point to the same direction; they represent the size of a merge scenario. Pointing to another direction, *#ConfLoC*, *#ConfFiles*, and *#ConfChunks* represent the merge conflict size. The *#Devs* point to a third direction and, hence, we call it social activity. The factors *CodeComplexity*, *%FormattingChanges*, and *%IntegratorKnowledge* compose the fourth group, which we named integrator's prior knowledge/type of change.

To summarise, we see that (i) by clustering our independent variables into four groups, we do not need all of them in our regression model which increases internal validity avoiding overfitting and multicollinearity, as we explain in Section 6.2.2.3; (ii) measures from the *integrator's prior knowledge/type of change* group are almost orthogonal to the merge conflict resolution time which means a small share of variance among these variables with the merge conflict resolution time; and, (iii) measures from the *merge scenario size* and *social activity* groups have a stronger relation with the merge conflict resolution time than measures from the groups *merge conflict size* and *integrator's prior knowledge/type of change*.

6.2.2.3 *Multiple Regression Model Analysis*

All independent variables presented in Section 6.2.1.1 may be in our model because there is a belief that these variables influence the merge conflict resolution [180, 196, 214], which is confirmed in with our survey (Section 6.3). However, including all independent variables would increase overfitting (i.e., a model that contains more parameters that can be justified by the data) and multicollinearity (i.e., high correlation between two or more independent variables) in our model. To minimise overfitting and multicollinearity in our model, we

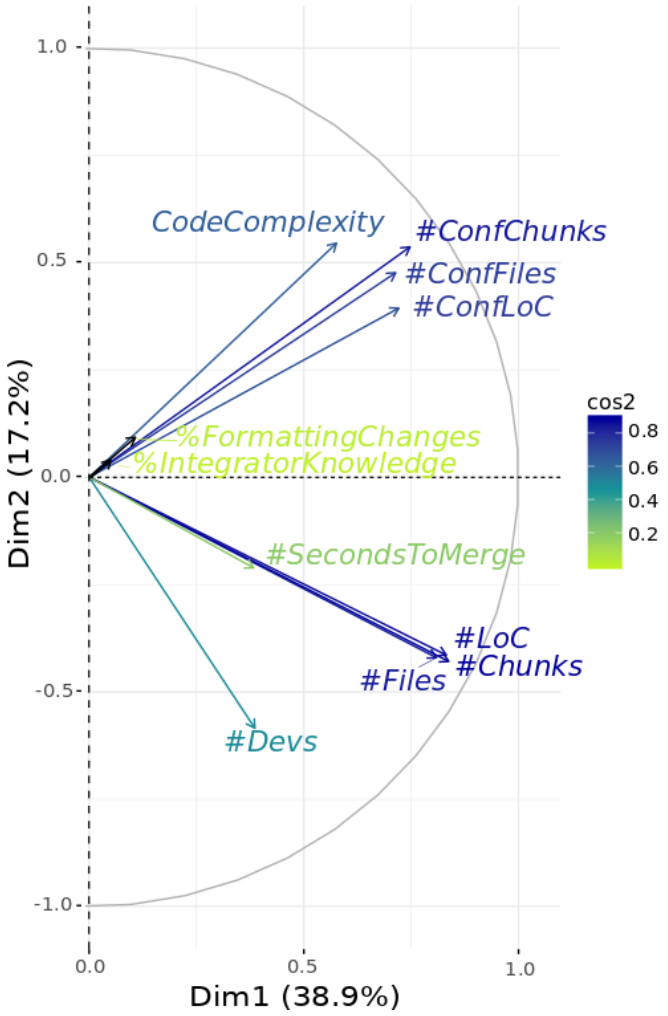

Figure 6.5: Principal Component Analysis of our Variables

perform a transparent process, as suggested by different researchers [100, 113, 151]. Of course, we could use a variable of each group of the PCA. Nonetheless, we do not know which variables better fit the multiple regression model and we may ignore hidden relationships. In our case, this process consists of four steps: (i) create a preliminary model and learn with this model; (ii) create further models with observations made from the first preliminary model; (iii) compare the variance of the created models; and, (iv) choose the model that represents the investigated relationship most accurately. We present the details for these models in Table 6.3.

The *Full Model* column of Table 6.3 presents the correlation coefficients for the 10 independent variables that compose our preliminary model. Looking at the correlation coefficients of this preliminary model, we can make three observations: (i) the coefficient of six independent variables (i.e., *#LoC*, *#ConfChunks*, *#Devs*, *CodeComplexity*, *#Chunks*, and *#ConfLoC*) are significant with a confidence interval of 95 %; (ii) the four independent variables with greatest correlation coefficients belong to distinct groups of our PCA (see Section 6.2.2.2); and (iii) from the two remaining variable with coefficient significant (i.e., *#Chunks* and *#ConfLoC*), only *#Chunks* provides a different view from the variable that belongs to the same group.

Table 6.3: Correlation Coefficients for Independent Variables in the Multiple Regression Model Analysis

Measure	Full Model	Simplest Model	Balanced Model
#LoC	0.2538^{***}	0.2268^{***}	0.2931^{***}
#ConfChunks	0.1239^{**}	0.1752^{***}	0.1782^{***}
#Devs	0.1221^{***}	0.1171^{***}	0.1251^{***}
CodeComplexity	-0.1067^{***}	-0.0870^{***}	-0.0841^{**}
#Chunks	-0.1013^{*}	-	-0.0783^{*}
#ConfLoC	0.0799^{**}	-	-
#Files	0.0525	-	-
#ConfFiles	0.0146	-	-
%FormattingChanges	−0.0048	-	-
%IntegratorKnowledge	−0.0041	-	-

$^{***}\ p-value < 0.001$, $^{**}\ p-value < 0.01$, $^{*}\ p-value < 0.05$

In other words, while *#LoC* has a positive correlation coefficient in the *Full Model*, *#Chunks* has a negative correlation coefficient. On the other hand, *#ConfLoC* and *#ConfChunks* have a positive correlation coefficient. Therefore, adding both does not provide a different view to our model. Hence, choosing only *#ConfChunks* which has a greater correlation coefficient in the *Full Model* is more promising to avoid overfitting and multicollinearity.

Taking these observations into account, we build other two regression models: *simplest model* has only the four variables with greatest correlation coefficients (see observation ii) and *balanced model* has *#Chunks* and all other variables in the simplest model (see observation iii). The correlation coefficients of these models can be seen in Table 6.3. While the *simplest model* and the *balanced model* minimise overfitting and multicollinearity, the *balanced model* shows the hidden relationship among *#LoC* and *#Chunks* that we did not see in the correlation analysis. From that perspective, the *balanced model* seems to be the correct model to choose, although the analysis of variance supports a more data-oriented choice.

In the *analysis of variance*, a significant p-value (i.e., < 0.05) means that adding variables to the model, in fact, adds relevant information to the regression model. We compare the balanced model with the other two models because these comparisons allow us to find out which model better represents the investigated relationship. The p-value of the analysis of variance among the *balanced model* and the *full model* is 0.1. Therefore, adding *#ConfLoC*, *#Files*, *#ConfFiles*, *%FormattingChanges*, and *%Integratornowledge* to the balanced model do not add relevant information to it. On the other hand, the p-value of the analysis of variance among the *simplest model* and the *balanced model* is 0.004. Therefore, adding *#Chunks* to the simplest model add relevant information to it. In conclusion, the *balanced model* fits better on the investigated relationship than both the simplest and the full model.

Once we have chosen the model that best represents the relationship among our dependent and independent variables, we discuss its correlation coefficients as follows. Column *Balanced*

Model of Table 6.3 presents the coefficients obtained from the chosen regression model. We can see that *#LoC*, *#ConfChunks*, and *#Devs* show a positive correlation with *#SecondsToMerge*. Hence, if these variables increase, the time to resolve merge conflicts also increases. On the other hand, *#Chunks* and *CodeComplexity* have a negative correlation coefficient with *#SecondsToMerge*. Increasing these two variables is associated with less time to resolve merge conflicts. Note that *#LoC*, which is the variable with the highest correlation in the correlation matrix (Figure 6.4), remains with the highest correlation in the multiple regression model analysis.

Our regression model has a significant explanation value at any significance level (p-value $< 1/10^{16}$). It has R^2 and adjusted R^2 equal to 0.122 and 0.12. Following Falk and Miller [101] classification, $R^2 < 0.1$ is negligible and $R^2 \geq 0.1$ is adequate. Therefore, the R^2 of our model is adequate. Our model has a residual standard error of 706 on 2602 degrees of freedom. It means that on average, our estimate is 706 above or below the observed value and it considers 2602 out of the 2608 conflicting merge scenarios investigated. Note that the interpretation of our regression model needs to be associated with the *ceteris paribus* concept. In other words, a correct interpretation of the model has to account that changing the value of one independent variable, all other variables' values have to be equal.

A simple way to interpret our regression model for *#LoC* and *#Chunks* is described as follows. Adding 1000 LoC in the merge scenario is associated with an increase in time by approximately 293 seconds or 5 minutes to solve the merge conflicts, for a fixed amount of *#ConfChunks*, *#Chunks*, *#Devs*, and *CodeComplexity*, on average. Regarding *#Chunks*, for a fixed number of *#LoC*, *#ConfChunks*, *#Devs*, and *CodeComplexity*, adding 1000 chunks in the merge scenario leads to a decrease in time by approximately 78 seconds or 1.2 minute. At first sight, the negative correlation coefficients for *CodeComplexity* and *#Chunks* in the regression model seem counter-intuitive, but, in Section 6.4, we discuss why it is not.

6.2.2.4 *Effect-Size Analysis*

Finally, to answer our RQ and be able to quantify the influence of the independent variables on merge conflict resolution time, we performed an effect-size analysis. As described in Section 6.2.1.3, we chose Cohen's f^2 effect size since it is adequate when using multiple regression models [64]. In Table 6.4, we present the results of our effect-size analysis ordered by the highest to the lowest effect-size. We can see that the effect-size of *#Chunks*, *#Devs*, *#LoC*, *#ConfChunks*, and *CodeComplexity* are 0.298, 0.135, 0.129, 0.105, and 0.064, respectively. Following Cohen's classification, *#Chunks* has a medium effect-size on merge conflict resolution time, while the other four variables have a small effect-size. Interestingly, despite *#Chunks* is the variable in the chosen regression model with weak correlation, it has the highest effect-size.

Similar values are also found for η^2 and ω^2. *#Chunks* with a medium effect-size and the other variables with a small or very small effect-size. All variables in our effect-size analysis has a p-value $\leq$ than 0.001 and our analysis has a confidence interval level of 90%. Therefore, our analysis covers 90% of the subject conflicting merge scenarios.

Surprisingly, the three variables with highest effect-size (*#Chunks*, *#Devs*, and *#LoC*) are not directly related to merge conflicts. By combining the results from the regression model and effect-size analysis, we can see that the number of chunks shows a negative correlation,

Table 6.4: Effect-size Analysis

Measure	f^2	f^2 GV	η^2	η^2 GV	ω^2	ω^2 GV
#Chunks	0.298		0.078		0.078	
#Devs	0.135		0.016		0.017	
#LoC	0.129		0.015		0.014	
#ConfChunks	0.105		0.010		0.011	
CodeComplexity	0.064		0.004		0.003	

GV stands for graphical visualisation of the target measure. In the case of Cohen's f^2, it is divided into three groups: small, medium, and high effect-size. In the case of η^2 and ω^2, it has an additional group very small when compared with Cohen's f^2.

whereas the other two variables indirectly related with merge conflicts (*#LoC* and *#Devs*) show a positive one. Hence, we conclude 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 small effect-size.

Even though the correlation coefficients of the multivariable regression model are low, we obtained medium and small effect-sizes for the targeted independent variables. It highlights the importance of an effect-size analysis on measuring the impact of independent variables on the dependent variable. It may be an incentive for researchers to perform similar studies that do not stop on the correlation analysis.

Note that we present results only for the variables that compose the *balanced model* since it is the model that best represents the investigated relationship. As mentioned, further variables would only add noise to our analysis (see Sections 6.2.1.3 and 6.2.2.3) because they are highly correlated with variables that compose the regression model or do not correlate with *#SecondsToMerge*. Anyway, some of the hidden variables have a similar effect-size.

In Figure 6.6, we show an overview of our results considering all independent variables. Full and dashed lines represent explicit and implicit relationships investigated in the effect-size analysis, respectively. As we can see, *#ConfLoC* and *#ConfFiles* provide a similar effect-size to *#ConfChunks*. Hence, we added a dashed line from these variables to *#SecondsToMerge*. Similarly, *#Files* is correlated with *#LoC*. Hence, they have a similar effect-size with *#SecondsToMerge*. Since *%FormattingChanges* and *%IntegratorKnowledge* do not have a significant correlation with *#SecondsToMerge*, they also do not present an effect-size on *#SecondsToMerge*. For that reason, there is no line among them and *#SecondsToMerge*. We postpone a discussion of independent variables and their relationships to Section 6.4.

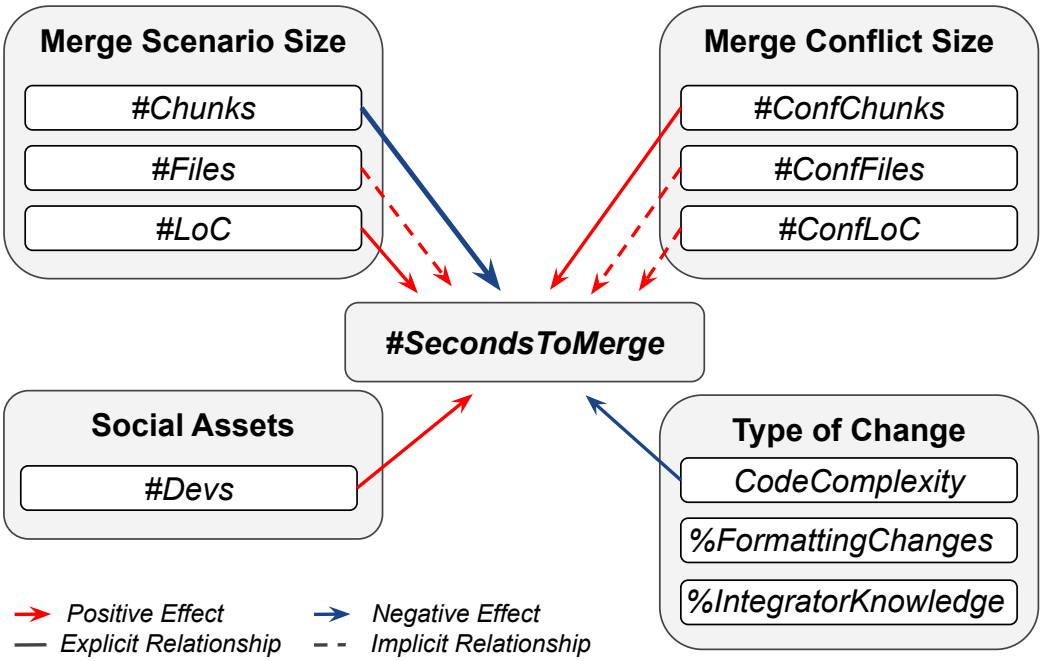

Figure 6.6: Overview on our Effect-size Results

Results Summary: Our *correlation analysis* indicates that some variables are strongly correlated and, for that reason, we classified them into four groups which supported the construction of our regression model. Our *multiple regression model analysis* shows that *#LoC*, *#ConfChunks*, *#Devs*, *CodeComplexity*, and *#Chunks* are correlated with *#SecondsToMerge*. *#Chunks* and *CodeComplexity* have a negative influence while the others show a positive influence. Our *effect-size analysis* reveals that *#Chunks* has a medium effect-size on the merge conflict resolution time while *#Devs*, *#LoC*, *#ConfChunks*, and *CodeComplexity* have a small effect-size on the merge conflict resolution time.

6.3 Survey

In this section, we report on a survey of software developers from our subject projects (Section 6.2.1.2). The goal of the survey was to cross-validate our results and reduce threats to the validity of our quantitative findings. Next, we present the setting (Section 6.3.1) and results (Section 6.3.2) of our survey.

6.3.1 Study Setting

We created a seven-question survey, of which the first and last questions are open-ended. The other five questions are close-ended questions (5-point Likert-type scales). Aiming at grouping and systematically generating a theory from the answers of our open-ended questions, we used two Grounded Theory techniques [279, 282]: *open coding* and *axial coding*.

We followed four steps of which two authors performed the first three steps separately. First, we extracted data from open-ended questions. Second, we segmented answers into meaningful expressions and described them in a short sequence of words (the open coding technique). Third, we relate short sequences of words to each other, combining inductive and deductive thinking (the axial coding technique). Fourth, all authors combined and discussed the outcome data repeating the second and third steps until we had a concise answer for a given question.

The survey is divided into three parts. First, we are interested in understanding factors that make the merge conflict longer/harder to resolve (Q_1). Then, with Q_2 to Q_6, we address potential threats to validity asking developers for confirmation about results in the empirical study (Section 6.2). In the third part, we are interested in the experience survey participants had when dealing with merge conflicts (Q_7).

We recruited participants from subject projects that faced merge conflicts (obtained as described in Section 6.2.1.2) by directly contacting them via e-mail. We followed a learn-and-improve approach, in which we adapt questions based on the participants' feedback. For instance, in the first version of the survey, we asked about participants' experience and team size. However, a few developers replied to our email or reported in their survey response that, despite their interest in the topic, the survey was too long containing unnecessary questions and, for that reason, they or their colleagues did not answer the survey. Aiming at getting more informative answers, we decided to modify/shorten the survey.

The survey was available for about 6 months. We received 140 responses (response rate around 2%). Individual parts of the survey had varying response rates since open-ended questions were optional. The full set of survey versions are available at our Appendix D.

6.3.2 Results

We divide the discussion of the results of our survey in three sections according to the parts as previously described.

6.3.2.1 *Participants' Perception of Factors that Make Merge Conflict Longer/Harder to Resolve*

To understand factors that make the merge conflict resolution longer/harder, we asked the survey participants Q_1: *How do you estimate how hard/time-consuming a merge conflict is to be resolved?* We got 89 answers for this question. In Table 6.5, we present 25 measures pointed out by survey participants sorted by the number of suggestions. Measures used in the empirical study (Section 6.2) are highlighted with the acronym in parenthesis. As seen, the top three suggested measures are: the number of conflicting lines of code (*#ConfLOC*), the number of conflicting chunks (*#ConfChunks*), and the number of lines of code changed (*#LOC*). Notably, we used these three measures in our empirical study (Section 6.2). Interestingly, five participants mentioned they do not believe that it is possible to measure how hard/time-consuming is the resolution of merge conflicts.

A few participants mentioned that the difficulty/time of resolving merge conflicts is somehow related to the files' characteristics. For instance, nine participants mentioned

Table 6.5: Measures to Estimate the Difficulty/time to Resolve Merge Conflicts

Measure	**#Sug.**
Number of conflicting lines of code (#*ConfLOC*)	19
Number of conflicting chunks (#*ConfChunks*)	16
Number of lines of code changed (#*LOC*)	13
Number of files changed (#*Files*)	9
Time between the base commit and the merge commit	5
Developer experience responsible for conflicting changes (∼%*IntegratorKnowledge*)	4
Number of conflicting files (#*ConfFiles*)	4
Frequency target file changed	4
Semantically diff between conflicting code	4
Number of active developers (#*Devs*)	3
Number of commits with conflicts	3
Developer knowledge on the project (∼%*IntegratorKnowledge*)	3
Number of callers and callees functions in the conflicting code	3
Conflicts location	3
Number of chunks (#*Chunks*)	2
Number of commits	2
Number of conflicting lines per file in conflict	1
Number of commits affecting a file	1
Number of whitespace changes (∼%*FormattingChanges*)	1
Code complexity of conflicting code (*CodeCompexity*)	1
Number of conflicts per file	1
Average size of conflicting chunks	1
Ratio number of chunks by the number of conflicting chunks	1
Number of conflicts multiplied by the average of the number of conflicting lines of code	1
Character diff	1

#Sug. stands for the number of participants suggested a target measure.

Table 6.6: Responses on 5-point Likert-type Scale Indicating the Agreement with Questions (1 Means Hardly Ever True, 5 Means Nearly Always True)

#Q	Description	1 2 3 4 5	$\tilde{x}$	$\bar{x}$
Q_2	The more time it takes to resolve a conflict, the more difficult the conflict		3	3.4
Q_3	I merge my changes right after addressing an issue		4	3.9
Q_4	I resolve merge conflicts right after they occur		4	4.2
Q_5	I look at non-conflicting changes to resolve conflicts		3	3.4
Q_6	I change non-conflicting code to resolve merge conflicts and avoid introducing unexpected behaviour to the project		3	2.8

#Q, $\tilde{x}$, and $\bar{x}$ stand for questions, median, and mean, respectively.

that it is related to the number of files changed in the merge scenario (*#Files*), other four participants mentioned that it is related to the number of conflicting files (*#ConfFiles*), and another mentioned that it is related to the frequency files are changed, respectively. Further three participants mentioned that the difficulty/time of resolving conflicts is somehow related to the conflict's location. One of them mentioned *"fixing a conflict in the view layer* (i.e., referring to the Model-View-Controller design pattern) *is simpler than resolving a conflict in the controller layer"*.

First Part Summary: As expected and in line with previous work [180, 214], the measures used in our empirical study reflect what survey participants think about merge conflict resolution. Surprisingly, measures not directly related to conflicts are among the most suggested ones.

6.3.2.2 *Cross-validating the Quantitative Results*

In the second part of our survey, we address potential threats to validity and cross-validate our quantitative results from the empirical study (Section 6.2). In Table 6.6, we present statements and answers for Q_2 to Q_6. The 5-point Likert-type scale means: 1 – hardly ever true, 2 – rarely true, 3 – sometimes true, 4 – often true, and 5 – nearly always true.

As the term "difficulty" is subjective, in Q_2, we asked whether survey participants agree with the statement *"the more time it takes to resolve a conflict, the more difficult the conflict is"*. 15% of them mentioned that it is hardly ever true or rarely true, 37.9% mentioned that it is sometimes true, and 47.1% that it is often true or nearly always true. In Q_1, a participant, who mentioned that this statement is rarely true, made an interesting comment: *"Time is not perfect because there may be lots of simple changes but it's time consuming to rectify and potentially*

error prone". Despite her mentioning that time is not perfect, she indirectly assumes that small changes may become a difficult task due to potential bug introduction. Even though most survey participants agree with time as a measure of difficulty, our study is more straightforward by searching for factors that make conflict resolution longer.

With Q_3 and Q_4, we investigate whether survey participants merge their changes right after addressing an issue and whether they resolve merge conflicts right after they occur. These questions were motivated by the fact that we are not able to detect unexpected events that happened on the merge-conflict resolution (e.g., the developer responsible for resolving the conflict had a break). Yet, 72.9% and 77.5% of the participants agree that statements of Q_3 and Q_4 are often true or nearly always true (median 4 and mean around 4). Of course, this does not automatically mean that *#SecondsToMerge* precisely measures the time a developer spent resolving merge conflicts. However, with the survey answers, we have evidence that they normally merge their changes right after addressing an issue and resolve conflicts right after they occur.

With Q_5 and Q_6, we investigate whether survey participants look at non-conflicting changes to resolve conflicts and whether they change non-conflicting code to resolve conflicts avoiding introducing a new bug. The main motivation for these two questions is a result of our quantitative study indicating that merge conflict resolution time is strongly correlated with measures not directly related to the merge conflicts (e.g., the number of chunks changed and the number of developers active in the merge scenario). Regarding Q_5, 50.7% of the survey participants mentioned they often or nearly always look at non-conflicting code, and 25.7% of the participants sometimes look at non-conflicting code. Regarding Q_6, 25.7% said that they often or nearly always change non-conflicting code and 34.3% of the participants sometimes change non-conflicting code. We expected that developers would often look at non-conflicting code, but change it only rarely. In any event, scanning all changes in the merge scenario is time-consuming and influences the merge conflict resolution. These results provide evidence that resolving merge conflicts is much more than only fixing lines in conflict (see Section 6.4).

Second Part Summary: We found evidence that developers from our subject projects usually think that, the more time it takes to resolve a conflict, the more difficult the conflict is. With our survey, we confirm our assumption that developers usually merge after addressing an issue and resolve merge conflicts right after they occur, increasing construct validity of the dependent variable of our empirical study. Finally, we found that developers often look at non-conflicting code and sometimes change non-conflicting code when fixing merge conflicts to avoid bug introduction, cross-validating a not very intuitive result from our empirical study.

6.3.2.3 *Experience of Dealing with Merge Conflicts*

In the third part of the survey, we asked participants to share experience of dealing with merge conflicts (Q_7). We got 43 responses to this part.

Challenges of merge conflict resolution. In Tables 6.7 and 6.8, we present the 4 main challenges on merge conflict resolution brought up by our survey participants. We describe these challenges next.

Table 6.7: Challenges on Merge Conflict Resolution (Part 1)

Challenge	Sub-Challenge	Solution
Lack of coordination	Lack of communication and awareness	Create communication channels for all stakeholders and channels (e.g. slack or Microsoft teams) focused on developers or specific components (e.g. backend and frontend developers)
		Fix conflicts as soon as you are aware
		Keep others aware of refactoring changes
		Use adequate tool support to avoid developers working on the same region of code (see solution for the sub-challenge *mismanaging the backlog*)
	Large commits and rare merges	Create minimal commits (i.e., small chunks)
		Pull/push changes often (i.e., merge often)
		Create tasks/pull-requests small and focused
	Monitor changes at coarse-grained level	Manage code changes at fine-grained level (e.g., at method- or at chunk-level)
	Lack of an overall workflow	Create well-defined and documented development process
		Create and document contribution rules (e.g. formatting styling)
Lack of tool support	Inappropriate development environment	Use appropriate IDEs and, if possible, developers should use the same IDE
	Inappropriate tools for showing diffs and support merge conflicts resolution	Some IDEs provide support for that. If it is not your case, use (ad-hoc) tools to support this task
	Mismanaging the backlog	Use issue trackers (e.g., GitHub or Bitbucket) and/or tools for managing work (e.g., Jira or Asana)
Flaws in the system architecture	Highly coupled code	Refactoring code to minimise coupling and increasing cohesion
		Create an architecture that follows well-known design patterns (e.g., Singleton, Decorator, and Observer)
	Technical debt introduction	Always review code changes. Especially, more experienced developers should carefully review code changes from less experienced developers

Table 6.8: Challenges on Merge Conflict Resolution (Part 2)

Challenge	Sub-Challenge	Solution
Lack of testing suite or pipeline for continuous integration	Lack of tests and their maintenance	Always create test cases for new features and integrate them with existing test cases ensuring that no unexpected behaviour was introduced
		Update test cases always such that something changes in the project related to existing test cases
	Lack of continuous integration pipeline and its maintenance	Create and maintain a pipeline for continuous integration

Lack of coordination. We found four sub-challenges that deteriorate coordination:

- Lack of communication and awareness,
- Large commits and rare merges,
- Monitoring changes at coarse-grained level, and
- Lack of an overall workflow.

A participant mentioned "*good communication might avoid most hard conflicts*". Another participant suggested that most time-consuming conflicts arise from refactoring: "*Code moving from place to place is also a very hard scenario (in part because it makes diffs harder to obtain)*". Thirteen participants reported that their strategy to avoid conflicts is simply based on small commits and repeated merging. Interestingly, 8 participants mentioned that they rebase their changes often. We discuss rebase scenarios in Section 6.4. Related to the third sub-challenge, a participant mentioned "*I manage changes at the chunk level (not as files)*". Regarding lack of overall workflow, a participant mentioned "*good development processes avoid most merge conflicts*".

Lack of tool support. We identified three sub-challenges related to tool support:

- Inappropriate development environment,
- Inappropriate tools for showing diffs and supporting merge conflicts resolution, and
- Mismanaging the backlog.

A participant stated "*Never resolve conflicts by hand. Use a tool*". Other six participants mentioned that merge conflict resolution is much easier with an appropriate IDE. One of them said: "*I use the included git merge conflict tool in IntelliJ. The 'magic wand' is a really powerful tool which can solve some merge conflicts, for example if there are 20 diffs in a file that magic wand button can usually figure out what to change, and only leave you with one or two lines which it*

can't figure out by itself". Other participants mentioned tools they use to support diffing and merge conflict resolution. The reported tools are: P4MERGE[4], FILEMERGE[5], BEYONDCOMPARE[6], OPENDIFF[7], BBEDIT[8], TORTOISE[9], GIT DIFF[10], and GITK[11]. A few of them reported the reasons for choosing a tool. For instance, a participant mentioned that she uses TORTOISE because it shows her changes and the remote changes side by side and the file for merging them below. Other participants just mentioned avoiding duplicated work (e.g., avoiding addressing the same JIRA task) and working on the same parts of the source code at the same time.

Flaws in the system architecture. We found two sub-challenges related to system architecture flaws:

- Highly coupled code and
- Technical debt introduction.

A participant mentioned that conflicting code that is highly coupled is normally harder to resolve, since this requires looking into files that have not changed in the merge scenario or have changed but have no conflict. Another participant complemented this by mentioning that non-trivial merge conflicts tend to be a symptom of architectural flaws that make it difficult to apply a given change without touching a lot of different files/systems. Regarding the introduction of technical debt, a participant suggested that the introduction of new features/code should be reviewed by more experienced developers aiming at reducing the introduction of technical debt. This case is either related to the deterioration of the system code/design or to future refactorings. As mentioned by participants, both are prone to introduce conflicts. In fact, previous studies have reported the relation of merge conflicts with code smells [5, 8] and their effects on software quality [41]. In most cases, merge conflicts deteriorate the software quality. Furthermore, researchers [67, 219] have found that sometimes developers may not have the expertise or knowledge to make the right decisions, which might degrade the quality of the merged code. This highlights the importance of proper code review by experienced developers.

Lack of testing suite or pipeline for continuous integration. We classified factors related to this challenge into:

- Lack of tests and their maintenance and
- Lack of continuous integration pipeline and its maintenance

A participant stated: "*my harder conflicts are often when integrating two different large feature branches, the tests may at most ensure that specific isolated scenarios keep working, not that the involved features interact well, until newer tests are written for that purpose*". Another participant mentioned that, when describing her process on resolving merge conflicts, she tries to merge

4 https://www.perforce.com/products/helix-core-apps/merge-diff-tool-p4merge
5 https://developer.apple.com/xcode/
6 https://www.scootersoftware.com/
7 https://developer.apple.com/xcode/
8 https://developer.apple.com/xcode/
9 https://tortoisegit.org/
10 https://git-scm.com/docs/git-difftool
11 https://git-scm.com/docs/gitk

everything as much automated as possible. If it does not parse, or does not build, or does not pass on tests, then she uses reverse engineering.

Participants' desires, needs, and alerts. Participants articulated four desires, needs, or alerts.

Improve diffs. A participant mentioned "*a semantic diff would be amazing*", another articulated the desire of a diff of each version against the common ancestor, and several participants highlighted the importance of good visualisation/interfaces in diff tools.

Keep awareness when others are refactoring. As mentioned, keeping awareness in the project is very important. There are some awareness tools proposed in the literature, for instance, CollabVS [81], Palantír [252], Cassandra [163], and FASTDash [32]. However, what called our attention is a participant expressing the interest in a tool informing when others are refactoring the source code. As mentioned, for some participants, hard merge conflicts normally occur because of refactoring changes. Related to that, Mahmood et al. [192] found that refactoring was the most frequent change, which often collided with other refactoring or feature introductions and enhancements on the other branch. Furthermore, Mahmoudi et al. [193] have found evidence that refactoring operations are involved in 22% of merge conflicts and that conflicts that involve refactoring are more complex than conflicts with no refactoring. Putting awareness and refactoring together, Shen et al. [260] have proposed IntelliMerge, a graph-based refactoring-aware merging algorithm for Java programs and Cavalcanti et al. [59] have proposed jFSTMerge the state-of-the-art semi-structured merging algorithm for Java programs.

Show a merge-conflict difficulty estimation. A participant suggested a tool to show the merge conflict difficulty. We see it as an opportunity for tool builders building tools that work either reactively (i.e., when developers pull code from other branches) or proactively (i.e., the tool checks changes from other pre-selected branches periodically). Indeed we found some studies predicting indicators for merge conflicts [82, 180, 226], however, we did not find any tool that estimate the merge-conflict difficulty.

Improve Git *conflict message and merge strategy*. We got three opinions specific to Git. A participant complained about Git's conflict report: "*often the most confusing parts are the guides informing about incoming and current changes. Literally 2 in 3 times I have to refresh my memory about those*". Another participant reported: "*sometimes the 'differ' erroneously show the conflict to be across two functions while in reality just one function was changed significantly (also happens often when merging xml files) such cases should best be approached with a differ/merger that is aware of the underlying semantic but detecting such cases and assigning them high merge difficulty would be nice*". A third participant mentioned: "*The worst problems are when* Git *doesn't detect a merge conflict because the change appears to merge cleanly, but then bugs are introduced*".

Third Part Summary: Based on the participants' reported experience, we defined four challenges on the merge conflict resolution: lack of coordination, lack of tool support, flaws in the system architecture, and lack of testing suite or pipeline for CI. Furthermore, we collected desires, needs, and alerts reported by survey participants in which we classified into: improve diffs, keep awareness when others are refactoring, show a merge conflict difficulty estimator, and improve Git's conflict message and merge strategy.

6.4 Discussion

Aiming at achieving a deep understanding of what happens when developers are resolving merge conflicts, we triangulate our analyses with a manual analysis of the 100 shortest and 100 longest merge scenarios (Section 6.4.1). Next, we discuss the outcome of variables individually (Section 6.4.2) as well as some relationships among them (Section 6.4.3). After that, we provide a comparison of our results with previous work results (Section 6.4.4), followed by a reflection of the importance of any improvement in the merge conflict life-cycle (Section 6.4.5).

6.4.1 Manual Analysis

In this section, we inspect our data deeply and manually to understand the resolution of merge conflicts and support further discussion points.

Is it possible to identify any difference between the most quickly resolved conflicts and the ones that took the longest to be resolved? To answer this question, we manually investigated the 100 shortest conflicting merge scenarios (Group 1) and the 100 longest conflicting merge scenarios (Group 2) performing three analyses. First, we check if the independent variables are statistically significant across these two groups to confirm the results presented in Section 6.2.2 and to increase the internal validity of our study. Second, we compare the file extension of the conflicting files for both groups to see if the content of files of specific extensions might influence the merge conflict resolution. Third, we look at each conflicting code and observe how developers resolved them to investigate merge conflicts resolution from a different perspective that our independent variables might have not been able to catch.

Regarding time, we see a huge difference: while the shortest conflicting merge scenarios took less than 40 seconds to be resolved, the median for longest ones is 6.62 days. In Figure 6.7, we present a comparison of the two groups for the ten independent variables investigated in the study. Except *CodeComplexity*, *%IntegratorKnowledge*, and *%FormattingChanges*, the other variables show a statistically significant difference for these two groups (Wilcoxon signed-rank test with $p\text{-}value < 0.001$). This result is similar to the results we presented in Section 6.2.2, except that there is no significant difference for *CodeComplexity*. Note that, although significant, the difference for variables measuring the merge conflict size is small. For instance, the average number of conflicting files is 1.06 and 1.75 for the shortest and longest conflicting merge scenarios, respectively. In other words, the 100 shortest and the 100 longest conflicting merge scenarios have on average around 1 and 2 files in conflict, respectively. It explains why the correlation between the merge conflict size measures and *#SecondsToMerge* is not strong, as we found in Section 6.2.2. In fact, previous studies [110, 192] found that merge conflicts are normally small. For instance, Mahmood et. al [192] found that 28 out of 40 investigated conflicts had only one line of code conflicting in the merge branches.

Regarding the second analysis, in Table 6.9, we present a comparison of the extension of conflicting files for both groups divided into five categories:

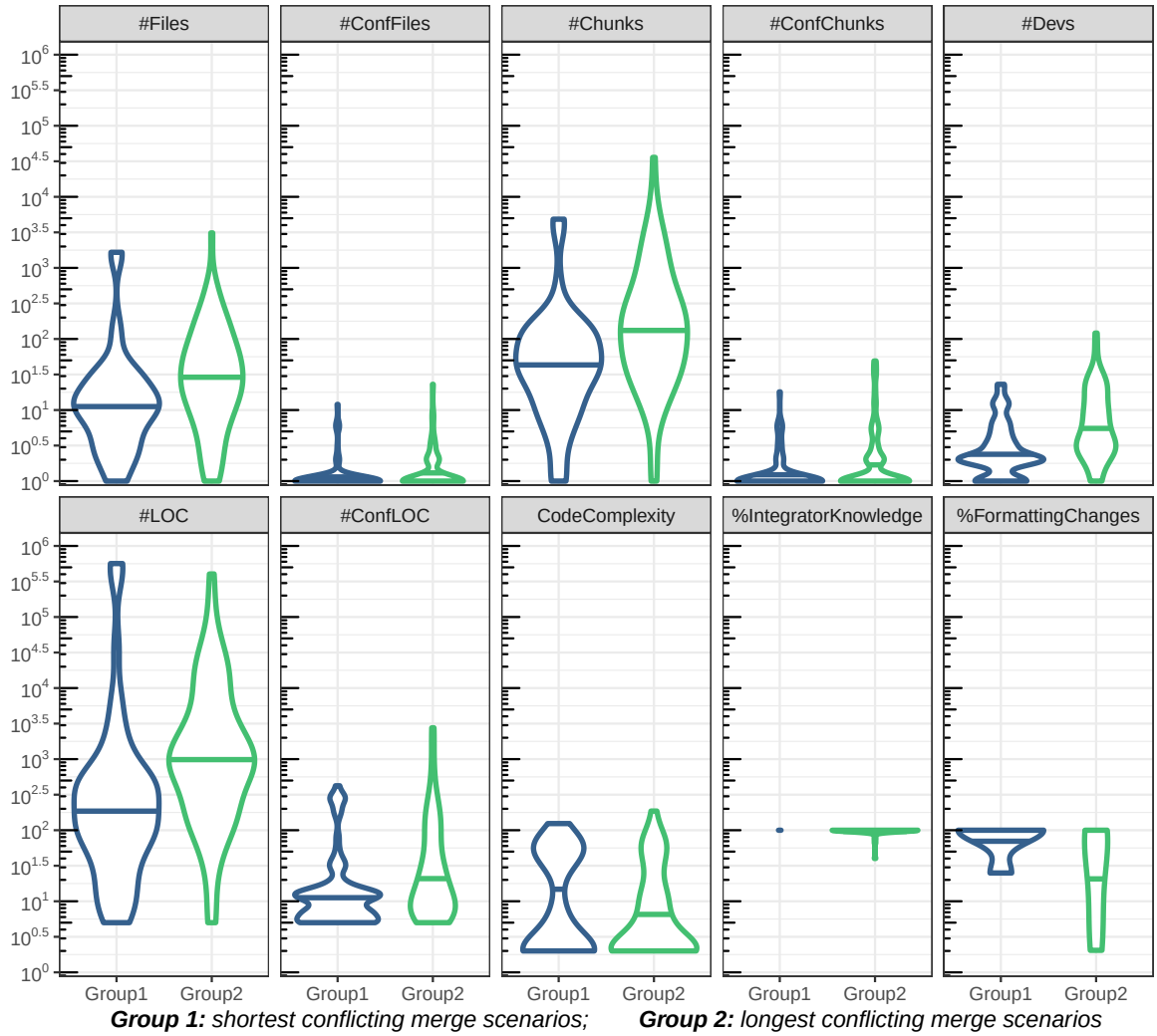

Figure 6.7: Violin Plots Distinguishing Shortest and Longest Conflicting Merge Scenarios

1. *Minified files* pass for a minification process for markup Web pages and script files, for example. Although the minification process reduces readability, it dramatically improves site speed and accessibility [127, 248]. This minification process is normally automated by tools, such as Minify[12] and JSCompress[13].

2. *Markdown files* describe the next category of files. Markdown is a lightweight markup language with plain text formatting syntax.

3. *Package manager files* are files automatically generated to manage the project. It includes, for instance, pom.xml and package.json files that are used in Java and by Node Package Manager (NPM), respectively, to identify the project as well as to handle project dependencies.

12 https://www.minifier.org/
13 https://jscompress.com/

Table 6.9: Comparing the Number (and Percentage) of Conflicting Files per Category in the Shortest (Group 1) and the Longest (Group 2) Scenarios

File Extension	Group 1	Group 2
Minified files	45 (42.45%)	16 (9.14%)
Markdown files	39 (36.79%)	21 (12.00%)
Package manager files	7 (6.61%)	14 (8.00%)
Programming language files extension	14 (13.21%)	119 (68.00%)
Other files	1 (0.94%)	5 (2.86%)

4. *Programming language files* are related to software development including programming project files, source code files, code libraries, header files, class files, and compiled objects. They are files not included in previous categories and have extensions, such as *.js*, *.rb*, *.css*, *.c*, *.h*, *.py*, and *.java*.

5. *Other files* represent files without extension (e.g., change-log files) or .gitignore files.

We argue that developers need more time to resolve merge conflicts in files with native programming language extensions because of the inherent structure including dependencies among methods, functions, procedures, class, modules, or components that may exist in the conflicting code of these files. Such dependencies may not happen in plain text files and can be automatically generated in package manager and minified files. Survey participants mentioned that number of callers and callees for added/removed functions and the conflict's location might influence the merge conflict resolution time (see Table 6.5). Furthermore, previous studies [42] [82] investigated the diffusion changes and the conflict's location. Brindescu et al. [42], found that diffusion changes (e.g., the number of files changed and the dependency among files) are important when predicting the difficulty of merge conflicts. Dias et al., [82] found that the likelihood of merge conflict occurrence significantly increases when contributions to be merged are not modular in the sense that they involve files from the same MVC slice (related model, view, and controller files). As we can see in Table 6.9, while only 13.21% of the conflicting files in Group 1 are files with native programming language extensions, 68.00% of the conflicting files in Group 2 are files with native programming language extensions. It might be an indication that source code files influence the time of resolving merge conflicts. Aiming for a more adequate answer, we look at the conflicting code as well as at how developers resolve merge conflicts in these files next.

Conflicts in Group 1. Looking at the conflicting code for Group 1, we found that:

- For all 45 minified files found with conflicts, the original file was also changed in the merge scenario. Hence, to resolve the merge conflicts developers only regenerated the minified files after the merge;

- For all package manager files found with conflicts, the number of the version of the document or of some dependencies were different. Hence, developers only chose the newest version to solve the conflicts;

- For the 14 files in native programming language extension, 5 of them had a timestamp problem in the header of the document and, hence, developers only chose the newest timestamp. For 4 files, we found code additions between existing methods in both branches. Hence, to remove the merge conflicts, developers only removed the conflict markers (e.g., "`>>>>>>>`"). For the remaining 5 files, we found a combination of formatting, with small refactorings (e.g., renaming) and fixing small issues and typos (e.g., adding an extra condition in an existing if-statement, or changing the background colour of an object, or removing a "\9" that appeared in a JavaScript file). In these cases, developers chose the changes of one branch or a combination of both;

- For 39 markdown files and the `.gitignore` file with conflicts, conflicts were basically emerged from refactoring (e.g., rewriting phrases improving grammar) and code addition (e.g., adding new phrases to give more details about some topic) without any dependency with other files. The conflicts of these files are similar to the ones we found with the files in native programming language extension. To resolve the merge conflicts, developers chose one version, or removed the conflict markers. Even without knowing the code before, with a diff checker tool, we quickly understood why the conflict arose and we would resolve conflicts similar to how developers did.

Conflicts in Group 2. Looking at the conflicting code for Group 2, the longest conflicting merge scenarios, we found that:

- Regarding the 21 markdown files, 10 of them are in merge scenarios of which there are also merge conflicts in files from the native programming language extension category. Files from the native programming language extension category might have taken longer to resolve the merge conflicts, which might have influenced the resolution of conflicts in the 10 markdown files. The remaining 11 markdown files present URLs for other files changed in the merge scenario or with external links. We do not believe that this is the only reason that developers took longer to resolve merge conflicts of these files. However, it was a pattern that we noted;

- Regarding the 16 minified files, in 13 of them the original files also changed and in 1 of these 13 files the original file also had merge conflicts. For the remaining 3 files, the original files did not change. Looking only at code changes, we could not find a plausible reason for developers taking so long to resolve merge conflicts in these files;

- Regarding the 14 package manager files, 8 of them had changes in the structure that were beyond formatting and version of dependencies. For the other 6 files, we found only formatting and versioning problems. However, 5 of them are in a merge scenario with other conflicting files. Hence, the other conflicting files might be the reason why these conflicts took so long to be resolved;

- The changes in all files in the *other files* category were simply code addition, although, all of them are in merge scenarios that have additional conflicting files. These additional files might be the reason for the long time needed to resolve their merge conflicts;

- For the 119 files with native programming language, 78 have at least another file in conflict and 117 of them have at least another file that was changed in the merge scenario. Looking at the code changes, in 79 of the 119 files, we could not understand the code changes only by looking at the file in conflict because it contained a call to a method, procedure, or import file or module also changed in the merge scenario. Therefore, in 66.39% of the cases of source code files, we found a dependency that made the merge conflict resolution longer to resolve. For the remaining 40 files without any explicit dependency, in 29 there is at least another conflicting file with further dependency in the merge scenario. Hence, at the end, 90.76% of native programming language files are related to a non-trivial solution (i.e., involving complex dependencies with libraries and other packages). Only 11 of them do not have any other file with dependency and we could not find a plausible reason for developers needing so much time to resolve these merge conflicts.

As shown in Section 6.2.2.1, merge conflicts normally take up to 11 minutes to be resolved and this result is in line with previous work [43]. In that study, researchers measured the time spent resolving merge conflicts directly by observing developers. Even looking at the 100 longest merge scenarios, we could not find plausible reasons for why the merge conflict resolution took so long for only 11 of them. With the survey (Section 6.3), participants confirmed that they normally merge their changes and resolve conflicts right after they arise. Therefore, based on our manual analysis and on the answers of survey participants, we found that extraordinary events occur in practice, but not too often to the point of biassing our results. These results altogether, strengthens that *#SecondsToMerge* is a reasonable choice as a dependent variable for a post-hoc analysis as we presented in our empirical study.

Analysis Summary. Indeed, the longest conflicting merge scenarios are larger and more complex than the shortest conflicting merge scenarios for most of the independent variables, which is in line with the results presented in Section 6.2.2 and with survey participants. It shows that our choice of *#SecondsToMerge* as a dependent variable is plausible. Our subsequent analysis shows an indication that developers need more time to resolve merge conflicts in programming language files. With the follow-up analysis, we see that the content of files with some extensions remains an indicator for the merge conflict resolution time. However, the dependency among the code in conflict with files changed in the merge scenario may be a better indicator of the merge conflict resolution time. While in the files of Group 1 we did not find a dependency among the conflicting code, we found such dependency for 90.76% of the programming language files in Group 2 (see a concrete example in Section 6.4.3).

6.4.2 Investigating Non-correlated Variables

In this section, we investigate reasons of why the *%IntegratorKnowledge* and *%FormattingChanges* are not correlated with *#SecondsToMerge*.

Do integrators have knowledge of conflicting files? Are there differences between conflicting merge scenarios that are resolved by integrators with previous knowledge on the involved files and those integrators without previous knowledge? To answer these

two questions, we distinguished between merge scenarios for which the merge-scenario integrator had previous knowledge of the involved files from those merge scenarios that the integrator did not have previous knowledge of. As an immediate result, we found that about 56 % of the conflicting merge scenarios have been integrated by a developer with some knowledge on the files in conflict. From these scenarios, integrators previously changed all files in conflict in about 94 % of such cases. Therefore, more than half of the merge scenarios are integrated by developers that already touched all files in conflict.

For further investigation, we found that the number of chunks tends to be higher, on average, when integrators have some knowledge about the conflicting files (see the violin plots in Figure 6.8) with a statistically significant difference (Wilcoxon signed-rank test, $p \approx 0.001$, $W = 891\,559$). We choose the number of chunks for this further analysis because it is the variable with the greatest impact on the merge conflict resolution time (see Section 6.2.2.4). Furthermore, as can be seen in Figure 6.8, integrators with previous knowledge seem to handle more complex merge scenarios with respect to the other measures that represent size and complexity of the merge scenario (e.g., *#Files* ($p \approx 0.001$, $W = 13\,030$), *CodeComplexity* ($p \approx 0.001$, $W = 13\,207$), and *#Devs* ($p \approx 0.002$, $W = 12\,382$)). Given that, we can assume that these developers are tasked with handling merge scenarios that spread farther across the code base and potentially inherit more semantic changes, resulting in locally restricted merge scenarios given to less knowledgeable developers.

Inspired by the response of survey participants (Section 6.3.2) and previous work [43, 214], we see two main reasons for this finding in the previous knowledge of the integrators itself:

- Integrators that have profound knowledge of the involved files of a merge scenario tend to ask more questions regarding the proposed code and are able to identify more potential semantic problems avoiding future problems [43] and
- When developers feel that their experience is not sufficient to resolve the merge conflict, they generally seek help from other developers to resolve the merge conflict [214].

Hence, an integrator without previous knowledge finds the merge conflict, but, at the end, a knowledgeable integrator that will normally solve the problem. This transition among integrators may take some time. Still, we cannot draw a conclusion that prior knowledge on conflicting files supports integrators solving merge conflicts faster than integrators without prior knowledge mainly because of the inherent structure including dependencies among methods, functions, class, modules, or components that may exist in the conflicting code as discussed in Section 6.4.1.

Implications for practitioners: In more than half of the conflicting merge scenarios, integrators with previous knowledge on the conflicting files are the ones that resolve merge conflicts and they normally solve more complex and larger merge scenarios. We argue that integrators with previous knowledge on the merge scenario are recommended to resolve more complex and larger conflicting merge scenarios because they may provide more solid solutions. However, we cannot affirm that previous knowledge makes the merge conflict resolution faster.

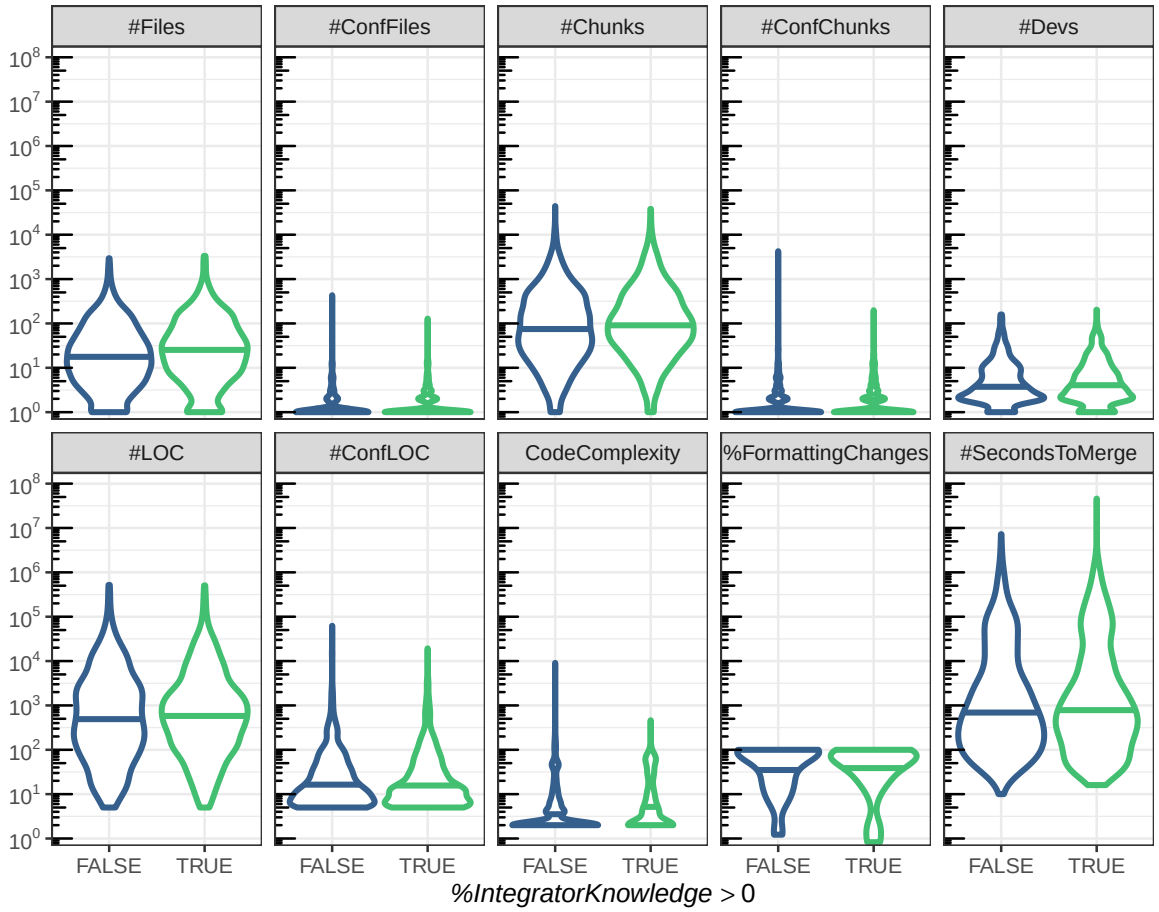

Figure 6.8: Violin Plots Distinguishing Merge Scenarios by the predicate *%IntegratorKnowledge* > 0

What is the percentage of conflicting merge scenarios due to formatting changes? 2.42 % of the conflicting merge scenarios occurred because of formatting changes. This small percentage is likely the reason that it does not correlate with other variables. For short, from the 66 subject projects, 30 have, at least, one merge scenario with merge conflicts arising due to formatting changes; 15 have, at least, one merge scenario of which all merge conflicts arose from formatting changes. What intrigued us, was the fact that despite the effort of researchers on proposing merge strategies [10, 11] and that simple definitions of contribution rules (e.g., defining expected code style) could extinguish this type of conflict, they are still present in some subject projects. As suggested by survey participants, formatting style from different IDEs might be an indicator of why this kind of conflict occurs even when contribution rules are well-defined in the project. An investigation aiming at finding out the reasons of why the merge conflicts due to formatting changes emerged and the actions developers took over the evolution of the projects would be welcome to both researchers and practitioners. As it is far from our goal, we leave it for future work.

Implications for researchers: Our results imply that formatting changes are not really relevant for the merge conflict resolution time since, despite 30 (out of 66) projects have conflicting merge scenarios due to formatting changes, it represents only 2.42 % of the merge scenarios analysed. A deeper investigation on domains and a temporal analysis of the contribution rules may be fruitful to better understand the reasons why developers proposed them and how it impacts on the occurrence of merge conflicts as well as on their resolution time.

6.4.3 Investigating Relationships Among Subject Variables

In this section, we investigate relationships between subject variables that at the first glance seem counter-intuitive.

Why merge conflict resolution time is stronger correlated with merge scenario size measures than with merge conflict size measures? Some merge conflicts are not trivial to resolve and, for that reason, there are many studies investigating it [2, 10, 11, 32, 49, 81, 110, 123, 163, 180, 196, 214, 252, 302]. In some cases, unexpected results are found. For instance, Leßenich et al. [180] aimed at predicting merge conflicts, but even though they have used factors that practitioners indicated to be related to the emergence of merge conflicts (e.g., scattering degree among classes, commit density, and number of files), none of these factors showed a strong correlation with the occurrence of merge conflicts. One may say that the merge conflict resolution depends on the type of conflict [2], others say that it depends on the language constructs that are involved in the conflict [110]. It is reasonable to expect that resolving merge conflicts involves much more than only taking a look at the conflicting code, especially when there is a dependency between conflicting and non-conflicting code introduced in the merge scenario (see Section 6.4.1). Resolving merge conflicts without looking at dependent code changes may introduce unexpected behavior to the project, even when it passes the test suite. Hence, the most prominent action is to understand, at least, the non-conflicting code related to the conflicting code.

To get a concrete example, we choose a merge scenario from project NODE[14]. Despite of being a large merge scenario (#*LoC* = 36794, #*Chunks* = 7305, and #*Files* = 669), we see only 86 lines in conflict (#*ConfLoC*) in three conflicting chunks (#*ConfChunks*) of three conflicting files (#*ConfFiles*). The developers needed around 40 hours to solve the merge conflict. Looking at one of the three files in conflict, *"src/node_crypto.cc"*, we could see that this file refers to five other non-conflicting files changed in the merge scenario. In addition, despite this file having only one conflicting chunk, other 87 chunks were changed in this file. Therefore, despite the merge conflict being small, the merge scenario changes are quite large and we found some dependencies among conflicting and non-conflicting code. These dependencies may explain the large amount of time needed to resolve this small conflict. In this direction, a recent study [43] accompanied by 7 developers resolving 10 merge conflicts noted that developers usually first look at files changed and then resolve the merge conflicts. Our results give nuance to this finding in a broader perspective and using different research methods.

14 https://github.com/nodejs/node – COMMIT HASH: 61CCAF

Implications for researchers and tool builders: Merge scenario characteristics impact more on the merge conflict resolution time than merge conflict characteristics especially when dependencies among conflicting and non-conflicting code are found. Therefore, researchers should pay more attention to measures related to the merge scenario when exploring merge conflicts and tool builders should consider them when creating solutions to support practitioners on resolving merge conflicts.

Why does the number of chunks and the code complexity of the conflicting code show a negative correlation with the time needed to resolve merge conflicts? Before a data-driven answer, let us make an analogy. Once you find a very long method, it might be an indicator that this method does more than it should do. In several cases, one or more methods can be extracted from this very long method to make it more concise. The same is valid for commits. Previous studies have shown that committing small chunks of code make it easier to understand the code changes [7, 138]. In the context of merge conflicts, thirteen survey's participants reported that their strategy to avoid conflicts is simply based on small commits and merge often. As a data-driven and simpler discussion, our arguments here focus on three measures used in our regression model: *#Chunks*, *#LoC*, and *CodeComplexity*. It is worth remembering that the results of our regression model are valid for one variable only when the values of the others remain the same. Therefore, by increasing the number of chunks and keeping the same number of lines of code and the complexity of the conflicting code, we have small chunks which are much easier to understand and resolve [7, 138, 214]. In other words, a fast understanding of each chunk makes it easier to figure out which ones have a dependence on conflicting code. Hence, integrators can focus only on the dependent ones to resolve the merge conflict, also avoiding unexpected behaviours. In the end, we have merge scenarios easier to understand which will also reflect on a faster merge conflict resolution.

As a concrete example related to the number of chunks, we selected two merge scenarios from VSCode[15] with similar *#LoC* and *CodeComplexity*, but different values of *#Chunks*. In each of these merge scenarios, developers have changed around 10 thousand lines of code and have only one conflicting chunk with code complexity equal 2. However, while in the first scenario the code is divided into 1458 chunks, in the second it was divided into 268 chunks. Taking all arguments into account that we discussed before, we assume that the first merge scenario would be easier to understand. In this particular case, it was true. The (same) integrator needed around 9 minutes to resolve the first merge scenario while she needed around 24 minutes to resolve the second. Looking at the code changes, we found nested scenarios in both cases. In other words, a developer was working in the source branch to add a feature while other developers were working in parallel to address other issues in other branches. At the end, their changes were integrated into the target branch before the subject source branch. At the end, most of the changes occurred in the target branch. As already discussed in Section 6.4.1, the location, content, and dependencies among non-conflicting code and conflicting code might have influenced the resolution time. It is worth mentioning that in both exemplified scenarios the changes in the source branch were quite simple, however, the merge scenario took around one week. Extracting some further information from GitHub, we found a possible reason that might have made these

15 https://github.com/Microsoft/vscode – commit hashes: bd8108 and 44c395

two merge scenarios longer. In both scenarios, the contributor was participating for the first time in the project. As Microsoft requests a Contribution Licence Agreement (CLA) for first-time contributors, it might have taken some time. Note that it does not impact on the merge conflict resolution, however, it might have influenced the time the merge scenario lasted, opening space for other integrations and introduction of merge conflicts.

To illustrate the code complexity of conflicting code, we selected two merge scenarios from NEXT.JS[16] with similar *#LoC* (387 vs. 661), *#Chunks* (107 vs. 147), *#ConfChunks* (9 vs. 9), and *#ConfFiles* (2 vs. 2), but different values of *CodeComplexity* (2 vs. 20). It is more intuitive to think that the conflicts resolution of the first scenario was faster than the second one since it is slightly smaller and less complex in terms of *#LoC* and *CodeComplexity*. However, while the same integrator took 47 minutes to resolve the first scenario, she took 18 minutes to resolve the second. Deeply looking at the code changes, we found that in the first scenario, there were changes in the yarn.lock and package.json files which only some of these changes were in conflict with. Ignoring the file content/extension/location (discussed in Section 6.4.1) and the slight difference in the number of chunks discussed above, we assume that the integrator fixed the conflicts fast and missed some dependencies. Once rebuilding and running the project, other dependency errors were found and she needed more time to complete the merge. In the second scenario, 8 of the 9 conflicts were in the taskfile.js with some nested loops. However, most of the changes are related to formatting, renaming, addition of new functions in the same region of code, and new parameters in asynchronous functions. All in all, for the first scenario, we assume that non-conflicting code influenced the long resolution time, for the second scenario, we assume that the type of the changes supported a faster resolution even though it looked more complicated in the beginning.

Implications for practitioners: Committing small chunks of code makes the code understanding easier and, consequently, merge conflict resolution faster. Indeed, we are not the first ones to recommend developers committing small chunks of code. However, to the best of our knowledge, we did not find any study showing that committing small chunks of code makes the merge conflict resolution faster. Related to the code complexity of the conflicting code, we see that changes that look more complex on first sight might have simpler solutions depending on their type and location as discussed in Section 6.4.1.

6.4.4 Comparison of Previous Work Results

What factors related to the merge conflict resolution have been explored in the literature and how do our results relate to them? In Section 2.8, we presented previous work on merge conflict resolution. Next, we compare our results with two other studies [110, 214]. As it is hard to quantitatively compare the results, we put all factors into Table 6.10 differing them as: (i) factors that make merge conflict resolution longer/harder (↗), (ii) factors that do not impact on the time/difficulty of resolving merge conflicts (→), and (iii) factors that make merge conflict resolution faster/easier (↘).

16 https://github.com/vercel/next.js – COMMIT HASHES: F34262 AND C92BDE

Nelson et al. [214] investigated nine factors while Ghiotto et al. [110] investigated four factors of which one is in both studies (see Table 6.10). Our study explores six factors from Nelson et al. and two factors from Ghiotto et al. In addition, we explore five factors not explored by previous studies. Our study is in-line with previous studies for most of the factors. The only exceptions are with factors: *complexity of conflicting lines of code* and *expertise in the area of conflicting code*. We acknowledge that our result in this case is counter-intuitive at first sight. The justification for it is given by the setup and results of our study. We can use the same argumentation when explaining the negative correlation between the number of chunks of the merge scenario and the merge conflict resolution time (see Section 6.4.3). For short, in our study the increase of the complexity of conflicting lines of code has a negative influence on the merge conflict resolution time only when the other variables in the regression model (*#LoC*, *#Chunks*, *#Devs*, and *#ConfChunks*) remain fixed. This way, we would keep with the same merge-conflict and merge-scenario size which are factors that strongly impact the merge conflict resolution time. In other words, since the chunks remain small it is not a problem they are slightly more complex. Regarding the expertise in areas of conflicting code, prior knowledge in the conflicting files does not always help people to resolve their tasks [112]. Hence, we believe that developers with previous knowledge do not always resolve merge conflicts faster than developers without prior knowledge, as discussed in Section 6.4.2.

Related to the factor *time to resolve a conflict*, we consider the results from our survey comparable with the results of Nelson et al. [214]. We both asked for the developer's agreement with a similar statement "the more time it takes to resolve a conflict, the more difficult the conflict" in a 5-point Likert-type scale. Despite our developers' set being different from theirs, we got a median of 3 and a mean of 3.4, while they got a median of 3 and a mean of 2.82. Therefore, we both found that the time of resolving merge conflicts is perceived by the developers as a factor of difficulty of merge conflicts.

The factor *non-functional changes in the code base* is similar to the factor *formatting changes in the code in conflict*. We prefer to classify them separately because the formatting changes in the code in conflict are only a subset of the non-functional changes in the code base. We consider that non-functional changes in the code base also include refactoring (e.g., renaming and reordering methods). Taking into account previous work [192] and survey participants, one of the main changes related to merge conflicts is refactoring. We agree with the developers surveyed by Nelson et al. [214] that non-functional changes in the code base (when refactorings are included) make the merge conflict resolution longer/harder. However, when considering only formatting changes in the code in conflict (i.e., excluding refactorings), it does not influence the merge conflict resolution time. In any event, it is worth remembering that the setup of our empirical study considers all variables together while previous work [214] asks developers individually.

Looking at Table 6.10 and considering the results of our effect-size analysis, we see that most factors that only we explore are related to the merge scenario size, i.e., not directly related to the merge conflict. In addition, *#Chunks*, *#Devs*, and *#LoC* are the three factors in our study with the highest effect on the merge conflict resolution time. Taking into account our cross-validation surveying developers, we are confident that, when resolving merge conflicts, developers usually are aware of the changes in the merge scenario. Being aware of the changes might take some time which influences the merge conflict resolution

Table 6.10: Comparison of our Results with Previous Studies

Factors	Nel.	Ghi.	Our Study
Factors directly related to merge conflicts			
Complexity of conflicting lines of code	↗	-	↘
Expertise in area of conflicting code	↗	-	→
Complexity of files with conflicts	↗	-	-
Number of conflicting lines of code	↗	↗	↗
Time to resolve a conflict	↗	-	↗
Atomicity of changesets in conflict	↗	-	-
Dependencies of conflicting code	↗	-	↗*
Number of files in the conflict	↗	-	↗
Non-functional changes in codebase	↗	-	-
Number of chunks in conflict	-	↗	↗
Language constructs involved in conflicting chunks	-	↗	-
Language constructs involving dependencies among conflicting chunks	-	↗	-
Formatting changes in the code in conflict	-	-	→
Factors indirectly related to merge conflicts			
Number of lines of code of the merge scenario	-	-	↗
Number of chunks of the merge	-	-	↘
Number of files of the merge scenario	-	-	↗
Number of developers involved in the merge scenario	-	-	↗

Nel. and Ghi. stand for Nelson et al. [214] and Ghiotto et al. [110].
*"↗" means that the factor makes the merge conflict resolution longer/harder, "→" means that the factor does not impact on the time/difficulty of resolving merge conflicts, and "↘" means that the factor makes the merge conflict resolution faster/easier. * highlights that the conclusion for this factor came by further analysis.*

time. Together with our previous discussions it shows that, in practice, merge scenario characteristics should be considered when exploring merge conflicts resolution.

Implications for practitioners and researchers: Merge conflict resolution theory and practice are in-line for most of the factors involved in both types of investigation. As mentioned, researchers and practitioners should also consider factors not directly related to merge conflicts (e.g., *#Chunks*, *#Devs*, and *#LoC*) in their analyses since these factors influence more the merge conflict resolution time than factors directly related to merge conflicts (e.g., *#ConfChunks* and *#ConfLoC*).

6.4.5 Reflections on the Merge Conflict Life-Cycle

Is rebasing a good solution for avoiding/dealing with merge conflicts? In our survey, 8 participants reported that they use rebases to integrate branches, whence, it is worth discussing this topic. Despite rebases drawn in a more linear evolutionary history view, rebase commits change the order code changes in fact occurred damaging the project history. Therefore, rebase should be used with care. The main difference between *git merge* and *git rebase* scenarios on the merge conflict resolution is that, in *git merge* scenarios, code changes are shown once all together and in *git rebase* scenarios, Git individually reapplies the commits off the to-do list. Hence, developers need to resolve conflicts first, and then Git continues to reapply the remaining commits. There is a chance that these resolutions would conflict with these remaining commits in the to-do list [161]. Ji et. al [154] investigated merge conflicts in Git rebases. Their results show that conflicts arise in 24.3% – 26.2% of rebase scenarios and no significant difference was found between the likelihood of conflicts arose given *git merge* from previous work [110, 317, 322] and the *git rebase* scenarios from their study. Considering that real-evolutionary history can be used to support developers on different types of tasks, we leave an open question to practitioners: *is the rebase really worth the damage in the project's history?*

Policy for resolving merge conflicts. In the third question of our survey, we asked the participants to share their experience of dealing with merge conflicts. Unfortunately, we did not get any answer explicitly reporting policies to deal with conflicts at project level. In fact, we searched in the GitHub page of each subject project, but we also did not find any report of it. Creating a policy might provide an organised and planned way to deal with conflicts. The creation of such a policy might be an opportunity for practitioners to improve their work-day tasks, as well as, for researchers to collect different experiences from practitioners and create a catalogue of best practices for dealing with merge conflicts.

Why is even a small improvement regarding the time to resolve single merge conflicts relevant for practitioners? The results we presented in Section 6.2.2 do not indicate that the time to resolve merge conflicts can be improved by a very large extent when solely referring to the observed variables. Even though this improvement would be enough to resolve 15.84% of the conflicting merge scenarios of our dataset, this is still a small fraction of them. Considering the difficulty of predicting merge conflicts as well as of creating strategies to resolve them faster, we argue that even a small improvement may help developers in practice. To this end, we see five relevant points of discussion to support our claim:

(i) Developers may get frustrated when they are unaware of merge conflicts [214], it may diminish their satisfaction to work on a project since it is a tedious and error-prone task [180, 203]. (ii) Developers potentially need to handle multiple merge conflicts during the evolution of the project. For instance, the developers of the project D3[17] have resolved 286 conflicting merge scenarios; if it was possible to save five minutes per conflicting merge scenario, they would have saved around 3 full-time working days. (iii) Developers may lose focus on the tasks they were doing to resolve merge conflicts. As merge conflicts normally interfere with other developers' work, they have high priority and developers should stop what they are doing to resolve the conflicts. A break in the software development may lead them to lose track of previous tasks; (iv) By reducing the time/difficulty of merge conflicts, which is a key challenge for developers [180, 196, 214], may also decrease the chance of introducing an error during the merge. (v) An anticipated reduction of merge-conflict difficulty will also benefit other parts of the software project. A reduction in time to resolve a merge conflict is only the result of improving the developers' daily work. For example, when developers introduce one-off contributions, they may also introduce more modularized code that may result also in an improvement to the software architecture.

6.5 Threats to Validity

In this section, we discuss limitations as well as internal and external threats to the validity of our study.

Internal Validity. There are basically six main threats to internal validity. First, we did not measure the software development experience of developers integrating conflicting code. This is a limitation of our study, since more experienced developers may need less time to solve the same merge conflict than less experienced developers. In any case, we argue that large samples average this effect out. Second, we selected subject projects from different programming languages, hence, one language could have dominated our dataset (see the programming language of each subject project in our Appendix B. We checked whether a programming language dominated half of our subject projects. Fortunately, it did not happen. Third, we rebuild merge scenarios by using the standard *git merge* command; if developers used other merge strategies, merge conflicts emerged that may differ from the ones we found. However, as developers normally use standard *git merge*, and avoid using external tools when merging [214], it does not affect our results considerably. Fourth, we could have classified merge scenarios based on their type (e.g., merge scenarios integrated using pull-requests and not using pull-requests) or based on the type of change (e.g., refactoring changes). We minimise this limitation by looking at characteristics of merge scenarios and merge conflicts and by differing formatting changes from other types of changes. Survey participants suggested that refactoring operations are conflict-prone. In that direction there is a previous work already [193]. Unfortunately, this is a limitation of our study and we suggest investigations in that direction in future work (see Section 6.6).

Fifth, we are not able to measure unexpected events that happened on the merge-conflict resolution (e.g., the developer responsible for solving the conflict had a break or asked other developers for support). As discussed in Sections 6.3 and 6.4, we believe that it does not

17 https://github.com/d3/d3

change our results considerably. Sixth, there may be a better measure than *#SecondsToMerge*; changing the measure may also change our results. We are confident that we chose the best option for a post-hoc analysis based on the following arguments: 1) we could not find plausible reasons for only 11 out of the 100 longest conflicting scenarios of our dataset having taken so long. It suggests that unexpected events occur, however, they do not occur very often in practice; 2) comparing the values of *#SecondsToMerge* with the values of a previous study [43] that precisely measured the time spent to resolve merge conflicts, our results are not that far from their results. For instance, while developers in their study needed at least 389 seconds (6.48 minutes) to resolve the half of the longest merge conflicts (i.e., 5/10), developers from our subject projects took at least 697 seconds (11.62 minutes) to resolve the half of the longest merge conflicts (1 304/2 608); and, 3) survey participants agreed that they usually merge their code changes right after addressing an issue/task, and resolve conflicts right after they occur (see Section 6.3.2). Again, it does not mean that *#SecondsToMerge* indeed measures the time developers spent resolving merge conflicts, however, it is an evidence that subject developers normally proceed as we expect (i.e., what we aim to measure with *#SecondsToMerge*).

External Validity. External validity is threatened mainly by three factors. First, our restriction to Git and GitHub as a platform, the three-way merge pattern as well as to the set of measures. Generalizability to other platforms, projects, development patterns, and set of measures is limited. This limitation of the sample was necessary to reduce the influence of confounds, increasing internal validity [263]. While more research is needed to generalise to other version control systems, development patterns, and measures, we are confident that we selected and analysed a practically relevant setting, measures estimating different software properties, 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 active projects (see Section 6.2.1.2). Second, 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 usually small in software projects. Third, the response rate of our survey study is very small (%2), it might be because of external factors that we are not able to control. For instance, (i) emails are not valid anymore given developers that used student emails and finished their studies or developers that used corporative emails and moved to another company or (ii) the great number of surveys in pandemic times as reported by a few developers that replied to our invitation. We tried to increase our response rate following guidelines of previous studies [79] [207] [268], however, it remained low. In any event, we got 140 answers for our survey, which is similar to previous studies investigating merge conflicts (e.g., [196], [214], and [180] had 162, 102, and 41 participants, respectively).

6.6 Conclusion

In this study, we investigated the main challenges on merge conflict resolution with a two-phase study. First, we empirically looked at thousands of merge scenarios from 66 subject projects aiming at identifying factors that make the merge conflict resolution longer. Second,

we minimise threats to validity of our empirical study and cross-validated non-intuitive results by surveying developers from subject projects. Furthermore, we manually checked hundreds of merge scenarios to dive deep in merge conflict resolutions. In Tables 6.7 and 6.8, we presented 4 major challenges detailed into 11 sub-challenges and with a body of 19 solutions to minimise the emergence of merge conflicts as well as make conflict resolutions faster.

Despite several studies investigating merge conflict resolution, we are the first to investigate factors that influence the merge conflict resolution in practice with a triangulated approach (mining, survey and manual analyses). In fact, some of our results were already mentioned in the literature. However, our quantitative and qualitative results complement and add nuance to the recommendations from previous work.

In the next chapter, we summarise the thesis up, present implications of our study, and present suggestions for future work.

7 Conclusion

In this chapter, we provide a summary of our thesis (Section 7.1) followed by the takeaways and implications of the thesis (Section 7.2) and suggestions for future work (Section 7.3).

7.1 Thesis Summary

Software development is a social task and, to succeed, software teams need to coordinate social and technical assets [157]. In Chapter 2, we reviewed several existing studies investigating the social perspective supporting developers to better coordinate, collaborate, and communicate with each other. There are also a number of studies investigating the merge conflict life-cycle taking the technical perspective into account. Hence, either social and technical assets have been extensively investigated in the literature. However, studies investigating the merge conflict life-cycle often ignore the social perspective. Aiming at understanding the role of social assets on the occurrence or avoidance of merge conflicts, we developed 4CsNet, a framework to collect technical and social information from open source repositories (Appendix A). With the data collected with 4CsNet, we performed four empirical studies which we present an overview below.

In the first empirical study (Chapter 3), we investigated the relation between the communication activity and merge conflicts motivated by the popular belief that communication and collaboration success are mutually dependent. We found that active GitHub communication is not associated with the emergence or avoidance of merge conflicts even though developers communicate with each other. Among our discussions, we have seen that (i) the number of commits should be used with care when trying to understand the three-way merge pattern and that (ii) merge scenarios integrated with pull requests are related to fewer merge conflicts than merge scenarios integrated with the standard command in the three-way merge (i.e., `git merge`).

In the second empirical study (Chapter 4), we investigated whether it is possible to predict merge conflicts with social measures (i.e., developer roles at coarse- and fine-grained levels). The motivation is that effectively predicting merge conflicts decreases the cost of constantly pulling and merging a large number of branch combinations (i.e., speculative merging), which makes awareness tools reliable and feasible in practice. Our results show that it is possible to predict merge conflicts taking the social perspective into account with 100% of recall (i.e., all real conflicts are correctly identified). However, to achieve state-of-the-art performance (i.e., combination of recall, prediction, accuracy, AUC), technical measures are necessary. On one hand, our results highlight the importance of investigating the social perspective, especially when developers coordinate themselves without sophisticated tool

support. On the other hand, this study demonstrates that the technical perspective is still essential to predict merge conflicts. Our results show that (i) the touched branch is an important factor in investigating merge conflicts in general and (ii) a deep understanding on the developer code changes, as well as, historical information might be useful to predict and to avoid merge conflicts.

In the third empirical study (Chapter 5), we investigated developer roles and specific developer code changes. The motivation is that understanding social aspects of conflicting contributors and their activity on changing source files can help managers or developers themselves to decide which developers to instruct and when with the purpose of avoiding merge conflicts. Our results show that 80% of contributors are involved in one or two merge scenarios, and only 3.8% of the developers are involved in more than 10 conflicting scenarios. We also found that some developers are often related to merge conflicts independent of the number of merge scenarios they contributed to and that in 42 out of the 66 subject projects, the top contributor is also the top conflicting contributor (i.e., the one involved in more conflicting scenarios in the project). Furthermore, in 48.49% of the projects these top contributors participated in more than 50% of the conflicting merge scenarios in their project. This is evidence that these developers are related to merge conflicts and a deep understanding of their code changes and better coordination might be beneficial to reduce the number of merge conflicts in their projects. By investigating the profile and the code changes of five of these top contributors in detail, we found that (i) contribution rules are useful to avoid simple merge conflicts and, (ii) similar to developers, merge conflicts are concentrated in a few files. Hence, using this information together with historical data in the repository might be useful to make developers more aware when these files are changed which significantly supports the reduction of merge conflicts.

In the fourth empirical study (Chapter 6), we moved to the end of the merge conflict life-cycle investigating the challenges and factors related to the merge conflict resolution. The main motivation is that by understanding which kind of merge conflicts are time-consuming to resolve, developers should focus on avoiding this kind of conflicts mostly. Hence, they minimise delays delivering core-tasks on the project, such as developing new features and fixing critical bugs. Our results show that measures indirectly related to merge conflicts (i.e., measures related to the merge scenario changes) are more strongly correlated with merge conflict resolution time than measures directly related to merge conflicts (i.e., merge conflict characteristics). More specifically, the number of chunks, developers, and lines of code, for instance, have a greater impact on the merge conflict resolution time than the number of conflicting chunks, conflicting files, and conflicting lines of code. Aiming at cross-validating our results and searching for new findings, we surveyed 140 developers. With this survey, we identified four main challenges on merge conflict resolution: *lack of coordination*, *lack of tool support*, *flaws in the system architecture*, and *lack of testing suite or pipeline for continuous integration*. Investigating further characteristics of conflicting scenarios, we identified that the file extension influences the merge conflict resolution time. For instance, resolving merge conflicts in minified and markdown files take less time than resolving merge conflicts in programming language files, such as Java files. Our results pointed the need of creating policies for resolving merge conflicts and to the reasons that even small improvements on the merge conflict resolution time are relevant for practitioners. For short, reducing the merge conflict resolution time, influence on the team mood, decrease the chance of

introducing an error during the merge, and improve the task coordination (e.g., keeping the team in the planned schedule).

7.2 Takeaways and Implications

In this thesis, we investigated the merge conflict life-cycle by taking the social dimension into account. Our investigations included several approaches (e.g., build developer communication networks and derive developer roles) on different merge conflict life-cycles (e.g., when avoiding or resolving merge conflicts). For instance, we provide investigations before the conflicts happen to understand types of changes related to merge conflicts and investigations after the conflict happen to understand characteristics that make merge conflicts harder to resolve. We are confident that we provide substantial and innovative contributions for researchers, tool builders, and practitioners in the field of coordination on software engineering as we detail below.

7.2.1 Implications for Researchers

Our results show the usefulness of social perspective on dealing with merge conflicts. We suggest researchers to investigate the social perspective not only when investigating merge conflicts, but also when investigating other software engineering topics (e.g., coordination issues, bug and technical debt introduction, and quality assurance). In the end, software development and architecture are social tasks and the social perspective influences technical assets.

We found that measures not directly related to merge conflicts influenced the merge conflict resolution time more than measures directly related to merge conflicts. This is evidence that merge conflict is a complex topic and researchers should have a broader view to investigate this topic. We suggest researchers consider the whole code changes in a merge scenario and not only the code changes involved in merge conflicts. In a similar direction, we found that changes in the source branch can be three times more conflict-prone than changes in the target branch. Researchers should take this information into account to investigate merge conflicts since it might explain high conflict frequency in a few merge scenarios.

We provide a broad literature review on merge conflicts and our merge conflict taxonomy might support researchers to use a common naming for types of conflicts in future studies. Researchers might also explore opportunities highlighted in our literature review. For instance, we saw a lack of studies comparing merge conflict resolution strategies and policies.

We found that merge conflicts are normally introduced by a few developers and merge conflicts are recurrently concentrated in only a few files. This is crucial information to researchers creating customised models using historical information and improving the state-of-art of merge conflict prediction.

We have seen that some measures that were widely used in the past, such as the number of commits, should be used with care when investigating merge conflicts, since with the three-

way merge the development process changed over time. Our suggestion is to understand whether a given measure is still meaningful in the context instead of choosing it because other researchers used it in the past.

7.2.2 Implications for Tool Builders

We found which developer roles, files, and the branch developers are touching are conflict-prone. Tool builders might use such information to propose tools to support practitioners avoiding and predicting merge conflicts. This information might also be useful for proposing a tool for recommending developer roles and files to address or tasks that can be addressed simultaneously with low or no risk to raise merge conflicts. Still, tool builders can use this information to create mechanisms to support the coordination of these small groups of developers and effectively reduce the number of merge conflicts.

Developers reported desires, needs, and alerts when dealing with merge conflicts. Among them, we observed that developers do not want to switch from one to another tool. A reasonable solution is to incorporate standalone tools into IDEs. Hence, developers can enable plug-ins once they desire a given functionality.

Another desire mentioned by practitioners tool builders can explore is to improve the visualisation of code changes and merge conflicts. These improvements go from better gathering useful information to resolve merge conflicts until showing a merge-conflict difficulty estimation to support developers choosing whether they resolve a conflict or ask for support, for instance.

7.2.3 Implications for Practitioners

We found that code integrated by using pull requests instead of a common merge command reduces the number of conflicts. We suggest developers more often integrate their code using pull requests. We also found that contribution rules reduce the number of lower-order merge conflicts. Therefore, defining rules not just guide newcomers, but also avoid integration issues (e.g., ordering and formatting conflicts). Contribution rules might also support top contributors reducing their involvement within merge conflicts.

We found that the merge scenarios with small chunks, even though numerous, negatively influence the merge conflict resolution time. Hence, we believe that developers addressing small issues and/or tasks and committing small pieces of code, help not only to avoid merge conflicts, but also to make it simpler to understand and consequently to resolve.

Developers reported (see Section 6.3) that some challenges when resolving merge conflicts are related to the lack of tool support. While it is true that there is space for proposing new tools, in Chapter 2, we reported a bunch of tools that practitioners can use to support themselves. Proposed tools go from keeping developers in the backlog to supporting merge conflict resolution.

We found that practitioners barely define policies for resolving conflicts. Given the injury and loss caused by merge conflicts (see Section 2.4.1), we suggest practitioners define policies to guide themselves on how to deal with merge conflicts. Such policies might include from

recommendation of developers prone to supporting others to how they should proceed in specific scenarios. These policies might also support more solid solutions reducing technical debt introduction.

We found that GitHub communication itself is not associated with merge conflicts. We suggest that developers worry more about their coordination in terms of tasks they are working on and possibly changed files than trying to improve communication about their ongoing tasks. Simply marking the issue they are addressing and linking the commits with a given issue might be enough to keep others aware.

7.3 Future Work

The research presented in this thesis lays the foundation for future work on the methodology and application of strategies and approaches to deal with merge conflict in their whole life-cycle. We suggest studies that cover: (i) a larger set of subject systems (including proprietary systems), (ii) involving other variables that we are not able to properly control, such as, the dependencies among files, and (iii) different version control systems, programming languages, and classifiers (when predicting merge conflicts). With the knowledge acquired in this thesis, we opened up a multitude of further research avenues which we discuss next:

Explore other communication channels. Related to the first empirical study (Chapter 3), we suggest a study investigating other communication channels (e.g., mailing lists, Slack[1], Microsoft Teams[2], Discord[3], Stack Overflow[4], Telegram[5], WhatsApp[6], and Gitter[7]) and the impact of the use of GitHub communication when developers use other channels. Given the popular belief that communication and collaboration success are mutually dependent (see Chapter 2), our results can be seen as a *negative result*, finding no indication for a global monotonic relationship between the amount of GitHub communication and the occurrence of merge conflicts for the majority of merge scenarios. As negative results are often suspected to be due to the failure of the research design [85, 180, 223, 239], this suggestion for a further study may overcome limitations of our study as discussed in Section 3.5.1 and increase the generalizability of our results outside of the GitHub communication activity.

Predicting merge conflicts using historical information. With the results from the studies presented on Chapters 4 and 5, we gathered evidence that some files are more conflict-prone compared to other files. This way, we suggest a work that creates a model/approach to predict merge conflicts taking historical information from files into account. Our assumption is that such a customised model/approach will be able to improve the state-of-the-art of merge conflict predictions.

Interview developers often related to merge conflicts. We suggest a study interviewing top conflicting developers to understand their perspective and, whenever possible, propose procedures and guidelines to minimise the number of merge conflicts. Such a study would

1 https://slack.com/
2 https://www.microsoft.com/en/microsoft-teams/group-chat-software
3 https://discord.com/
4 https://stackoverflow.com/
5 https://web.telegram.org/k/
6 https://web.whatsapp.com/
7 https://gitter.im/

not only confirm and add nuance to our results, but also support the construction of procedures and guidelines to avoid merge conflicts taking the practitioners' perspective into account. This study would be able to evaluate how these procedures and guidelines support the reduction of merge conflicts in multiple repositories, as well as, measure how effective the coordination of this small group of developers might positively impact the software development of subject projects.

Interview developers asking them about specific outlier merge scenarios and how merge conflicts impact on their mood and their contribution to the project. With this study we would find technical or social reasons that influence the resolution of merge conflicts from the practitioners' perspective. For instance, why did a merge conflict of a given characteristic take so long to be addressed? Was that because of lack of experience, lack of knowledge in the piece of code, or because of dependencies with other conflicts or changed code?

An investigation on the impact of the merge conflict resolution time comparing merge scenarios with refactorings and without refactorings. Survey participants (Chapter 6) reported that merge conflicts with refactorings are harder to resolve. Having a study that distinguishes merge scenarios which involve refactoring or not, might provide useful insights and add nuance to our finding of the impact of non-conflicting changes on the merge conflict resolution time, as well as, bring justifications beyond our study.

Similar investigations to our empirical study classifying the different types of conflicts (e.g., textual, syntactic, and semantic). As mentioned, our decision to cover several programming languages makes this investigation harder in practice since it is necessary an analysis in the AST and in several cases the conflicting code would not be enough. A study focusing on specific programming languages instead might be feasible in practice and provides an understanding on the influence of the merge conflict type on its emergence or resolution.

Use code ownership information to predict merge conflicts. Given the difficulty of predicting merge conflicts discussed in previous work [2, 110, 180], we suggest an exploration of the lack of code ownership [108, 119] and the centrality of files on the occurrence of merge conflicts. Some files change more often than others during a project's lifetime. Hence, they may have lack of code ownership or be too central to the project influencing the occurrence of merge conflicts. By providing evidence that these files are conflicting-prone, developers should treat them differently to avoid the introduction of merge conflicts.

A

4CsNet Framework

In this Appendix, we present 4CsNet our open-source framework to collect social and technical information. 4CsNet is the first tool to integrate technical and social perspectives making it simpler to relate GitHub events with code changes that belong to the same merge scenario and not, only looking at contributions that occur in the same series. Being open-source and integrating technical and social assets are the main reasons why we did not use any other tool and consider 4CsNet a contribution of our work.

In Section A.1, we present an overview of the framework. In Section A.2, we present prerequisites and how to run our framework. In Section A.3, we describe the architecture and which information our framework collects. Finally, in Section A.4, we conclude this Appendix.

A.1 Overview

4CsNet is a framework to build merge scenario data, communication data, and networks to help understanding code changes and interactions among contributors. The first part of the name come up from the 4 *c*s:

- Coordination
- Contribution/Collaboration
- Communication
- Conflicts

The *Net* part comes up from the related data and networks that we can build with data collected in our framework to investigate different aspects in the repository history. To understand for instance, the influence on the communication activity on the occurrence of merge conflicts.

4CsNet uses Git[1] and GitHub[2] to collect information. 4CsNet is developed upon two tools (GitWrapper[3] and GitHubWrapper[4]). GitWrapper is a tool that builds a set of classes for working with Git repositories. A given repository is examined using a wrapper around native Git calls. The current feature set is limited to extraction of merge commits and

1 https://git-scm.com/
2 https://github.com/
3 https://github.com/se-passau/GitWrapper
4 https://github.com/se-sic/GitHubWrapper

associated operations. GitHubWrapper is an extension to the GitWrapper for interaction with the GitHub issue and pull-request API. 4CsNet is open-source and mostly built in Java programming language.

A.2 Prerequisites and Execution

In addition to Git, GitWrapper, and GitHubWrapper, the user must have Gradle[5] and MySQL[6] installed to run 4CsNet. Gradle is a tool with a focus on build automation and support for multi-language development. MySQL is an open source relational database. Optionally, the user can also install Workbench[7] to visualise the schema used in 4CsNet. Be sure that the *build.gradle* file in the root of the project has the dependencies to GitWrapper and GitHubWrapper repositories. All files necessary to make the tool run, for instance, database configuration file should be in the resource directory. See details in our GitHub repository[8].

Once the requirements are fulfilled search for the main file (../CN_Main.java). To run the tool, it is expected four arguments:

1. Mode of the tool you want to run
2. File path with GitHub project URLs to be analysed
3. File path with GitHub tokens, and
4. File path with merge commit hashes case you want to analyse only them and not all the merge commits of the target project. If this file is empty, all merge commits identified will be proceeded.

Below we show an example to build all merge scenarios from a list of projects and after we describe what are the tool modes and what are the GitHub tokens:

```
-ms resources/urls.txt resources/tokens.txt resources/mergeCommits.txt
```

Tool Modes. The tool can run in several modes. For instance, building merge scenarios (*ms*), issues (*i*), contribution networks (*contnet*), communication networks (*comnet*), project metrics (*mepr*), merge scenario metrics (*mems*), store developers (*storeDevs*), committers (*storeCommitter*), integrators (*storeIntegrators*). See all options in the main file (i.e., ../CN_Main.java). As the communication network is based on merge scenarios and, to get them, we need the contribution network or at least the merge scenario mode to be run before running the communication networks mode. If the communication mode is selected, but no contribution network or merge scenario is stored before, it will just build issues and data related to issues. The *ms* and *i* modes will just set the merge scenarios and issues in the database, respectively.

5 https://gradle.org/
6 https://www.mysql.com/
7 https://www.mysql.com/products/workbench/
8 https://github.com/se-passau/CommunicationNetwork/tree/master

GitHub Tokens. To retrieve full information from GitHub repositories, the user needs to supply your own OAuth token. Since GitHub API is limited to 60 requests per hour and repositories normally contain hundreds of GitHub issues and events, tokens from multiple users will speed up the analysis. See details in the GitHub setting tokens page[9]. Interestingly, 4CsNet manages the number of requests per token and uses them extensively without exceeding its limit.

A.3 Architecture and Model

4CsNet contains mainly five packages (builder, crawler, metricsExtractor, model, and util). The *builder* package follows the idea of the builder design pattern for constructing complex objects step by step. The *crawler* package is basically used for navigating and exploring Git repositories. It is our crawlers that we define the number of threads will run simultaneously, for instance. The *metricsExtractor* package is a shortcut functionality to (re)compute measures for projects and merge scenarios, for instance. The *model* package contains our business logic. In this model, we define our beans, Data Access Object (DAO)), and database writers, for instance. The *util* package contains various utility classes and interfaces. For instance, it contains our input/output handlers and loggers.

In Figure A.1, we present all tables present in our database. The tables can be divided into four main layers (merge scenario analysis, issue analysis, social analysis, measurement). The only table that is out of layers is the *project table* which has an identifier, name and URL of the repository. We explain each layer in the following sections:

A.3.1 The Merge Scenario Analysis Layer

In Figure A.2, we present the tables for the merge scenario analysis layer. In this layer, we mapped merge scenarios data for each project. Our strategy for merge scenario data acquisition consists of six steps:

1. We select a list of repositories to be investigated. As the repository list depends on the empirical study, we provide details about it in its respective chapter.
2. For each selected repository, we clone a subject project's repository.
3. We identify merge scenarios by filtering commits with multiple parent commits (merge commits are identified in Git when the number of parent commits is greater than one).
4. For each merge commit, we retrieve a common ancestor for both parent commits (i.e., the base commit).
5. We (re)merge the parent commit of the source branch into the parent commit of the target branch by using the standard *git merge* command and retrieve measurement data by comparing the changes that occurred from the base commit until the merge

9 https://github.com/settings/tokens

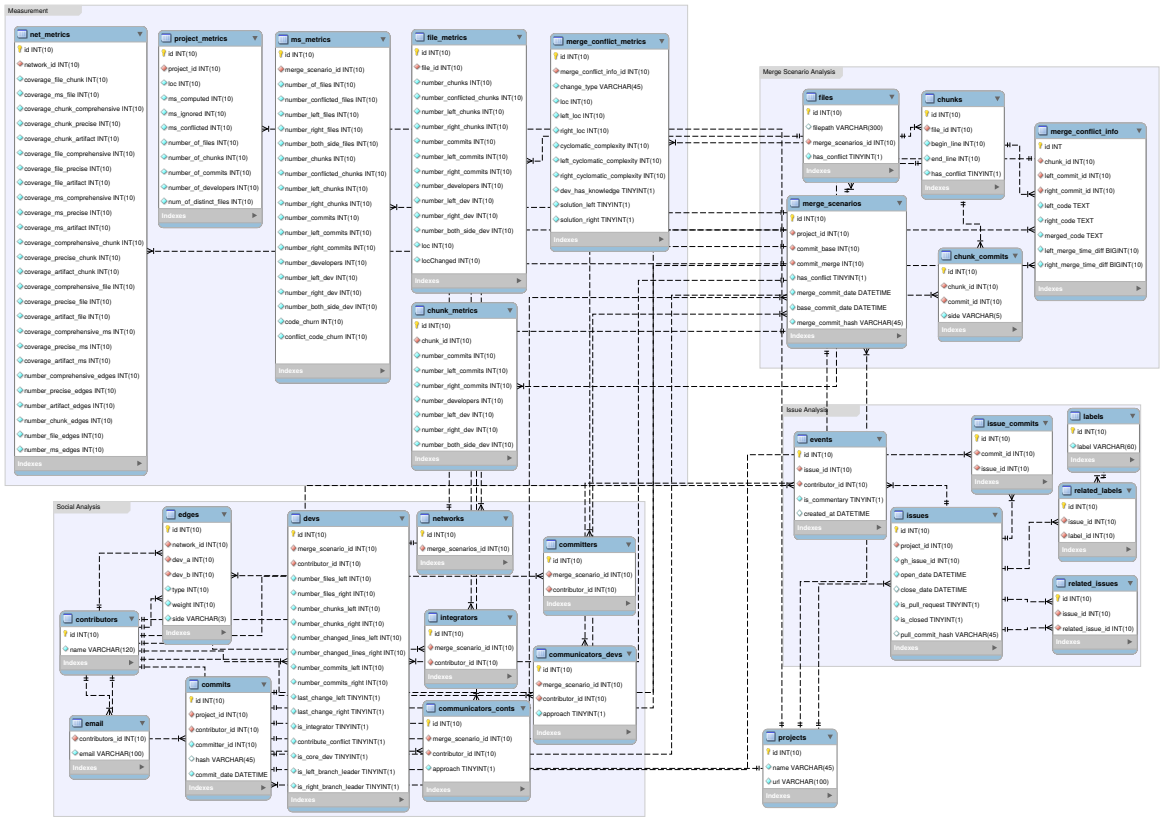

Figure A.1: 4CsNet Database Schema

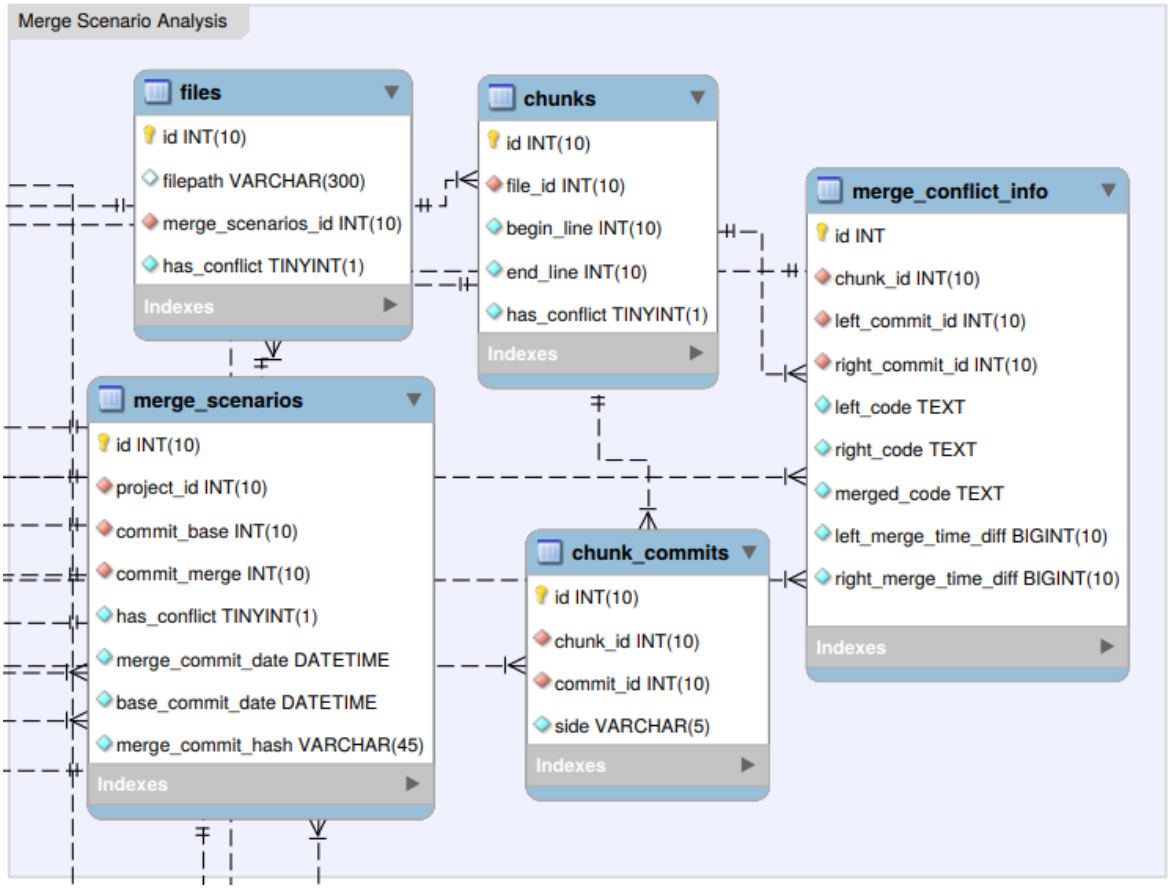

Figure A.2: Merge Scenario Analysis Layer Tables

commit. Commit information includes author, date, lines of code, and files changed (as we will see in the social analysis layer A.3.3). Therewith, we know, for instance, which developer (by the commit's author) changed each line of code at each branch.

6. We store all data and repeat steps 4 to 6 for each merge scenario found in step 3.

Note that we have excluded merge scenarios that do not have a base commit (e.g., rebase, fast-forward, or squash integrations [161]), and we ignore binary files, because we cannot track their changes. Note also that the integration of two branches is not tied to pull-requests. Once we identified an integration of one or more branches into another, we rebuilt the merge scenario. It is important to highlight that all investigated merge scenarios integrate only two branches (i.e., no octopus merges).

Therefore, at the end, for each merge scenario, we store the touched files. For each file, we store the touched chunks. For each chunk we store the commit related to them and, for the conflicting chunks we still store the code related in each branch (target and source) and the merged code.

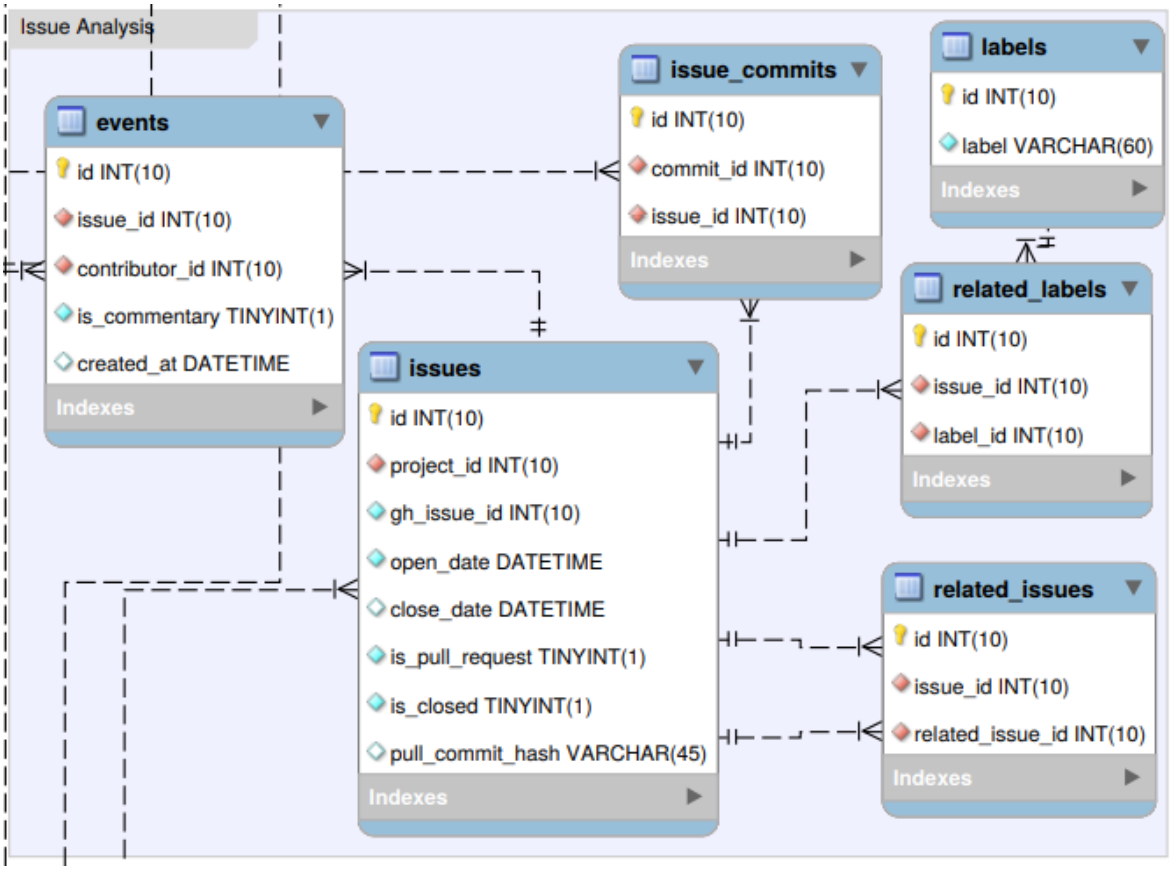

Figure A.3: Issue Analysis Layer Tables

a.3.2 The Issue Analysis Layer

In Figure A.3, we present the tables for the issue analysis layer. In this layer, we mapped GitHub data for each project and linked them with merge scenarios whenever possible. As mentioned in Section 2.3, GitHub data consists of retrieving issues from a project and the developers, events, commits, labels and other issues related to them. In Figure A.4, we show an example where an issue with title "Testing Issues communicationNetwork" is created (#2), a label *enhancement* is added, the a developer assigned herself, and created a comment where she related a commit (hash: 44a2add) and a previous issue (#1). Note that at the end, the issue is closed and set as completed.

a.3.3 The Social Analysis Layer

In Figure A.5, we present the tables for the social analysis layer. In this layer, we mapped data for each project in a more complex way than the previous layers. For each contributor (i.e., a person who committed to a repository or created an event in a GitHub issue), we store her name and email. We also store every commit this contributor participated in. With this information we extracted some refined information, for instance, who are the

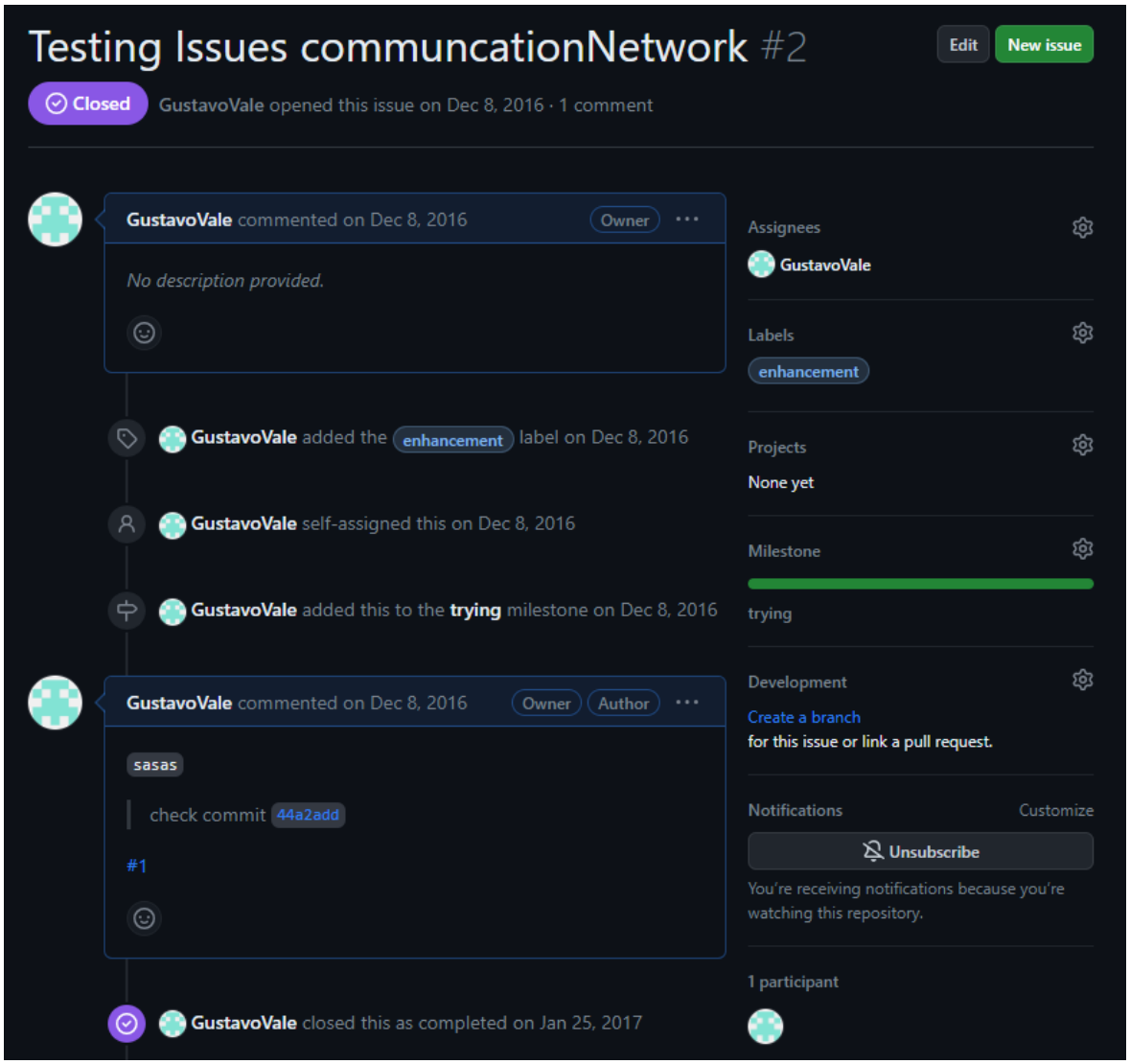

Figure A.4: Example of GitHub Issue

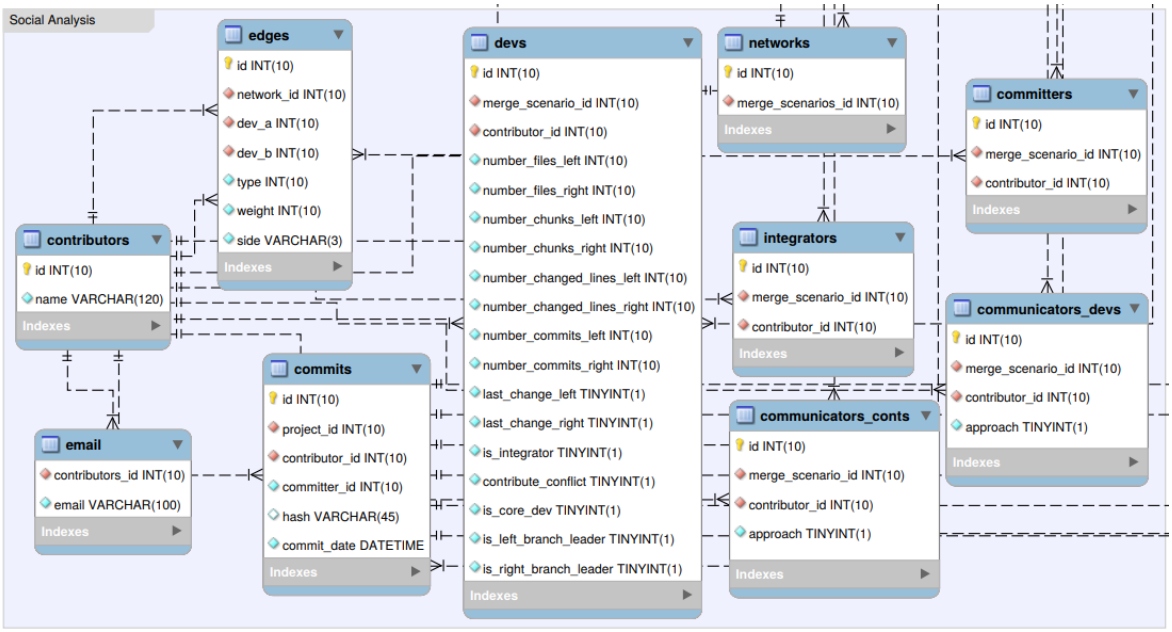

Figure A.5: Social Analysis Layer Tables

developers (i.e., person who commits code), integrators (i.e., developers that integrated branches) and communicators (i.e., person who communicated on GitHub). Later, when presenting empirical studies, we explain, for instance, how we build networks, the type of networks and developer roles we investigate.

a.3.4 The Measurement Layer

In Figure A.6, we present the tables for the measurement layer. The measures used in our empirical studies are not limited to them. We decided to compute these measures from coarse-grained level (such as project and merge scenario level) to fine-grained level (such as chunks and merge conflict level) to have a great understanding and build visualisations of our data. For instance, knowing the number of LOC), the number of merge scenarios computed (ms_computed), the number of merge scenarios ignored (ms_conflicted), and number of files (number_of_files) from table project_metrics help us to understand the size and practices used in a project to see whether it should be included or not into our dataset selected projects in an empirical study. On the other hand, having fine-grained measures at chunk level, like the total number of commits and developers involved in a chunk as well as the number of developers and commits for the target and source branch supports an initial understanding of the chunk characteristics.

a.4 Final Remarks

In this Chapter, we present 4CsNet a framework to build merge scenario data, communication data, and networks to help understanding code changes and interactions among

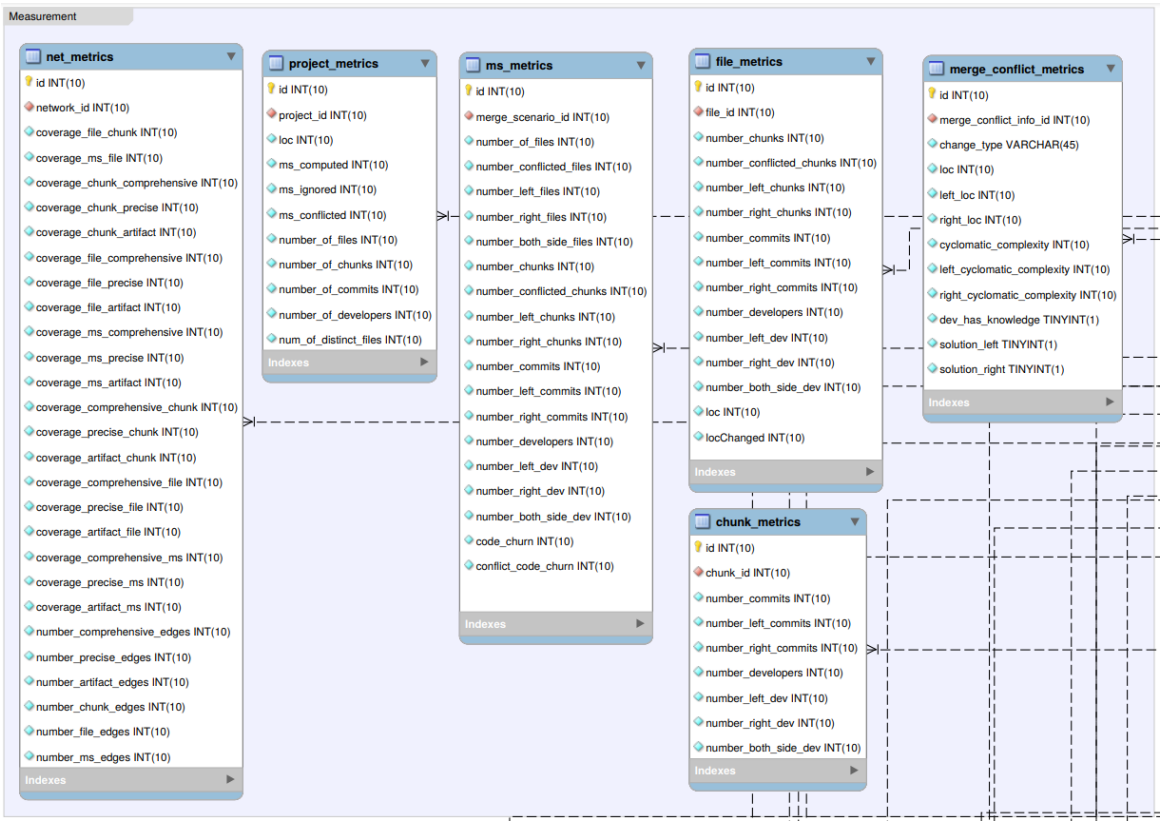

Figure A.6: Measurement Layer Tables

contributors. In this thesis, 4CsNet is used to automate our analyses and, as there are common procedures on our empirical investigations.

As mentioned, several tools and frameworks can be found in the literature. However, 4CsNet is the first one integrating the technical and the social perspective enabling investigations on these topics individually or together providing a broader and more robust investigation. All empirical studies presented in Chapters 3, 4, 5, and 6 were powered by our framework.

B

Appendix

In this appendix, we show the subject project list for our empirical studies (Section B.1) and the excluded projects as well as the reason the projects were excluded from our analysis (Section B.2).

b.1 Subject Projects

In Tables B.1, B.2, B.3, and B.4, we show all subject projects sorted by the number of stars at the moment we selected them. We split the projects into multiple tables to make them fit in the page. Note that in all of these tables, we present project ID, their name and URL (*Name*), the main programming language used in the project (*Prog. Lang.*), the number of starts (*#Stars*), the number of merge scenarios identified in the project at the moment of our analyses (*#MS*), the domain of the project (*Domain*), and the type of contribution rules in the project (*Contr. Rules*). The contribution rules information was used in an analysis from Chapter 5.

In the subject project list, we see a variety 13 of programming languages such as JavaScript, Python, Java, Rust, Go, and C. JavaScript is the most common programming language appearing in 32 of the subject projects. Despite this, 996.ICU, the repository with the greatest number of stars, is developed using Rust. A couple of programming languages. such as C, C++, Shell, and CSS were found in only one subject project.

Looking at the domain of subject projects, we see a list with repositories with five different domains (framework, library, learning, programming language, and tool). Tool appeared 22 times in our list followed by framework that appeared 18 times. Programming language appeared only 4 times in our list being the least mentioned in our list.

Table B.1: Subject Projects for Empirical Studies (Part 1)

ID	Name	Prog. Lang.	#Stars	#MS	Domain	Contr. Rules
1	996.ICU[1]	Rust	243370	1280	Library	No rules
2	vue[2]	JavaScript	137975	140	Framework	Yes, well defined
3	bootstrap[3]	JavaScript	133212	4355	Framework	Yes, well defined
4	react[4]	JavaScript	128811	2486	Library	Yes, well defined
5	oh-my-zsh[5]	Shell	87925	1579	Framework	Yes, well defined
6	d3[6]	JavaScript	84554	688	Library	No rules
7	react-native[7]	JavaScript	76955	650	Framework	Yes, well defined
8	vscode[8]	TypeScript	75689	4222	Tool	Yes, well defined
9	electron[9]	C++	73088	4198	Framework	Yes, well defined
10	create-react-app[10]	JavaScript	67352	80	Tool	Yes, well defined
11	awesome-python[11]	Python	67037	452	Learning	Yes, but not clear
12	CS-Notes[12]	Java	61165	493	Learning	Yes, but not clear
13	node[13]	JavaScript	61047	391	Framework	Yes, well defined
14	Font-Awesome[14]	JavaScript	59749	156	Learning	Yes, but not clear
15	angular.js[15]	JavaScript	59509	36	Framework	Yes, well defined
16	animate.css[16]	CSS	59336	111	Library	Yes, but not clear
17	axios[17]	JavaScript	59120	188	Library	Yes, well defined
18	go[18]	Go	57868	147	Prog. Lang.	Yes, well defined
19	public-apis[19]	Python	56704	624	Library	Yes, well defined
20	models[20]	Python	52510	970	Library	Yes, but not clear
21	laravel[21]	PHP	52169	1455	Framework	Yes, well defined

Prog. Lang. *stands for programming language.* ***Contr. Rules*** *stands for contribution rules*

1 https://github.com/996icu/996.ICU
2 https://github.com/vuejs/vue
3 https://github.com/twbs/bootstrap
4 https://github.com/facebook/react
5 https://github.com/robbyrussell/oh-my-zsh
6 https://github.com/d3/d3
7 https://github.com/facebook/react-native
8 https://github.com/Microsoft/vscode
9 https://github.com/electron/electron
10 https://github.com/facebookincubator/create-react-app
11 https://github.com/vinta/awesome-python
12 https://github.com/CyC2018/CS-Notes
13 https://github.com/nodejs/node
14 https://github.com/FortAwesome/Font-Awesome
15 https://github.com/angular/angular.js
16 https://github.com/daneden/animate.css
17 https://github.com/mzabriskie/axios
18 https://github.com/golang/go
19 https://github.com/toddmotto/public-apis
20 https://github.com/tensorflow/models
21 https://github.com/laravel/laravel

Table B.2: Subject Projects for Empirical Studies (Part 2)

ID	Name	Prog. Lang.	#Stars	#MS	Domain	Contr. Rules
22	jquery[22]	JavaScript	51510	250	Library	Yes, well defined
23	youtube-dl[23]	Python	50264	1381	Tool	Yes, but not clear
24	TypeScript[24]	TypeScript	48760	7616	Prog. Lang.	Yes, well defined
25	webpack[25]	JavaScript	48729	2269	Tool	Yes, but not clear
26	atom[26]	JavaScript	48675	4336	Tool	Yes, well defined
27	redux[27]	JavaScript	48489	675	Tool	Yes, well defined
28	angular[28]	TypeScript	47939	300	Framework	Yes, well defined
29	java-design-patterns[29]	Java	47481	393	Library	Yes, well defined
30	material-ui[30]	JavaScript	46700	2115	Tool	Yes, well defined
31	socket.io[31]	JavaScript	46218	342	Framework	Yes, but not clear
32	ant-design[32]	TypeScript	46194	2108	Prog. Lang.	Yes, well defined
33	reveal.js[33]	JavaScript	46084	447	Framework	Yes, but not clear
34	Semantic-UI[34]	JavaScript	45344	658	Framework	Yes, but not clear
35	30-seconds-of-code[35]	JavaScript	44247	1579	Learning	Yes, well defined
36	flask[36]	Python	43923	874	Framework	Yes, well defined
37	express[37]	JavaScript	43723	523	Framework	Yes, but not clear
38	thefuck[38]	Python	43601	359	Tool	Yes, but not clear
39	awesome-go[39]	Go	43581	928	Learning	Yes, but not clear
40	Chart.js[40]	JavaScript	43322	602	Tool	Yes, but not clear
41	html5-boilerplate[41]	JavaScript	42755	214	Library	Yes, well defined
42	lantern[42]	Go	41818	2717	Tool	No rules

Prog. Lang. *stands for programming language.* ***Contr. Rules*** *stands for contribution rules*

22 https://github.com/jquery/jquery
23 https://github.com/rg3/youtube-dl
24 https://github.com/microsoft/TypeScript
25 https://github.com/webpack/webpack
26 https://github.com/atom/atom
27 https://github.com/reactjs/redux
28 https://github.com/angular/angular
29 https://github.com/iluwatar/java-design-patterns
30 https://github.com/callemall/material-ui
31 https://github.com/socketio/socket.io
32 https://github.com/ant-design/ant-design
33 https://github.com/hakimel/reveal.js
34 https://github.com/Semantic-Org/Semantic-UI
35 https://github.com/30-seconds/30-seconds-of-code
36 https://github.com/pallets/flask
37 https://github.com/expressjs/express
38 https://github.com/nvbn/thefuck
39 https://github.com/avelino/awesome-go
40 https://github.com/chartjs/Chart.js
41 https://github.com/h5bp/html5-boilerplate
42 https://github.com/getlantern/lantern

Table B.3: Subject Projects for Empirical Studies (Part 3)

ID	Name	Prog. Lang.	#Stars	#MS	Domain	Contr. Rules
43	django[43]	Python	41460	656	Framework	Yes, well defined
44	httpie[44]	Python	41082	101	Tool	Yes, well defined
45	moment[45]	JavaScript	41071	964	Library	Yes, but not clear
46	meteor[46]	JavaScript	41041	2222	Tool	Yes, well defined
47	keras[47]	Python	41022	763	Tool	Yes, but not clear
48	json-server[48]	JavaScript	40013	55	Learning	No rules
49	awesome-machine-learning[49]	Python	39700	555	Learning	No rules
50	lodash[50]	JavaScript	39043	192	Library	Yes, well defined
51	RxJava[51]	Java	38851	1591	Library	Yes, but not clear
52	requests[52]	Python	38528	1494	Learning	Yes, well defined
53	netdata[53]	C	38034	1617	Tool	Yes, well defined
54	Python[54]	Python	37965	317	Learning	Yes, well defined
55	ionic[55]	TypeScript	37911	923	Framework	Yes, well defined
56	markdown-here[56]	JavaScript	37750	70	Tool	No rules
57	jekyll[57]	Ruby	37609	2064	Tool	Yes, well defined
58	storybook[58]	JavaScript	37574	6137	Tool	Yes, well defined
59	element[59]	Vue	37476	614	Tool	Yes, well defined
60	next.js[60]	JavaScript	37217	102	Framework	Yes, but not clear
61	ansible[61]	Python	37035	5609	Tool	Yes, well defined
62	redis[62]	C	36307	1094	Tool	Yes, but not clear

Prog. Lang. *stands for programming language.*

Contr. Rules *stands for contribution rules*

43 https://github.com/django/django
44 https://github.com/jakubroztocil/httpie
45 https://github.com/moment/moment
46 https://github.com/meteor/meteor
47 https://github.com/keras-team/keras
48 https://github.com/typicode/json-server
49 https://github.com/josephmisiti/awesome-machine-learning
50 https://github.com/lodash/lodash
51 https://github.com/ReactiveX/RxJava
52 https://github.com/requests/requests
53 https://github.com/netdata/netdata
54 https://github.com/TheAlgorithms/Python
55 https://github.com/ionic-team/ionic
56 https://github.com/adam-p/markdown-here
57 https://github.com/jekyll/jekyll
58 https://github.com/storybooks/storybook
59 https://github.com/ElemeFE/element
60 https://github.com/zeit/next.js
61 https://github.com/ansible/ansible
62 https://github.com/antirez/redis

Table B.4: Subject Projects for Empirical Studies (Part 4)

ID	Name	Prog. Lang.	#Stars	#MS	Domain	Contr. Rules
63	react-router[63]	JavaScript	36151	617	Tool	Yes, but not clear
64	rust[64]	Rust	36029	26421	Prog. Lang.	Yes, but not clear
65	materialize[65]	JavaScript	35695	746	Framework	Yes, well defined
66	yarn[66]	JavaScript	35695	207	Tool	Yes, well defined

Prog. Lang. *stands for programming language.* ***Contr. Rules*** *stands for contribution rules*

Table B.5: Projects in the Initial Selection, but Excluded (Part 1)

ID	Name	Programming Language	#Stars	Exclusion Filter
67	freeCodeCamp[67]	JavaScript	302551	(4)
68	tensorflow[68]	C++	127312	(3)
69	free-programming-books[69]	No language	122687	(1)
70	awesome[70]	No language	108144	(1)
71	You-Dont-Know-JS[71]	No language	101509	(1)
72	javascript[72]	JavaScript	84845	(4)
73	gitignore[73]	No language	83537	(1)
74	developer-roadmap[74]	No language	80910	(1)
75	coding-interview-university[75]	No language	77388	(1)
76	linux[76]	C	74733	(3)
77	flutter[77]	Dart	62213	(4)

B.2 Excluded Projects

In Tables B.5, B.6, and B.7, we show the list of projects that were initially selected by the number of stars criteria, but were later excluded by one of our filters. GitHub projects were excluded from our analyses due to four reasons:

63 https://github.com/ReactTraining/react-router
64 https://github.com/rust-lang/rust
65 https://github.com/Dogfalo/materialize
66 https://github.com/yarnpkg/yarn
67 https://github.com/freeCodeCamp/freeCodeCamp
68 https://github.com/tensorflow/tensorflow
69 https://github.com/EbookFoundation/free-programming-books
70 https://github.com/sindresorhus/awesome
71 https://github.com/getify/You-Dont-Know-JS
72 https://github.com/airbnb/javascript
73 https://github.com/github/gitignore
74 https://github.com/kamranahmedse/developer-roadmap
75 https://github.com/jwasham/coding-interview-university
76 https://github.com/torvalds/linux
77 https://github.com/flutter/flutter

Table B.6: Projects in the Initial Selection, but Excluded (Part 2)

ID	Name	Prog. Lang.	#Stars	Excl. Filter
78	system-design-primer[78]	Python	62818	(4)
79	moby[79]	Go	53239	(3)
80	kubernetes[80]	Go	52454	(3)
81	three.js[81]	JavaScript	51325	(3)
82	programming-books-zh_CN[82]	No language	49712	(1)
83	puppeteer[83]	JavaScript	48965	(2)
84	javascript-algorithms[84]	JavaScript	48676	(2)
85	swift[85]	C++	47677	(2)
86	awesome-vue[86]	No language	45190	(1)
87	build-your-own-x[87]	No Language	44566	(1)
88	rails[88]	Ruby	43032	(3)
89	elasticsearch[89]	Java	40905	(3)
90	computer-science[90]	No language	40679	(1)
91	resume.github.com[91]	JavaScript	40644	(1)
92	Front-end-Developer-Interview-Questions[92]	HTML	40626	(1)
93	the-art-of-command-line[93]	No language	40439	(1)
94	JavaGuide[94]	Java	39125	(1)
95	bitcoin[95]	C++	38234	(3)
96	material-design-icons[96]	CSS	37905	(4)
97	spring-boot[97]	Java	37697	(3)
98	every-programmer-should-know[98]	No language	37474	(1)

Prog. Lang. *stands for programming language.* ***Excl. Filter*** *stands for exclusion filter*

78 https://github.com/donnemartin/system-design-primer
79 https://github.com/moby/moby
80 https://github.com/kubernetes/kubernetes
81 https://github.com/mrdoob/three.js
82 https://github.com/justjavac/free-programming-books-zh_CN
83 https://github.com/GoogleChrome/puppeteer
84 https://github.com/trekhleb/javascript-algorithms
85 https://github.com/apple/swift
86 https://github.com/vuejs/awesome-vue
87 https://github.com/danistefanovic/build-your-own-x
88 https://github.com/rails/rails
89 https://github.com/elastic/elasticsearch
90 https://github.com/ossu/computer-science
91 https://github.com/resume/resume.github.com
92 https://github.com/h5bp/Front-end-Developer-Interview-Questions
93 https://github.com/jlevy/the-art-of-command-line
94 https://github.com/Snailclimb/JavaGuide
95 https://github.com/bitcoin/bitcoin
96 https://github.com/google/material-design-icons
97 https://github.com/spring-projects/spring-boot
98 https://github.com/mtdvio/every-programmer-should-know

Table B.7: Projects in the Initial Selection, but Excluded (Part 3)

ID	Name	Prog. Lang.	#Stars	Excl. Filter
99	shadowsocks-windows[99]	C#	36972	(4)
100	node-v0.x-archive[100]	No language	35512	(2)

Prog. Lang. *stands for programming language.* ***Excl. Filter*** *stands for exclusion filter*

1. Projects that do not have a classified programming language as the main file extension since we are interested in programming language projects;
2. Projects with less than two commits per month in the last six months, since we are interested in active community projects on GitHub;
3. Projects in which it was not possible to reconstruct at least 50% of the merge scenarios, since we are interested in projects that use the three-way merge pattern in the majority of integrations; and,
4. Balancing the programming language of projects consists of excluding less popular JavaScript projects until they are not the majority of subject projects.

In Figure 6.2 (shown in Chapter 6), we can visually see how many projects were excluded by each filter. For short, we found 15 projects that were not classified as programming language projects, 4 that were not active at the moment of our analyses, 9 that we were not possible to reconstruct at least 50% of the identified merge commits, and 6 that did not have merge conflicts at the moment of our analyses.

One interesting finding we can mention related to the excluded projects is that developers and students are using GitHub to host books or other materials to study programming supporting road-maps to grow fast in the career or to pass in code interviews.

99 https://github.com/shadowsocks/shadowsocks-windows
100 https://github.com/nodejs/node-v0.x-archive

C

Appendix

In this appendix, we present complementary material of our second empirical study presented in Chapter 4. Hence, the appendix is organised as follows. In Section C.1, we present the balancing techniques used in the study. In Section C.2, we briefly describe the machine learning classifiers used in the study. Finally, in Section C.3, we present the complete results for the predictions of our study.

C.1 Balancing Techniques Description

Below, we briefly describe the seven balancing techniques of our study:

Under-sampling balances the dataset by randomly reducing the size of the majority class until it has the size of the minority class.

Over-sampling balances the dataset by randomly increasing the size of the minority class until it has the size of the majority class.

Both-sampling is a mix of under- and over-sampling. It randomly reduces the majority class and increases the minority class until the sample has a size of around the initial majority plus the minority class divided by two.

SMOTE synthesises elements for the minority class in the vicinity of already existing elements, similar to over-sampling [62].

Borderline Synthetic Minority Oversampling TEchnique (BS) is a variant of the original SMOTE algorithm; however, in BS, samples will be detected and used to generate new synthetic samples [129].

Support Vector Machine SMOTE (SVMS) is a variant of the SMOTE algorithm that uses an SVM algorithm to detect samples for generating new synthetic samples [215].

Adasyn (Adaptive Synthetic) is an algorithm that generates synthetic data. Its greatest advantages are not copying the same minority data and generating more data for "harder to learn" examples. Adasyn is similar to SMOTE, but it generates different samples depending on a local distribution estimation of the oversampled class.

C.2 Machine Learning Classifiers Description

We provide a brief description of the three used machine learning techniques as follows:

Decision tree is a non-parametric supervised algorithm that learns from simple decision rules inferred from the data features.

Random forest is an ensemble learning method for classification that operates by constructing a multitude of decision trees at training time.

KNN algorithm is a non-parametric supervised learning method. The input consists of the k closest training examples in a data set.

C.3 Predictions

In this Section, we present the predictions for each investigated machine learning classifier (i.e., decision tree, random forest, and KNN) considering the greatest recall values obtained when tuning our models. In Tables C.1 and C.2, we present the prediction performance for the decision tree classifier. In Tables C.3 and C.4, we present the prediction performance for random forest classifier. In Tables C.5 and C.6, we present the prediction performance for KNN classifiers. We present two tables per machine learning classifier given space constraints and to better visualise it.

All these tables mentioned above follow the same structure. In the *Model* column, we describe each model considered in our study. The three options are: (i) the social model which is built using only social measures; (ii) the technical model which is built using only technical measures; and, (iii) both which stands for the model built using social and technical measures. In the *Bal.* column, we describe the balancing technique used to balance the data used in our predictions. The balance technique list and description can be found in Section C.1. In the *C* column, we show the results for safe-scenarios which are represented by "0" and conflicting scenarios which are represented by "1". In the *Ac.*, *P*, *R*, *F1*, and *AUC* columns we present the performance measures, as we did in Section 4.4.3 when answering the research questions of Chapter 4. Finally, in the *Hyper Parameter* column we presented the hyper parameters and their values to achieve the performance. As mentioned, in this Section, we present only the best performance for each subject machine learning classifier for a given model, balancing technique, and hyper parameters. The complete list of our predictions can be found in our supplementary Website [297].

Table C.1: Decision Tree Predictions (Part 1)

Model	Bal.	C	Ac.	P	R	F1	AUC	Hyper Parameters
social	Under	0	0.7	1.00	0.69	0.81	0.82	max_depth: 10, max_feature: sqrt, min_samples_leaf: 10, min_samples_split: 10, criterion: entropy, splitter: random
social	Under	1	0.7	0.14	0.95	0.24	0.82	max_depth: 10, max_feature: sqrt, min_samples_leaf: 10, min_samples_split: 10, criterion: entropy, splitter: random
social	Over	0	0.62	1.00	0.60	0.75	0.8	max_depth: 10, max_feature: auto, min_samples_leaf: 10, min_samples_split: 3, criterion: entropy, splitter: random
social	Over	1	0.62	0.12	0.99	0.21	0.8	max_depth: 10, max_feature: auto, min_samples_leaf: 10, min_samples_split: 3, criterion: entropy, splitter: random
social	Both	0	0.84	0.99	0.85	0.91	0.82	max_depth: 10, max_feature: auto, min_samples_leaf: 2, min_samples_split: 10, criterion: entropy, splitter:, best
social	Both	1	0.84	0.21	0.80	0.34	0.82	max_depth: 10, max_feature: auto, min_samples_leaf: 2, min_samples_split: 10, criterion: entropy, splitter:, best
social	SMOTE	0	0.63	1.00	0.62	0.76	0.78	max_depth: 10, max_feature: sqrt, min_samples_leaf: 1, min_samples_split: 10, criterion: entropy, splitter: random
social	SMOTE	1	0.63	0.12	0.95	0.21	0.78	max_depth: 10, max_feature: sqrt, min_samples_leaf: 1, min_samples_split: 10, criterion: entropy, splitter: random
social	BS	0	0.78	0.99	0.78	0.87	0.83	max_depth: 10, max_feature: sqrt, min_samples_leaf: 10, min_samples_split: 5, criterion: gini, splitter: random
social	BS	1	0.78	0.17	0.87	0.29	0.83	max_depth: 10, max_feature: sqrt, min_samples_leaf: 10, min_samples_split: 5, criterion: gini, splitter: random
social	SVMS	0	0.78	0.99	0.77	0.87	0.82	max_depth: 10, max_feature: auto, min_samples_leaf: 10, min_samples_split: 2, criterion: entropy, splitter: random
social	SVMS	1	0.78	0.17	0.86	0.28	0.82	max_depth: 10, max_feature: auto, min_samples_leaf: 10, min_samples_split: 2, criterion: entropy, splitter: random
social	Adasyn	0	0.67	1.00	0.65	0.79	0.81	max_depth: 10, max_feature: log2, min_samples_leaf: 10, min_samples_split: 2, criterion: entropy, splitter: random
social	Adasyn	1	0.67	0.13	0.96	0.22	0.81	max_depth: 10, max_feature: log2, min_samples_leaf: 10, min_samples_split: 2, criterion: entropy, splitter: random
tech	Under	0	0.88	1.00	0.87	0.93	0.94	max_depth: 10, max_feature: log2, min_samples_leaf: 3, min_samples_split: 5, criterion: entropy, splitter: random
tech	Under	1	0.88	0.30	1.00	0.46	0.94	max_depth: 10, max_feature: log2, min_samples_leaf: 3, min_samples_split: 5, criterion: entropy, splitter: random
tech	Over	0	0.88	1.00	0.88	0.93	0.94	max_depth: 10, max_feature: auto, min_samples_leaf: 5, min_samples_split: 5, criterion: entropy, splitter: random
tech	Over	1	0.88	0.30	1.00	0.46	0.94	max_depth: 10, max_feature: auto, min_samples_leaf: 5, min_samples_split: 5, criterion: entropy, splitter: random
tech	Both	0	0.88	1.00	0.88	0.93	0.94	max_depth: 10, max_feature: log2, min_samples_leaf: 10, min_samples_split: 3, criterion: entropy, splitter: random
tech	Both	1	0.88	0.30	1.00	0.46	0.94	max_depth: 10, max_feature: log2, min_samples_leaf: 10, min_samples_split: 3, criterion: entropy, splitter: random
tech	SMOTE	0	0.88	1.00	0.88	0.93	0.94	max_depth: 10, max_feature: auto, min_samples_leaf: 1, min_samples_split: 2, criterion: entropy, splitter: random
tech	SMOTE	1	0.88	0.30	1.00	0.46	0.94	max_depth: 10, max_feature: auto, min_samples_leaf: 1, min_samples_split: 2, criterion: entropy, splitter: random
tech	BS	0	0.88	1.00	0.88	0.93	0.94	max_depth: 10, max_feature: auto, min_samples_leaf: 2, min_samples_split: 3, criterion: entropy, splitter: random
tech	BS	1	0.88	0.30	1.00	0.46	0.94	max_depth: 10, max_feature: auto, min_samples_leaf: 2, min_samples_split: 3, criterion: entropy, splitter: random

Bal. *stands for balancing technique.* ***C*** *stands for whether the sample has merge conflicts or not.* ***Ac.*** *stands for accuracy.* ***P*** *stands for precision.* ***R*** *stands for recall.* ***F1*** *stands for F1-score.*

Table C.2: Decision Tree Predictions (Part 2)

Model	Bal.	C	Ac.	P	R	F1	AUC	Hyper Parameters
tech	SVMS	0	0.88	1.00	0.87	0.93	0.93	max_depth: 10, max_feature: sqrt, min_samples_leaf: 3, min_samples_split: 5, criterion: entropy, splitter: random
tech	SVMS	1	0.88	0.29	1.00	0.45	0.93	max_depth: 10, max_feature: sqrt, min_samples_leaf: 3, min_samples_split: 5, criterion: entropy, splitter: random
tech	Adasyn	0	0.88	1.00	0.88	0.93	0.94	max_depth: 10, max_feature: sqrt, min_samples_leaf: 3, min_samples_split: 3, criterion: entropy, splitter: random
tech	Adasyn	1	0.88	0.30	1.00	0.46	0.94	max_depth: 10, max_feature: sqrt, min_samples_leaf: 3, min_samples_split: 3, criterion: entropy, splitter: random
both	Under	0	0.86	1.00	0.86	0.92	0.93	max_depth: 10, max_feature: sqrt, min_samples_leaf: 5, min_samples_split: 10, criterion: entropy, splitter: random
both	Under	1	0.86	0.27	1.00	0.42	0.93	max_depth: 10, max_feature: sqrt, min_samples_leaf: 5, min_samples_split: 10, criterion: entropy, splitter: random
both	Over	0	0.87	1.00	0.87	0.93	0.93	max_depth: 10, max_feature: log2, min_samples_leaf: 10, min_samples_split: 2, criterion: entropy, splitter: random
both	Over	1	0.87	0.29	1.00	0.44	0.93	max_depth: 10, max_feature: log2, min_samples_leaf: 10, min_samples_split: 2, criterion: entropy, splitter: random
both	Both	0	0.9	1.00	0.89	0.94	0.94	max_depth: 10, max_feature: sqrt, min_samples_leaf: 5, min_samples_split: 10, criterion: entropy, splitter: random
both	Both	1	0.9	0.33	0.98	0.49	0.94	max_depth: 10, max_feature: sqrt, min_samples_leaf: 5, min_samples_split: 10, criterion: entropy, splitter: random
both	SMOTE	0	0.88	1.00	0.88	0.93	0.93	max_depth: 10, max_feature: log2, min_samples_leaf: 2, min_samples_split: 5, criterion: gini, splitter: random
both	SMOTE	1	0.88	0.30	0.99	0.46	0.93	max_depth: 10, max_feature: log2, min_samples_leaf: 2, min_samples_split: 5, criterion: gini, splitter: random
both	BS	0	0.88	1.00	0.88	0.93	0.93	max_depth: 10, max_feature: auto, min_samples_leaf: 5, min_samples_split: 10, criterion: entropy, splitter: random
both	BS	1	0.88	0.30	0.98	0.46	0.93	max_depth: 10, max_feature: auto, min_samples_leaf: 5, min_samples_split: 10, criterion: entropy, splitter: random
both	SVMS	0	0.88	1.00	0.87	0.93	0.93	max_depth: 10, max_feature: log2, min_samples_leaf: 10, min_samples_split: 5, criterion: entropy, splitter: random
both	SVMS	1	0.88	0.29	0.99	0.45	0.93	max_depth: 10, max_feature: log2, min_samples_leaf: 10, min_samples_split: 5, criterion: entropy, splitter: random
both	Adasyn	0	0.88	1.00	0.88	0.93	0.93	max_depth: 10, max_feature: auto, min_samples_leaf: 1, min_samples_split: 3, criterion: entropy, splitter: random
both	Adasyn	1	0.88	0.30	0.99	0.46	0.93	max_depth: 10, max_feature: auto, min_samples_leaf: 1, min_samples_split: 3, criterion: entropy, splitter: random

Bal. *stands for balancing technique.* ***C*** *stands for whether the sample has merge conflicts or not.* ***Ac.*** *stands for accuracy.* ***P*** *stands for precision.* ***R*** *stands for recall.* ***F1*** *stands for F1-score.*

Table C.3: Random Forest Predictions (Part 1)

Model	Bal.	C	Ac.	P	R	F1	AUC	Hyper Parameters
social	Under	0	0.6	1.00	0.58	0.74	0.79	max_depth: 1, max_feature: sqrt, min_samples_leaf: 2, min_ss: 3, estimator: 10, criterion: gini, warm_start: False
social	Under	1	0.6	0.11	1.00	0.20	0.79	max_depth: 1, max_feature: sqrt, min_samples_leaf: 2, min_ss: 3, estimator: 10, criterion: gini, warm_start: False
social	Over	0	0.58	1.00	0.56	0.72	0.78	max_depth: 1, max_feature: auto, min_samples_leaf: 1, min_ss: 10, estimator: 10, criterion: gini, warm_start: True
social	Over	1	0.58	0.11	1.00	0.19	0.78	max_depth: 1, max_feature: auto, min_samples_leaf: 1, min_ss: 10, estimator: 10, criterion: gini, warm_start: True
social	Both	0	0.82	0.99	0.81	0.89	0.83	max_depth: 200, max_feature: sqrt, min_samples_leaf: 10, min_ss: 10, estimator: 10, criterion: entropy, warm_start: False
social	Both	1	0.82	0.19	0.85	0.31	0.83	max_depth: 200, max_feature: sqrt, min_samples_leaf: 10, min_ss: 10, estimator: 10, criterion: entropy, warm_start: False
social	SMOTE	0	0.6	1.00	0.58	0.74	0.79	max_depth: 1, max_feature: auto, min_samples_leaf: 5, min_ss: 2, estimator: 10, criterion: entropy, warm_start: False
social	SMOTE	1	0.6	0.11	1.00	0.20	0.79	max_depth: 1, max_feature: auto, min_samples_leaf: 5, min_ss: 2, estimator: 10, criterion: entropy, warm_start: False
social	BS	0	0.44	1.00	0.41	0.59	0.71	max_depth: 1, max_feature: auto, min_samples_leaf: 1, min_ss: 3, estimator: 1, criterion: gini, warm_start: False
social	BS	1	0.44	0.08	1.00	0.15	0.71	max_depth: 1, max_feature: auto, min_samples_leaf: 1, min_ss: 3, estimator: 1, criterion: gini, warm_start: False
social	SVMS	0	0.21	1.00	0.17	0.29	0.59	max_depth: 1, max_feature: sqrt, min_samples_leaf: 1, min_ss: 3, estimator: 1, criterion: entropy, warm_start: True
social	SVMS	1	0.21	0.06	1.00	0.11	0.59	max_depth: 1, max_feature: sqrt, min_samples_leaf: 1, min_ss: 3, estimator: 1, criterion: entropy, warm_start: True
social	Adasyn	0	0.6	1.00	0.58	0.74	0.79	max_depth: 1, max_feature: sqrt, min_samples_leaf: 3, min_ss: 2, estimator: 10, criterion: gini, warm_start: False
social	Adasyn	1	0.6	0.11	1.00	0.20	0.79	max_depth: 1, max_feature: sqrt, min_samples_leaf: 3, min_ss: 2, estimator: 10, criterion: gini, warm_start: False
tech	Under	0	0.91	1.00	0.91	0.95	0.95	max_depth: 50, max_feature: auto, min_samples_leaf: 1, min_ss: 2, estimator: 50, criterion: gini, warm_start: True
tech	Under	1	0.91	0.37	1.00	0.54	0.95	max_depth: 50, max_feature: auto, min_samples_leaf: 1, min_ss: 2, estimator: 50, criterion: gini, warm_start: True
tech	Over	0	0.91	1.00	0.91	0.95	0.95	max_depth: 10, max_feature: sqrt, min_samples_leaf: 3, min_ss: 3, estimator: 10, criterion: gini, warm_start: True
tech	Over	1	0.91	0.37	1.00	0.54	0.95	max_depth: 10, max_feature: sqrt, min_samples_leaf: 3, min_ss: 3, estimator: 10, criterion: gini, warm_start: True
tech	Both	0	0.91	1.00	0.91	0.95	0.95	max_depth: 10, max_feature: auto, min_samples_leaf: 1, min_ss: 5, estimator: 50, criterion: entropy, warm_start: False
tech	Both	1	0.91	0.37	1.00	0.54	0.95	max_depth: 10, max_feature: auto, min_samples_leaf: 1, min_ss: 5, estimator: 50, criterion: entropy, warm_start: False
tech	SMOTE	0	0.92	1.00	0.91	0.95	0.95	max_depth: 10, max_feature: log2, min_samples_leaf: 10, min_ss: 3, estimator: 10, criterion: gini, warm_start: True
tech	SMOTE	1	0.92	0.38	1.00	0.55	0.95	max_depth: 10, max_feature: log2, min_samples_leaf: 10, min_ss: 3, estimator: 10, criterion: gini, warm_start: True
tech	BS	0	0.91	1.00	0.91	0.95	0.95	max_depth: 10, max_feature: auto, min_samples_leaf: 5, min_ss: 3, estimator: 10, criterion: entropy, warm_start: False
tech	BS	1	0.91	0.36	1.00	0.53	0.95	max_depth: 10, max_feature: auto, min_samples_leaf: 5, min_ss: 3, estimator: 10, criterion: entropy, warm_start: False

Bal. *stands for balancing technique.* ***C*** *stands for whether the sample has merge conflicts or not.* ***Ac.*** *stands for accuracy.* ***P*** *stands for precision.* ***R*** *stands for recall.* ***F1*** *stands for F1-score.* ***min_ss*** *stands for minimum samples split*

Table C.4: Random Forest Predictions (Part 2)

Model	Bal.	C	Ac.	P	R	F1	AUC	Hyper Parameters
tech	SVMS	0	0.9	1.00	0.90	0.95	0.95	max_depth: 1, max_feature: auto, min_samples_leaf: 2, min_ss: 3, estimator: 10, criterion: gini, warm_start: True
tech	SVMS	1	0.9	0.34	1.00	0.51	0.95	max_depth: 1, max_feature: auto, min_samples_leaf: 2, min_ss: 3, estimator: 10, criterion: gini, warm_start: True
tech	Adasyn	0	0.92	1.00	0.91	0.95	0.95	max_depth: 10, max_feature: log2, min_samples_leaf: 1, min_ss: 3, estimator: 10, criterion: gini, warm_start: True
tech	Adasyn	1	0.92	0.38	1.00	0.55	0.95	max_depth: 10, max_feature: log2, min_samples_leaf: 1, min_ss: 3, estimator: 10, criterion: gini, warm_start: True
both	Under	0	0.92	1.00	0.92	0.96	0.96	max_depth: 10, max_feature: auto, min_samples_leaf: 1, min_ss: 2, estimator: 50, criterion: gini, warm_start: False
both	Under	1	0.92	0.39	1.00	0.56	0.96	max_depth: 10, max_feature: auto, min_samples_leaf: 1, min_ss: 2, estimator: 50, criterion: gini, warm_start: False
both	Over	0	0.92	1.00	0.92	0.96	0.96	max_depth: 10, max_feature: auto, min_samples_leaf: 1, min_ss: 2, estimator: 100, criterion: entropy, warm_start: False
both	Over	1	0.92	0.39	1.00	0.56	0.96	max_depth: 10, max_feature: auto, min_samples_leaf: 1, min_ss: 2, estimator: 100, criterion: entropy, warm_start: False
both	Both	0	0.91	1.00	0.90	0.95	0.95	max_depth: 1, max_feature: log2, min_samples_leaf: 2, min_ss: 10, estimator: 10, criterion: entropy, warm_start: False
both	Both	1	0.91	0.36	1.00	0.53	0.95	max_depth: 1, max_feature: log2, min_samples_leaf: 2, min_ss: 10, estimator: 10, criterion: entropy, warm_start: False
both	SMOTE	0	0.9	1.00	0.90	0.95	0.95	max_depth: 1, max_feature: auto, min_samples_leaf: 1, min_ss: 5, estimator: 10, criterion: gini, warm_start: True
both	SMOTE	1	0.9	0.35	1.00	0.51	0.95	max_depth: 1, max_feature: auto, min_samples_leaf: 1, min_ss: 5, estimator: 10, criterion: gini, warm_start: True
both	BS	0	0.9	1.00	0.90	0.95	0.95	max_depth: 1, max_feature: auto, min_samples_leaf: 2, min_ss: 5, estimator: 10, criterion: entropy, warm_start: True
both	BS	1	0.9	0.35	1.00	0.51	0.95	max_depth: 1, max_feature: auto, min_samples_leaf: 2, min_ss: 5, estimator: 10, criterion: entropy, warm_start: True
both	SVMS	0	0.9	1.00	0.90	0.95	0.95	max_depth: 1, max_feature: auto, min_samples_leaf: 1, min_ss: 10, estimator: 10, criterion: gini, warm_start: True
both	SVMS	1	0.9	0.35	1.00	0.51	0.95	max_depth: 1, max_feature: auto, min_samples_leaf: 1, min_ss: 10, estimator: 10, criterion: gini, warm_start: True
both	Adasyn	0	0.9	1.00	0.90	0.95	0.95	max_depth: 1, max_feature: sqrt, min_samples_leaf: 2, min_ss: 5, estimator: 10, criterion: entropy, warm_start: True
both	Adasyn	1	0.9	0.35	1.00	0.51	0.95	max_depth: 1, max_feature: sqrt, min_samples_leaf: 2, min_ss: 5, estimator: 10, criterion: entropy, warm_start: True

Bal. *stands for balancing technique.* ***C*** *stands for whether the sample has merge conflicts or not.* ***Ac.*** *stands for accuracy.* ***P*** *stands for precision.* ***R*** *stands for recall.* ***F1*** *stands for F1-score.* ***min_ss*** *stands for minimum samples split*

Table C.5: KNN Predictions (Part 1)

Model	Bal.	C	Ac.	P	R	F1	AUC	Hyper Parameters
social	Under	0	0.71	0.99	0.70	0.82	0.81	n_neighbors: 200, weights: uniform, metrics: minkowski, algorithm: auto
social	Under	1	0.71	0.14	0.93	0.24	0.81	n_neighbors: 200, weights: uniform, metrics: minkowski, algorithm: auto
social	Over	0	0.74	0.99	0.73	0.85	0.83	n_neighbors: 150, weights: uniform, metrics: minkowski, algorithm: auto
social	Over	1	0.74	0.15	0.92	0.26	0.83	n_neighbors: 150, weights: uniform, metrics: minkowski, algorithm: auto
social	Both	0	0.79	0.99	0.78	0.88	0.82	n_neighbors: 50, weights: uniform, metrics: manhattan, algorithm: kd_tree
social	Both	1	0.79	0.17	0.85	0.29	0.82	n_neighbors: 50, weights: uniform, metrics: manhattan, algorithm: kd_tree
social	SMOTE	0	0.75	0.99	0.74	0.85	0.83	n_neighbors: 150, weights: uniform, metrics: manhattan, algorithm: auto
social	SMOTE	1	0.75	0.15	0.92	0.26	0.83	n_neighbors: 150, weights: uniform, metrics: manhattan, algorithm: auto
social	BS	0	0.82	0.99	0.82	0.89	0.79	n_neighbors: 200, weights: uniform, metrics: manhattan, algorithm: auto
social	BS	1	0.82	0.18	0.77	0.29	0.79	n_neighbors: 200, weights: uniform, metrics: manhattan, algorithm: auto
social	SVMS	0	0.85	0.98	0.85	0.91	0.79	n_neighbors: 300, weights: uniform, metrics: minkowski, algorithm: ball_tree
social	SVMS	1	0.85	0.21	0.73	0.32	0.79	n_neighbors: 300, weights: uniform, metrics: minkowski, algorithm: ball_tree
social	Adasyn	0	0.73	1.00	0.72	0.83	0.83	n_neighbors: 200, weights: uniform, metrics: minkowski, algorithm: ball_tree
social	Adasyn	1	0.73	0.15	0.94	0.26	0.83	n_neighbors: 200, weights: uniform, metrics: minkowski, algorithm: ball_tree
tech	Under	0	0.77	0.99	0.76	0.86	0.82	n_neighbors: 20, weights: distance, metrics: manhattan, algorithm: auto
tech	Under	1	0.77	0.17	0.89	0.28	0.82	n_neighbors: 20, weights: distance, metrics: manhattan, algorithm: auto
tech	Over	0	0.8	0.99	0.79	0.88	0.84	n_neighbors: 200, weights: distance, metrics: manhattan, algorithm: auto
tech	Over	1	0.8	0.19	0.88	0.31	0.84	n_neighbors: 200, weights: distance, metrics: manhattan, algorithm: auto
tech	Both	0	0.87	0.98	0.88	0.93	0.78	n_neighbors: 20, weights: distance, metrics: manhattan, algorithm: auto
tech	Both	1	0.87	0.23	0.68	0.34	0.78	n_neighbors: 20, weights: distance, metrics: manhattan, algorithm: auto
tech	SMOTE	0	0.78	0.99	0.78	0.87	0.84	n_neighbors: 200, weights: distance, metrics: manhattan, algorithm: auto
tech	SMOTE	1	0.78	0.18	0.91	0.30	0.84	n_neighbors: 200, weights: distance, metrics: manhattan, algorithm: auto
tech	BS	0	0.92	0.97	0.94	0.95	0.7	n_neighbors: 1, weights: distance, metrics: manhattan, algorithm: auto
tech	BS	1	0.92	0.29	0.45	0.35	0.7	n_neighbors: 1, weights: distance, metrics: manhattan, algorithm: auto

Bal. *stands for balancing technique.* ***C*** *stands for whether the sample has merge conflicts or not.* ***Ac.*** *stands for accuracy.* ***P*** *stands for precision.*

R *stands for recall.* ***F1*** *stands for F1-score.*

Table C.6: KNN Predictions (Part 2)

Model	Bal.	C	Ac.	P	R	F1	AUC	Hyper Parameters
tech	SVMS	0	0.87	0.98	0.88	0.93	0.78	n_neighbors: 300, weights: distance, metrics: manhattan, algorithm: auto
tech	SVMS	1	0.87	0.23	0.68	0.34	0.78	n_neighbors: 300, weights: distance, metrics: manhattan, algorithm: auto
tech	Adasyn	0	0.76	1.00	0.75	0.85	0.84	n_neighbors: 300, weights: distance, metrics: manhattan, algorithm: auto
tech	Adasyn	1	0.76	0.16	0.93	0.28	0.84	n_neighbors: 300, weights: distance, metrics: manhattan, algorithm: auto
both	Under	0	0.78	0.99	0.77	0.87	0.82	n_neighbors: 10, weights: distance, metrics: manhattan, algorithm: ball_tree
both	Under	1	0.78	0.17	0.87	0.29	0.82	n_neighbors: 10, weights: distance, metrics: manhattan, algorithm: ball_tree
both	Over	0	0.8	0.99	0.80	0.88	0.84	n_neighbors: 150, weights: distance, metrics: manhattan, algorithm: kd_tree
both	Over	1	0.8	0.19	0.88	0.31	0.84	n_neighbors: 150, weights: distance, metrics: manhattan, algorithm: kd_tree
both	Both	0	0.87	0.98	0.88	0.93	0.79	n_neighbors: 20, weights: distance, metrics: manhattan, algorithm: kd_tree
both	Both	1	0.87	0.24	0.70	0.36	0.79	n_neighbors: 20, weights: distance, metrics: manhattan, algorithm: kd_tree
both	SMOTE	0	0.79	0.99	0.79	0.88	0.84	n_neighbors: 200, weights: distance, metrics: manhattan, algorithm: auto
both	SMOTE	1	0.79	0.18	0.89	0.30	0.84	n_neighbors: 200, weights: distance, metrics: manhattan, algorithm: auto
both	BS	0	0.82	0.99	0.82	0.90	0.83	n_neighbors: 150, weights: distance, metrics: manhattan, algorithm: auto
both	BS	1	0.82	0.20	0.83	0.32	0.83	n_neighbors: 150, weights: distance, metrics: manhattan, algorithm: auto
both	SVMS	0	0.86	0.98	0.86	0.92	0.8	n_neighbors: 150, weights: distance, metrics: manhattan, algorithm: auto
both	SVMS	1	0.86	0.22	0.74	0.34	0.8	n_neighbors: 150, weights: distance, metrics: manhattan, algorithm: auto
both	Adasyn	0	0.74	0.99	0.73	0.84	0.82	n_neighbors: 300, weights: uniform, metrics: manhattan, algorithm: auto
both	Adasyn	1	0.74	0.16	0.92	0.27	0.82	n_neighbors: 300, weights: uniform, metrics: manhattan, algorithm: auto

Bal. *stands for balancing technique.* ***C*** *stands for whether the sample has merge conflicts or not.* ***Ac.*** *stands for accuracy.* ***P*** *stands for precision.*

R *stands for recall.* ***F1*** *stands for F1-score.*

D

Appendix

In this appendix, we present the survey we sent to developers of subject projects. As mentioned in Chapter 6, we followed a learn-and-improve approach, in which we adapt questions based on the participants' feedback. In Figure D.1, we show a screenshot of our invitation email. To make it more personalised, we substituted the $<$ *developer name* $>$ by the name in the commit of the subject developer. When the developer clicks on the *Answer the survey now* button, she is redirected to a Google Forms to answer the survey.

In Section D.1, we present the questions from the initial version of our survey. In Section D.2, we present the final version of our survey. It is important to highlight that most of the answers of developers from subject projects came from the second survey.

d.1 Survey Initial Version

The initial version of our survey consists of 6 parts named *Introduction*, *Background*, *Contribution style*, *Linking Social and Technical Assets Merge Conflict Resolution*, *Email information*, and *Thanks*. In the introduction part we explain the context of the research, inform the developer of the estimated time to conclude the survey and inform how the survey is organised. The only question in the introduction part is shown in Table D.1.

In Table D.2, we show the questions related to the background part of the survey. In this part we are interested in getting to know the developer's background. Hence, we ask for the number of years of experience, sector, position, level of education, and gender.

In Table D.3, we ask questions related to the developer's contribution style. For instance, *n projects that you occasionally contribute to, are you more worried about addressing an issue than on the impact of your changes?*

In Tables D.4 and D.5, we show questions related to how developers link social and technical assets. Hence, we would like to know the number of developers in the team, how developers use GitHub events (e.g., issues, pull-requests, labels), how they normally commit their changes, and whether and how often they link GitHub events with commits and vice-versa.

In Table D.6, we show the questions related to patterns and procedures developers follow to resolve merge conflicts. Note that we also ask questions related to the difficulty of resolving merge conflicts to cross-validate the dependent variable used in Chapter 6.

The e-mail information part consists of an open-ended question to the developer optionally leaving her email to receive a copy of our study. We highlight that we will not link your email address to your survey responses, nor will we publish your e-mail address in

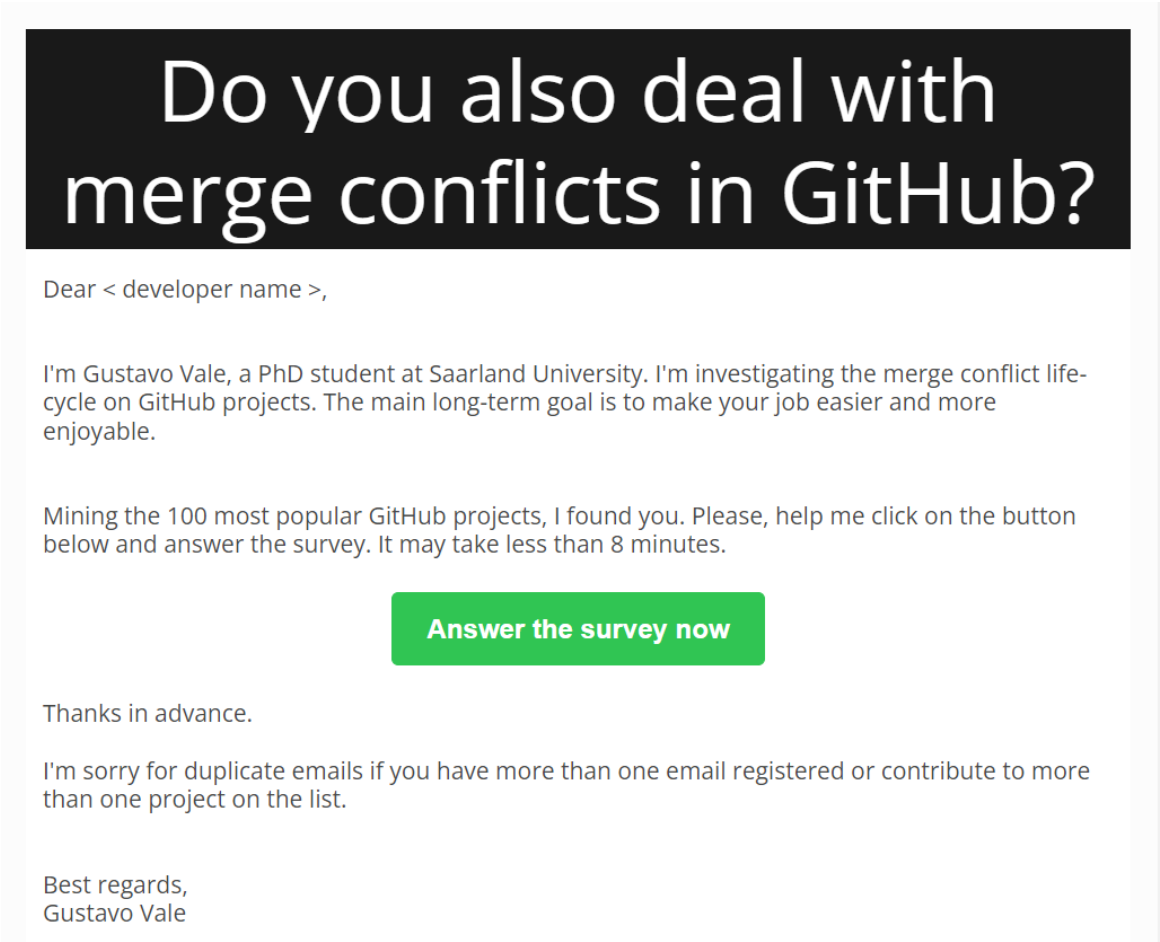

Figure D.1: Invitation Email Sent to Developers of Subject Projects

Table D.1: Survey Questions - Initial Version - Introduction

Question	**Options**
1. We plan to include the results of this survey in a scientific publication. Please select your choice below. Selecting the "Yes" option below indicates that: i) you have read and understood the above information, ii) you voluntarily agree to participate, iii) you agree that your answers can be used for research purposes, and iv) you are at least 18 years old. If you do not wish to participate in the research study, please decline participation by selecting "No". I consent to participate in this research study	☐ Yes ☐ No

Table D.2: Survey Questions - Initial Version - Background

Question	**Options**
2. How many years experience you have in software development?	☐ less than one ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 8 ☐ 10 ☐ more than 10
3. How many years experience you have in software development?	☐ Information Technology ☐ Automation ☐ Engineering ☐ Education ☐ Other: _____
4. What of these roles best fits your current position?	☐ Software Developer ☐ Software Architect ☐ Web Developer ☐ Software Tester ☐ Android Developer ☐ Manager ☐ Other: _____
5. What is your level of education?	☐ Less than high school ☐ Technical or high school ☐ Bachelor ☐ Master ☐ PhD
6. What is your gender?	☐ Male ☐ Female ☐ Prefer not to say

Table D.3: Survey Questions - Initial Version - Contribution-Style

Question	**Options**
7. In projects that you are the owner of the repository or among the top three developers in number of contributions, are you more worried about addressing an issue than on the impact of your changes?	strongly agree → ☐ ☐ ☐ ☐ ☐ ← strongly disagree
8. In projects that you occasionally contribute to, are you more worried about addressing an issue than on the impact of your changes?	strongly agree → ☐ ☐ ☐ ☐ ☐ ← strongly disagree
9. When you are going to integrate branches do you care more about arising merge conflicts than when others are going to integrate the code?	strongly agree → ☐ ☐ ☐ ☐ ☐ ← strongly disagree
10. Do you agree that it is better to address an issue fast (e.g., one hour) with unnecessary code changes than take more time (e.g., four hours) and change only the necessary code?	strongly agree → ☐ ☐ ☐ ☐ ☐ ← strongly disagree
11. Do you agree that it is better to contribute often to the project and often cause merge conflicts than contribute occasionally and rarely cause merge conflicts?	strongly agree → ☐ ☐ ☐ ☐ ☐ ← strongly disagree

Table D.4: Survey Questions - Initial Version - Linking Social and Technical Assets (Part 1)

Question	**Options**
12. How many members compose your team?	☐ 0 ☐ Less than 10 ☐ 10 - 50 ☐ 50 - 100 ☐ more than 100
13. While solving an issue, do you normally refer to this issue in the pull request description or body?	☐ Never ☐ Occasionally ☐ Normally ☐ Always ☐ Other: _____
14. Why do you think it is important to link issues with pull requests?	Open-ended question
15. When do you ignore the creation of this link among issues and pull requests?	Open-ended question
16. Before closing an issue that is solved, do you explicitly describe which pull request solves the problem?	☐ Never ☐ Occasionally ☐ Normally ☐ Always ☐ Other: _____
17. Why do you think it is important to explicitly mention the pull request that solves an issue?	Open-ended question
18.When do you ignore the creation of an explicit link among the pull request that solves an issue?	Open-ended question
19. Do you describe in the body of an issue about similar issues previously addressed?	☐ Never ☐ Occasionally ☐ Normally ☐ Always ☐ Other: _____
20. Why do you think it is important to explicitly mention similar or related issues?	Open-ended question
21. Do you explicitly refer to the issue you are addressing in the commit message?	☐ Never ☐ Occasionally ☐ Normally ☐ Always ☐ Other: _____
22. When do you ignore to explicitly refer to the issue in the commit message?	Open-ended question

Table D.5: Survey Questions - Initial Version - Linking Social and Technical Assets (Part 2)

Question	**Options**
23. In the issue body or description, do you explicitly refer the commit hash related to it?	☐ Never ☐ Occasionally ☐ Normally ☐ Always ☐ Other: _____
24. When do you ignore to explicitly refer to the commit hash in the issue body?	Open-ended question
25. Which type of labels do you use?	☐ Never ☐ Occasionally ☐ Normally ☐ Always ☐ Other: _____
26. Do you assign labels on GitHub?	☐ Never ☐ Occasionally ☐ Normally ☐ Always ☐ Other: _____
27. When do you think that it is important to link issues (including pull requests) with labels? Please give details when possible.	☐ Never ☐ Occasionally ☐ Normally ☐ Always ☐ Other: _____
28. Does your project have contribution rules that recommend links between technical (e.g., commits) and social assets (e.g., issues and labels)?	☐ No ☐ Yes, but it is optional ☐ Yes and it is mandatory

Table D.6: Survey Questions - Initial Version - Merge Conflict Resolution

Question	**Options**
29. Which metrics do you think that can automatically measure the difficulty of resolving a merge conflict?	Open-ended question
30. Do you agree that time is normally a good metric to measure the difficulty of a task? In other words, the more time someone needs to complete a task, the more difficult the task is.	strongly agree → ☐ ☐ ☐ ☐ ☐ ← strongly disagree
31. In merge scenarios with conflicts, do you agree that the time between the last commit in the working branch and the merge commit is often used to resolve the conflicts?	strongly agree → ☐ ☐ ☐ ☐ ☐ ← strongly disagree
32. Do you normally merge your changes after finishing to address an issue (e.g., developing a new feature or fixing a bug)	strongly agree → ☐ ☐ ☐ ☐ ☐ ← strongly disagree
33. Do you agree that you often have to look at non-conflicting code to resolve merge conflicts?	strongly agree → ☐ ☐ ☐ ☐ ☐ ← strongly disagree
34. Do you agree that you often have to change non-conflicting code to resolve merge conflicts?	strongly agree → ☐ ☐ ☐ ☐ ☐ ← strongly disagree

any which way. Finally, in the thanks part, we thank the developer for participating in our survey and display a figure with the word thanks in several languages.

D.2 Survey Final Version (Most Answered Version)

The final version of our survey consists of 4 parts (*Introduction*, *Merge Conflict Resolution*, and *Email information*, and *Thanks*). The *background* and *linking social and technical assets* parts from the initial version were excluded. Except from the *merge conflict resolution* part, the other remaining parts are basically the same. In Table D.7, we present the main questions of the final version of our survey.

The mapping of the answers from the initial version to the final version to the merge conflict resolution part is mapped as follows:

- Open-ended question: "*Which metrics do you think that can automatically measure the difficulty of resolving a merge conflict?*" became "*How do you think someone could estimate how hard a merge conflict is?*"
- Close-ended question: "*Do you agree that time is normally a good metric to measure the difficulty of a task? In other words, the more time someone needs to complete a task, the more difficult the task is.*" became "*The more time it takes to resolve a conflict, the more difficult the conflict.*"
- Close-ended question: "*In merge scenarios with conflicts, do you agree that the time between the last commit in the working branch and the merge commit is often used to resolve the conflicts?*" became "*I merge my changes right after addressing an issue.*"
- Close-ended question: "*Do you normally merge your changes after finishing to address an issue (e.g., developing a new feature or fixing a bug)*" became "*I resolve merge conflicts right after they occur.*"
- Close-ended question: "*Do you agree that you often have to look at non-conflicting code to resolve merge conflicts?*" became "*I look at non-conflicting changes to resolve conflicts.*"
- Close-ended question: "*Do you agree that you often have to change non-conflicting code to resolve merge conflicts?*" became "*I change non-conflicting code to resolve merge conflicts and avoid introducing unexpected behaviour to the project.*"
- Open-ended question: "*If you wish, add a commentary about of your experience of dealing with merge conflicts.*" was added in the final version.

We are confident that the final version better expresses what we intend to ask for the developers and the survey is much more concise and interesting to hold their attention.

Table D.7: Main Survey Questions - Final Version

Question	**Options**
1. How do you think someone could estimate how hard a merge conflict is?	Open-ended question
2. The more time it takes to resolve, the more difficult the conflict.	Nearly always true → ☐ ☐ ☐ ☐ ☐ ← Hardly ever true
3. I merge my changes right after addressing an issue.	Nearly always true → ☐ ☐ ☐ ☐ ☐ ← Hardly ever true
4. I resolve merge conflicts right after they occur.	Nearly always true → ☐ ☐ ☐ ☐ ☐ ← Hardly ever true
5. I look at non-conflicting changes to resolve conflicts.	Nearly always true → ☐ ☐ ☐ ☐ ☐ ← Hardly ever true
6. I change non-conflicting code to resolve merge conflicts and avoid introducing unexpected behaviour to the project	Nearly always true → ☐ ☐ ☐ ☐ ☐ ← Hardly ever true
7. If you wish, add a commentary about of your experience of dealing with merge conflicts.	Open-ended question

Bibliography

[1] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. "Analyzing conflict predictors in open-source Java projects." In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. Gothenburg, Sweden: ACM, 2018, pp. 576–586. ISBN: 9781450357166.

[2] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. "Understanding semi-structured merge conflict characteristics in open-source Java projects." In: *Empirical Software Engineering (EMSE)* 23.4 (2018), pp. 2051–2085. ISSN: 1573-7616.

[3] Bram Adams and Shane McIntosh. "Modern Release Engineering in a Nutshell – Why Researchers Should Care." In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 78–90.

[4] Evan Adams, Wayne Gramlich, Steven S. Muchnick, and Soren Tirfing. "SunPro engineering a practical program development environment." In: *Advanced Programming Environments*. Springer, 1986, pp. 86–96. ISBN: 978-3-540-47347-3.

[5] Iftekhar Ahmed, Caius Brindescu, Umme Ayda Mannan, Carlos Jensen, and Anita Sarma. "An Empirical Examination of the Relationship between Code Smells and Merge Conflicts." In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Markham, Ontario, Canada: IEEE, 2017, pp. 58–67. ISBN: 978-1-5090-4039-1.

[6] Ban Al-Ani, Erik Trainer, Roger Ripley, Anita Sarma, André van der Hoek, and David Redmiles. "Continuous Coordination within the Context of Cooperative and Human Aspects of Software Engineering." In: *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2008, 1–4. ISBN: 9781605580395.

[7] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I Maletic. "What's a typical commit? a characterization of open source software repositories." In: *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE, 2008, pp. 182–191.

[8] Luís Amaral, Marcos C. Oliveira, Welder Luz, José Fortes, Rodrigo Bonifácio, Daniel Alencar, Eduardo Monteiro, Gustavo Pinto, and David Lo. "How (Not) to Find Bugs: The Interplay between Merge Conflicts, Co-Changes, and Bugs." In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 441–452. ISBN: 978-1-7281-5619-4.

[9] Kenneth M. Anderson and Niels Olof Bouvin. "Supporting Project Awareness on the WWW with the IScent Framework." In: *SIGGROUP Bulletin* 21.3 (2000), 16–20.

[10] Sven Apel, Olaf Leßenich, and Christian Lengauer. "Structured merge with auto-tuning: balancing precision and performance." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2012, pp. 120–129.

[11] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. "Semistructured Merge: Rethinking Merge in Revision Control Systems." In: *Proceedings of the Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE)*. Szeged, Hungary: ACM, 2011, 190–200. ISBN: 9781450304436.

[12] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. "JDiff: A differencing technique and tool for object-oriented programs." In: *Automated Software Engineering* 14.1 (2007), pp. 3–36.

[13] Wolfgang Appelt. "WWW Based Collaboration with the BSCW System." In: *Proceedings of the International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*. Springer, 1999, pp. 66–78.

[14] Brad Appleton, Steve Berczuk, Ralph Cabrera, and Robert Orenstein. "Streamed Lines: Branching Patterns for Parallel Software Development." In: *Proceedings of the Pattern Languages of Programs Conference (PLoP)*. Vol. 98. ACM, 1998, p. 14.

[15] Jorge Aranda and Gina Venolia. "The secret life of bugs: Going past the errors and omissions in software repositories." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 298–308.

[16] Ritu Arora, Sanjay Goel, and Ravi Kant Mittal. "Using dependency graphs to support collaboration over GitHub: The Neo4j graph database approach." In: *Proceedings of the International Conference on Contemporary Computing (IC3)*. IEEE, 2016, pp. 1–7.

[17] Ritu Arora, Sanjay Goel, and Ravi Kant Mittal. "Supporting collaborative software development over GitHub." In: *Software - Practice and Experience* 47.10 (2017), pp. 1393–1416.

[18] Ritu Arora, Anand Wani, Ankur Vineet, Bhavik Dhandhalya, Yashvardhan Sharma, and Sanjay Goel. "Continuous conflict prediction during collaborative software development: A step-before continuous integration." In: *Proceedings of the International Conference on Software Engineering and Information Management (ICSIM)*. Sydney, NSW, Australia: ACM, 2020, pp. 105–109. ISBN: 9781450376907.

[19] Dimitar Asenov, Balz Guenat, Peter Müller, and Martin Otth. "Precise Version Control of Trees with Line-Based Version Control Systems." In: *Fundamental Approaches to Software Engineering*. Springer, 2017, pp. 152–169.

[20] Ulf Asklund. "Identifying Conflicts During Structural Merge." In: *Proceedings of the Nordic Workshop on Programming Environment Research (NWPER)* (1994), pp. 231–242.

[21] Jae Young Bang, Daniel Popescu, George Edwards, Nenad Medvidovic, Naveen Kulkarni, Girish M. Rama, and Srinivas Padmanabhuni. "CoDesign: a highly extensible collaborative software modeling framework." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 243–246.

[22] Titus Barik, Kevin Lubick, and Emerson Murphy-Hill. "Commit Bubbles." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. Vol. 2. IEEE, 2015, pp. 631–634.

[23] Matthias Barkowsky and Holger Giese. "Towards Development with Multi-version Models: Detecting Merge Conflicts and Checking Well-Formedness." In: *Proceedings of the International Conference on Graph Transformation (ICGT)*. Springer, 2022, pp. 118–136. ISBN: 978-3-031-09843-7.

[24] Victor Basili and H. Dieter Rombach. "The TAME project: Towards improvement-oriented software environments." In: *Transactions on Software Engineering (TSE)* 14.6 (1988), pp. 758–773.

[25] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. "Codebook: Discovering and Exploiting Relationships in Software Repositories." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2010, 125–134. ISBN: 9781605587196.

[26] James "Bo" Begole, John C. Tang, Randall B. Smith, and Nicole Yankelovich. "Work rhythms: analyzing visualizations of awareness histories of distributed groups." In: *Proceedings of the Conference on Computer supported cooperative work (CSCW)*. ACM, 2002, pp. 334–343.

[27] Lasse Bergroth, Harri Hakonen, and Timo Raita. "A survey of longest common subsequence algorithms." In: *Proceedings International Symposium on String Processing and Information Retrieval (SPIRE)*. IEEE, 2000, pp. 39–48.

[28] Brian Berliner. "CVS II: Parallelizing Software Development." In: *Proceedings of the Advanced Computing Systems Professional and Technical Association (USENIX)* (1990), pp. 22–26.

[29] Valdis Berzins. "Software merge: Models and methods for combining changes to programs." In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Springer, 1991, pp. 229–250.

[30] Valdis Berzins. "Software merge: semantics of combining changes to programs." In: *Transactions on Programming Languages and Systems* 16.6 (1994), pp. 1875–1903.

[31] Jacob T. Biehl, William T. Baker, Brian P. Bailey, Desney S. Tan, Kori M. Inkpen, and Mary Czerwinski. "Impromptu: A New Interaction Framework for Supporting Collaboration in Multiple Display Environments and Its Field Evaluation for Co-Located Software Development." In: *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. ACM, 2008, 939–948. ISBN: 9781605580111.

[32] Jacob Biehl, Mary Czerwinski, Greg Smith, George Robertson, and Brian Bailey. "FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams." In: *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. ACM, 2007, pp. 1313–1322.

[33] David Binkley, Susan Horwitz, and Thomas Reps. "Program integration for languages with procedure calls." In: *Transactions on Software Engineering and Methodology (TOSEM)* 4.1 (1995), pp. 3–35.

[34] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. "Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista." In: *Communications of the ACM* 52.8 (2009), 85–93.

[35] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. "Don't Touch My Code! Examining the Effects of Ownership on Software Quality." In: *Proceedings of the SIGSOFT Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, 4–14. ISBN: 9781450304436.

[36] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. "Latent Social Structure in Open Source Projects." In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2008, 24–35. ISBN: 9781595939951.

[37] Christian Bird and Thomas Zimmermann. "Assessing the value of branches with what-if analysis." In: *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012, pp. 1–11.

[38] Walter R. Bischofberger, Thomas Kofler, K.-U. Matzel, and B. Schaffer. "Computer supported cooperative software engineering with Beyond-Sniff." In: *Proceedings Software Engineering Environments*. IEEE, 1995, pp. 135–143.

[39] Wai Fong Boh, Sandra A. Slaughter, and J. Alberto Espinosa. "Learning from experience in software development: A multilevel analysis." In: *Management Science* 53.8 (2007), pp. 1315–1331.

[40] Hudson Borges and Marco Tulio Valente. "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform." In: *Journal of Systems and Software (JSS)* 146 (2018), pp. 112–129.

[41] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. "An empirical investigation into merge conflicts and their effect on software quality." In: *Empirical Software Engineering (EMSE)* 25.1 (2020), pp. 562–590. ISSN: 1573-7616.

[42] Caius Brindescu, Iftekhar Ahmed, Rafael Leano, and Anita Sarma. "Planning for untangling: Predicting the difficulty of merge conflicts." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. Seoul, South Korea: ACM, 2020, pp. 801–811. ISBN: 978-1-4503-7121-6.

[43] Caius Brindescu, Yenifer Ramirez Anita Sarma, and Carlos Jensen. "Lifting the Curtain on Merge Conflict Resolution: A Sensemaking Perspective." In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 534–545. ISBN: 978-1-7281-5619-4.

[44] Petra Brosch, Uwe Egly, Sebastian Gabmeyer, Gerti Kappel, Martina Seidl, Hans Tompits, Magdalena Widl, and Manuel Wimmer. "Towards Semantics-Aware Merge Support in Optimistic Model Versioning." In: *Models in Software Engineering*. Springer, 2012, pp. 246–256.

[45] Bernd Bruegge, Andrea De Lucia, Fausto Fasano, and Genoveffa Tortora. "Supporting Distributed Software Development with fine-grained Artefact Management." In: *Proceedings of the International Conference on Global Software Engineering (ICGSE)*. IEEE, 2006, pp. 213–222.

[46] Bernd Bruegge and Allen A. Dutoit. *Object-Oriented Software Engineering; Conquering Complex and Changing Systems*. Prentice Hall PTR, 1999. ISBN: 0134897250.

[47] Bernd Bruegge, Allen H. Dutoit, and Timo Wolf. "Sysiphus: Enabling informal collaboration in global software development." In: *Proceedings of the International Conference on Global Software Engineering (ICGSE)*. IEEE, 2006, pp. 139–148.

[48] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. "Crystal: precise and unobtrusive conflict warnings." In: *Proceedings of the SIGSOFT symposium and the European conference on Foundations of software engineering (ESEC/FSE)*. ACM, 2011, pp. 444–447.

[49] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. "Proactive Detection of Collaboration Conflicts." In: *Proceedings of the SIGSOFT Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE)*. Szeged, Hungary: ACM, 2011, 168–178. ISBN: 9781450304436.

[50] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. "Early Detection of Collaboration Conflicts and Risks." In: *Transactions on Software Engineering (TSE)* 39.10 (2013), pp. 1358–1375. ISSN: 1939-3520.

[51] Jim Buffenbarger. "Syntactic software merging." In: *Lecture Notes in Computer Science*. Springer, 1995, pp. 153–172.

[52] John Businge, Simon Kawuma, Engineer Bainomugisha, Foutse Khomh, and Evarist Nabaasa. "Code Authorship and Fault-Proneness of Open-Source Android Applications: An Empirical Study." In: *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. ACM, 2017, 33–42.

[53] Sebastian Böcker, David Bryant, Andreas W.M. Dress, and Mike A. Steel. "Algorithmic Aspects of Tree Amalgamation." In: *Journal of Algorithms* 37.2 (2000), pp. 522–537.

[54] Michelle Cart and Jean Ferrie. "Asynchronous reconciliation based on operational transformation for p2p collaborative environments." In: *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*. IEEE, 2007, pp. 127–138.

[55] Luís Carvalho and João Costa Seco. "Deep Semantic Versioning for Evolution and Variability." In: *Proceedings of the International Symposium on Principles and Practice of Declarative Programming (PPDP)*. Tallinn, Estonia: ACM, 2021, pp. 1–13. ISBN: 9781450386890.

[56] Marcelo Cataldo and James D. Herbsleb. "Factors leading to integration failures in global feature-oriented development." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 161–170.

[57] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. "Identification of coordination requirements." In: *Proceedings of the conference on Computer supported cooperative work (CSCW)*. ACM, 2006, pp. 353–362.

[58] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. "Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment." In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015.

[59] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. "Evaluating and Improving Semistructured Merge." In: *Proceedings of the ACM on Programming Languages (OOPSLA)* 1 (2017), pp. 1–27.

[60] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. "The impact of structure on software merging: Semistructured versus structured merge." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. San Diego, California: IEEE, 2019, pp. 1002–1013. ISBN: 9781728125084.

[61] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. "Change detection in hierarchically structured information." In: *ACM SIGMOD Record* 25.2 (1996), pp. 493–504.

[62] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. "SMOTE: synthetic minority over-sampling technique." In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357.

[63] Mark C. Chu-Carroll and Sara Sprenkle. "Coven: Brewing Better Collaboration through Software Configuration Management." In: *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2000, 88–97. ISBN: 1581132050.

[64] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[65] Catarina Costa, Jair Figueiredo, Leonardo Murta, and Anita Sarma. "TIPMerge: Recommending Experts for Integrating Changes across Branches." In: *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*. Seattle, WA, USA: ACM, 2016, 523–534. ISBN: 9781450342186.

[66] Catarina Costa, Jair Figueiredo, Joao Felipe Pimentel, Anita Sarma, and Leonardo Murta. "Recommending Participants for Collaborative Merge Sessions." In: *Transactions on Software Engineering (TSE)* 47.6 (2021), pp. 1198–1210. ISSN: 00985589.

[67] Catarina Costa, José JC Figueiredo, Gleiph Ghiotto, and Leonardo Murta. "Characterizing the problem of developers' assignment for merging branches." In: *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* 24.10 (2014), pp. 1489–1508. ISSN: 0218-1940.

[68] Catarina Costa, Jose Menezes, Bruno Trindade, and Rodrigo Santos. "Factors that Affect Merge Conflicts: A Software Developers' Perspective." In: *Proceedings of the Brazilian Symposium on Software Engineering (SBES)*. ACM, 2021, pp. 233–242. ISBN: 9781450390613.

[69] Catarina Costa and Leonardo Murta. "Version Control in Distributed Software Development: A Systematic Mapping Study." In: *Proceedigns of the International Conference on Global Software Engineering (ICGSE)*. IEEE, 2013, pp. 90–99.

[70] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. "Core and Periphery in Free/Libre and Open Source Software Team Communications." In: *Proceedings of the Annual Hawaii International Conference on System Sciences (HICSS)*. Vol. 6. IEEE, 2006, 118a–118a.

[71] Davor Cubranic and Gail C. Murphy. "Hipikat: recommending pertinent software development artifacts." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2003, pp. 408–418.

[72] Bill Curtis. "Insights from empirical studies of the software design process." In: *Future Generation Computer Systems* 7.2 (1992), pp. 139–149. ISSN: 0167-739X.

[73] Bill Curtis, Herb Krasner, and Neil Iscoe. "A Field Study of the Software Design Process for Large Systems." In: *Communications of the ACM* 31.11 (1988), 1268–1287.

[74] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and Joao Moisakis. "Detecting Semantic Conflicts via Automated Behavior Change Detection." In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 174–184. ISBN: 978-1-7281-5619-4.

[75] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. "Social coding in github: transparency and collaboration in an open software repository." In: *Proceedings of the Computer Supported Cooperative Work (CSCW)*. ACM, 2012, pp. 1277–1286.

[76] David A. Luqi Dampier and Valdis Berzins. "Automated merging of software prototypes." In: *Journal of Systems Integration* 4.1 (1994), pp. 33–49.

[77] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. "Adams: Advanced artefact management system." In: *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE. 2006, 2–pp.

[78] Andrea De Lucia, Fausto Fasano, Giuseppe Scanniello, and Genny Tortora. "Enhancing collaborative synchronous UML modelling with fine-grained versioning of software artefacts." In: *Journal of Visual Languages & Computing* 18.5 (2007), pp. 492–503.

[79] Rafael Maiani De Mello and Guilherme Horta Travassos. "Surveys in software engineering: Identifying representative samples." In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2016, pp. 1–6.

[80] Prasun Dewan. "Dimensions of Tools for Detecting Software Conflicts." In: *Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE)*. ACM, 2008, 21–25.

[81] Prasun Dewan and Rajesh Hegde. "Semi-synchronous conflict detection and resolution in asynchronous software development." In: *Proceedings of the European Conference on Computer-Supported Cooperative Work (ECSCW)*. Springer, 2007, pp. 159–178.

[82] Klissiomara Dias, Paulo Borba, and Marcos Barreto. "Understanding predictive factors for merge conflicts." In: *Information and Software Technology* 121 (2020), p. 106256. ISSN: 0950-5849.

[83] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. "Untangling fine-grained code changes." In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 341–350.

[84] Martín Dias, Guillermo Polito, Damien Cassou, and Stéphane Ducasse. "Deltaimpactfinder: Assessing semantic merge conflicts with dependency analysis." In: *Proceedings of the International Workshop on Smalltalk Technologies (IWST)*. Brescia, Italy: ACM, 2015, pp. 1–6. ISBN: 9781450338578.

[85] Kay Dickersin, Yuan-I Min, and Curtis L Meinert. "Factors Influencing Publication of Research Results: Follow-up of Applications Submitted to Two Institutional Review Boards." In: *Journal of the American Medical Association (JAMA)* 267.3 (1992), pp. 374–378.

[86] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. "Effective software merging in the presence of object-oriented refactorings." In: *Transactions on Software Engineering (TSE)* 34.3 (2008), pp. 321–335. ISSN: 00985589.

[87] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. "Refactoring-aware configuration management for object-oriented programs." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 427–436. ISBN: 978-0-7695-2828-1.

[88] Danny Dig, Tien N. Nguyen, Kashif Manzoor, and Ralph Johnson. "MolhadoRef: a refactoring-aware software configuration management tool." In: *Proceedings of the Companion to the Symposium on Object-oriented programming systems, languages, and applications (OOPSLA)*. ACM, 2006, pp. 732–733.

[89] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu Lahiri. "DEEPMERGE: Learning to Merge programs." In: *Transactions on Software Engineering (TSE)* 49.4 (2022), pp. 1–16. ISSN: 23318422.

[90] Trung T. Dinh-Trong and James M. Bieman. "The FreeBSD project: a replication case study of open source development." In: *Transactions on Software Engineering (TSE)* 31.6 (2005), pp. 481–494.

[91] Stephen E. Dossick and Gail E. Kaiser. "CHIME: A Metadata-Based Distributed Software Development Environment." In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Springer, 1999, 464–475. ISBN: 3540665382.

[92] Georg Dotzler and Michael Philippsen. "Move-Optimized Source Code Tree Differencing." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2016, 660–671.

[93] W. Keith Edwards. "Flexible conflict detection and management in collaborative applications." In: *Proceedings of the Symposium on User interface software and technology (UIST)*. ACM, 1997, pp. 139–148.

[94] Clarence A Ellis and Simon J Gibbs. "Concurrency control in groupware systems." In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 1989, pp. 399–407.

[95] Max Ellis, Sarah Nadi, and Danny Dig. "A Systematic Comparison of Two Refactoring-aware Merging Techniques." to appear. 2022.

[96] H. Christian Estler, Carlo A. Nordio Martinand Furia, and Bertrand Meyer. "Unifying configuration management with merge conflict detection and awareness systems." In: *Proceedings of the Australian Software Engineering Conference (ASWEC)*. IEEE, 2013, pp. 201–210. ISBN: 978-0-7695-4995-8.

[97] H.-Christian Estler, Martin Nordio, Carlo A. Furia, and Bertrand Meyer. "Awareness and merge conflicts in distributed software development." In: *Proceedings of the International Conference on Global Software Engineering (ICGSE)*. IEEE. IEEE, 2014, pp. 26–35. ISBN: 978-1-4799-4360-9.

[98] Jacky Estublier and Rubby Casalles. "The Adele Configuration Manager." In: *Configuration Management* (1995), 99–134.

[99] Jacky Estublier and Sergio Garcia. "Process model and awareness in scm." In: *Proceedings of the International Workshop on Software Configuration Management (SCM)*. ACM, 2005, pp. 59–74.

[100] Ludwig Fahrmeir, Thomas Kneib, Stefan Lang, Brian Marx, Ludwig Fahrmeir, Thomas Kneib, Stefan Lang, and Brian Marx. *Regression models*. Springer, 2013.

[101] R. Frank Falk and Nancy B. Miller. *A primer for soft modeling.* University of Akron Press, 1992.

[102] J. Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. "Fine-grained and accurate source code differencing." In: *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 313–324.

[103] Yuanrui Fan, Xin Xia, David Lo, Y. Shanping Li. Fan, X. Xia, D. Lo, and S. Li. "Early prediction of merged code changes to prioritize reviewing tasks." In: *Empirical Software Engineering (EMSE)* 23.6 (2018), pp. 3346–3393. ISSN: 1573-7616.

[104] Shukor Sanim Mohd Fauzi, Paul L. Bannerman, and Mark Staples. "Software Configuration Management in Global Software Development: A Systematic Map." In: *Asia Pacific Software Engineering Conference (APSEC)*. 2010, pp. 404–413.

[105] Andy Field. *Discovering statistics using IBM SPSS statistics*. Sage, 2013.

[106] Geraldine Fitzpatrick, Tim Mansfield, Simon Kaplan, David Arnold, Ted Phelps, and Bill Segall. "Augmenting the Workaday World with Elvin." In: *Proceedings of the European Conference on Computer Supported Cooperative Work (ECSCW)*. Springer, 2002, pp. 431–450.

[107] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction." In: *Transactions on Software Engineering (TSE)* 33.11 (2007), pp. 725–743.

[108] Matthieu Foucault, Jean-Rémy Falleri, and Xavier Blanc. "Code Ownership in Open-Source Software." In: *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 2014.

[109] Matthieu Foucault, Cédric Teyton, David Lo, Xavier Blanc, and Jean-Rémy Falleri. "On the usefulness of ownership metrics in open-source software projects." In: *Information and Software Technology (IST)* 64 (2015), pp. 102–112.

[110] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André van der Hoek. "On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub." In: *Transactions on Software Engineering (TSE)* 46.8 (2020), pp. 892–915. ISSN: 1939-3520.

[111] Sergio Luis Gonzalez and Piero Fraternali. "Almost Rerere: Learning to resolve conflicts in distributed projects." In: *Transactions on Software Engineering (TSE)* 49.4 (2022), pp. 2255–2271.

[112] Natassia Goode and Jens F Beckmann. "You need to know: There is a causal relationship between structural knowledge and control performance in complex problem solving tasks." In: *Intelligence* 38.3 (2010), pp. 345–352.

[113] Rachel A Gordon. *Regression analysis for the social sciences*. Routledge, 2015.

[114] Georgios Gousios. "The ghtorent dataset and tool suite." In: *Proceedings of the working conference on mining software repositories (MSR)*. IEEE, 2013, pp. 233–236.

[115] Georgios Gousios, Martin Pinzger, and Arie van Deursen. "An exploratory study of the pull-based software development model." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 345–355.

[116] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. "Work practices and challenges in pull-based development: the contributor´s perspective." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 285–296.

[117] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. "Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. Florence, Italy: IEEE, 2015, 358–368. ISBN: 9781479919345.

[118] Judith E. Grass. "Cdiff: A Syntax Directed Diff for C++ Programs." In: *Proceedings of the Advanced Computing Systems Professional and Technical Association (USENIX)* (1992), pp. 181–193.

[119] Michaela Greiler, Kim Herzig, and Jacek Czerwonka. "Code Ownership and Software Quality: A Replication Study." In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE, 2015, pp. 2–12.

[120] Rebecca E. Grinter. "Using a Configuration Management Tool to Coordinate Software Development." In: *Proceedings of the Conference on Organizational Computing Systems (COCS)*. ACM, 1995, 168–177. ISBN: 0897917065.

[121] Rebecca E. Grinter. "Recomposition: Putting It All Back Together Again." In: *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*. ACM, 1998, 393–402.

[122] Rebecca E. Grinter, James D. Herbsleb, and Dewayne E. Perry. "The Geography of Coordination: Dealing with Distance in R&D Work." In: *Proceedings of the International Conference on Supporting Group Work (GROUP)*. ACM, 1999, 306–315.

[123] Mário Luís Guimarães and António Rito Silva. "Improving early detection of software merge conflicts." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 342–352. ISBN: 978-1-4673-1067-3.

[124] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie van Deursen. "Communication in open source software development mailing lists." In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 277–286.

[125] Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. "Supporting Developers' Coordination in the IDE." In: *Proceedings of the Conference on Computer Supported Cooperative Work And Social Computing (CSCW)*. ACM, 2015, pp. 518–532.

[126] Anja Haake and Jörg M. Haake. "Take CoVer: Exploiting Version Support in Cooperative Systems." In: *Proceedings of the INTERACT and Conference on Human Factors in Computing Systems (CHI)*. ACM, 1993, 406–413.

[127] Matthew Hague, Anthony W. Lin, and Chih-Duo Hong. "CSS Minification via Constraint Solving." In: *Transactions on Programming Languages and Systems (TOPLAS)* 41.2 (2019).

[128] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. "Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack." In: *Empirical Software Engineering (EMSE)* 24.1 (2019), pp. 674–717.

[129] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. "Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning." In: *International Conference on Intelligent Computing*. Springer, 2005, pp. 878–887.

[130] Juris Hartmanis. "Computers and Intractability: A Guide to the Theory of NP-Completeness (Michael R. Garey and David S. Johnson)." In: *SIAM Review* 24.1 (1982), pp. 90–91.

[131] Lile Hattori and Michele Lanza. "Syde: A Tool for Collaborative Software Development." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. Vol. 2. Cape Town, South Africa: ACM, 2010, 235–238. ISBN: 9781605587196.

[132] Lile Hattori, Michele Lanza, and Marco D'Ambros. "A qualitative user study on preemptive conflict detection." In: *Proceedings of the International Conference on Global Software Engineering (ICGSE)*. IEEE, 2012, pp. 159–163.

[133] Rajesh Hegde and Prasun Dewan. "Connecting Programming Environments to Support Ad-Hoc Collaboration." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2008, pp. 178–187.

[134] Johannes Henkel and Amer Diwan. "CatchUp! Capturing and Replaying Refactorings to Support API Evolution." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ICSE '05. ACM, 2005, 274–283. ISBN: 1581139632.

[135] James D Herbsleb and Rebecca E Grinter. "Splitting the organization and integrating the code: Conway's law revisited." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. 1999, pp. 85–95.

[136] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. "Distance, Dependencies, and Delay in a Global Collaboration." In: *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*. ACM, 2000, 319–328. ISBN: 1581132220.

[137] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. "An empirical study of global software development: distance and speed." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2001, pp. 81–90.

[138] Abram Hindle, Daniel M German, and Ric Holt. "What do large commits tell us? a taxonomical study of large commits." In: *Proceedings of the International working conference on Mining software repositories (MSR)*. IEEE, 2008, pp. 99–108.

[139] Reid Holmes and Robert J. Walker. "Promoting Developer-Specific Awareness." In: *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2008, 61–64. ISBN: 9781605580395.

[140] Reid Holmes and Robert J. Walker. "Customized awareness: recommending relevant external change events." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 465–474.

[141] Susan Horwitz, Jan Prins, and Thomas Reps. "Integrating non-interfering versions of programs." In: *ACM Transactions on Programming Languages and Systems*. Vol. 11. ACM, 1989, pp. 345–387.

[142] Susan Horwitz and Thomas Reps. "The use of program dependence graphs in software engineering." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 1992, pp. 392–411.

[143] Susan Horwitz, Thomas Reps, and David Binkley. "Interprocedural slicing using dependence graphs." In: *ACM Transactions on Programming Languages and Systems*. Vol. 12. ACM, 1990, pp. 26–60.

[144] Watts S Humphrey. "Introducing the personal software process." In: *Annals of Software Engineering* 1.1 (1995), pp. 311–325.

[145] James J. Hunt and Walter F. Tichy. "Extensible language-aware merging." In: *Proceedings of the International Conference on Software Maintenance (ICSME)*. IEEE, 2002, pp. 511–520.

[146] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. "Delta algorithms: An empirical analysis." In: *Transactions on Software Engineering and Methodology (TOSEM)* 7.2 (1998), pp. 192–214.

[147] James W. Hunt and Thomas G. Szymanski. "A fast algorithm for computing longest common subsequences." In: *Communications of the ACM* 20.5 (1977), pp. 350–353.

[148] James Wayne Hunt and M. Douglas McIllroy. *An Algorithm for Differential File Comparison*. Murray Hill: Bell Laboratories, 1976.

[149] Phan Thi Thanh Huyen and Koichiro Ochimizu. *A Change Support Model for Distributed Collaborative Work*. 2012.

[150] Jackson and Ladd. "Semantic Diff: a tool for summarizing the effects of modifications." In: *Proceedings International Conference on Software Maintenance (ICSM)*. IEEE, 1994, pp. 243–252.

[151] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*. Vol. 112. Springer, 2013.

[152] H.-Zar Jerrold. "Significance Testing of the Spearman Rank Correlation Coefficient." In: *Journal of the American Statistical Association* 67.339 (1972), pp. 578–580.

[153] Tao Ji, Liqian Chen, Xiaoguang Mao, Xin Yi, and Jiahong Jiang. "Automated regression unit test generation for program merges." In: *Science China Information Sciences* 65 (2020). ISSN: 23318422.

[154] Tao Ji, Liqian Chen, Xin Yi, and Xiaoguang Mao. "Understanding merge conflicts and resolutions in git rebases." In: *Proceegins of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 70–80. ISBN: 978-1-7281-9870-5.

[155] Miguel Jimenez, Mario Piattini, and Aurora Vizcaino. "Challenges and improvements in distributed software development: A systematic review." In: *Advances in Software Engineering* 16.3 (2009), pp. 1–16.

[156] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. "Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 164–174.

[157] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. "Evolutionary trends of developer coordination: a network approach." In: *Empirical Software Engineering (EMSE)* 22.4 (2017), pp. 2050–2094.

[158] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. "From Developer Networks to Verified Communities: A Fine-Grained Approach." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. Vol. 1. IEEE, 2015, pp. 563–573.

[159] Ian T. Jolliffe. *Principal Component Analysis*. Springer, 2002, p. 487.

[160] Frederick Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1974. ISBN: 978-0-201-00650-6.

[161] Sascha Just, Kim Herzig, Jacek Czerwonka, and Brendan Murphy. "Switching to Git: The Good, the Bad, and the Ugly." In: *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 400–411.

[162] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damia. "The promises and perils of mining GitHub." In: *Proceedings of the working conference on mining software repositories (MSR)*. ACM, 2014, pp. 92–101.

[163] Bakhtiar Khan Kasi and Anita Sarma. "Cassandra: Proactive conflict minimization through optimized task scheduling." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 732–741.

[164] Inderdeep Kaur, Parminder Kaur, and Hardeep Singh. "Analysis of workspace awareness tools in distributed software development." In: *Proceedings of the International Conference on Intelligent Communication and Computational Techniques (ICCT)*. 2018, pp. 233–238.

[165] Mik Kersten and Gail C. Murphy. "Using task context to improve programmer productivity." In: *Proceedings of the Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2006, pp. 1–11.

[166] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. "A Formal Investigation of Diff3." In: *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. Springer, 2007, pp. 485–496.

[167] Seongho Kim. "ppcor: An R Package for a Fast Calculation to Semi-partial Correlation Coefficients." In: *Communication for Statistical Applications and Methods* 22.6 (2015), pp. 665–674.

[168] Sunghun Kim, E. James Whitehead, and Yi Zhang. "Classifying Software Changes: Clean or Buggy?" In: *Transactions on Software Engineering (TSE)* 34.2 (2008), pp. 181–196.

[169] Maximilian Koegel, Jonas Helming, and Stephan Seyboth. "Operation-based conflict detection and resolution." In: *Workshop on Comparison and Versioning of Software Models (CVSM)*. 2009, pp. 43–48.

[170] Maximilian Koegel, Helmut Naughton, Jonas Helming, and Markus Herrmannsdoerfer. "Collaborative model merging." In: *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA-C)*. ACM, 2010, pp. 27–34.

[171] Oleksii Kononenko, Tresa Rose, Olga Baysal, Michael Godfrey, Dennis Theisen, and Bart de Water. "Studying Pull Request Merges: A Case Study of Shopify's Active Merchant." In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Gothenburg, Sweden: ACM, 2018, 124–133. ISBN: 9781450356596.

[172] Thomas D. LaToza, Gina Venolia, and Robert DeLine. "Maintaining Mental Models: A Study of Developer Work Habits." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2006, 492–501.

[173] Michele Lanza, Marco D'Ambros, Alberto Bacchelli, Lile Hattori, and Francesco Rigotti. "Manhattan: Supporting real-time visual team activity awareness." In: *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 207–210.

[174] Simon Larsén, Jean-Rémy Falleri, Benoit Baudry, and Martin Monperrus. "Spork: Structured Merge for Java with Formatting Preservation." In: *Transactions on Software Engineering (TSE)* (2022), pp. 1–1. ISSN: 00985589.

[175] Janne Lautamäki, Antti Nieminen, Johannes Koskinen, Timo Aho, Tommi Mikkonen, and Marc Englund. "CoRED: browser-based Collaborative Real-time Editor for Java web applications." In: *Proceedings of the conference on Computer Supported Cooperative Work (CSCW)*. ACM, 2012, pp. 1307–1316.

[176] Xuan-Bach. Le, Ferdian Thung, David Lo, and Claire Le Goue. "Overfitting in semantics-based automated program repair." In: *Empirical Software Engineering (EMSE)* 23.1 (2018).

[177] David B Leblang, Robert P Chase Jr, and Howard Spilke. "Increasing Productivity with a Parallel Configuration Manager." In: *Proceedings of the International Workshop Software Version and Configuration Control*. 1988, pp. 21–37.

[178] David B. Leblang and Robert P. Chase. "Computer-Aided Software Engineering in a distributed workstation environment." In: *ACM SIGPLAN Notices* 19.5 (1984), pp. 104–112.

[179] Olaf Leßenich, Sven Apel, and Christian Lengauer. "Balancing precision and performance in structured merge." In: *Automated Software Engineering* 22.3 (2015), pp. 367–397. ISSN: 1573-7535.

[180] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian Kästner, and Claus Hunsen. "Indicators for merge conflicts in the wild: survey and empirical study." In: *Automated Software Engineering* 25.2 (2018), pp. 279–313. ISSN: 1573-7535.

[181] Timothy R Levine and Craig R Hullett. "Eta squared, partial eta squared, and misreporting of effect size in communication research." In: *Human Communication Research* 28.4 (2002), pp. 612–625.

[182] Olaf Leßenich, Sven Apel, Christian Kästner, Georg Seibt, and Janet Siegmund. "Renaming and shifted code in structured merging: Looking ahead for precision and performance." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 543–553.

[183] Anund Lie, Reidar Conradi, Tor M. Didriksen, and E.-A. Karlsson. "Change Oriented Versioning in a Software Engineering Database." In: *Software Engineering Notes* 14.7 (1989), 56–65.

[184] Max Lillack, Ştefan Stănciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wasowski. "Intention-Based Integration of Software Variants." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. Montreal, Quebec, Canada: IEEE, 2019, 831–842.

[185] Johan Linåker, Hussan Munir, Krzysztof Wnuk, and Carl-Eric Mols. "Motivating the contributions: An open innovation perspective on what to share as open source software." In: *Journal of Systems and Software (JSS)* 135 (2018), pp. 17–36.

[186] Tancred Lindholm. "A three-way merge for XML documents." In: *Proceedings of the symposium on Document engineering*. ACM, 2004, pp. 1–10.

[187] Ernst Lippe and Norbert van Oosterom. "Operation-based merging." In: *Proceedings of the Symposium of Software Development Environments*. Vol. 17. ACM, 1992, pp. 78–87.

[188] Jing Liu, Jiahao Li, and Lulu He. "A Comparative Study of the Effects of Pull Request on GitHub Projects." In: *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. IEEE, 2016, pp. 313–322.

[189] David Lubkin. "Heterogeneous configuration management with DSEE." In: *Proceedings of the International Workshop on Software Configuration Management (SCM)*. ACM, 1991, pp. 153–160.

[190] Chandra Maddila, Nachiappan Nagappan, Christian Bird, Georgios Gousios, and Arie van Deursen. "ConE: A Concurrent Edit Detection Tool for Large-Scale Software Development." In: *Transactions on Software Engineering and Methodology (TOSEM)* 31.2 (2021). ISSN: 1049-331X.

[191] Boris Magnusson and Ulf Asklund. "Fine grained version control of configurations in COOP/Orm." In: *Proceedings of the International Workshop on Software Configuration Management (SCM)*. Springer, 1996, pp. 31–48.

[192] Wardah Mahmood, Moses Chagama, Thorsten Berger, and Regina Hebig. "Causes of merge conflicts: A case study of ElasticSearch." In: *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS)*. Magdeburg, Germany: ACM, 2020, pp. 1–9. ISBN: 9781450375016.

[193] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. "Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts." In: *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. IEEE, 2019, pp. 151–162. ISBN: 978-1-7281-0591-8.

[194] Jose Martinez, Luke Lucas, and Alessandro Donati. "Dependency Finder: Surprising Relationships in Telemetry." In: *Proceedings of the SpaceOps Conference*. American Institute of Aeronautics and Astronautics, 2018.

[195] Stina Matthiesen, Pernille Bjørn, and Claus Trillingsgaard. "Implicit bias and negative stereotyping in global software development and why it is time to move on!" In: *Journal of Software: Evolution and Process (S:EP)* 35.5 (2023), e2435.

[196] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. "Software practitioner perspectives on merge conflicts and resolutions." In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 467–478. ISBN: 978-1-5386-0992-7.

[197] Ahmed-Nacer Mehdi, Pascal Urso, and François Charoy. "Evaluating Software Merge Quality." In: *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE)*. London, England, United Kingdom: ACM, 2014, pp. 1–10. ISBN: 9781450324762.

[198] Andrew Meneely and Laurie Williams. "Secure Open Source Collaboration: An Empirical Study of Linus' Law." In: *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2009, 453–462.

[199] José W. Menezes, Bruno Trindade, Felipe P. João, Alexandre Plastino, Leonardo Murta, and Catarina Costa. "Attributes that may raise the occurrence of merge conflicts." In: *Journal of Software Engineering Research and Development (JSERD)* 9 (2021), 14:1–14:14.

[200] José William Menezes, Bruno Trindade, João Felipe Pimentel, Tayane Moura, Alexandre Plastino, Leonardo Murta, and Catarina Costa. "What causes merge conflicts?" In: *Proceedings of the Brazilian Symposium on Software Engineering (SBES)*. Natal, Brazil: ACM, 2020, pp. 203–212. ISBN: 9781450387538.

[201] Tom Mens. "Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution." In: *Applications of Graph Transformations with Industrial Relevance*. Springer, 2000, pp. 127–143.

[202] Tom Mens. "A formal foundation for object-oriented software evolution." In: *Proceedings International Conference on Software Maintenance (ICSM)*. IEEE. 2001, pp. 549–552.

[203] Tom Mens. "A state-of-the-art survey on software merging." In: *Transactions on Software Engineering (TSE)* 28.5 (2002), pp. 449–462. ISSN: 1939-3520.

[204] Victor Cacciari Miraldo and Wouter Swierstra. "An Efficient Algorithm for Type-Safe Structural Diffing." In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–29.

[205] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. "Two Case Studies of Open Source Software Development: Apache and Mozilla." In: *Transactions on Software Engineering and Methodology (TOSEM)* 11.3 (2002), 309–346.

[206] Audris Mockus and David M. Weiss. "Predicting risk of software changes." In: *Bell Labs Technical Journal* 5.2 (2002), pp. 169–180.

[207] Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. "Survey guidelines in software engineering: An annotated review." In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2016, pp. 1–6.

[208] Pascal Molli, Hala Skaf-Molli, and Christophe Bouthier. "State Treemap: an awareness widget for multi-synchronous groupware." In: *Proceedings of the International Workshop on Groupware (CRIWG)*. IEEE, 2001, pp. 106–114.

[209] Leticia Montalvillo, Oscar Díaz, and Thomas Fogdal. "Reducing Coordination Overhead in SPLs: Peering in on Peers." In: *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. Vol. 1. Gothenburg, Sweden: ACM, 2018, 110–120. ISBN: 9781450364645.

[210] Jonathan P. Munson and Prasun Dewan. "A flexible object merging framework." In: *Proceedings of the Conference on Computer supported cooperative work (CSCW)*. ACM, 1994, pp. 231–242.

[211] Kivanç Muslu, Luke Swart, Yuriy Brun, and Michael D. Ernst. "Development History Granularity Transformations (N)." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 697–702.

[212] Ward Muylaert and Coen De Roover. "Prevalence of Botched Code Integrations." In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. Buenos Aires, Argentina: IEEE, 2017, 503–506. ISBN: 9781538615447.

[213] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. "The Influence of Organizational Structure on Software Quality: An Empirical Case Study." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2008, 521–530. ISBN: 9781605580791.

[214] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. "The life-cycle of merge conflicts: processes, barriers, and strategies." In: *Empirical Software Engineering (EMSE)* 24.5 (2019), pp. 2863–2906. ISSN: 1573-7616.

[215] Hien M Nguyen, Eric W Cooper, and Katsuari Kamei. "Borderline over-sampling for imbalanced data classification." In: *International Journal of Knowledge Engineering and Soft Data Paradigms* 3.1 (2011), pp. 4–21.

[216] Hoai Le Nguyen and Claudia-Lavinia Ignat. "An Analysis of Merge Conflicts and Resolutions in Git-based Open Source Projects." In: *Proceedings of the European Conference on Computer Supported Cooperative Work (ECSCW)*. Vol. 27 (3). Springer, 2018, pp. 741–765.

[217] Hoai Nguyen and Claudia-Lavinia Ignat. "Parallelism and conflicting changes in Git version control systems." In: *Proceedings of the International Workshop on Collaborative Editing Systems (IWCES)*. HAL-Inria, 2017.

[218] Hung Viet Nguyen, My Huu Nguyen, Son Cuu Dang, Christian Kästner, and Tien N. Nguyen. "Detecting semantic merge conflicts with variability-aware execution." In: *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy: ACM, 2015, pp. 926–929. ISBN: 9781450336758.

[219] Antti Nieminen. "Real-time collaborative resolving of merge conflicts." In: *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*. IEEE, 2012, pp. 540–543.

[220] Yuichi Nishimura and Katsuhisa Maruyama. "Supporting merge conflict resolution by using fine-grained code change history." In: *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE, 2016, pp. 661–664. ISBN: 978-1-5090-1855-0.

[221] Nan Niu, Fangbo Yang, Jing-Ru C. Cheng, and Sandeep Reddivari. "A cost-benefit approach to recommending conflict resolution for parallel software development." In: *Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 2012, pp. 21–25.

[222] Nan Niu, Fangbo Yang, Jing-Ru C. Cheng, and Sandeep Reddivari. "Conflict resolution support for parallel software development." In: *Institution of Engineering and Technology Software* 7.1 (2013), pp. 1–11.

[223] Carin M. Olson, Drummond Rennie, Deborah Cook, Kay Dickersin, Annette Flanagin, Joseph W. Hogan, Qi Zhu, Jennifer Reiling, and Brian Pace. "Publication Bias in Editorial Decision Making." In: *Journal of the American Medical Association (JAMA)* 287.21 (2002), pp. 2825–2828.

[224] Päivi Ovaska, Matti Rossi, and Pentti Marttiin. "Architecture as a coordination tool in multi-site software development." In: *Software Process: Improvement and Practice* 8.4 (2003), pp. 233–247.

[225] Moein Owhadi-Kareshk and Sarah Nadi. "Scalable software merging studies with MERGANSER." In: *Proceedings of the International Conference on Mining Software Repositories (MSR)*. Montreal, Quebec, Canada: IEEE, 2019, pp. 560–564.

[226] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. "Predicting Merge Conflicts in Collaborative Software Development." In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11. ISBN: 978-1-7281-2968-6.

[227] Rohan Padhye, Senthil Mani, and Vibha Sinha. "A study of external community contribution to open-source projects on GitHub." In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 332–335.

[228] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike Kaufman. "Can program synthesis be used to learn merge conflict resolutions? An empirical analysis." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. 2021, pp. 785–796. ISBN: 978-0-7381-1319-7.

[229] Sebastiano Panichella, Gabriele Bavota, Massimiliano Di Penta, Gerardo Canfora, and Giuliano Antoniol. "How Developers' Collaborations Identified from Different Sources Tell Us about Code Changes." In: *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 251–260.

[230] Fabrizio Pastore, Leonardo Mariani, and Daniela Micucci. "BDCI: Behavioral driven conflict identification." In: *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2017, pp. 570–581.

[231] Dewayne E. Perry and Gail E. Kaiser. "Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems." In: *Proceedings of the Annual Conference on Computer Science (CSC)*. ACM, 1987, pp. 292–299.

[232] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. "Parallel changes in large scale software development: an observational case study." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1998, pp. 308–337.

[233] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. "Parallel changes in large-scale software development." In: *Transactions on Software Engineering and Methodology (TOSEM)* 10.3 (2001), pp. 308–337.

[234] Shaun Phillips, Jonathan Sillito, and Rob Walker. "Branching and Merging: An Investigation into Current Version Control Practices." In: *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2011, 9–15.

[235] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. "Can Developer-Module Networks Predict Failures?" In: *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2008, 2–12.

[236] Ranjith Purushothaman and Dewayne E. Perry. "Toward understanding the rhetoric of small source code changes." In: *Transactions on Software Engineering (TSE)* 31.6 (2005), pp. 511–526.

[237] Foyzur Rahman and Premkumar Devanbu. "Ownership, Experience and Defects: A Fine-Grained Study of Authorship." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2011, 491–500. ISBN: 9781450304450.

[238] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. "Feature toggles: practitioner practices and a case study." In: *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. ACM, 2016.

[239] Ehud Reiter, Roma Robertson, and Liesl M. Osman. "Lessons from a failure: Generating tailored smoking cessation letters." In: *Artificial Intelligence* 144.1 (2003), pp. 41–58.

[240] Jungkyu Rho and Chisu Wu. "An efficient version model of software diagrams." In: *Proceedings of the Asia Pacific Software Engineering Conference (APSEC)*. IEEE, 1998, pp. 236–243.

[241] Peter Rigby, Daniel German, Laura Cowen, and Margaret-Anne Storey. "Peer review on open-source software projects: Parameters, statistical models, and theory." In: *Transactions on Software Engineering and Methodology (TOSEM)* 23.4 (2014), pp. 1–33.

[242] Roger M. Ripley, Ryan Y. Yasui, Anita Sarma, and André van der Hoek. "Workspace awareness in application development." In: *Proceedings of the OOPSLA workshop on eclipse technology eXchange*. ACM, 2004, pp. 17–21.

[243] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Israel Herraiz. "Evolution of the core team of developers in libre software projects." In: *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*. IEEE, 2009, pp. 167–170.

[244] Gregorio Robles, Jesús González-Barahona, Carlos Cervigón, Andrea Capiluppi, and Daniel Izquierdo-Cortázar. "Estimating development effort in free/open source software projects by mining software repositories: A case study of OpenStack." In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 222–231.

[245] Thaís Rocha, Paulo Borba, and João Pedro Santos. "Using acceptance tests to predict files changed by programming tasks." In: *Journal of Systems and Software* 154 (2019), pp. 176–195. ISSN: 0164-1212.

[246] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. "Replicated abstract data types: Building blocks for collaborative applications." In: *Journal of Parallel and Distributed Computing* 71.3 (2011), pp. 354–368.

[247] Sebastian Rönnau, Christian Pauli, and Uwe M. Borghoff. "Merging changes in XML documents using reliable context fingerprints." In: *Proceedings of the symposium on Document engineering*. ACM, 2008.

[248] Yasutaka Sakamoto, Shinsuke Matsumoto, Seiki Tokunaga, Sachio Saiki, and Masahide Nakamura. "Empirical study on effects of script minification and HTTP compression for traffic reduction." In: *Proceedings of the International Conference on Digital Information, Networking, and Wireless Communications (DINWC)*. IEEE, 2015, pp. 127–132.

[249] Jadson Santos and Uira Kulesza. "Quantifying and assessing the merge of cloned web-based system: An exploratory study." In: *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Redwood City, USA, 2016, pp. 583 –588.

[250] Santonu Sarkar, Renuka Sindhgatta, and Krishnakumar Pooloth. "A Collaborative Platform for Application Knowledge Management in Software Maintenance Projects." In: *Proceedings of the Bangalore Annual Compute Conference (COMPUTE)*. ACM, 2008.

[251] Anita Sarma, Zahra Noroozi, and André van der Hoek. "Palantír: raising awareness among configuration management workspaces." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2003, pp. 444–454.

[252] Anita Sarma, David F. Redmiles, and André van der Hoek. "Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes." In: *Transactions on Software Engineering (TSE)* 38.4 (2012), pp. 889–908. ISSN: 1939-3520.

[253] Trevor Savage, Bogdan Dit, Malcom Gethers, and Denys Poshyvanyk. "Topicxp: exploring topics in source code using latent dirichlet allocation." In: *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–6.

[254] Kjeld Schmidt and Carla Simonee. "Coordination mechanisms: Towards a conceptual foundation of CSCW systems design." In: *In Computer Supported Cooperative Work (CSCW)* 5 (1996), 155–200.

[255] Till Schümmer and Jörg M. Haake. "Supporting distributed software development by modes of collaboration." In: *Proceedings of the European Conference on Computer Supported Cooperative Work (ECSCW)*. Springer, 2001, pp. 79–98.

[256] Todd Sedano, Paul Ralph, and Cécile Péraire. "Software Development Waste." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 130–140.

[257] Georg Seibt, Florian Heck, Guilherme Cavalcanti, Paulo Borba, and Sven Apel. "Leveraging Structure in Software Merge: An Empirical Study." In: *Transactions on Software Engineering (TSE)* 48 (2022), pp. 4590–4610. ISSN: 00985589.

[258] Francisco Servant, James A. Jones, and André van der Hoek. "CASI: preventing indirect conflicts through a live visualization." In: *Proceedings of the Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2010, pp. 39–46.

[259] Bo Shen, Wei Zhang, Ailun Yu, Yifan Shi, Haiyan Zhao, and Zhi Jin. "SoManyConflicts: Resolve Many Merge Conflicts Interactively and Systematically." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 2021, pp. 1291–1295. ISBN: 978-1-6654-0337-5.

[260] Bo Shen, Wei Zhang, Haiyan Zhao, Guangtai Liang, and Qianxiang Wang. "Intellimerge: A refactoring-aware software merging technique." In: *Proceedings of the ACM on Programming Languages (OOPSLA)* 3 (2019), 170:1–170:28. ISSN: 24751421.

[261] Bowen Shen, Cihan Xiao, Na Meng, and Fei He. "Automatic detection and resolution of software merge conflicts: Are we there yet?" to appear. 2021.

[262] Emad Shihab, Christian Bird, and Thomas Zimmermann. "The Effect of Branching Strategies on Software Quality." In: *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2012, pp. 301–310.

[263] Janet Siegmund and Jana Schuman. "Confounding parameters on program comprehension: a literature survey." In: *Empirical Software Engineering (EMSE)* 20.1 (2015), pp. 1159–1192.

[264] Isabella A. da Silva, Ping H. Chen, Christopher Van der Westhuizen, Roger M. Ripley, and André van der Hoek. "Lighthouse: coordination through emerging design." In: *Proceedings of the workshop on eclipse technology eXchange*. ACM, 2006, pp. 11–15.

[265] Léuson Da Silva, Paulo Borba, and Arthur Pires. "Build conflicts in the wild." In: *Journal of Software: Evolution and Process* 34.4 (2022).

[266] Leif Singer, Fernando Figueira Filho, Brendan Cleary, Christoph Treude, Margaret-Anne Storey, and Kurt Schneider. "Mutual Assessment in the Social Programmer Ecosystem: An Empirical Investigation of Developer Profile Aggregators." In: *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*. ACM, 2013, 103–116.

[267] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. "Is the cure worse than the disease? overfitting in automated program repair." In: *Proceedings of the Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 532–543.

[268] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. "Improving developer participation rates in surveys." In: *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2013, pp. 89–92.

[269] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. "Verified Three-Way Program Merge." In: *Proceedings of the ACM on Programming Languages (OOPSLA)* 2 (2018).

[270] Cleidson R. B. Souza and David Redmiles. "The Awareness Network: To Whom Should I Display My Actions? And, Whose Actions Should I Monitor?" In: *Proceedings of the European Conference on Computer-Supported Cooperative Work (ECSCW)*. Springer, 2007, pp. 99–117.

[271] Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. "How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development." In: *SIGSOFT Software Engineering Notes* 29.6 (2004), 221–230.

[272] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. ""Breaking the code", moving between private and public work in collaborative software development." In: *Proceedings of the International Conference on Supporting Group Work (GROUP)*. ACM, 2003, pp. 105–114.

[273] Cleidson R. de Souza, Stephen Quirk, Erik Trainer, and David F. Redmiles. "Supporting collaborative software development through the visualization of socio-technical dependencies." In: *Proceedings of the international ACM conference on Conference on supporting group work (GROUP)*. ACM, 2007, pp. 147–156.

[274] Cleidson de Souza and David Redmiles. "An empirical study of software developers' management of dependencies and changes." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2008, pp. 241–250.

[275] Lucas Batista Leite de Souza and Marcelo de Almeida Maia. "Do software categories impact coupling metrics?" In: *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 217–220.

[276] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. "Pydriller: Python framework for mining software repositories." In: *Proceedings of the Symposium and the European Conference on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 908–911.

[277] Hans Spanjers, Maarten ter Huurne, Bas Graaf, Marco Lormans, and Dan Bendas. "Tool Support for Distributed Software Engineering." In: *Proceedings of the International Conference on Global Software Engineering (ICGSE)*. IEEE, 2006, pp. 187–198.

[278] Nancy A. Staudenmayer and Michael A. Cusumano. "Managing Multiple Interdependencies in Large Scale Software Development Projects." PhD thesis. Massachusetts Institute of Technology, 1997.

[279] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. "Grounded theory in software engineering research." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 120–131.

[280] Margaret-Anne Storey, Leif Singer, Brendan Cleary, Fernando Figueira Filho, and Alexey Zagalsky. "The (R) Evolution of Social Media in Software Engineering." In: *Future of Software Engineering Proceedings (FOSE)*. ACM, 2014, 100–116.

[281] Margaret-Anne Storey, Alexey Zagalsky, Fernando Figueira Filho, Leif Singer, and Daniel M. German. "How Social and Communication Channels Shape and Challenge a Participatory Culture in Software Development." In: *Transactions on Software Engineering (TSE)* 43.2 (2017), pp. 185–204.

[282] Anselm Strauss and Juliet M Corbin. *Grounded theory in practice*. Sage, 1997.

[283] Chengzheng Sun and Clarence Ellis. "Operational transformation in real-time group editors: issues, algorithms, and achievements." In: *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*. ACM, 1998, pp. 59–68.

[284] Chungha Sung, Shuvendu K. Lahiri, Mike Kaufman, Pallavi Choudhury, and Chao Wang. "Towards Understanding and Fixing Upstream Merge Induced Conflicts in Divergent Forks: An Industrial Case Study." In: *Proceedings of the International Conference on Software Engineering: Companion (ICSE-C)*. ACM, 2020, pp. 320–321. ISBN: 9781450371223.

[285] Junichi Suzuki and Yoshikazu Yamamoto. "SoftDock: a distributed collaborative platform for model-based software development." In: *Proceedings of the International Workshop on Database and Expert Systems Applications (DEXA)*. IEEE, 1999, pp. 672–676.

[286] Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri. "Program merge conflict resolution via neural transformers." In: *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2022, pp. 822–833.

[287] Chakkrit Tantithamthavorn and Ahmed E. Hassan. "An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges." In: *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 2018, 286–295. ISBN: 9781450356596.

[288] Alberto Trindade Tavares, Paulo Borba, Guilherme Cavalcanti, and Sérgio Soares. "Semistructured merge in Javascript systems." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. San Diego, California: IEEE, 2019, pp. 1014–1025. ISBN: 9781728125084.

[289] Timothy Teo. *Handbook of quantitative methods for educational research*. Springer, 2013, p. 404.

[290] Antonio Terceiro, Luiz Romario Rios, and Christina Chavez. "An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects." In: *Proceedings of the Brazilian Symposium on Software Engineering (SBES)*. IEEE, 2010, pp. 21–29.

[291] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. "Revisiting Code Ownership and Its Relationship with Software Quality in the Scope of Modern Code Review." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2016, 1039–1050. ISBN: 9781450339001.

[292] Walter F. Tichy. "The string-to-string correction problem with block moves." In: *ACM Transactions on Computer Systems* 2.4 (1984), pp. 309–321.

[293] Walter F. Tichy. "Rcs — a system for version control." In: *Software: Practice and Experience* 15.7 (1985), pp. 637–654.

[294] Marina Bianca Trif and Radu Razvan Slavescu. "Towards Predicting Merge Conflicts in Software Development Environments." In: *Proceedings of the International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 2021, pp. 251–256.

[295] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. "Accurate and efficient refactoring detection in commit history." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 483–494.

[296] Jason Tsay, Laura Dabbish, and James Herbsleb. "Influence of Social and Technical Factors for Evaluating Contribution in GitHub." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 356–366.

[297] Gustavo Vale. *Investigating the Merge Conflict Life-Cycle Taking the Social Dimension into Account*. https://gustavovale.github.io/investigating-the-merge-conflict-life-cycle-taking-the-social-dimension-into-account/. [Online; accessed 30-October-2023]. 2024.

[298] Gustavo Vale, Heitor Costa, and Sven Apel. "Predicting Merge Conflicts Considering Social and Technical Assets." In: *Empirical Software Engineering (EMSE)* (2023), pp. 1–26.

[299] Gustavo Vale, Eduardo Fernandes, and Eduardo Figueiredo. "On the proposal and evaluation of a benchmark-based threshold derivation method." In: *Software Quality Journal* 27 (2019), pp. 275–306.

[300] Gustavo Vale, Eduardo Fernandes, Eduardo Figueiredo, and Sven Apel. "Behind Developer Contributions on Conflicting Merge Scenarios." In: *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society Press. IEEE, 2023, pp. 1–11.

[301] Gustavo Vale, Claus Hunsen, Eduardo Figueiredo, and Sven Apel. "Challenges of Resolving Merge Conflicts: A Mining and Survey Study." In: *Transactions on Software Engineering (TSE)* 48.12 (2022), pp. 4964–4985. ISSN: 00985589.

[302] Gustavo Vale, Angelika Schmid, Alcemir Santos, Eduardo Almeida, and Sven Apel. "On the relation between Github communication activity and merge conflicts." In: *Empirical Software Engineering (EMSE)* 25.1 (2020), pp. 402–433. ISSN: 1573-7616.

[303] Gustavo Vale, Angelika Schmid, Alcemir Santos, Eduardo Almeida, and Sven Apel. *On the Relation Between Coordination Activities and Merge Conflicts - Supplementary Web site*. https://sites.google.com/view/vale-emse2019. [Online; accessed 30-October-2023]. 2023.

[304] Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. "Copies convergence in a distributed real-time collaborative environment." In: *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*. ACM, 2000, pp. 171–180.

[305] Stéphane Weiss, Pascal Urso, and Pascal Molli. "Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks." In: *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2009, pp. 404–412.

[306] Bernhard Westfechtel. "Structure-oriented merging of revisions of software documents." In: *Proceedings of the International Workshop on Software Configuration Management (SCM)*. ACM, 1991, pp. 68–79.

[307] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. "Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models." In: *Empirical Software Engineering (EMSE)* 13.5 (2008), pp. 539–559.

[308] Jan Wloka, Barbara Ryder, Frank Tip, and Xiaoxia Ren. "Safe-commit analysis to facilitate team software development." In: *Proceedings of the International Conference on Software Engineering (ICSE)*. 2009, pp. 507–517. ISBN: 978-1-4244-3452-7.

[309] Claes Wohlin, Per Runeson, Martin Höst, Magnus Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. 1st. Springer, 2012.

[310] S.-Paul Wright. "Adjusted P-Values for Simultaneous Inference." In: *Biometrics* 48.4 (1992), pp. 1005–1013.

[311] Thorsten Wuensche, Artur Andrzejak, and Sascha Schwedes. "Detecting Higher-Order Merge Conflicts in Large Software Projects." In: *Proceedings of the International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 353–363. ISBN: 978-1-7281-5778-8.

[312] Yang Wuu. "How to merge program texts." In: *Journal of Systems and Software* 27.2 (1994), pp. 129–135.

[313] WenPeng Xiao, ChangYan Chi, and Min Yang. "On-Line Collaborative Software Development via Wiki." In: *Proceedings of the International Symposium on Wikis (WikiSym)*. ACM, 2007, 177–183.

[314] Xiaoqian Xing and Katsuhisa Maruyama. "Automatic Software Merging using Automated Program Repair." In: *Proceedings of the International Workshop on Intelligent Bug Fixing (IBF)*. IEEE. 2019, pp. 11–16. ISBN: 978-1-7281-1809-3.

[315] Wuu Yang. "Identifying syntactic differences between two programs." In: *Software: Practice and Experience* 21.7 (1991), pp. 739–755.

[316] Wuu Yang, Susan Horwitz, and Thomas Reps. "A program integration algorithm that accommodates semantics-preserving transformations." In: *Transactions on Software Engineering and Methodology (TOSEM)* 1.3 (1992), pp. 310–354.

[317] Ryohei Yuzuki, Hideaki Hata, and Kenichi Matsumoto. "How we resolve conflict: an empirical study of method-level conflict resolution." In: *Proceedings of the International Workshop on Software Analytics (SWAN)*. IEEE, 2015, pp. 21–24.

[318] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K. Lahiri. "Can pre-trained language models be used to resolve textual and semantic merge conflicts?" to appear. 2021.

[319] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. "The Impact of Continuous Integration on Other Software Development Practices: A Large-scale Empirical Stud." In: *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 60–71.

[320] Fengmin Zhu and Fei He. "Conflict Resolution for Structured Merge via Version Space Algebra." In: *Proceedings of the ACM on Programming Languages (OOPSLA)* 2 (2018), 166:1–25. ISSN: 24751421.

[321] Fengmin Zhu, Fei He, and Qianshan Yu. "Enhancing Precision of Structured Merge by Proper Tree Matching." In: *Proceedings of the International Conference on Software Engineering: Companion (ICSE-C)*. IEEE, 2019, 286–287.

[322] Thomas Zimmermann. "Mining Workspace Updates in CVS." In: *Proceedings of the International Workshop on Mining Software Repositories (MSR)*. IEEE, 2007, pp. 11–11.

[323] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. "Mining version histories to guide software changes." In: *Transactions on Software Engineering (TSE)* 31.6 (2005), pp. 429–445.