Pós-Graduação em Ciência da Computação

Paola Rodrigues de Godoy Accioly

Understanding Collaboration Conflicts Characteristics

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2018

Paola Rodrigues de Godoy Accioly

Understanding Collaboration Conflicts Characteristics

A Ph.D. Thesis presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.

Concentration Area: computer science
Advisor: Paulo Henrique Monteiro Borba

Recife
2018

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

A171u Accioly, Paola Rodrigues de Godoy
Understanding collaboration conflicts characteristics / Paola Rodrigues de Godoy Accioly. – 2018.
102 f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2018.
Inclui referências.

1. Engenharia de software. 2. Desenvolvimento colaborativo de software. I. Borba, Paulo Henrique Monteiro (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2018-123

Paola Rodrigues Godoy Accioly

Understanding Collaboration Conflicts Characteristics

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Aprovado em: 26/02/2018.

_____

Orientador: Prof. Dr. Paulo Henrique Monteiro Borba

BANCA EXAMINADORA

_____

Prof. Dr. Vinicius Cardoso Garcia
Centro de Informática / UFPE

_____

Prof. Dr. Leopoldo Teixeira Motta
Centro de Informática / UFPE

_____

Prof. Dr. Leonardo Gresta Paulino Murta
Instituto de Computação / UFF

_____

Prof. Dr. Tiago Lima Massoni
Departamento de Sistemas e Computação / UFCG

_____

Profa. Dra. Anita Sarma
School of Electrical Engineering and Computer Science
Oregon State University

I dedicate this thesis to all my family, friends and professors who gave me the necessary support to get here.

ACKNOWLEDGEMENTS

Agradecimentos especiais a todos que contribuiram diretamente para a realização deste trabalho. Primeiramente ao meu orientador, professor Paulo Borba, pela sua orientação cuidadosa durante todas as fases desse trabalho. Gostaria de agradecer aos membros do SPG e do LabES pelos vários conselhos, momentos de apoio e também de descontração. Em particular gostaria de agradecer a Guilherme, Roberto e Léuson por terem trabalhado em colaboração comigo. Além disso, também faço um agradecimento especial as minhas "irmãs de pesquisa": Gabriela, Thaís e Klissiomara. Agradeço ao CIn, aos seus professores e aos seus funcionários pela estrutura e formação de excelência que conheço desde quando ingressei em 2005 para a graduação em Ciência da Computação. Agradeço ao INES — Instituto Nacional de Ciência e Tecnologia para Engenharia de Software — e a FACEPE por financiarem a minha pesquisa. Claro, não poderia deixar de agradecer a minha família e ao meu marido Italo, por todo o apoio que me foi dado durante essa longa jornada.

ABSTRACT

Empirical studies show that collaboration conflicts frequently occur, impairing developers' productivity, since merging conflicting contributions often is a demanding and error-prone task. However, to the best of our knowledge, the structure of changes that lead to conflicts has not been studied yet. Understanding the underlying structure of conflicts, and the involved syntactic language elements might shed light on how to better avoid merge conflicts. To this end, in this thesis we derive a catalog of conflict patterns expressed in terms of code changes (considering Java programs) that lead to merge conflicts. We focus on conflicts reported by a semistructured merge tool that exploits knowledge about the underlying syntax of the artifacts. This way, we avoid analyzing a large number of spurious conflicts often reported by typical line based merge tools. To assess the occurrence of such patterns in different systems, we conduct an empirical study showing that most semi-structured merge conflicts in our sample happen because developers independently edit the same or consecutive lines of the same method. We also observe that using more sophisticated merge tools might decrease integration effort. As a complementary result, we also discuss that developers often do not take full advantage of version control systems when they copy and paste pieces of code around different branches, and merge conflicts usually involve more than two developers which might suggest that they are not so easy to resolve. This study was a first exploration into merge conflicts characteristics. As a practical consequence of our results, a possible strategy to avoid conflicts would be improving existing development tools to alert when developers independently edit the same method, ideally before code integration. However, it is possible that developers edit unrelated parts of the same method without having to deal with conflicts. Because of that, this strategy might have the potential to raise false alarms. In order to assess this strategy precision, we conduct a second empirical study using Travis Continuous Integration service to learn if changing the same method increases the chance of having build and test problems. In addition, we evaluate how frequently having different developers editing directly dependent methods leads to build and test problems as well. Our results indicate that detecting editions to the same method would be a reasonable conservative strategy to detect conflicts early. Moreover, we also provide recommendations that could provide even more precise results. For example, we could generate test cases to expose contributions interaction to detect test conflicts more efficiently. In addition, we could detect refactoring changes to decrease the number of false alarms in an awareness tool. To sum up, in this thesis we conducted two empirical studies to understand different characteristics of collaboration conflicts. Based on both studies evidence we were able to derive different recommendations to detect conflicts early.

Key-words: Collaborative software development. Collaboration conflicts. Empirical software engineering.

RESUMO

Estudos empíricos mostram que conflitos de integração acontecem frequentemente. Essas ocorrências impactam a produtividade dos desenvolvedores, já que integrar contribuições conflitantes é uma tarefa cansativa e propensa a erros. No entanto, de acordo com nosso conhecimento, características como a estrutura das mudanças que levam a conflitos ainda não foram estudadas. Conhecer essas características, e os elementos sintáticos das linguagens envolvidos em conflitos pode lançar uma luz em como evitar conflitos de forma mais otimizada. Com essa finalidade, nesta tese nós derivamos um catálogo de padrões de conflito reportados por uma ferramenta de integração semiestruturada para programas escritos em Java. Esses padrões são expressos em termos das estruturas das mudanças de código feitas e que levaram a conflitos de integração. Nós focamos em conflitos reportados por uma ferramenta semiestruturada que possui conhecimento sobre a sintaxe dos artefatos a serem integrados, para que possamos evitar analisar uma grande quantidade de conflitos irrelevantes tipicamente reportados for ferramentas de integração baseadas em diferença de linhas. Para verificar a ocorrência dos padrões de conflito na prática, nós conduzimos um estudo empírico mostrando que a maioria (aproximadamente 84%) dos conflitos semiestruturados em nossa amostra acontecem quando desenvolvedores editam de forma independente as mesmas linhas ou linhas consecutivas de um mesmo método. Como consequência prática dos nossos resultados, uma possível estratégia para evitar conflitos seria modificar as ferramentas de desenvolvimento para alertar desenvolvedores quando eles editarem o mesmo método, preferencialmente antes da integração de código. No entanto, é possível que desenvolvedores editem partes não relacionadas de um mesmo método, o que não implicaria em um conflito de integração. Dessa forma, essa estratégia de prevenção de conflitos pode tem o potencial de levantar muitos alarmes falsos. Para avaliar a a eficiência dessa estratégia, nós conduzimos um segundo estudo empírico utilizando a plataforma de integração contínua Travis CI para verificar se modificar o mesmo método sem causar conflitos de integração aumenta as chances de se ter problemas de compilação e testes. Outro possível preditor de conflitos que testamos, é editar métodos diferentes, mas com dependências diretas entre si. Nossos resultados mostram que detectar edições no mesmo método pode ser uma estratégia razoável considerando-se uma postura conservadora de detecção de conflitos. Além disso também discutimos idéias que podem trazer melhoras na precisão dos nossos resultados. Em resumo, nesta tese conduzimos dois estudos empíricos com o objetivo de entender diferentes características de conflitos de colaboração. Baseados nas evidências trazidas por esses estudos nós fazemos diferentes tipos de recomendação para se detectar conflitos de forma mais proativa.

Palavras-chaves: Desenvolvimento colaborativo de software. Conflitos de Integração. Engenharia de Software Empírica.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In a collaborative development environment, tasks are usually assigned to developers that work separately without much need of communication using individual copies of project files. As a result, while trying to integrate different contributions back together, one might have to deal with collaboration conflicts. Such conflicts might occur during the merge step (due to merge conflicts), while building the system (due to build conflicts), or when running tests (due to test conflicts).

Previous studies bring evidence that such conflicts might occur frequently, and developers dedicate substantial effort to resolve them since understanding conflicting changes often is a demanding and error-prone task. As a matter of fact, developers might even introduce new code issues while attempting to fix conflicts (ZIMMERMANN, 2007; BRUN et al., 2013; KASI; SARMA, 2013; SARMA; REDMILES; HOEK, 2012; BIRD; ZIMMERMANN, 2012). Perhaps even worse, test conflicts might not be detected during testing, escaping to production releases and compromising correctness. Thus, collaboration conflicts might impair development productivity, and even impact the resulting products' quality.

Such evidence has guided and motivated the development of different tools and strategies that try to mitigate or resolve conflicts. For example, Crystal (BRUN et al., 2013), proactively integrates commits from different developers working on the same project before they push their changes to a shared repository. In contrast, Palantír (SARMA; REDMILES; HOEK, 2012), a workspace awareness tool, informs different developers of ongoing activities in the same repository. Moreover, Syde (HATTORI; LANZA, 2010) is a tool that provides team awareness by capturing developers' changes as atomic AST changes and warns developers if they change the same nodes. Either way, by using these strategies, developers obtain information about conflicts sooner, and are able to resolve the problematic integration scenario before it becomes too complex. Given that it is not always possible to detect merge conflicts before the actual merge, FSTMerge (APEL et al., 2011) is a more elaborated merge tool than the traditional line-based merge tools, such as diff3 (KHANNA; KUNAL; PIERCE, 2007), commonly used by different open-source Version Control Systems (VCS). Because of that, FSTMerge is able to resolve some conflicts automatically that the traditional tools cannot. Consequently, FSTMerge reduces integration effort.

However, despite the existing evidence about conflicts frequency and their impact on collaborative software development, to the best of our knowledge, the structure of the changes that most likely lead to conflicts, expressed in terms of changed syntax elements, has not been studied yet. We believe that understanding conflicts underlying structure and the involved syntactic language elements might shed light on how to better avoid and resolve conflicts. Our goal in this work is to learn through empirical studies about the different types of changes and how frequently they lead to conflicts. In other words, we

want to identify different conflict predictors— as changes leading to conflicts when merged together—, and analyze their effectiveness. Based on such evidence, we intend to suggest improvements to existing strategies or even propose new ones. To achieve this goal, in this thesis we propose, execute and discuss the results of two main empirical studies.

In our first empirical study, discussed in Chapter 3, and also published at the Empirical Software Engineering Journal (ACCIOLY; BORBA; CAVALCANTI, 2017), we focus on merge conflicts. We define merge conflicts as conflicts that can be identified by merge tools while integrating two different code contributions together. To learn about what kinds of changes might lead to merge conflicts, we systematically analyze FSTMerge's semistructured merge algorithm for Java programs to extract all situations that lead to conflicts detected by this tool. As a result, we present a merge conflict pattern catalog containing 9 conflicts patterns, expressing the structure of changes made by developers leading to merge conflicts that can be detected by FSTMerge.

For example, one of the patterns represents the situation when different developers edit the same lines, or consecutive lines from the same method. Another pattern represents the conflict arising from different developers editing different parts of the same class field. For example, if one developer changes the field type, while the other developer changes the initialization value. While trying to integrate these contributions back together, FSTMerge is able to report such conflict.

After deriving the conflict patterns catalog, we used the same FSTMerge tool to reproduce merges and compute how frequently each one of the patterns occur in practice. Among our main findings, we observed that most merge conflicts reported in our sample happen because developers independently edit the same or consecutive lines of the same method. Moreover, we also learn that editing methods is one of the change types that most likely leads to merge conflicts. Based on these findings, we discuss possible implications for existing awareness tools. For example, it seems like an efficient way to proactively detect merge conflicts would be to detect when developers edit the same method, and warn them before the conflict becomes too complex.

However, it is possible that developers edit different unrelated statements inside the same method without causing merge conflicts. If this situation occurs often, then a tool that alerts developers whenever they edit the same method would raise too many false alarms. A tool that raises too many alarms tends to become obsolete or simply ignored by its users. Therefore, we needed further studies to investigate whether editing the same method is an effective conflict predictor in practice. This is the main motivation behind our second empirical study. In particular, we are interested in investigating this conflict predictor's precision, that is, how frequently the conflict predictor presence is associated to a conflict, and its recall, that is, what percentage of conflicts we might avoid by using such predictors.

In addition, a previous work (LIMA, 2014) suggested that changes to directly dependent

methods is also a frequent cause for conflicts. This happens when one developer edits a method that calls a second method edited by another developer. Although this situation does not lead to merge conflicts, it is reasonable to consider that it might increase the chance of having other types of conflicts, such as build and test conflicts. Build conflicts happens when the system build process fails right after integrating the contributions together. For example, if one developer renames a local variable that is later used in a new statement added by the second developer. After integrating the code, the compiler will not find the variable declaration that was renamed. Alternatively, test conflicts happen when one of the system's test cases starts to fail after the integration. This happens when developers edit the same method without causing merge nor build conflicts, but alters the code semantic causing method's observable behavior to change.

Therefore, in Chapter 4 we describe our second empirical study where we investigate the efficiency of two conflict predictor— edits to the same method, and edits to directly dependent methods— by measuring their precision and recall. Our results indicate that the predictors combined together have a precision of 57.99% and a recall of 82.67%. Such results help to guide different strategies for early conflict detection. For example, a more conservative strategy would be to alert developers about a large part of potential conflicts at the cost of dealing with some false positives. In this case, warning developers about all predictor occurrences (editing the same method or the same class field) is a reasonable strategy. In contrast, a strategy that aims at precision, even at the cost of loosing conflicts, would be alerting developers only when they edit the same lines of the same method.

In our second study we also conduct a manual analysis of the false positive instance from our sample, that is, conflict predictors that did not cause conflicts, providing insights about strategies that could further increase the precision and the recall of our results. For instance, one strategy would be to detect when the considered contributions clearly do not interfere with each other. This happens when one of the contributions consists in refactoring changes, for example. When this happens, the awareness tool would not need to trigger an alarm for a potential conflict. Our second empirical study was published at the International Conference on Mining Software Repositories (ACCIOLY et al., 2018).

In conclusion, Chapters 3 and 4 describe the core of this thesis, all of our studies and main conclusions. Then, to link such findings considering previous studies, in Chapter 5 we discuss related work. Finally, Chapter 6 sums up our main conclusions and future work derived from them.

In this chapter we briefly motivate the problem that this thesis tackles— understanding structural characteristics of merge conflicts— and how we managed to address this problem by conducting two different empirical studies to learn what are the most frequent merge conflicts causes in terms of changes performed by different contributions, and how often the occurrence of such changes lead to conflicts in practice. Next, on Chapter 2, we review essential concepts used throughout this work. Namely, commonly uses open-source

version control systems; conflicts arising from collaborative development, and their consequences; unstructured and semistructured merge strategies; and continuous integration practices.

2 BACKGROUND

In this chapter we explain concepts forming the basis to understand this work. First, Section 2.1 describes key concepts about commonly used open-source Version Control Systems (VCS). In particular, we focus on describing the characteristics of Git (GIT, 2018), and GitHub (GITHUB, 2018), which we use throughout this work. Subsequently, Section 2.2 explains how developers work collaboratively and how their work might end up conflicting with each other, thus, creating unwanted consequences such as integration errors, loss of productivity, or even escaped defects. We also present a taxonomy of collaboration conflicts according to the moment when developers commonly detect them. Next, Section 2.3 presents two merge strategies we use in this work: the unstructured merge (KHANNA; KUNAL; PIERCE, 2007) which is widely used in practice and by different open-source VCSs, and the semistructured merge, proposed by Apel et al. (APEL et al., 2011), and improved by Cavalcanti et al. (CAVALCANTI; ACCIOLY; BORBA, 2015), which is able to automatically resolve some of the conflicts that the unstructured merge cannot. Finally, Section 2.4 discusses continuous integration practices covering the basics of Maven (APACHE, 2018) and Travis CI (TRAVIS, 2018) tools.

2.1 VERSION CONTROL SYSTEMS

Version Control Systems (VCS) are fundamental tools in any software development project. They help to keep track of project features, products and keep backed-up versions of the project just in case anything goes wrong. Moreover, they are one of the key enablers of collaborative software development because they provide a shared repository where developers can work together.

We can classify open-source VCSs in two types, the centralized VCSs, and the decentralized ones. The main difference between these two types is that centralized open-source VCSs keep the history of changes on a central repository from which developers pull and push work to. This means that everyone sharing the repository is also sharing everyone's work. The two most popular centralized open-source VCSs are the CVS (FOUNDATION, 2015), and SVN (APACHE, 2015).

In contrast, on decentralized open-source VCSs, also called as distributed VCSs, everyone has a local copy of the entire development history. In practice there is not a central entity in charge of the work's history, so that anyone can sync with any other team member. This helps avoid failure due to a crash of the central versioning server. The most popular decentralized open-source VCSs are Git (GIT, 2018) and Mercurial (MERCURIAL, 2018).

Throughout this work we analyze projects using decentralized open-source VCSs.

Moreover we focus on projects using Git, and GitHub, a web-based Git repository hosting service. GitHub, besides being one of the most important sources of software artifacts on the internet (BIRD et al., 2009), hosts Git repositories, which facilitates the task of understanding conflicts due to its version history model, as we detail further.

Git models the repository version history as a graph where the vertices represent commits, and each commit has an edge pointing to its predecessor commit— the so called parent commit. Figure 1 illustrates such model. In this example, the blue commits represent the main development branch— also called the *master* branch— and the purple commits represent the commits made by a developer in her local repository.

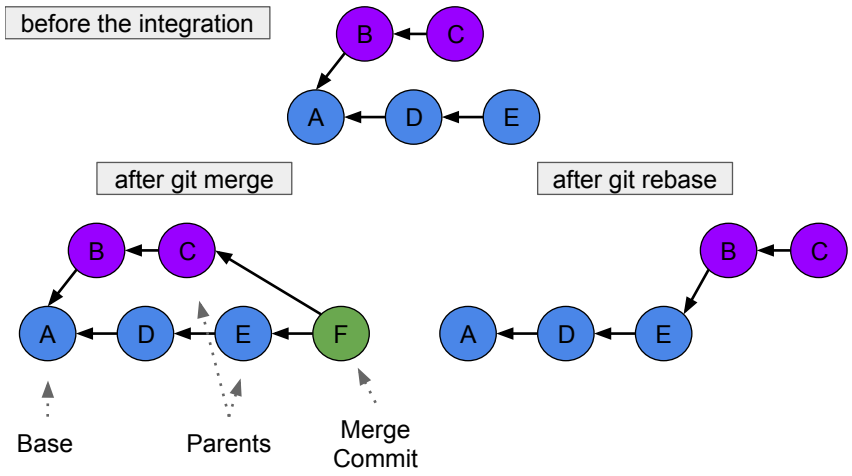

Figure 1 – The difference between *git merge* and *git rebase* commands.

After finishing the task, the developer needs to push her contributions back to the master branch. However, another developer has already evolved the state of the master branch— commits D and E. This means that the developer needs to merge her contributions together with the other developer contributions directly to the master branch. Alternatively, if the developer has no authorization to push changes to the *master* branch, than he or she can open a *pull request*, that notifies authorized developers that new changes are ready to be merged in.

Either way, to merge one's contributions, Git offers two alternatives, the *merge* command, and the *rebase* command. If the developer uses the merge command, Git performs a three-way merge considering commits C and E as parents, and commit A as the base revision from which both branches derived. The result is the merge commit F containing references to both of its parents. This way, in order to identify when developers worked independently and had to merge their work back together, we just need to look for the merge commits— commits having more than one parent— in the repository version history. Alternatively, if the developer uses the *rebase* command, Git replays B and C changes on top of E commit. In such scenario, because there is no commit with more than one parent, the history remains linear, and we cannot track contributions being integrated

back together.

2.2 COLLABORATION CONFLICT TYPES

In a collaborative development environment, it is common to assign different development tasks to developers who implement them using their own copy of the system's files. These developers usually work with the support of a VCS such as the ones we cited in the last section. In addition, they commonly work in a independent way without much need of communication. Then, after implementing the tasks, they try to push their changes to the shared repository. While doing that, they might have to deal with conflicting changes made by other developers contributions.

As a motivating example for that scenario, consider the code snippets in Figure 2 showing that the tasks *Authentication* and *Research Group* were each assigned to different developers, and both had to edit the *Member* class. The developer responsible for the *Authentication* task adds a new field declaration representing a member's username. In addition, both developers add a `toString` method containing the same signature but returning different values.

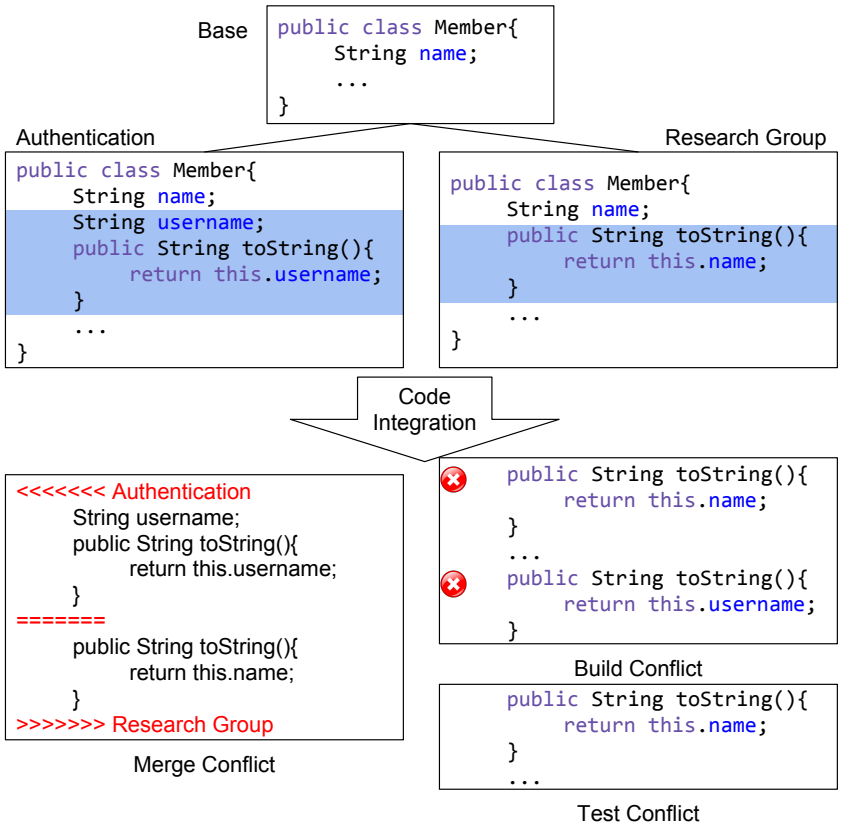

Figure 2 – Merging development tasks might lead to different types of conflicts.

During the task integration process, the first step is to merge the system's files often using merge tools such as Unix diff3 program. The merge tool executes a process called

three way merge, since it consists in applying the changes made by left and right versions of the file, using the base version from which they derived. During this step, the merge program tries to merge new lines from each version, but when developers edit the same line, or consecutive lines, the tool reports merge conflicts. In the current example, this conflict is shown as in the lower left part of Figure 2. At this moment, the merge process is interrupted until the developer manually resolves this conflict.

Imagine that, in order to resolve this merge conflict, the developer deletes the `toString` method added by the *Authentication* task. Then, the merge conflict would be resolved, but probably task *Authentication* would have an unexpected behaviour, which might be detected by the system tests. We classify such test failure as a test conflict. We also consider situations where the automatic merge could integrate both contributions without merge or build conflicts, but with at least one of the tests failing as test conflicts.

Alternatively, a different scenario happens if the developer responsible for resolving the merge conflict decides to resolve the merge conflict by adding both `toString` method versions in different areas of the file. In this case, the merge tool would merge the files successfully, without the presence of merge conflicts. However, the Member class would contain two methods with the same signature. Consequently the compiler would not be able to build this class, resulting in a build conflict due to a duplicate method error. Thus, to resolve this conflict, the developer would probably have to rename one of the `toString` methods.

As a result, collaboration conflicts might impair the development productivity because they require rework, and might not be simple do understand and resolve. Even worse, if test conflicts cannot be detected by the system's existing tests, they might become escaped defects, affecting directly the final product's quality.

In the existing literature, previous studies have measured collaboration conflicts frequency in practice. Zimmermann (ZIMMERMANN, 2007) analyzed 4 projects hosted on CVS, and reports that merge conflicts occurred in a range of 23% to 47% of all files' integration. Meanwhile, Brun et al. (BRUN et al., 2013) and Kasi and Sarma (KASI; SARMA, 2013), which analyzed projects from GitHub, found that merge conflicts occurred in a total of 16%, and 13.3% of project merge scenarios, respectively. Moreover, Brun et al. reported that 33% of all analyzed scenarios in their study represent cases of build or test conflicts. Kasi and Sarma describes this information for each kind of conflict; their results show build conflicts occurrence ranged from 2% up to 15% and test conflicts appear in a range from 6% up to 35% of the merge scenarios respectively.

Kasi and Sarma also provide evidence that conflicts usually persist for days in the repository before someone fixes them. As a matter of fact, Sarma et al. (SARMA; REDMILES; HOEK, 2012) reports that it is common for developers to race to finish their tasks before others, so that they do not have to deal with conflicts while pushing their changes to the shared repository. Altogether, those previous studies provide evidence that merging

is a tiresome and error prone activity which negatively impacts developers' productivity, and might even impact the product's quality, since test conflicts might go on undetected eventually leading to escaped defects.

2.3 MERGE STRATEGIES

After implementing the tasks using their own version of the system's files, developers need to integrate their changes back to the shared repository. For that, there are different merge strategies. In this section we briefly explain and compare two of them: the unstructured, and the semistructured merge strategies which are used throughout this work.

2.3.1 Unstructured Merge

Unstructured merge, also known as line-based merge, is commonly used by various VCSs as their default merge strategy, thus, it is widely used in the industry nowadays. For instance, GNU Merge (GNU, 2015) is one of the tools that implement this strategy. Like all line-based merge tools, they work similarly to the Unix's diff3 algorithm described next.

In a nutshell, the main idea of the diff3 algorithm is to compare the files line by line, detecting the smallest sets of differing lines (chunks). As *diff3* (KHANNA; KUNAL; PIERCE, 2007) describes, for each of the detected chunks, the algorithm checks whether there is a common element in all three revisions, separating the chunk's content into two distinct areas according to the differences between the three revisions. Thus, if the developers modify the content of the same lines, or consecutive lines, the algorithm reports a conflict, separating the different parts of the same chunk.

In this work we use diff3 as an unstructured merge tool. In particular, we execute diff3 in two different ways. First, by calling the `diff3 --merge -E` command, we get the merge conflict display shown on the lower left part of Figure 2. Note that the conflict markers isolate each version of the code — left and right — where the same lines, or consecutive lines changed. With the `E` parameter, diff3 outputs only unmerged changes from the left and right versions of the chunk, ignoring the base version of the chunk. As a consequence, if left and right revisions make identical changes, diff3 would be able to merge them successfully, as shown in Figure 3. Otherwise, if we remove the `E` parameter, diff3 is not able to merge those versions. Instead, it would report a conflict considering the base version of the code, as Figure 3 shows.

The benefits of the unstructured merge strategy are its generality and its performance. It can be applied to all non-binary files, thus, in a system with source code files written in different programming languages, you only need one tool to merge them. In addition, since it compares files' text lines, it has no knowledge of the underlying structure of the files, which makes the merging process really fast. However it might miss conflicts, detect too

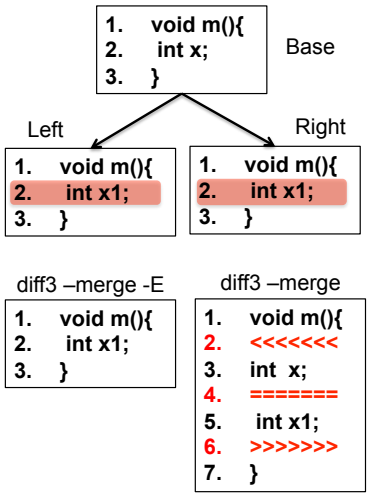

Figure 3 – Running diff3 with and without parameter E.

many conflicts, and it might also produce syntactically incorrect outputs. For instance, in the example of Figure 2, if the developers had put the `toString` methods in separate text areas, the merge would be successful, but there would be a build conflict.

2.3.2 Semistructured Merge

Conversely, the semistructured merge tool FSTMerge (APEL et al., 2011) represents software artifacts as partial trees, the so-called program structure trees, and provides information (through an annotated grammar) about how nodes of certain types (methods, classes, fields, etc.), and its subtrees can be merged. This way, FSTMerge is able to resolve conflicts based on the information that the order of certain elements (classes, methods, fields, imports, and so on) does not matter — the so-called ordering conflicts. For example, Figure 4 shows the difference between diff3 and FSTMerge when merging the same example. While diff3 reports a conflict, FSTMerge knows that for Java files, the order of the methods inside the class does not matter, and places both methods in an arbitrary order. Note that in Figure 2 example where different developers added two `toString` methods, even if they had added such methods in different parts of the file, FSTMerge would match those methods by their signature, thus, reporting a conflict. In summary, FSTMerge not only solves ordering conflicts, but also captures conflicts that diff3 do not report.

Furthermore, we call the trees built by FSTMerge as partial trees because, at the leaves level, FSTMerge represents code elements as pure text. For example, class fields, and method declarations are leaves. Thus, when FSTMerge reaches the leaves, it does not have enough information on how to merge them. So it uses a conventional line-based merge tool to merge the leaves text content.

Previous studies have compared lined-based and semistructured merge tools. FSTMerge provides evidence that semistructured merge reduced the number of conflicts in

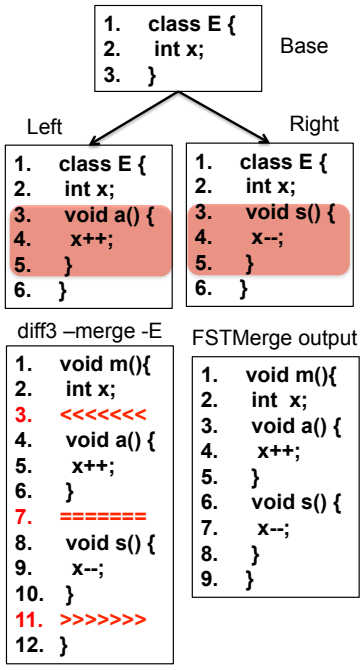

Figure 4 – Ordering conflicts.

60% of the sample merge scenarios by, on average, 34%, compared to unstructured merge. In a replication of this study, Cavalcanti et al. (CAVALCANTI; ACCIOLY; BORBA, 2015) also found similar benefits.

2.4 CONTINUOUS INTEGRATION

Fowler (FOWLER, 2006) defines Continuous Integration (CI) as " a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily— leading to multiple integrations per day". To enable this practice, each integration, or merge, should be verified by an automated build— including tests— to detect build and test errors as quick as possible. According to Yangyang et al. (ZHAO et al., 2017) CI practices have the potential to speed up development and help maintain code quality.

In practice, CI is seeing a broad adoption with the increasing popularity of decentralized VCSs such as Git and their web-based repository hosts such as GitHub. Among the most popular GitHub-compatible, cloud-based CI tools are Travis CI (TRAVIS, 2018), CloudBees,[1] and CircleCI.[2] Moreover, frameworks such as Maven (APACHE, 2018), Gradle[3], and Ant[4] are commonly used to automate projects' build and testing.

In this work we focus on projects from GitHub using Travis CI and Maven as build manager. In this context, each project repository has a Maven configuration file (*pom.xml*)

[1] <https://www.cloudbees.com/>
[2] <https://circleci.com/>
[3] <https://gradle.org/>
[4] <http://ant.apache.org/>

containing instructions about that project's build and tests automation, together with a Travis CI configuration file (*.travis.yml*) describing the Maven commands used to execute the build process.

The CI workflow starts whenever a developer pushes new commits, or accepts a pull request, to the remote shared repository on GitHub and Travis CI gets notified to start the build process. There are 4 possible outcomes for the build automation process on Travis CI. If the build fails, the status is *errored*, if the build executes successfully, but the tests fail, the status is *failed*, and if build and tests run successfully, the status is *passed*. Finally, if an external agent cancels the build process before it finishes, the status is *canceled*. In the context of projects using CI platforms, developers might detect conflicts faster since the build or the test conflicts might fail after the merge. Therefore, it becomes even more important to resolve conflicts faster when using CI to remove build and test problems without compromising other developers productivity who might refrain to merge their changes to a master HEAD containing build and test errors.

2.5 CONCLUSION

In this chapter we explain the different software engineering techniques and concepts that are essential to understand the remaining of this thesis. On the next chapters we present in detail the two empirical studies we conduct using the definitions described here. For example, in the first study we use the FSTMerge tool to derive a catalog of semistructured merge conflict patterns. Then we use this catalog to assess how frequently each pattern occurs in practice. To do so, we analyze the development history of different open-source projects hosted on GitHub to assess in practice how often developers merging contributions ended up having to deal with conflicts. Then, in our second empirical study we analyze how frequently the most common conflict patterns end up causing merge, build and test conflicts. To identify build and test conflicts we rely on the status of building and testing processes executed by the Travis CI (TRAVIS, 2018) service.

3 UNDERSTANDING MERGE CONFLICTS FREQUENCY AND THEIR UNDERLYING STRUCTURE

In Chapters 1 and 2 we argue, based on previous works, how previous studies report that collaboration conflicts occur frequently and impair the development productivity (ZIMMERMANN, 2007; KASI; SARMA, 2013; BRUN et al., 2013). However, to the best of our knowledge, the structure of changes that lead to conflicts has not been studied yet. Understanding the underlying structure of conflicts, and the involved syntactic language elements, might shed light on how to better avoid them. For example, awareness tools that inform users about ongoing parallel changes such as Palantír (SARMA; REDMILES; HOEK, 2012) can benefit from knowing the most common conflict patterns to become more efficient.

With that aim, in this chapter we focus on understanding the underlying structure of *merge* conflicts. At first one might think that merge conflicts do not have a direct impact on software productivity and quality, as the state-of-the-practice merge tools identify merge conflicts, and developers solve them before resuming implementation activities. However, previous studies (SARMA; REDMILES; HOEK, 2012; BIRD; ZIMMERMANN, 2012) suggest the contrary by reporting, based on experimental observations, that resolving merge conflicts is not so trivial. It might take considerable time, and is an error-prone activity.

To better understand merge conflict characteristics, we derive a catalog of conflict patterns for Java programs expressed in terms of the structure of code changes that lead to conflicts. In particular, we focus on conflicts reported by FSTMerge (APEL et al., 2011), a *semistructured* merge tool that is able to automatically resolve a large number of spurious conflicts often reported by typical unstructured, line based merge tools (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015; CAVALCANTI; BORBA; ACCIOLY, 2017). For example, FSTMerge automatically resolves conflicts due to changes involving commutative and associative declarations, such as two methods inserted in the same text area— the so-called *ordering* conflicts. Moreover, FSTMerge is able to detect conflicting situations that line-based tools cannot. Namely, when developers add methods with the same signature— but with different behaviors— to the same file. Such situation likely leads to a build conflict. However, unless developers add both methods to the same text area, line-based tools will not be able to detect such a conflict. From now on, when we mention merge conflicts, we refer to the conflicts that FSTMerge reports.

To derive our conflict pattern catalog, we analyzed FSTMerge's implementation and systematically derived conflict patterns by abstracting **all** kinds of conflicts that can be detected by this tool. Each pattern captures the language syntax elements involved in a conflict, besides the interaction between two revisions that should be integrated, and their common base revision —the so-called *merge scenario*. In particular, we are interested in

the structure of individual changes performed along two different development branches or repository clones, and how they lead to a conflict. For example, the pattern *"Different edits to the same class field declaration"* captures the situation when two developers, working independently, edit the same class field declaration.

To assess the occurrence of such conflict patterns in different systems, we conduct an empirical study that reproduces 70,047 merges from 123 GitHub Java projects. This sample has 10 times more projects, and more than 15 times merge scenarios than related previous studies (BRUN et al., 2013; KASI; SARMA, 2013). Like such studies, we analyze commits having more than one parent— the so-called merge commits— that appear in the development history of the project. These commits might be the result of a pull-request merge or a simple branch merge. However, as we discuss in Chapter 2, sometimes developers hide merge commits from the project development history by using Git commands such as rebase. It is not possible to identify when a rebase happened just by looking at the development history of the repository. For this reason we do not consider rebase merges in our analysis.

Our results show that 84.57% of semistructured merge conflicts in our sample happen because developers independently edit the same or consecutive lines of the same method. This result might seem obvious at first, as most code in Java files appear inside methods. However this is only the case because we used a more sophisticated merge tool that is able to solve ordering conflicts. Contrasting, if we had used a line-based merge tool, such as GNU *diff3* (Free Software Foundation, 2017), a significant part of the reported conflicts would likely be ordering conflicts, which appear outside methods, or even crossing different methods boundaries, as indicated by previous studies (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015; CAVALCANTI; BORBA; ACCIOLY, 2017). Moreover, even if one expects most conflicts inside methods, it would be hard to guess the frequency proportions among different conflict patterns considering edited language syntax elements.

By normalizing the number of conflicts considering the number of changes made to the different language syntax elements, we found out that edits to method lines, class fields, and modifier lists show similar probabilities of leading to merge conflicts. With the obtained evidence, we might say that a reasonable strategy to avoid merge conflicts is to monitor ongoing development activities, and alert developers editing the same method lines, class field or modifier lists so that they can communicate and solve potential conflicts early instead of having to resolve a merge conflict hours or even days after implementing such changes.

Additionally, our results show that merge conflicts happened in 9.38% of the analyzed merges from our sample, with a median of 6.64%, and an IQR (Interquartile Range) of 8.81%. However, we noticed that part of these semistructured merge conflicts are spurious conflicts simply caused by changes to code indentation or consecutive line edits. This motivated us to implement an improved version of FSTMerge that automatically re-

solves such conflicts. Using this adapted tool dropped the total conflicting merge scenario rate to 8.39% (median of 6%, and IQR of 7.21%). This result reinforces the existing evidence (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015) that semistructured merge indeed reduces the number of reported conflicts. And there is still room for improvements.

As a complementary result, by analyzing some patterns of behavior in Git, we notice that developers often do not take full advantage of proper version control systems. Instead of performing merges, they rather copy and paste code around different branches, editing them, and then merging them back together, creating the risk of conflicts. This problem evidences the need for tools that enable partial merges in which developers, instead of merging entire sequences of commits, can break commits into smaller parts/pieces of code and then choose what commits they want to merge.

Finally, our data indicates that merge scenarios often involve more than two developers' contributions, suggesting that merging branches is not likely to be a simple task, since one needs to understand and merge contributions made by different developers probably working on different assignments. This evidence reinforces the need for tools such as TIP-Merge (COSTA et al., 2016) which recommends expert developers for integrating changes across branches.

In summary, in this chapter we make the following contributions:

- Derive a conflict pattern catalog with 9 patterns, and collect evidence on how frequently each pattern occurs using different metrics;
- Implement a slightly improved version of the FSTMerge tool that further eliminates spurious conflicts, and provide evidence that such improvement effectively decreases the number of reported conflicts;
- Report new evidence on merge conflict frequency, by measuring how frequently merges end up with conflicting changes, which allows us to compare our results to previous studies. Moreover, we use a much larger sample, and adopt a more advanced merge tool then other previous studies did;
- Reveal the need for new research studies, and suggest potential improvements to tools that support collaborative software development.

The material used to execute our study, including sample description, tools, and results can be found in our Appendix. The remainder of this chapter is organized as follows. In Section 3.1 we define the study goals, the research questions we analyze and the metrics used to answer them. Then, in Section 3.2 we describe the tools and strategies we implement to conduct the study. Section 3.3 describes all study results. Section 3.4 presents the discussion and actions supported by our results. Finally, in Section 3.5 we present the possible threats to the validity of this study. This chapter was published as a journal at the Empirical Software Engineering Journal (ACCIOLY; BORBA; CAVALCANTI, 2017).

3.1 UNDERSTANDING MERGE CONFLICTS CHARACTERISTICS

Considering the context described in the previous section, our goal in this chapter is to understand characteristics— such as structural patterns, causes, and frequency— of merge conflicts reported when reproducing real merge scenarios from the development history of different software projects. To achieve this goal, we investigate the following research questions.

3.1.1 Research Question 1 (RQ1): What are the structural conflict patterns that can be found by a semistructured merge tool?

To answer **RQ1** we need to derive a conflict pattern catalog, highlighting conflict structures in terms of the program elements independently changed in each of the revisions that lead to the conflict. We derive such catalog by abstracting the kinds of conflicts that can be detected by merge tools. Because we focus on merge conflicts, in this study we do not use tools and strategies such as *Semantic Diff* (JACKSON; LADD, 1994), which compute semantic differences between two versions of the same method, but does not try to integrate them. This leaves us with the following strategies for merge tools: unstructured, semistructured, and structured (APEL et al., 2011).

As explained in Chapter 2, unstructured, line-based merge tools such as diff3 might report too many ordering conflicts. Besides that, it would be hard to systematically derive a catalog of conflict patterns based on edited language syntax elements, as unstructured tools analyze text lines, and have no knowledge about the underlying artifact syntax. We need to have a systematic way to derive these patterns because we want to have patterns that represent all conflicts that can be detected by a merge tool. So, mainly to avoid biasing our sample with a large number of spurious merge conflicts, we decided not to use unstructured tools to derive the conflict patterns.

In contrast, structured merge tools such as JDime (APEL; LESSENICH; LENGAUER, 2012) operate on Abstract Syntax Trees (ASTs), and incorporate full information on the underlying language syntax. However, a drawback of this strategy is that it might introduce defects in the merged version of the code. Consider, for example, when one developer edits the initialization statement of a *for* declaration while the other developer edits the condition statement. In such context, merging these contributions might introduce build or semantic conflicts. Nonetheless, because they edit different statements, the structured strategy is able to successfully merge them, increasing the chance of having build or test conflicts. In addition, there is a considerable performance overhead to use structured tools because they have to build and match the full artifacts' ASTs for every merge scenario we wish to analyze.

Finally, the FSTMerge tool inherits part of the strengths from both unstructured and structured strategies by partially representing software artifacts as trees. It builds the

structured tree until the method level. Method bodies are represented as simple text. It also provides information (through an annotated grammar) about how nodes of certain types (methods, classes, etc.), and their subtrees can be merged. Thus, FSTMerge is able to resolve ordering conflicts based on the information that the order of certain elements (classes, methods, fields, and so on) does not matter. The code elements represented as text— the method bodies— are merged using a conventional line-based merge tool.

In the example described above where two contributions edit the same *for* declaration, if both statements (variable initialization and for condition) are in the same line, or in consecutive lines, FSTMerge and diff3 will report it as a merge conflict. In such cases, we believe that reporting the conflict is a better strategy than merging the code without raising any alarm.

Moreover, besides resolving ordering conflicts, FSTMerge is capable of detecting some types of conflicts that line-based merge cannot. For example, if two developers add to the same class, but in different parts of the text, methods with the same signature, but with different bodies, FSTMerge reports a conflict. Such strategy prevents subsequent build problems while trying to build files with duplicate methods.

We chose FSTMerge to derive our conflict patterns because it has a more sophisticated merge mechanism than diff3, while allowing less false negatives than JDime. However, the list of conflicts detected by FSTMerge is not exhaustive. This tool, as any other merge tool that we could have chosen to derive the conflict catalog, has false positives and false negatives. In fact, in our threats to validity section (Section 3.5) we present a list of FSTMerge false positives and false negatives, together with a discussion on the impact that such cases have on our results.

In order to find the conflict patterns, our starting point was FSTMerge's annotated Java grammar. This file describes a Java grammar with annotations on nodes declarations describing how FSTMerge should handle conflicts in each type of node. For example, the method declaration node has an annotation saying that conflicts within this type of node should be handled by calling the line-based merge approach.[1] Thus, we checked **all** node annotations in the Java grammar and derived the conflict patterns based on the syntactic elements involved. Then, after performing this first analysis, we noticed that two nodes (the class extends declaration, and the enumeration constant declaration) did not have annotations. So we changed FSTMerge annotated grammar to add annotations to these nodes as well. Finally, if our conflict analyzer tool cannot match the conflict with any of the defined 14 patterns, it classifies the conflict in a pattern called "No Pattern". However, none of the conflicts from our sample were classified in this category.

We know describe the resultant catalog containing 9 conflict patterns for Java programs, expressed in terms of the performed kinds of changes to the involved syntactic language structures. With these 9 conflict patterns we answer **RQ1** by categorizing all

[1] <https://goo.gl/9BXCmn>

the conflicts that can be detected by FSTMerge.

EditSameMC

This conflict happens when different contributions edit the same or consecutive lines of the same method or constructor, including lines with the list of modifiers and exceptions. Figure 5 describes this pattern.

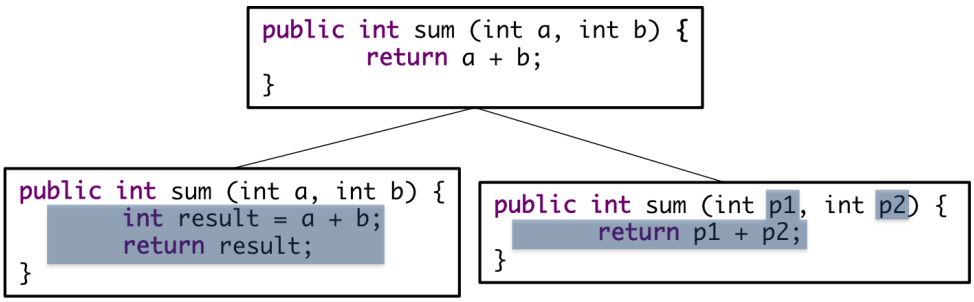

Figure 5 – EditSameMC conflict pattern.

EditSameFd

This conflict happens when different contributions edit the same class field declaration. Figure 6 describes this pattern.

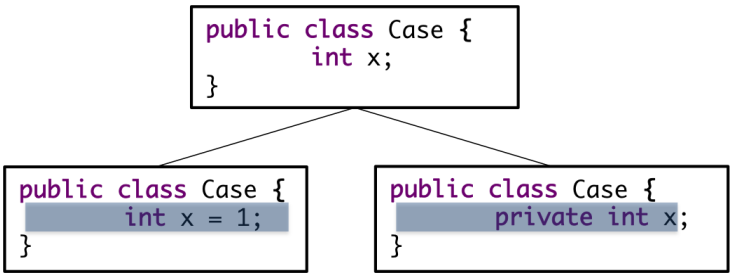

Figure 6 – EditSameFd conflict pattern.

SameSignatureMC

This conflict happens when different contributions add methods or constructors with the same signature and different bodies to the same class. Figure 7 describes this problem.

AddSameFd

This conflict happens when different contributions add a class field declaration with the same identifier and different types or modifiers. Figure 8 describes this conflict.

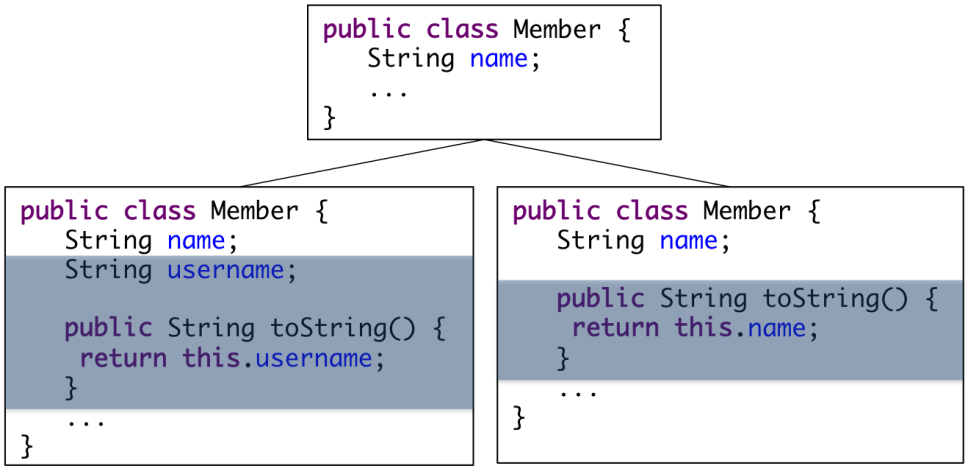

Figure 7 – SameSignatureMC conflict pattern.

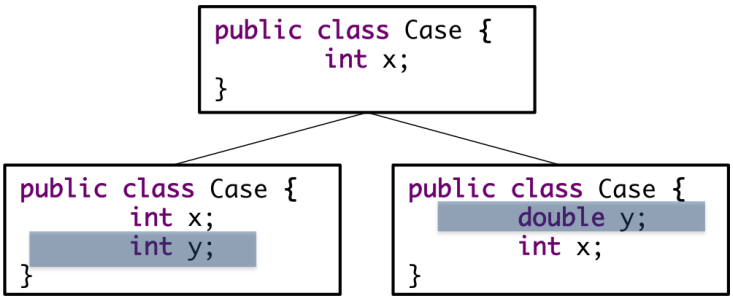

Figure 8 – AddSameFd conflict pattern.

ModifiersList

This conflict happens when different contributions edits the modifier list of the same type declaration (class, interface,annotation or enum types). Figure 9 describes this conflict.

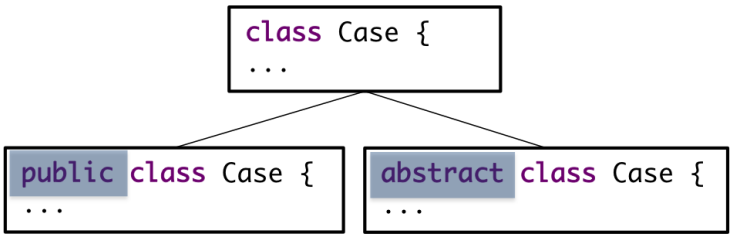

Figure 9 – ModifiersList conflict pattern.

ImplementsList

This conflict happens when different contributions edit the same implements class declaration. Figure 10 describes this conflict.

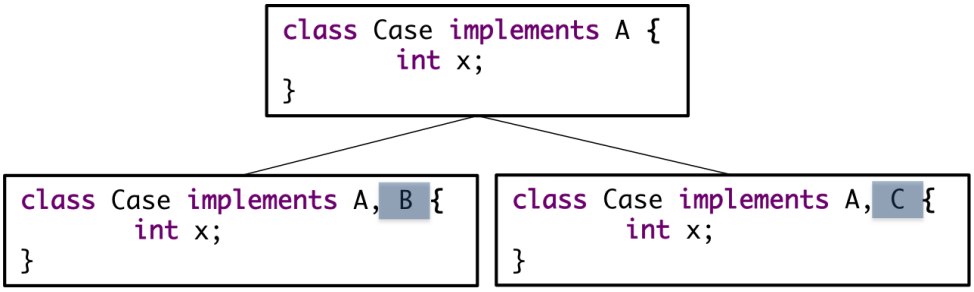

Figure 10 – ImplementsList conflict pattern.

ExtendsList

This conflict happens when different contributions edit the same class extends declaration. Figure 11 describes this conflict.

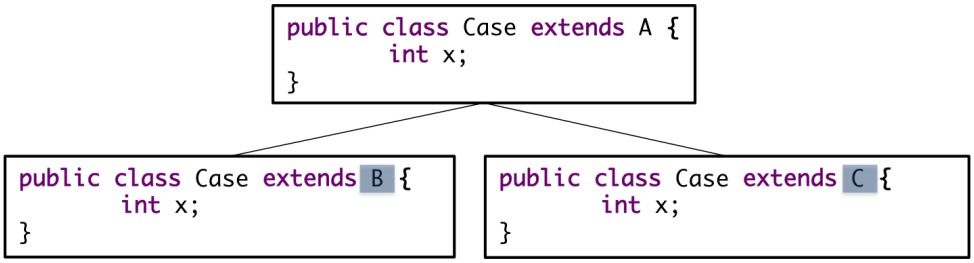

Figure 11 – ExtendsList conflict pattern.

EditSameEnumConst

This conflict happens when different contributions edit the same Enum constant declaration. Figure 12 describes this conflict.

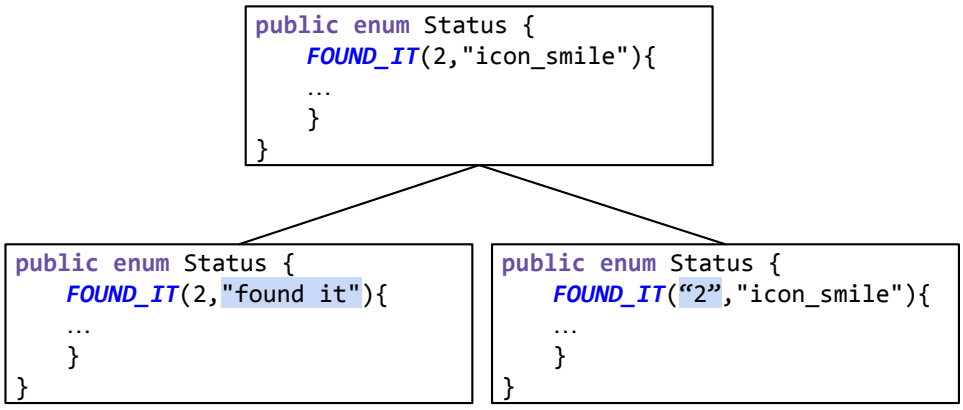

Figure 12 – EditSameEnumConst conflict pattern.

DefaultValueA

This conflict happens when different contributions edit to the same annotation method default value declaration. Figure 13 describes this conflict.

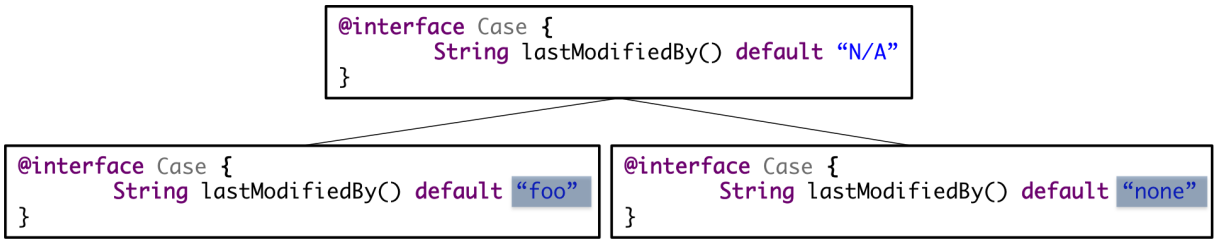

Figure 13 – DefaultValueA conflict pattern.

We add the word "different" in our edit related conflict patterns to remind that, if developers make equal edits— such as adding the same *get* method— there is no conflict, since their contributions do not interfere with each other. In this case, a straightforward solution would be to simply merge the contributions choosing the first developer's version.

To better illustrate our conflict patterns, Figure 14 shows an example of the EditSameMC pattern. We found this conflict while analyzing the OpenTripPlanner project, an open-source trip planner application.[2] Note that, in this example, both developers edited the declaration of variable *optionsBefore*. Hence, the merge tool reported this conflict so that it could be manually resolved.

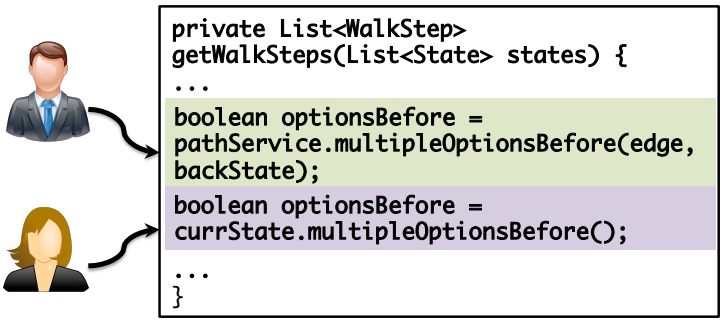

Figure 14 – Example of the EditSameMC pattern occurence.

Although FSTMerge currently supports code written in Java, C#, and Python, for simplicity, we only analyze Java projects. Considering other languages would require different conflict pattern catalogs and associated analysis. Deriving a conflict pattern in a different language that FSTMerge supports would not be difficult. One would need to inspect the language grammar file on FSTMerge project looking for the annotated nodes, and then derive the patterns expressed in terms of the changes involving the specific syntax elements.

[2] <http://www.opentripplanner.org/>

However, some of our general patterns, such as EditSameMC and EditSameFd, would also apply for C# and Python. But each language syntax particularities could add or remove patterns from our catalog. For example, C# would have a pattern for when developers edit a directive with the same alias. In contrast, a catalog for Python would not consider the ModifiersList pattern, since Python does not have explicit access modifiers. In the threats to validity section (Section 3.5) we discuss how such a decision affect our study.

3.1.2 Research Question 2 (RQ2): How frequently does each merge conflict pattern occur?

After deriving the conflict pattern catalog, we are able to answer **RQ2** by reproducing real merge scenarios from the entire development history of different Java projects, while collecting the absolute number of conflict occurrences for each conflict pattern from our catalog, using the following metric:

- Number of conflicts

By answering **RQ2** we will learn how frequently the different conflict patterns occur, and the frequency proportions among them. In Section 3.2 we describe the experiment setup used to answer **RQ2** and the remaining research questions. Then, in Section 3.3 we present our results.

3.1.3 Research Question 3 (RQ3): What kinds of conflict patterns most likely lead to conflicts?

While the number of conflicts shows conflict patterns that occur more frequently, we complement this information by understanding the probability of ending up with a merge conflict when editing different language syntax elements. To this end we compute the following metric:

- Normalized number of conflicts = $\frac{Number\ of\ conflicts}{Number\ of\ changes}$

We compute the normalized number of conflicts by dividing the number of conflict occurrences from each pattern by the number of involved syntax elements changed during the entire project development history. For example, if, during the development history of a particular project, we observe 50 EditSameMC conflicts, and 500 edits to method or constructor elements, the normalized number of conflicts for the EditSameMC pattern would be 0.1, meaning that, when editing a method or constructor, there is a 10% chance of introducing EditSameMC conflicts. In contrast, if, in this same project, we observe 5 EditSameFd conflicts, and 10 edits to class field elements, the normalized number of conflicts for EditSameFd would be 0.5 or 50%. In such context, although EditSameMC

conflicts are 10 times more frequent than EditSameFd conflicts, the probability of having EditSameFd conflicts when editing class fields is 5 times higher.

We compute both metrics because they complement each other. The number of conflicts shows which conflict patterns occur more frequently, whereas the normalized number of conflicts is useful to understand what kinds of code changes most likely lead to merge conflicts. One of the goals in our study is to provide recommendations for detecting conflicts more efficiently while working collaboratively. Therefore, computing both metrics is useful to improve collaborative development tools' recall by helping them to detect the most frequent conflicts while being careful about important conflict predictors.

Furthermore, because FSTMerge runs *diff3* to merge methods and constructor elements, we compute the normalized number of conflicts for the EditSameMC pattern considering not only the number of changed method and constructor elements, but also the number of changed line chunks, that is, blocks of commands edited together, and the total number of changed lines inside methods and constructors. Thus, we have different alternatives to analyze the results.

3.1.4 Research Question 4 (RQ4): How frequently do merge conflicts occur?

By answering this question we would like to know how frequently developers have to deal with conflicting changes when merging different code revisions. For this purpose we use the following metric:

- Conflicting scenarios = $\frac{Merge\ scenarios\ with\ conflicts}{Merge\ scenarios}$

The conflicting scenarios metric measures the ratio of merge scenarios having at least one merge conflict by the total number of analyzed merge scenarios. Thus, it gives us the intuition of how often the merge process fails. This metric was also used in previous studies (KASI; SARMA, 2013; BRUN et al., 2013). This way we can compare our results to theirs.

3.1.5 Pilot Study Outcome

With the purpose of testing our infrastructure, which we describe in Section 3.2, we ran a pilot version of this study with a subsample of 40 projects from a larger sample that we selected according to the requirements we describe in Section 3.2.5. A surprising result was that SameSignatureMC was the second most frequent pattern, representing approximately 13% of the conflict occurrences. It seems unlikely that developers working independently would so often add, to the same class, methods with the exact same signature and different bodies. To better understand the situation, we manually analyzed a few examples of SameSignatureMC conflicts to understand their underlying causes.

During this analysis, we noticed that some of those duplicate methods were simple methods such as getters and setters, which seems to be reasonable. However, some of

those duplications seemed odd as they were large methods (more than 100 lines, for example), and only small parts of them differed. For example, Figure 15 illustrates a conflict extracted from project Jitsi, an instant messenger application. In this example the *sendFile* method is long— it has more than 100 lines— and only the highlighted part of the figure differs. When we checked Jitsi development history we noticed that in a certain commit one developer added the *sendFile* method. Then, on a different branch, another developer copied this method and made a few changes. Finally, when merging the changes, the conflict occurred. We saw other examples like these. In fact, we even found examples where, instead of copying one method, the developer copied the entire file from one repository to the other.

```
FileTransfer sendFile(...){
    //(...)
    OperationSetMultiUserChat mucOpSet = jabberProvider
        .getOperationSet(OperationSetMultiUserChat.class);
    if(mucOpSet != null
        && mucOpSet.isPrivateMessagingContact
            (toContact.getAddress()))
    {
        fullJid = toContact.getAddress();
    }
    //(...)
}
```

```
FileTransfer sendFile(...){
    //(...)
    if(jabberProvider.getOperationSet
        (OperationSetMultiUserChat.class)
        .isPrivateMessagingContact(toContact.
            getAddress()))
    {
        fullJid = toContact.getAddress();
    }
    //(...)
}
```

Figure 15 – SameSignatureMC example from Graylog2-server project.

In other examples, the duplicated method existed previously in the base revision and was equally renamed in two different branches. For example, on project Async-http-client, an asynchronous Http Client for Java programs, there was a method called *onHttpError* in the base revision. Then, on subsequent commits from different branches this method was equally renamed to *onHttpHeaderError*. By analyzing the project development history, we found out that this method is overridden from the Grizzly project API.[3] This method was renamed in the new API version, and when the dependency was updated in the Async-http-client project, the system build no longer worked, causing developers to rename this method across different repositories.

After this analysis, we decided to further detail our study setup to automatically analyze SameSignatureMC occurrences matching them with their underlying causes, as further explained in Section 3.2. Understanding such causes is useful to derive new requirements for tools supporting collaborative development. For this reason, we added an extra research question that we describe as follows.

3.1.6 Research Question 5 (RQ5): How frequent are the underlying causes of the SameSignatureMC pattern?

We consider the following causes for SameSignatureMC occurrences and measure their frequency:

[3] <https://grizzly.java.net/>

- Copied files: a developer copies an entire file from one branch to another, changes it and tries to merge the branches;
- Copied methods: a developer copies a method from one branch to another, changes it and tries to merge the branches;
- Small methods (getters or setters): simple methods, containing no more than 3 lines of code, that could have been added by two developers working independently;
- Renamed methods: methods that are equally renamed in different branches that are merged;
- Others: all SameSignatureMC conflicts that we cannot classify in any of the previour categories.

We consider that copying and pasting across repositories does not necessarily imply code cloning because, after the copy, the developer merged the branches. In this scenario, after resolving the merge conflict, the code is no longer duplicated (cloned) in the repository. If the branches were never meant to be merged (branches for different products, for example) then we would have a code clone across software systems, like the ones detected in previous studies (SVAJLENKO et al., 2014).

3.2 STUDY SETUP

In this section we describe the setup of the designed study, including how we implement the tools and scripts to measure the metrics defined in the previous section, and also how we select the sample projects to conduct the study.

3.2.1 Conflict Analysis

The infrastructure that we built to run our study can be divided in two steps: mining and merging. Figure 16 describes the elements involved in our experiment setup and how they relate with each other. In the mining step we have a script that clones a GitHub project locally and runs the command `git log --merges` which provides a list containing information about all merge commits of that project— commits that were the result of a *git merge* command. Subsequently we parse the result of this command to retrieve a list of all merge commit *ids* and their parent *ids*. Each merge commit has two parents that we call from now on as left and right revisions.

After retrieving the list of merge commits, we use the *JGit* API (ECLIPSE, 2015) to checkout and copy the three revisions involved in the merge scenario— the common base revision, and the left and right revisions derived from the base revision and later merged into the merge commit. Then we perform three-way merges using the *git merge* command— which uses *diff3* as default merge tool, and therefore can be applied to any

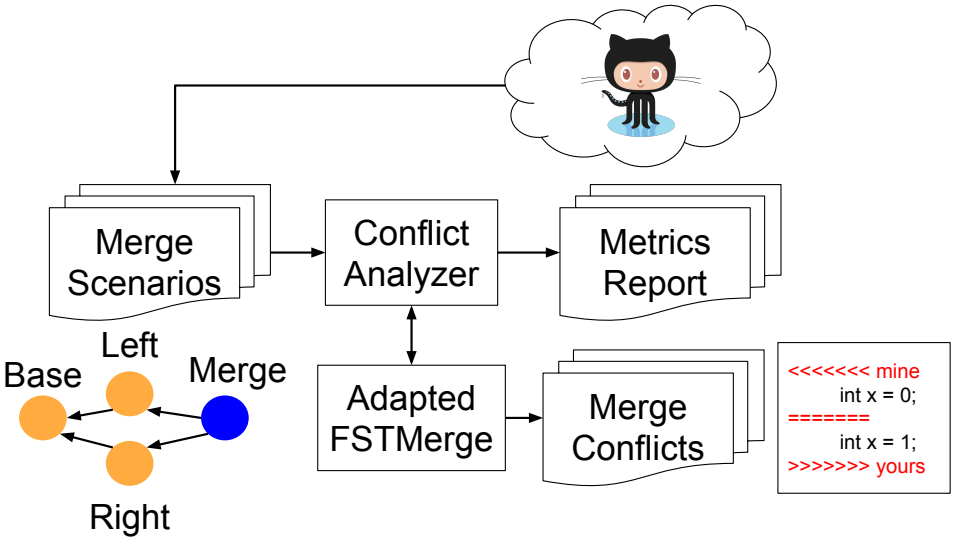

Figure 16 – Study infrastucture setup. Everything starts when it clones locally a project repository from GitHub. Then a scripts retrieves all merge commits in the master branch development history. Next, for each merge scenario, we run the Conflict Analyzer tool which calls our adapted version of FSTMerge to reproduce the merge scenario. Everytime FSTMerge reports a conflict, the Conflict Analyzer captures it and computes the metrics defined to answer our research questions.

text file— to merge non Java files, and an adapted version of FSTMerge to merge Java files. This way we compute the conflicting scenario rate considering all files in the revisions, and not only Java files.

Our adapted version of FSTMerge contains the following new features:

1. An observer (GAMMA et al., 1995) that intercepts FSTMerge main mechanism and collects all reported conflicts;

2. We changed FSTMerge's annotated grammar to report the ExtendsList and EditSameEnumConst patterns, which were missing in the original version;

3. We had to discard Java files that could not be parsed by FSTMerge. In particular, it cannot parse the constructs related to lambda expressions and type annotations from Java 8. However, these Java 8 new features are not the single cause for parser errors. We observed that some parsed files had syntax issues as well. For example, some files were committed to the main repository containing conflict headers. The bad parsed files correspond to 0.16% of the total number of Java files in our sample;

4. We changed FSTMerge's default line-based merge tool from Revision Control System's (RCS)[4] to Unix's *diff3*; RCS is no longer maintained for *mac os* and we wanted

[4] <http://www.gnu.org/software/rcs/>

to run the same tool across different operating systems to increase the study replicability. This change does not impact the results, as RCS uses *diff3* in the background to merge files

We also implement a component called Conflict Analyzer which calls FSTMerge to integrate the merge scenarios. When our adapted version of FSTMerge calls *diff3* to merge tree leaves, it first executes the command *diff3 –merge -E*. The parameter *E* makes *diff3* ignore the distinction between tabs and spaces on input. Then, if the output contains conflict markers, we call *diff3* again, but this second time without parameter *E*. We do that because the parameter *E* removes the base code version from the conflict body, and we need the base version in order to run further analysis on conflicts, as explained later.

Then, every time FSTMerge is about to merge the content of a node containing conflicts, our observer method notifies the Conflict Analyzer about it. The Conflict Analyzer receives as argument the node containing the three versions of the code (left, base, and right). Then, depending on the node type it can classify the conflict directly to its respective pattern. For example, if it is a class modifier node, this conflict is classified as a ModifiersList conflict. In contrast, if the node is a class field, we need to check if this field exists in the base version of the code. If it exists, we classify this conflict as a EditSameFd conflict. Otherwise, we classify it as a AddSameFd conflict. At the end, if we cannot match the conflict to any of our defined patterns, then we classify it as a noPattern conflict. The method responsible for classifying conflicts can be found in our GitHub repository.[5]

Besides classifying the conflicts according to their specific pattern, the Conflict Analyzer tool is responsible for computing the other metrics described in Section 3.1 and we describe in more details in Section 3.2.2

3.2.2 Identifying Different Spacing, and Consecutive Line Edit Conflicts (Potential False Positives)

After classifying conflicts to their respective patterns, the Conflict Analyzer tool evaluates the conflict body content (base, left, and right versions) to identify two situations that likely represent spurious conflicts: different spacing, and consecutive line edit conflicts. Figure 17 illustrates both cases. The left side of the figure illustrates a different spacing conflict example. On both revisions, the same line was edited, but only the right revision made significant changes to the code: added a parameter to the method declaration. The left revision simply altered code formatting.

Our tool is able to identify this type of false positive by comparing left and right revisions to the base revision, ignoring tabs, spaces and line breaks. We treat the three versions of the methods as strings. If one of the revisions (left or right) is equal to the base, we classify this conflict as a different spacing conflict. We can then factor out this

[5] <https://goo.gl/tkYrSh>

kind of conflict in our results, focusing on the analysis of conflicts that are likely more relevant.

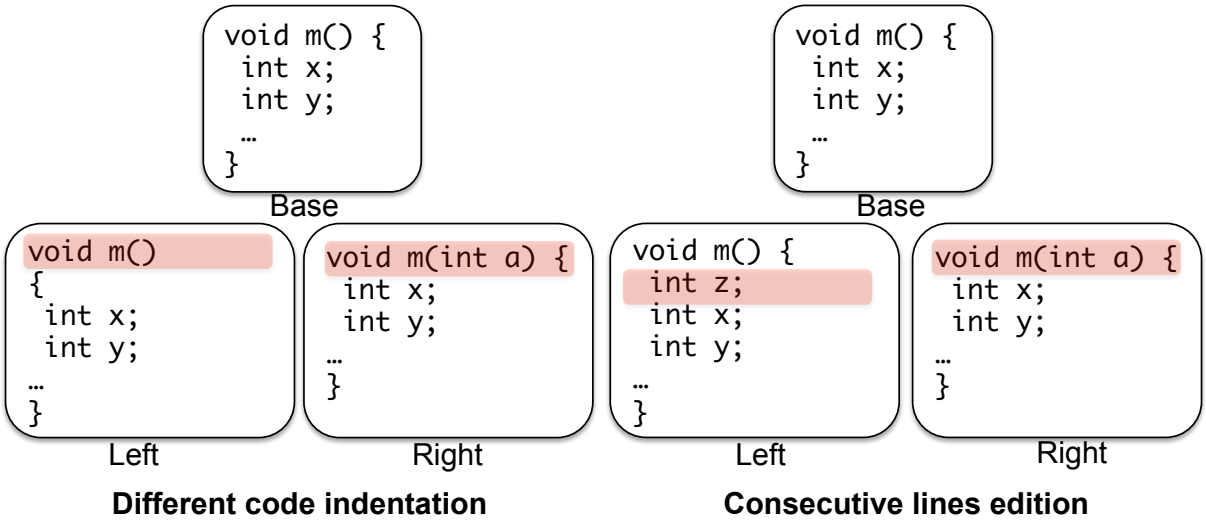

Figure 17 – Types of conflicts that diff3 cannot merge.

On the right side of Figure 17, both revisions apply significant changes, but *diff3* algorithm is not able to merge them because consecutive lines were edited (KHANNA; KUNAL; PIERCE, 2007). While we analyze spacing conflicts for all conflict patterns in our catalog, we only look for consecutive lines edition conflicts on EditSameMC occurrences, because it is the only case when FSTMerge calls the *diff3* algorithm. We identify such conflicts using string comparison to check if only consecutive lines were edited.

While spacing conflicts always represent false positives, there is a risk that consecutive line edits conflicts might lead to semantic conflicts. For example, if the definition of a *string* variable takes more than one code line, and each developer edits one of the lines. So the analysis factoring out this kind of conflict should be considered with care. For this reason we present separate results to consider the incidence of conflicts with and without consecutive line edits conflicts.

We identify such cases to compare the overall numbers with narrowed down numbers that try to focus on the more interesting merge conflicts, that is, conflicts that have a greater chance of representing an interference between development contributions. Besides that, both conflict types are simple to resolve automatically. A straightforward solution to resolve spacing conflicts is to replace the conflict body with the significant changed version of the code. Moreover, allowing *diff3* to merge code when consecutive lines are edited could solve the second example from Figure 17.

3.2.3 Identifying the underlying causes of SameSignatureMC conflicts

After identifying spacing and consecutive line edit conflicts, there is one extra step applied for the SameSignatureMC conflicts to understand and quantify its underlying causes. To

automate this analysis, we first check if the file containing the conflicting method or constructor exists in the base revision. In case it does not, we classify this occurrence in the "Copied file" category. Such situation happens when one developer adds one file to her repository. Subsequently, another developer copies (instead of pulling and merging) this file to her local workspace, and alters one of the methods body.

Conversely, if the file containing the method exists in the base revision, we check the method size and its name. If the method name contains the words *get* or *set*, or it is a small method— with no more than 3 lines of code— we consider the conflict to be in the "Small method" category. This situation we believe is more reasonable to expect: two developers working independently felt the need to add getters, setters or other kind of simple methods to the same class.

However, if the conflict was not classified in the "Copied file" or in the "Small method" categories, there are still three following categories: "Copied method", "Renamed method", and "Others", whose classification is more elaborate. First, because the SameSignatureMC conflict body has no base version, we first compare both left and right method versions using the Levenshtein distance algorithm (LEVENSHTEIN, 1966) to check for string similarity. If the method bodies are considered similar enough— we discuss our similarity analysis in the following paragraphs—, we consider the methods to be the same. Otherwise, we classify them in the "Others" category.

If the methods are similar enough, we look for a method in the base AST that FSTMerge was not able to match with any other method node and that is similar enough to the other two matched methods. If we find this method, we classify this conflict in the "Renamed method" category. That is, two contributions renamed the same method in different branches, and then tried to merge them. Finally, if we do not find a similar method in the base file, we classify the conflict in the "Copied method" category.

Regarding our string similarity analysis, we use Levenshtein original algorithm version considering insertion, deletion, and substitution of characters. The extended version also considers the transposition of two adjacent characters. This extension would be useful to measure the distance between smaller strings such as words, when, for example, two adjacent characters are displaced in a typo. In our work, we compare larger strings (entire method declarations with more than 3 lines of code), so this feature would be less useful. It would capture, for example, situations like when a local variable has its name slightly changed, but we believe that our threshold— as discussed next— is able to consider such cases.

We consider methods to be the same if the similarity value is greater than or equal to 70%. At first, we conducted some manual tests and found that 70% could be a reasonable threshold. However, to gain more confidence in this choice, we executed our analyses considering 68 randomly selected projects from our original sample, using 3 different similarity thresholds (>=60%, >=70%, and >=80%). We found out that, for the Renamed

Method category, a total of 78.6% of the renamed methods in the sub sample fall in the >=80% category, and an additional 11% is considered if we use the >=70% category. For the Copied Method category, 84.4% fall in the >=80% category, and an additional 8% is considered using the >=70% category. Hence, we considered 70% to be an acceptable threshold value, since we get most part of the renamed and copied methods (more than 80% similar), and we are still able to get some of the renamed, and copied methods having similarities between 70% and 80%.

3.2.4 Normalized number of conflicts analysis

To measure the normalized number of conflicts we need to compute the number of conflicts for each conflict pattern, and divide this number by the sum of all changes made to the involved language syntax elements during the entire project history. Figure 18 illustrates how we compute both metrics. The top part of the figure shows a graph representing the development history of a project hosted on Git. In this graph, the vertices represent the commits, and each commit has an edge pointing to its parent (or parents, in the case of merge commits such as commits E and G).

We compute the number of conflicts while using Conflict Analyzer and FSTMerge to reproduce the merge scenarios. Then, to compute the total changes for a given kind of syntactic element in the project history, we run the *git log* command that provides information about all commits in that project, and parse the output to retrieve the list of all commits ids and their parents ids. Then we use our adapted version of FSTMerge to compute the difference between each commit and its parent in terms of the kinds and numbers of nodes changed between each commit and its parents.

Note that in the *CHANGES* formula we sum changes only from regular commits, and exclude changes from merge commits. Otherwise we would be summing up most of the changes twice, because changes made on regular commits before the merge are replicated on merge commits. Moreover, since FSTMerge calls *diff3* to merge changes inside methods and constructors, besides computing the number of changed method and constructor nodes, we also compute the number of line chunks, and number of lines changed inside methods and constructors, so that we can have different metrics to compare.

3.2.5 Sample

To select our subjects sample, we used GitHub's advanced search page,[6] and configured the search to filter Java projects with more than 500 stars ordered by projects' recent activity. By filtering GitHub search by the number of stars we likely select more meaningful and popular projects, avoiding toy projects. From this search results, we randomly selected 123 projects out of 1,963.

[6] <https://github.com/search/advanced>

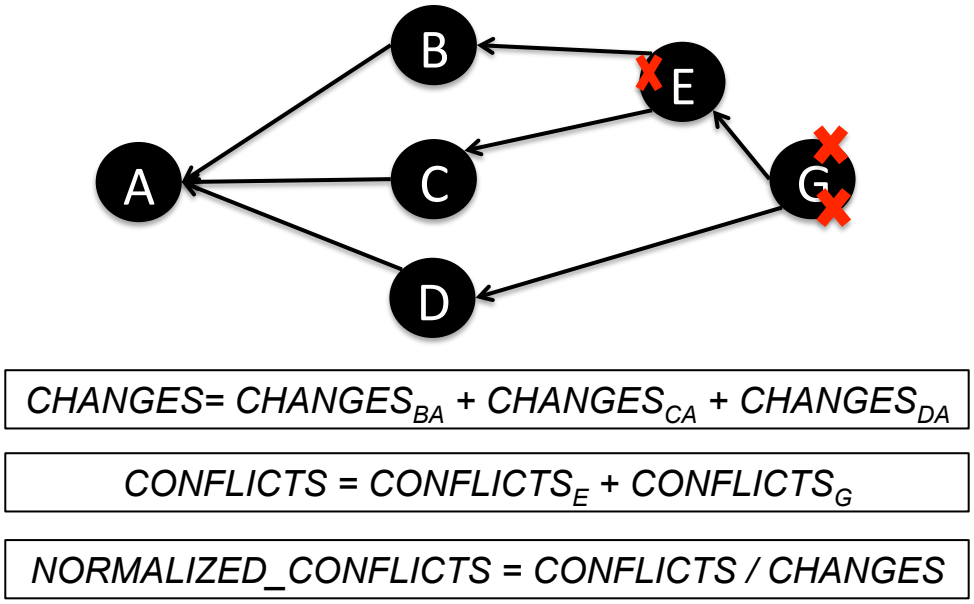

Figure 18 – Computing the number of conflicts, and the probability of ending up with conflicts while editing different language syntax elements.

Although we have **not** systematically targeted representativeness or even diversity (NAGAPPAN; ZIMMERMANN; BIRD, 2013), by inspecting our sample we observe some degree of diversity with respect to the following dimensions: size, domain, and number of collaborators. Our sample contains projects from different domains such as databases, search engines, games, and frameworks. They also have varying sizes. For example, SimianArmy, a cloud computing tool suite from Netflix, has only 4 KLOCs, while Osmand, a navigation application, has approximately 640 KLOCs. Moreover, Exhibitor has 20 collaborators, while Cassandra has 112 collaborators. Besides that, we also selected projects that are widely used by software developers, and that were analyzed in previous studies, including Junit, Jenkins, Cassandra, Gradle, and Voldemort (KASI; SARMA, 2013; BRUN et al., 2013; CAVALCANTI; ACCIOLY; BORBA, 2015). For further information on our sample, we provide a complete subject list in our Appendix.

3.3 RESULTS

In this empirical study we analyze 70,047 merge scenarios considering the entire version history of 123 projects hosted on GitHub. In this section, we present descriptive statistics of the results structured according to the research questions.

3.3.1 RQ2: How frequently does each merge conflict pattern occur?

To answer **RQ2**, we collected a total of 28,883 conflicts reported by our adapted version of FSTMerge, from the total of 4,141 merge scenarios with conflicts on Java files. Table 1 describes the absolute number of conflicts collected according to its pattern, with, and

without the potential false positives (spacing and consecutive lines conflicts). Also, none of the collected conflicts was classified in the NoPattern category. Figure 19 describes the conflict pattern distribution. We found out that EditSameMC was, by far, the most frequent conflict pattern, representing 84.57% of the collected conflicts. The second most frequent pattern was EditSameFd, followed by SameSignatureMC, AddSameFd, ModifierList, ExtendsList, and ImplementList. Moreover, we did not collect any conflicts from the DefaultValueA, which happens when two revisions edit the same default value of an annotation method declaration.

The lower part of Figure 19 shows the bar chart after removing the potential false positive conflicts, that is, conflicts due to different spacing, and consecutive line edits. 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 edits, and the remaining 14.31% were due to both reasons. EditSameMC was the pattern that had most occurrences of those conflict types (32.21%). However, after removing spacing and consecutive line edit conflicts, EditSameMC is still the most frequent pattern, representing a total of 80.71% of the collected conflicts, without changing the big picture of our results.

Table 1 – Absolute number of conflicts.

	Absolute number	**Removing potential FP**
EditSameMC	24427	16557
EditSameFd	1578	1386
SameSignatureMC	1505	1275
ModifiersList	1020	1004
AddSameFd	136	112
ExtendsList	102	95
ImplementList	81	52
EditSameEnumConst	34	34
DefaultValueAnnotation	0	0
TOTAL	28883	20515

Since the total conflict percentages depicted by Figure 19 could be biased by outliers with high occurrence of EditSameMC conflicts, we checked whether conflict pattern occurrences follow a similar tendency across projects. The boxplots in Figure 20 show the conflict pattern percentage distributions considering all projects in our sample after removing the spacing and consecutive line edition conflicts. By analyzing these boxplots, we observe that, for more than 75% of the projects from our sample, more than 60% of the collected conflicts were from the EditSameMC pattern. Moreover, the EditSameMC boxplot is heavily skewed to the right, whereas the other pattern boxplots are heavily skewed to the left. Therefore, we conclude that for most, but not all projects, we confirm the

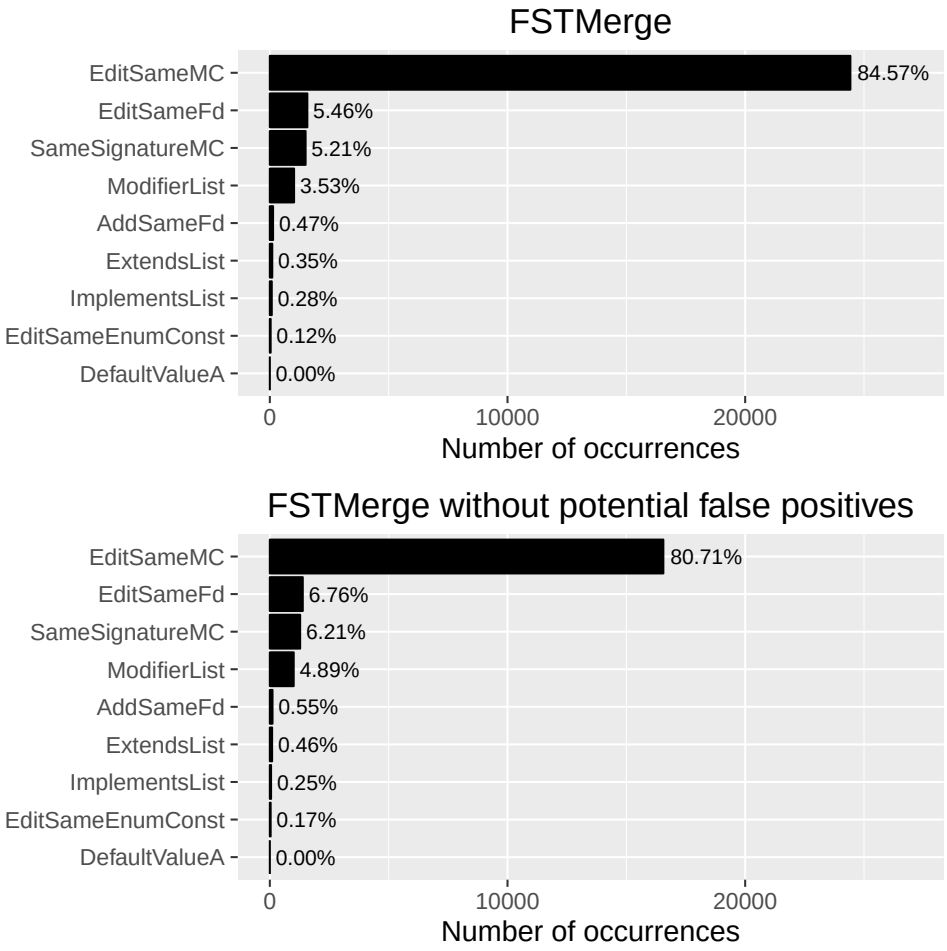

Figure 19 – Bar charts showing the conflicts pattern distribution with and without potential false positive conflicts.

same tendency of the total sample of conflicts, since the pattern with higher percentages is indeed EditSameMC.

3.3.2 RQ3: What conflict patterns most likely lead to conflicts?

We answer **RQ3** by computing the ratio between the number of conflicts and the number of nodes changed during the project development history. In addition, for the EditSameMC conflicts we use three different metrics. We divide the number of conflicts by the number of changes in method nodes, by the number of line chunks— block of lines that were edited together—, and by the total number of lines changes inside methods.

Table 2 summarizes the aggregated results. While EditSameMC Nodes and EditSameMC Chunks are by far the most conflict prone changes, the others have probabilities lower than 0.1%. To further compare them we use the Wilcoxon signed rank test (WILCOXON; WILCOX, 1964) combined with the Bonferroni correction method for multiple comparisons (BONFERRONI, 1936). Comparing the normalized number of conflicts The results show a statistically significant difference when we compare EditSameMC

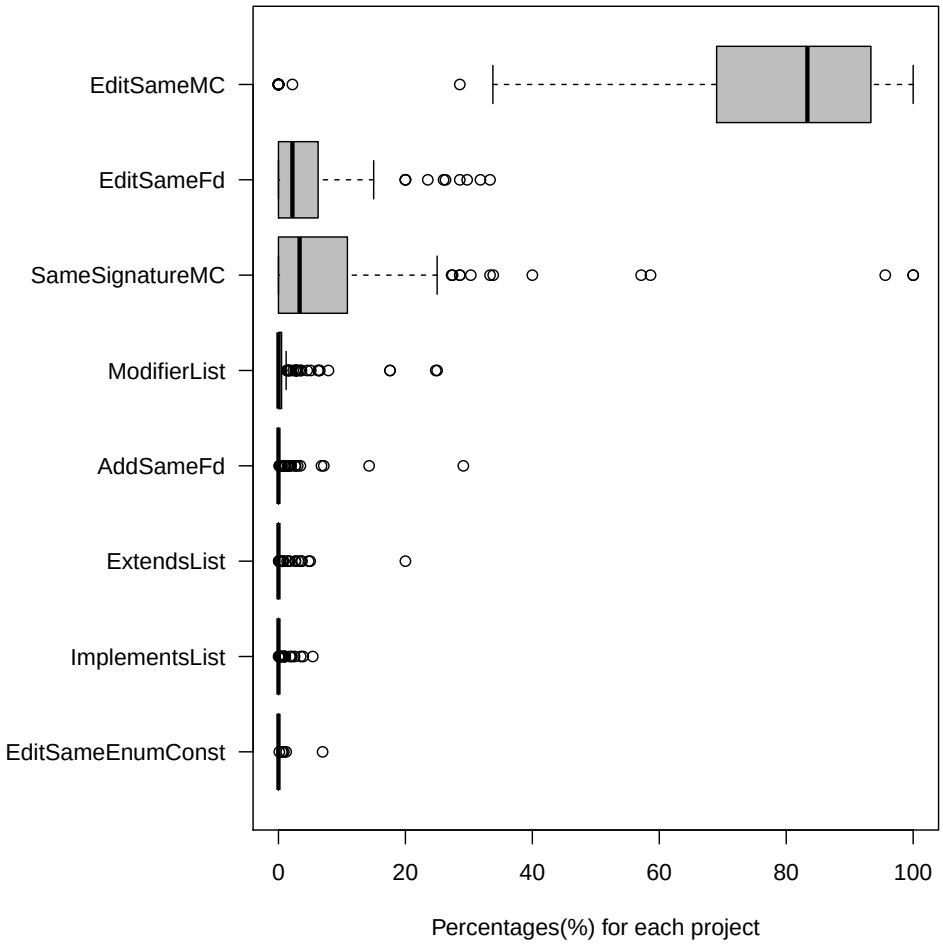

Figure 20 – Boxplots showing the dispersion of the conflict patterns percentages across projects.

Chunks to EditSameFd, SameSignatureMC, and ModifiersList (p-values < 0.01). However, there is no statistically significant difference when we compare EditSameMC Lines (each line edit inside a method counts 2 changes) to EditSameFd, SameSignatureMC, and ModifiersList.

Figure 21 top part depicts the boxplots containing normalized number of conflicts per project, computing EditSameMC normalization by the number of changed lines. The lower part of Figure 21 shows the boxplots describing the absolute number of conflicts per project. We can see that in the lower figure boxplots, EditSameMC is by far the most frequent conflict pattern, followed by SameSignatureMC, and EditSameFd. However, in the top graph of Figure 21 although there is not a statistically significant difference between the observations, EditSameFd has indeed higher values than EditSameMC, and SameSignatureMC.

Table 2 – Probability of having merge conflicts while editing different language syntax elements.

Pattern	**Probability**
EditSameMC Nodes	0.30%
EditSameMC Chunks	0.26%
EditSameMC Lines	0.03%
SameSignatureMC	0.03%
EditSameFd	0.06%
AddSameFd	0.01%
EditSameEnumConst	0.07%
ExtendsList	0.04%
ModifiersList	0.06%
ImplementsList	Approximately 0.00%

3.3.3 RQ4: How frequently do merge conflicts occur?

We answer **RQ4** by reproducing 70,047 merge scenarios and computing the conflicting scenario rate, which measures the percentage of merge scenarios with at least one merge conflict (See Section 3.1). We also compute this metric without considering spacing and consecutive line edit conflicts. Table 3 describes the conflicting scenario rates for a few of the projects from our sample. The complete table is in our Appendix.

In addition, Table 4 describes conflicting scenario rate values with and without different spacing and consecutive line edition conflicts. Because our data is not normally distributed, we used the Wilcoxon signed rank test (WILCOXON; WILCOX, 1964) combined with the Bonferroni correction method for multiple comparisons (BONFERRONI, 1936) to run our hypotheses tests as we describe further. We also measure the effect size for each test as defined by Rosenthal (ROSENTHAL, 1994), where an effect size of 0.1 means a small effect, 0.3 a medium effect, and 0.5 a large effect.

Our first hypothesis test was to compare the observed conflicting scenario rates for each project with and without spacing conflicts. We found that there is a statistically significant difference between these populations with a large effect size (p-value < 0.01, effect size $= 0.51$). Then, our second hypothesis test compares the conflicting scenario rates for each project with and without consecutive line edition conflicts. Again we found a statistically significant between these populations with a large effect size (p-value < 0.01, effect size $= 0.53$). Thus, such results suggest that removing such conflicts represents a statistically significant decrease on the conflicting scenario rate.

We further analyzed the conflicting scenario rate to check how many merge conflicts occur over the total number of commits. From the 70,047 analyzed merge scenarios, 4,141 (total of 5.91%, with a median of 4.43%, and an IQR of 5.54%) contain conflicts in Java Files considering the FSTMerge semistructured merge algorithm. In these scenarios,

Project	Size (KLOC)	Merges	CR	CR WFP
Antlr4	11.4	663	8.60%	7.99%
Javaee7-samples	65.5	207	0.97%	0.97%
AndEngine	40.7	115	6.96%	6.96%
Clojure	67	40	12.5%	10%
Elasticsearch	953	2,736	5.77%	5.26%
FBReaderJ	387	1,310	14.43%	13.28%
Graylog2-server	124	1,072	12.31%	12.13%
HoloEverywhere	48.6	82	8.54%	8.54%
OpenTripPlanner	15.9	734	12.67%	10.76%
cgeo	53	2,128	8.46%	7.71%
SimianArmy	4	218	9.17%	6.42%
Titan	251	488	16.19%	12.5%
Orientdb	319	1,752	9.13%	7.25%
Hector	27.6	404	12.62%	9.41%
Hive	1,003	244	42.21%	39.34%
Netty	182	169	6.51%	6.51%
Kotlin	412	588	9.01%	8.67%
ListViewAnimations	8.1	117	8.55%	5.98%
K-9	103	566	6.01%	4.95%
Droidparts	10.3	105	2.86%	2.86%
BroadleafCommerce	219	1,061	22.34%	20.74%
ShowcaseView	2.1	96	9.38%	8.33%
Jmxtrans	17.2	275	2.55%	2.18%
StickyListHeaders	2.8	97	3.09%	3.09%
Retrofit	8.9	526	0.57%	0.38%
Storm	139	1,689	12.31%	11.78%
Eureka	32.7	391	6.65%	6.14%
Spout	70.5	854	4.8%	3.98%
Druid	160.3	2,061	5.34%	4.85%
Conversations	39.2	514	5.25%	5.06%
Generator-jhipster	19.3	1781	4.72%	4.72%
Mongo-hadoop	15.2	92	15.22%	11.96%
Rstudio	494	1,840	5.82%	5.49%
HikariCP	8.5	189	12.7%	12.17%
Jitsi	381	94	9.57%	9.57%
Gradle	550	975	28.72%	28.51%
Bukkit	32.6	19	15.79%	15.79%
Cucumber-jvm	39.9	560	16.61%	14.82%
Groovy-core	311	674	9.64%	9.2%

Table 3 – Examples of projects from our sample. CR means conflicting scenario rate considering all files in the revisions, and WFP means without false positives, that is, spacing and consecutive line edit conflicts.

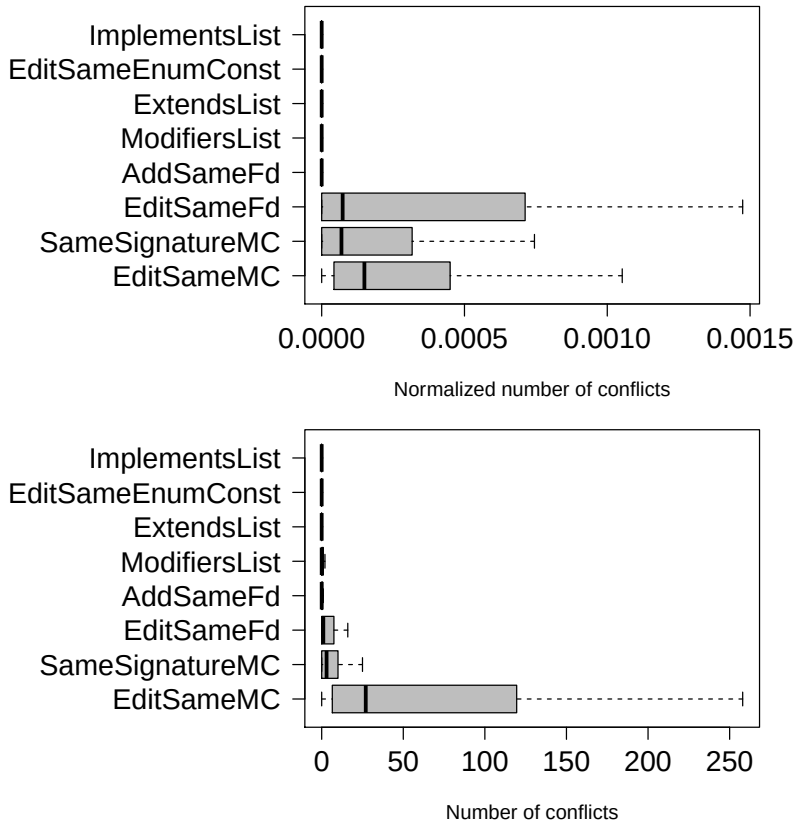

Figure 21 – The top of the image shows the normalized number of conflicts per project boxplots, computing EditSameMC changes by the number of changed lines. Conversely, the lower part of the image shows the absolute number of conflicts per project boxplots.

Table 4 – Conflicting Scenario Rate Description. DS means different spacing conflicts, CL means consecutive line edit conflicts, and IQR means interquartile range.

	Total	**Median**	**IQR**
CR	9.38%	6.64%	8.81%
CR Without DS Conflicts	9.04%	6.50%	8.72%
CR Without CL Conflicts	8.64%	6.39%	7.76%
CR Without DS and CL Conflicts	8.39%	6.00%	7.21%

28,883 conflicts were detected.

3.3.4 RQ5: How frequent are the underlying causes of the SameSignatureMC pattern?

Moving on with the analysis, our aim with **RQ5** is to understand why the SameSignatureMC pattern was the third most frequent pattern from our catalog. Figure 22 shows the distribution of the underlying causes for this pattern. We notice that more than half of the occurrences happened because files existing in both left and right revisions, did not

exist in the base revision. In such cases, the entire file was copied from one repository or branch to another. Thus, for each method that was edited by a commit at least one of the subsequent commits, there was a conflict. The second most frequent causes for method duplication were small methods, such as getters and setters, followed by copied methods, and renamed methods. Finally, a total of 6.4% of the SameSignatureMC conflicts were not classified in any of the previous categories.

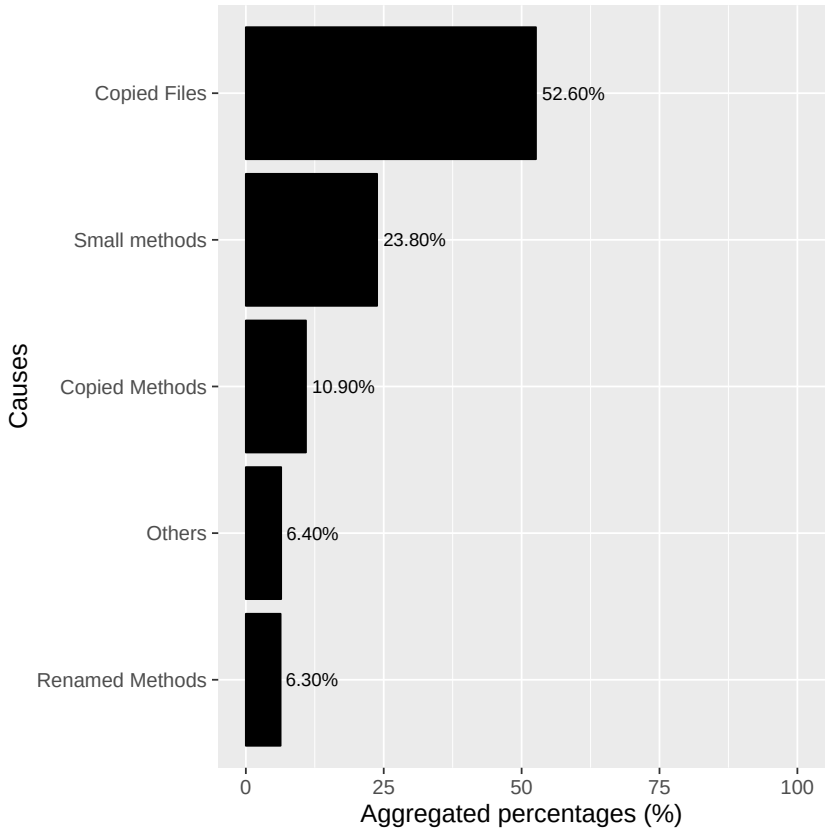

Figure 22 – SameSignatureMC different causes frequency.

One interesting aspect we noticed during the manual analysis conducted to understand the underlying causes of the SameSignatureMC pattern, is that developers made conflicting changes between her own branches. In fact, there are legitimate reasons for this work flow, as there are legitimate reasons for conflicting changes between different developers branches. We consider that conflicts involving contributions from a single developer are likely less problematic to resolve than conflicts involving more than one developer. For this reason we decided to run a complementary analysis to learn to which extent do merge conflicts occur involving different developers.

In addition, more than 50% of the SameSignatureMC conflicts happened because entire files were copied from one workspace to another, changed, and then merged back together. If a single developer was involved in such a operation, it is more likely that she copied files across her own branches, which could be considered less problematic. However this situation could be more problematic if two or more developers were involved.

Because of that we also check the number of developers involved in merge scenarios with SameSignatureMC conflicts caused by copy of files.

To accomplish such task, we collect, for each analyzed merge scenario, the number of different developers (considering their username and e-mail on Git) who authored commits between the merge commit and its base commit. We then classify the merge scenarios into three categories: single developer scenario, two developers scenario, and more than two developers scenario. With this data we answer the following questions:

1. How many developers are involved in merge scenarios, conflicting merge scenarios, and merge conflicts?

2. How many developers are involved in SameSignatureMC conflicts caused by copied files?

We answer both research questions considering all merge conflicts collected by FST-Merge and not only SameSignatureMC ones. Furthermore we consider all developers involved in merge scenarios even if they do not edit the elements involved in conflicts. Table 5 describes data percentages resulting from the analysis to answer question 1. We describe the data using the total percentage, the observed median and the interquartile range.

Furthermore, we used the Wilcoxon signed rank test combined with the Bonferroni correction method for multiples comparisons to test hypotheses comparing the number of developers (one developer vs. two developers, one developer vs. more than two developers, and so on) and considering merge scenarios, conflicting merge scenarios, and merge conflicts. Table 6 describes the adjusted p-values, and effect sizes for each hypothesis test.

Table 5 – Description of the percentages in our data considering the number of developers (one, two, and more than two). IQR means interquartile range.

	Merge Scenarios			**Conflicting Merge Scenarios**			**Merge Conflicts**		
	Total	Median	IQR	Total	Median	IQR	Total	Median	IQR
One Dev	6.53	5.19	8.92	6.39	1.23	9.21	2.56	0.00	4.22
Two Devs	27.84	34.44	20.92	12.77	16.66	25.78	6.20	7.69	25.00
> Two Devs	65.63	55.36	28.42	80.84	75.00	36.89	91.24	87.50	46.78

Analyzing Table 5 we see that most conflicts (more than 60%), and merge scenarios with or without merge conflicts tend to have contributions from more than two developers. Thus, the cases we manually analyzed where contributions from the same developer

	One Developer vs Two Developers	**One Developer vs More than Two Developers**	**Two developers vs More than Two Developers**
Nº of Developers Involved in Merges	p-value<6.60e-16 eff. size=0.57	p-value<6.600e-16 eff. size=0.56	p-value=1.60e-06 eff size=0.32
Nº of Developers Involved in Conflicting Merges	p-value=2.44e-08 eff. size=0.36	p-value<6.60e-16 eff. size=0.56	p-value<6.60e-16 eff. size=0.57
Nº of Developers Involved in Merge Conflicts	p-value=8.38e-07 eff. size = 0.32	p-value<6.60e-16 eff. size=0.56	p-value=1.64e-14 eff. size=0.47

Table 6 – Description of the adjusted p-values and their corresponding effect sizes according to the hypothesis test being made comparing the observations from two different populations, and the research question.

conflicted with each other do not represent the majority of cases. The descriptive data is reinforced by the p-values and effect sizes described in Table 6. All p-values fall under our threshold of 0.01, with their respective effect size ranging from 0.32 up to 0.57.

Finally, considering question 2, in our data, 121 merge scenarios from 56 different projects had conflicts that happened because files were copied. From those merge scenarios, 20.66% involved a single developer, 14.87% involved two developers, and the remaining 64.47% involved more than two developers (the median was 4 developers, and the IQR was 7 developers).

3.4 DISCUSSION

In this section, we discuss the consequences of our results, and actions they support.

Most merge conflicts happen when developers edit the same or consecutive lines of the same method. However, perhaps awareness tools should be more careful with class field, and modifier list edits as well.

RQ2 results points out that most merge conflicts— 84.57% of the collected conflicts, and 80.71% after removing spacing and consecutive line edit conflicts— happen because developers edit the same or consecutive lines of the same method or constructor. At first this result might seem obvious due to the intuition that most part of the Java code is inside methods. Thus the probability of conflicts occurring inside methods or constructors would be higher.

However, we achieved such results because we used a more sophisticated merge tool. If we had used a line-based merge tool like previous studies (BRUN et al., 2013; KASI; SARMA, 2013), a significant part of the collected conflicts would likely be ordering conflicts (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015; CAVALCANTI; BORBA; ACCIOLY, 2017), contradicting the initial reasoning. According to our previous results (CAVALCANTI; BORBA; ACCIOLY, 2017), approximately 40% of the conflicts reported by unstructured merge are ordering conflicts. Therefore, the EditSameMC pattern would be the most frequent conflict regarding the remaining 60% of the conflicts reported by unstructured merge. Although ordering conflicts are still real conflicts, we can ignore they altogether by using FSTMerge or any other merge solution considering the ordering of the elements inside a class. Resolving ordering conflicts is simple in theory. One just needs to disentangle the parts of the different elements inside the conflict headers. However, depending on the number of lines involved in an ordering conflict, this task is laborious and error prone (CAVALCANTI; BORBA; ACCIOLY, 2017).

In addition, FSTMerge captures the SameSignatureMC pattern that line-based merge tools do not, which increases the recall of our numbers. So far, there has been no evidence about the frequency for this type of conflict. Finally, we are not aware of previous studies providing empirical evidence about the distribution among different conflict patterns considering the granularity of edited language syntax elements.

Such results can be useful to help awareness tools becoming more efficient in terms of performance and precision, without compromising their recall. For example, although we have not implemented and validated awareness tools considering different conflict predictors, we hypothesize that a tool monitoring developers working on different repositories, identifying when they edit the same method, and alerting them, would likely have a reasonable recall since it would detect most merge conflicts (approximately 85%).

However, our biggest concern about driving conclusions based only on **RQ2** results is that there is no baseline about the proportion of changes made to the repository and the number of conflicts. Most conflicts reported by FSTMerge involve method declarations, but that could happen not because method changes are more problematic but just because most changes occur inside method declarations. Perhaps, when comparing the frequency of conflicts against this baseline, our results could change. For this reason we add **RQ3** in this study. We believe that **RQ3** complements **RQ2** results because while the absolute number of conflicts is more useful for driving awareness tools towards preventing a larger part of the conflicts, normalizing this number is useful to understand the precision of each kind of code change as a conflict predictor.

In fact, after executing **RQ3** analysis we found that not only editing the same method, but editing the same class field and modifier list are important predictors to consider when trying to prevent conflicts. As a result, the hypothetical tool we describe could also warn about the possibility of EditSameFd, and ModifiersList conflicts with low risk of

being wrong. This way, developers could communicate early and avoid the occurrence of such conflicts. This hypothetical tool could monitor individual workspace changes as they happen, or, they could analyze changes that were already committed. This second option would be more flexible because it does not assume that developers should be all working together, at the same time. Moreover, it would consider changes that will be integrated eventually. In contrast, a tool that monitors changes in real-time can detect changes that are not meant to be committed, for example, when a developer adds variables to help her while debugging the code.

In practice, RQ2 and RQ3 combined results might be helpful to improve existing awareness tools. For example, Palantír (SARMA; REDMILES; HOEK, 2012), a workspace awareness tool, informs different developers of ongoing activities in the same project. It proactively detects merge conflicts by informing when developers edit the same files using a metric based on the number of lines changed. As developers edit more lines of the same file, the higher is the risk of ending up in merge conflicts. However, such metric could potentially report false positives. For instance, when developers implement independent methods in the same class. As an improvement, one could add different types of alarms to alert developers in the presence of EditSameMC, EditSameFd, and ModifiersList patterns. This way, Palantír is still able to report most part of the conflicts, but it avoids false alarms.

Differently from Palantír, Crystal (BRUN et al., 2013) proactively integrates commits from developer repositories with the purpose of warning them if their changes conflict. To produce conflict information sooner, Crystal has to run its analysis often. This can be expensive because it might involve complex build and testing activities. To mitigate this problem, Crystal could use our conflict patterns as predictors for conflicts. Then it could process code contributions using partial ASTs such as FSTMerge before performing the integration routine to check if they contain the most frequent patterns, and, in case they do not, Crystal could delay the integration until the subsequent time period. Although this suggestion likely reduces the cost of running Crystal, further studies are needed to verify if it does not compromise Crystal's accuracy regarding build and test conflicts.

Finally, Syde (HATTORI; LANZA, 2010) is a tool that provides team awareness by capturing developers' edits as atomic AST changes and warns developers if they change the same nodes. Syde uses an approach that is very close to the hypothetical tool we propose in this study to detect conflicts in real time. However, as described by Syde authors, information overload could be a problem. Perhaps in an industrial environment with very large teams and many simultaneous changes Syde could overload developers with potential conflicts' information which could impair their productivity. One possible solution to mitigate information overload in those contexts would be to capture changes concerning only methods, class fields, and modifiers as **RQ2** and **RQ3** results indicate that such changes are the most likely to lead to merge conflicts.

An important aspect of these three conflict awareness tools (Palantír, Crystal, and Syde) is that they assume different collaborative development setups. For example, Palantír and Syde consider that developers always work synchronously, since changes are being monitored in real time even before they are committed. In contrast, Crystal only analyzes changes that were previously committed but not merged to the shared repository. This means that Crystal assumes that developers do not have to work synchronously, which is a more flexible approach. This happens when developers work in different time zones, for example. Either way, our results provide recommendations for both synchronous and asynchronous development. However, if developers commit their changes locally, without being connected to the internet for a long period of time, then it would not be possible to detect potential conflicts until they connect again. In such cases, conflicts would probably not be detected before they become severe (too many changes need to be considered to resolve the merge).

Sophisticated merge tools reduce conflicts and might improve productivity and quality

Compared to previous studies, our results show lower conflicting scenarios rate values. Kasi and Sarma (KASI; SARMA, 2013), and Brun et al. (BRUN et al., 2013), respectively show average conflicting scenarios rates of 14.38%, and 17%, while the median of our conflicting scenarios rate was 6.64%. Moreover, by using a slightly improved merge algorithm to remove spacing and consecutive line edit conflicts, the median drops to 6%. As discussed in our sample description section, we include in our analysis the same Java projects that previous studies analyze. Such difference is likely due to the adoption of FSTMerge to merge Java files, naturally reducing the number of reported conflicts compared to line-based merge tools. This result reinforces the evidence provided by previous studies that investigates the benefits of adopting semistructured merge tools (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015).

Thus we believe the adoption of our adapted version of FSTMerge could help to further increase not only development productivity, since developers could spend less time dealing with spurious conflicts (BIRD; ZIMMERMANN, 2012), but also product quality, given that a frequent cause of integration errors are merge conflicts that are not resolved correctly.

Lastly, our results show that 90.68% of the merge scenarios have less than 10 merge conflicts which could be considered less problematic from a quantitative perspective. However, the conflict resolution effort depends on the nature of the conflicts, as fewer conflicts do not necessarily mean less effort resolving them. For example, our results show that **most merge conflicts involve more than 2 developers' contributions**, which suggests that resolving merge conflicts might not be simple. Moreover, Menezes (MENEZES, 2016) achieved similar numbers when he analyzed the distribution of conflict chunks, us-

ing a traditional line-based merge tool. He reports that most failed merges involved just 4 or fewer conflicting chunks, and more than half involved 1 or 2 conflicting chunks.

Depending on the project development practices, we might have only been "scratching the surface" on the number of conflicts

We also analyze some of our sample outliers— projects with a conflicting scenario rate significantly higher or lower than the median— to understand factors that might have influenced such disparity. During this analysis, we noticed that 14 projects, including Cassandra and Hive, had higher conflicting scenario rates, comparable to those of the previous studies (higher than 16%). If we had not used a more advanced merge tool, those rates might have been even higher.

By manually analyzing 4 of those projects— namely, Cassandra, Hive, Roboguice, and BroadleafCommerce— we observe that these **higher conflicting rates are accompanied by a greater number of collaborators working independently at the same period of time, and pushing their commits directly to the main repository instead of performing pull requests**. Such practices resemble the development environment of centralized version control systems such as SVN and CVS (GOUSIOS; PINZGER; DEURSEN, 2014). Particularly, Cassandra has a patch-based contribution process, with no specific strategy to avoid conflicts.[7]

Alternatively, projects such as JeroMQ and Dagger have no conflicts on Java files. In fact, after removing spacing and consecutive line edit conflicts, a total of 10 projects from our sample turned out to have no conflicts on Java files. We suspect, but have no hard evidence, that projects with no different spacing conflicts might use tools that fix code indentation before commits. However, by analyzing 4 of those projects— Generator-jhipster, Exhibitor, JeroMQ, and OkHttp— we observed that this happened mainly for two reasons.

First, projects such as Generator-jhipster only merge contributions via pull requests and after *rebasing*— a Git operation that effectively integrates code without creating a merge commit or leaving any trace about a merge being performed. This practice is explicitly mentioned in their contribution guide.[8]. Rebasing is a frequent practice in a development model known as pull-based software development, commonly used in version control systems such as Git. In this development model, instead of pushing changes to a central repository, developers work locally and register pull-requests to the master repository (GOUSIOS; PINZGER; DEURSEN, 2014). Then, the repository administrator reviews the pull-request, approves the changes and rebase the branch changes on top of the master branch head. In such situations, although conflicts are still happening, the merge commit

[7] http://wiki.apache.org/cassandra/HowToContribute
[8] <https://goo.gl/XQyygC>

does not appear in the development history and, consequently, we cannot identify that there is a merge just by looking at the project's development history.

The second reason for having a low conflicting scenario rate is that, for some of the projects, in spite of their popularity and large number of registered contributors in the project's Github page, **only one or two contributors were significantly active at the same period**. This is the case of Exhibitor, which had 20 registered contributors, but only one of them was responsible for 72% of the commits. In contrast, OkHttp, which has a very low conflict rate (0.25%) had more than one active contributor but they contributed on different periods of time, so their work never really interfered with each other.

In summary, the development model used (pull-based together with rebase vs. push to shared repository) may affect the number of merge commits in history. The pull-based model, together with the systematic use of Git commands that rewrites commits history, such as *rebase*,*squash* and *cherry-pick*, decreases the number of merge commits. Nevertheless, conflicts are still being solved locally, which means that our empirical results represent a lower bound for the actual number of merge conflicts.

We did not assess the reasons why so many developers and projects prefer to use rebase. Our intuition is that they do so because it might seem easier to analyze a linear development history than a history with divergent branches. However, the drawback of this habit is the fact that the true history gets lost. Consequently, it is not easy to go back to the topic branch since it would have the appearance of being a part of the main branch. It also makes reverting changes much more difficult, because one would have to cherry-pick one commit at a time, trying to remember which ones came from the original topic branch.

Another aspect that reinforces our intuition the we might have been only scratching the surface on the number of conflicts in our retrospective analysis is that large companies such as Google, Facebook, and Amazon are avoiding branches in general to reduce the need of merges (POTVIN; LEVENBERG, 2016). They do "trunk-based development" where only one branch, called trunk or master, is allowed and everyone commits to it. The intuitive idea behind trunk-based development is that frequent integration to the master containing smaller changes should produce less conflicts.

Trunk-based development resembles the development environment of centralized VCSs. Thus, one could assume that in a replication of our study using projects from centralized VCS or using trunk-based development the conflict ratios would decrease. However, Zimmermann (ZIMMERMANN, 2007) while studying systems in CVS reports that 23% up to 46% of files integration lead to merge conflicts, which is a relatively high number of conflicts. Moreover, the projects that we manually analyzed containing the highest conflicting scenario rates from our sample have multiple collaborators working at the same time while committing directly to the master branch. Finally, in centralized VCSs we would have access to all merge conflicts since the development history is not rewritten to hide merges.

Therefore, we believe that trunk-based development should be more investigated in order to learn about the frequency and severity of conflicts in this context.

Developers do not take full advantage of proper code version and end up creating conflicts

In order to answer **RQ5** we analyzed the occurrences of SameSignatureMC conflicts in our sample to understand their underlying causes. In fact we did not expect that it would be so common for developers working on different assignments to add methods with the same signature. Our automated analysis done with a total of 1,505 conflicts of the SameSignatureMC pattern, shows that 63.5% of these conflicts happened because developers copied methods, or entire files, from one repository or branch to the other. We even observed curious cases where the same developer, working on different repositories, copied methods across them.

As explained before, this is not the case of code cloning, since the developer copied that same piece of code from other branch to her branch on the same class that it was before. We believe the idea is to reuse pieces of code from branches that were not meant to be merged— different products' branches, for example— or they were not ready to be merged yet— the feature was not fully implemented and tested. Furthermore, the developer might have simply postponed the entire merge process to avoid having to deal with conflicting changes at that moment.

Either way, our results show that copy and paste across different branches or repositories is a common practice. This evidence suggests that developers do not take full advantage of proper code version, but rather copy and paste code around creating the risk of conflicts. Such finding supports the need for tools that enable partial merges, where developers, instead of merging entire sequences of commits, can break commits into smaller parts/pieces of code and then choose what commits they want to merge.

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 (BARIK; LUBICK; MURPHY-HILL, 2015; DIAS et al., 2015). For example, the goal of Commit Bubbles, and EpiceaUntangler is to help developers to build systematic commit histories that adhere to version control best practices. Moreover, Codebase Manipulation (MUSLU et al., 2015), is a tool that automatically records a fine-grained history and manages its granularity by applying granularity transformations. In addition to such tools, we suggest a partial merge tool where developers that already know which code parts (methods or files) they need at that moment, are able to isolate them in a different commit, and merge just those selected commits to their local repository/branch.

Conversely, besides copying pieces of code, an additional 6.3% of SameSignatureMC occurrences happened when a method from the base revision was equally renamed on

both derived revisions. At first this seemed like an odd coincidence, but through a manual analysis we found that this was often due to a renaming in an API method, and, as a result, when the dependency is updated, it breaks the build across different repositories. Consequently, developers have to fix both the method's name, and its calls, on their local branch to successfully compile the code.

For those renaming cases, or other refactoring related changes, a mechanism that allows "broadcasting" refactoring related changes across repositories could help. For example, the developer responsible for committing the refactoring changes could mark this changes to be replayed for all collaborators of the project. Of course, changes would be applied to a repository only when the developer accepts the patch. This way, developers would not need to reproduce the same code changes in different repositories. This is then extra evidence for the need of better supporting refactorings in API evolution (DIG; JOHNSON, 2005). For example, Catch up! (HENKEL; DIWAN, 2005), a tool that uses descriptions of refactorings to help application developers migrate their applications to a new version of a component, could be extended to support the cases we have observed.

Alternatively, a total of 23.8% of the occurrences were simple methods such as getters, setters, or methods with less than 3 lines of code. This situation we believe is more reasonable to expect. Two developers might independently feel the need for adding a *get* or an *equals* method to the same class. However, through a manual analysis we saw that some of them were copied or equally renamed methods as well, but because they had few lines of code, we did not run the analysis of copied and renamed methods on them.

Finally, the remaining 6.4% of the SameSignatureMC conflicts did not fit in any of the previously defined categories. Through manual analysis, we observed that in some cases the methods were copied or renamed as well. However, because they were significantly changed, our string similarity algorithm returned a score smaller than our threshold (70%). Nevertheless, most of the manually analyzed cases really reflected the name of the pattern— developers indeed added complex methods with the same signature and different behavior.

We noticed that those methods' names often contained common words from developers' vocabulary such as *initialize*, *execute*, *run*, and *load*. The example we describe back in Section 3.1.5 illustrates a duplicated method called "sendFile" from project Jitsi which could be a recurrent name for methods from an instant messenger application. For such cases, we could improve awareness tools to alert when developers add methods with the same signature, so that they can communicate and solve this conflict earlier.

Merge scenarios, conflicting merge scenarios, and merge conflicts usually involve more than two developers

The bottom line of the analysis collecting the number of developers involved in merge conflicts is that those conflicting scenarios involving a single developer that we found

while manually investigating underlying causes for SameSignatureMC conflicts are not so common after all. In fact, our data indicates that merge scenarios, conflicting merge scenarios, and merge conflicts often involve more than two developers. We also observe this tendency when analyzing merge scenarios containing SameSignatureMC conflicts caused by copied files.

Although the number of developers involved in merge conflicts does not measure directly the effort to resolve them, we believe that solving conflicts involving a single developer is probably easier than solving conflicts involving more developers. In addition, Costa et al. (COSTA et al., 2016) reported that developers usually have a hard time while merging branches 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 for integrating changes across branches. Our work reinforces their findings.

3.5 THREATS TO VALIDITY

Our empirical analyses and evaluations naturally leave open a set of potential threats to validity, which we explain in this section.

3.5.1 Construct Validity

A possible threat to the construct validity of our study is our choice of metrics. We tried to mitigate this threat by using metrics already used in well established studies in the area. For example, to learn about the frequency of conflicting merges in **RQ4**, we measure the proportion between conflicting merge commits and merge commits. This metric was also used by Kasi and Sarma (KASI; SARMA, 2013) and Brun et al. (BRUN et al., 2013). This way, we are also able to compare our results with theirs.

A different alternative to learn about conflicts' frequency would be to measure the ratio between conflicting merge commits and commits in general. However, we believe that such metric is not appropriate. For example, consider that one developer committed 8 times while performing task A, and a second developer committed 1 time while performing task B. Then, someone merged task A and task B contributions into a merge commit which resulted in a conflict. In this case, the conflict frequency would be 10% (1 conflicting merge commit out of 10 commits).

However, if the second developer had the habit of making smaller and more frequent commits, the conflict frequency would decrease, but, in the end, her contributions would have conflicted with the first developer contributions regardless of the number of commits. In summary, depending on developers' habits, they might commit too often or too rarely and this metric would vary according to that. Meanwhile, by analyzing the proportion

between conflicting merge commits and merge commits we have a better notion of how often developers' contributions conflict with each other.

Moreover, we tried to choose metrics that gives us alternative views about the same problem. For example, to learn about the most frequent conflict pattern, besides computing the number of conflicts, we also compute the normalized number of conflicts to complement our results. We compute this metric by dividing the number of conflict occurrences from each pattern by the number of involved syntax elements changed during the entire project development history.

However, the normalized number of conflicts resulted in values that were small compared to the relatively high number of conflicts in projects. Perhaps, an alternative to compute this metric in the future would be to divide the number of conflicts by the number of changes made only to the elements involved in these conflicts. To compute this metric we would need to track down the history of each element involved in a conflict to compute the number of changes in each one of them. Moreover, we could add different weights to compute this metric such as the number of developers involved in these changes eventually leading to a conflict.

Finally we could also consider the notion of the lift measure commonly used in data mining and association rule learning (TUFFERY, 2011). According to Tufféry, the lift metric measures the improvement of a rule at predicting cases as the enhancement of that rule measured against the normal population rates. For example, if we consider that the conflicting rate for all projects is 10%, but the chance of ending up with conflicts when two developers edit the same method is 50% then we can say that this conflict has a lift measure of 5.0 (50 divided by 10). We also leave this alternative metric as future work.

3.5.2 Internal Validity

In this work we analyzed 123 Java projects from Git. Three projects from our sample (JeroMQ, Dagger, and Closure-compiler) had no merge conflicts, however, the sum of their merge scenarios represents only 1.07% from our sample and do not compromise our general results.

Differently from previous studies, which used line-based merge tools, we use FSTMerge which is a semistructured merge tool with some knowledge about the underlying syntax of the artifacts. Thus FSTMerge is able to automatically solve ordering conflicts. In addition, by using FSTMerge we were able to systematically derive our conflict pattern catalog by analyzing FSTMerge annotated Java grammar and extracting all changes that leads to conflicts detected by this tool. Nevertheless, the decision of using an adapted version of FSTMerge also brings drawbacks to our analysis. Although it removes a large number of false positives (APEL et al., 2011), it might add small numbers of false negatives and other kinds of false positives, as we discuss next.

The added false negatives might happen when two developers independently add *import* declarations involving different *packages* and the same member name. For instance, if developer A adds *java.util.List*, and developer B adds *java.awt.List*. When using FSTMerge to integrate those contributions, it treats this case as an ordering conflict. In contrast, FSTMerge orders the import list declarations, likely leading to a build conflict (type ambiguity error). Thus, if one uses a line-based tool and the described contributions were added to the same line (or in consecutive lines), the conflict would be reported and the developer responsible for the integration could resolve it before it became a build problem.

Moreover, a different type of false negative might happen when one developer adds a method that calls a second method that was edited by another developer, which could lead to a test conflict. Likewise, if those developers edit the same or consecutive lines of the same text area, the line-based merge tool would report this conflict, while the FSTMerge would not.

To measure if those two types of false negatives would have occurred frequently in our data, we further analyzed all merge scenarios of 50 Java projects from our sample (CAVALCANTI; BORBA; ACCIOLY, 2017). To identify false negatives concerning conflicting import declarations, we used FSTMerge to identify when different contributions add import declarations to the same class. Then, we try to build the merge class and check if we get a type ambiguity error. In contrast, to identify the second type of false negative we use an overestimated metric. We compare the results from FSTMerge to the diff3 merge. Whenever diff3 reports a conflict involving different code elements (we parse the conflict body) we consider it as a potential FSTMerge false negative.

From that analysis we observed that, from all the merge scenarios, only 1.66% had changes matching those patterns, and are, therefore, false negatives. Thus, such conflicts do not happen very often. Nevertheless, FSTMerge could be slightly improved to detect such cases.

Regarding the possibly added false positives, FSTMerge fails to identify renaming changes. If a program element such as a method is renamed in one revision, the FSTMerge algorithm is not aware of this fact and cannot map the renamed method to its previous version, and it considers that the method was removed. If the method that was renamed in one revision, is edited by the other revision, FSTMerge will report a conflict. Conversely, a line-based tool would report a conflict only if the same or consecutive lines were edited. This means that a percentage of the EditSameMC conflict occurrences that we collected might fall into this category and, therefore, be false positives, possibly affecting some of our more detailed findings, such as the normalized results.

It is hard to guess whether a version of FSTMerge that properly handles renaming would improve our findings. In fact, avoiding renaming conflicts could lead to new kinds of false negatives. So fixing FSTMerge to properly handle renaming would demand careful

evaluation of the renaming detection strategy. In this paper, we decided to compute a conservative (overestimated) number of renamings to check if, even considering more renamings than expected, our main results would remain the same.

We collected a total of 24,427 EditSameMC occurrences. From this total, 9,206 might be false positives due to the renaming issue. This is an overestimation because FSTMerge cannot discern a deletion of an element from a renaming. To have evidence that the FSTMerge renaming issue did not affect our main conclusions, we made a preliminary analysis using a subsample of 60 projects from our original sample to check for references on the renamed or deleted method.

We observed that 30.21% of renamed methods occurrences seems to be false positives, but this is also an overestimation. So, considering that 30.21% of those 9206 occurrences are false positives, the percentage of EditSameMC conflict drops from 84.57% to 82.92%. Consequently, even with this overestimated amount of false positives, EditSameMC conflicts would still be the most frequent conflict pattern by far from our sample, without compromising our general results. More details on FSTMerge's false negatives and false positives analysis can be found in Cavalcanti et al.'s study (CAVALCANTI; BORBA; ACCIOLY, 2017).

Like previous works (KASI; SARMA, 2013; BRUN et al., 2013) did, we analyze *Git* projects, which support commands such as *rebase*, *squash*, and *cherry-pick*, that rewrite project development history. Consequently, depending on the development practices of each project, we may have lost merge scenarios where developers had to deal with merge conflicts, but that do not appear on *Git* history as merge commits (BIRD et al., 2009). When those commands are used in a systematic way they might dramatically decrease the number of merge commits. Consequently, to analyze all merge scenarios, we would need to have access to developers private repositories.

Thus, our results are actually a lower bound for the real conflicting scenarios rates. In fact, our assumption is that if we use a centralized version control system such as SVN, the number of conflicts and conflicting merge scenarios would increase. However, studying SVN history is challenging in our context because there is no systematic way to precisely select merge scenarios; SVN has no standard log entry type for merges. Previous studies (APEL et al., 2011) look for commit messages that suggest a commit is the result of merging, but that might be imprecise. So, unless one carefully filters the merge scenarios, the analysis could be biased.

Besides that, we could have used different merge tools to extract our pattern catalog. For example, JDime (APEL; LESSENICH; LENGAUER, 2012) is a merge tool that tunes the merge process on-line by switching between unstructured and structured merge, depending on the presence of conflicts. However, JDime has the same disadvantages of FSTMerge (renaming and import declaration problems). Moreover we managed to remove some of the false positives of FSTMerge that JDime can solve (spacing and consecutive line conflicts).

In addition, JDime inserts new false negatives with respect to FSTMerge. For instance, if both revisions edit different parts the same field declaration— the type definition, and the initialization— JDime will solve this conflict, most likely leading to a build or test conflict. Lastly, we would not be able to use JDime's autotuning strategy because we would risk missing the false negatives of the line based merge. For example, we would miss the occurrences of SameSignatureMC conflicts when the duplicated methods are added in different areas of the file.

Finally, regarding our consecutive line edit conflict analysis, for most of the cases, the edited lines can be merged safely. Nevertheless, there are cases were this merge might lead to a build or test conflict. For example, if a string variable is initialized in two consecutive lines (by string concatenation), and those lines were edited, the revisions would be editing the value of the same variable. If such lines are merged, this could lead to a semantic conflict. Thus, further studies are needed to analyze how frequent this situation happens. Perhaps, with structured merge tools, we could assess if the edits made to consecutive lines inside methods belonged to the same statement or variable initialization. In case they did not, we could perform the merge successfully. Nevertheless we showed that, by removing just the spacing conflicts there is a statistically significant difference in the conflicting scenarios rate.

3.5.3 External Validity

Our sample contains only open source Java projects hosted on GitHub. We only use Java projects for simplicity. Furthermore, by choosing only popular projects— projects with more than 500 stars on GitHub, we might miss diversity in our sample. To analyze projects in different languages, we would have to derive different catalogs as well. Thus, generalization to other languages and other version control systems is limited, and further studies would be needed to confirm our findings. We present these analysis as future work in Chapter 6.

However some of our main patterns, like editing the same method or the same class field, could also be present in other object-oriented languages similar to Java. However, it would be harder to automatically solve spacing conflicts for languages such as Python, given that indentation affects semantics. Also, the implications for future research discussed in this chapter, such as the concept of the tool that monitors developers, the concept of partial merges, and the refactoring broadcast mechanism could be useful for a number of different languages using different types of version control systems. Furthermore, Eirini et al. (KALLIAMVAKOU et al., 2015) showed in their survey that GitHub, with its common development practices (pull-based development), is being increasingly adopted in commercial projects as well.

3.6 CONCLUSIONS

In this chapter we present in detail the first empirical study we conducted throughout this thesis. This study aims at understanding the structure of the changes that lead to conflicts. In order to do so, we derived a conflict catalog containing 9 semistructured merge conflict patterns expressed in terms of the performed kinds of changes considering involved syntactic language structures.

To assess the occurrence of conflict patterns in practice, we reproduced 70,047 merge scenarios from 123 GitHub Java projects. Furthermore, we focused on conflicts reported by a semistructured merge tool, avoiding a large number of spurious conflicts often reported by typical line-based merge tools. Our results show that 84.57% of merge conflicts happen because developers edit the same lines, or consecutive lines of the same method. However, editing methods, class fields, or modifier lists have similar probabilities of leading to merge conflicts. This means that, if we improve awareness tools to alert developers in those cases, we might avoid most merge conflicts. In addition, merge conflicts occur in a total of 9.38% of the analyzed merge scenarios. Moreover, by slightly improving the merge algorithm to better handle spacing and consecutive line edit conflicts, we got statistically significant lower numbers. Compared to previous studies, our results show that using more advanced merge tools reduces the number of conflicting merge scenarios. We also found that developers often copy methods, or even entire files across repositories, which is evidence of the need for tools that enable partial merges. Finally, as a complementary result, our data indicates that merge scenarios, conflicting merge scenarios, and merge conflicts usually involve more than two developers. This result suggests that integrating different branches is not often an easy task since one needs to understand and merge contributions made by different developers.

In conclusion, this study was a first exploration into semistructured merge conflicts' structure and frequency. However, we need further analysis to understand if our conflict patterns, such as edits to the same method, would be efficient conflict predictors in practice, considering not only merge conflicts, but also build and test conflicts. We need to do so because in this study we collect merge conflict instances to trace back the changes that caused them. Which means that all of the EditSameMC instances we collect here indeed caused merge conflicts. However, we do not have any knowledge about developers editing the same method without causing merge conflicts. This happens, for example, when developers edit different lines concerning unrelated parts of the same method. If such situations happen frequently, then alerting developers whenever they edit the same method would raise too many false alarms. Consequently, it would not be an efficient strategy to avoid conflicts. To this end, in the next chapter we describe our second empirical study where we capture change patterns, such as edits to the same method, and measure how often they lead not only to merge conflicts, but also to build and test conflicts.

4 ANALYZING CONFLICT PREDICTORS IN OPEN-SOURCE JAVA PROJECTS FROM GITHUB AND TRAVIS CI

In Chapter 3 we learned that most semistructured merge conflicts happen when developers edit the same lines, or consecutive lines of the same method or constructor declaration.[1] Moreover, after normalizing the number of conflicts considering the types of changes made to a repository, we learned that editing method bodies is one of the change types that most likely leads to merge conflicts.

Moreover, developers might edit the same method without touching the same lines, avoiding merge conflicts, but increasing the chance of causing other types of conflicts, such as build an test conflicts. As explained in Chapter 2, build and test conflicts occur frequently and impair developers' productivity. Build conflicts happen when the system building process fails after the merge. This happens, for example, when developers independently introduce the same local variable declaration inside the same method body. In contrast, a test conflict happens when merged contributions interact with each other causing the system to have different observable outputs than the system tests expect. One example of a test conflict would be when one developer edits one method by adding a call to a method that was edited by another developer. Then, after the integration, one of the test cases of the system starts to fail.

According to the described evidence, it sounds that a good strategy to avoid conflicts would be to alert developers whenever they edit the same method. However, it is possible that developers edit unrelated pieces of code inside the same method, without causing merge, build or test conflicts. This might happen, for example, when a method contains pieces of code from different features (crosscutting concerns). Likewise, if two developers edit the same method, but one of them does not change the method semantics (refactoring changes), it will not cause collaboration conflicts. If such situations happen frequently, this alerting strategy we discuss might raise too many false alarms.

With regard to the other conflict patterns described in Chapter 3, we consider that they rarely report false positives because changes inside those elements are too semantically close together. Therefore, we recommend that such changes should be identified and always reported as conflicts.

Therefore, we need further studies to investigate if edits to the same method is a good conflict predictor not only for predicting merge conflicts, but also for build and test conflicts. Moreover, since we are investigating build and test conflicts, we decided to include new conflict predictors in our analysis. In particular, a previous work (LIMA, 2014) suggests that when one developer edits a method that calls a second method edited by another developer, conflicts often occur. Although this situation does not lead to merge

[1] From now on, we use method declarations to refer both to method and constructor declarations.

conflicts, it is reasonable to consider that it might cause build and test conflicts as well.

In summary, in this study we are interested at investigating the efficiency of the two following conflict predictors: edits to the same method— which we refer to as EditSameMC predictor from now on—, and edits to directly dependent methods— or EditDepMC changes. In particular, we are interested in investigating these conflict predictors precision, that is, how frequently the conflict predictor presence is associated with a conflict, and we also want to measure their recall, that is, what percentage of conflicts we can avoid by using such predictors.

For establishing build and test conflicts ground truth, we rely on the status of building and testing processes executed by the Travis CI (TRAVIS, 2018) service. Whereas this provides quite precise guarantees for build conflicts, the guarantees for test conflicts are as good as the project test suites. So, even for projects with strong test suites, actual semantic conflicts might be missed by the existing tests.

To this end, in this chapter we conduct an empirical study that analyzes 5,647 merge scenarios from 45 Java-maven-travis projects from GitHub to collect instances of conflicts and conflict predictors. Then, we compute how frequently a predictor occurrence is associated with a conflict occurrence, and the percentage of conflicts that can be captured by detecting predictor instances.

Additionally, we conduct a manual analysis to understand what other types of changes cause conflicts, and what changes were associated with predictor instances that did not cause conflicts. Based on the collected evidence we derive more appropriate requirements for detecting conflicts early, and suggest improvements to existing conflict awareness tools.

Our results indicate that, considering both EditSameMC and EditDepMC conflict predictors together, we achieve a precision of 57.99%. In particular, EditSameMC individual precision is 56.71%, and EditDepMC precision is 8.85%. Moreover, we achieve a recall of 82.67% if we consider both predictors together, while EditSameMC individual recall is 80.85% , and EditDepMC recall is 13.15%.

The manual analysis points out that part of predictor occurrences in our sample that are not associated with conflicts are actual missed merge conflicts, that is, although the contributions do not edit the same limes, they interfere with each other. For example, in one EditSameMC predictor from project Web Magic[2] while one developer changes an *if* statement condition, the other developer removes code located inside this same *if* statement. This is expected given the limitations of how we establish test conflicts ground truth— using projects existing test suites executed by Travis CI. Consequently, the precision results we report represent lower bounds of actual semantic conflicts, whereas the recall results are upper bounds because other missed semantic conflicts not caused by the predictors might have occurred as well.

Nevertheless, such evidence is useful to guide different conflict awareness strategies.

[2] <https://github.com/code4craft/webmagic>

For instance, a more conservative strategy would be to alert developers about a large part of potential conflicts at the cost of dealing with some false positives. In this case, warning developers about all predictor occurrences is a reasonable strategy. In contrast, a strategy that aims at precision, even at the cost of loosing conflicts, would be alerting developers about EditSameMC instances only when developers edit the same lines of the same method. In addition, based on our false positives and false negatives analysis, we discuss different strategies that could further increase the predictors' precision as well as increasing the recall by using other predictors.

The remainder of this chapter is organized as follows:

- In Section 4.1 we present the research questions, and the metrics used to answer them;
- Section 4.2 describes the study infrastructure we implement to analyze and collect the defined metrics;
- In Section 4.3 we report the study results;
- Sections 4.4 and 4.5 discuss the results and their implications;
- Section 4.6 presents the threats to the validity of our study.

All the material and data collected in this study is available in our Appendix. This study was published at the International Conference on Mining Software Repositories (ACCIOLY et al., 2018).

4.1 ANALYZING CONFLICT PREDICTORS

Considering the motivation described in the previous section, our goal is to analyze EditSameMC and EditDepMC effectiveness as conflict predictors. Specifically, we want to measure the conflict predictors' precision and recall. Besides that, we want to understand what happens when one conflict predictor occurrence is not associated with a merge, build or test conflict occurrences. In addition, we analyze what other change patterns, besides the defined predictors, could also be considered important conflict predictors.

Therefore, to achieve such a goal, we analyze merge scenarios from the development history of different software projects while answering the following research questions:

4.1.1 Research Question 1 (RQ1): How precise are EditSameMC and EditDepMC predictors?

To answer this question we measure the conflict predictors' precision. The Venn diagram depicted in Figure 23 illustrates how we compute this metric considering our context. When we reproduce project merge scenarios, we collect the occurrences of merge, build

and test conflicts, together with the occurrences of the conflict predictors. This way, when a merge scenario has a conflict predictor and a conflict occurrence, we classify it as a true positive instance. In practice, if an awareness tool had alerted developers about the occurrence of such predictor, it would have indeed detected a conflict. However, if a merge scenario has predictors but no detected conflicts, we classify it as a false positive instance. This means that the awareness tool might have raised a false alarm. In contrast, if the merge scenario has conflicts but no predictors, we classify it as a false negative instance because the awareness tool would not have triggered an alarm, and the conflict would only be detected during the merge process. Finally, if the merge scenario has no predictors nor conflicts, we consider it to be a true negative instance.

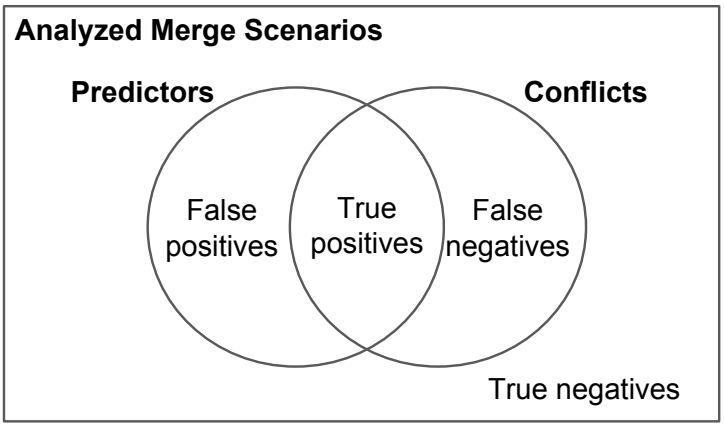

Figure 23 – Computing conflict predictors' precision and recall.

Therefore, we can compute the precision considering both predictors together, and for each predictor individually, using the following formula:

- Precision = $\frac{True\ positives}{True\ positives+False\ positives}$

4.1.2 Research Question 2 (RQ2): How many conflicts can we avoid by detecting EditSameMC and EditDepMC predictors?

To answer this question, we need to measure the conflict predictors' recall. We compute the recall considering the predictors together, and individually, using the following formula:

- Recall = $\frac{True\ positives}{True\ positives+False\ negatives}$

4.1.3 Research Question 3 (RQ3): Why EditSameMC and EditDepMC instances are not associated with merge, build, or test conflicts?

To answer this question, we conduct a manual analysis considering a sub sample of the false positives depicted in Figure 23. With this analysis, we aim to understand how developers managed to edit the same method or directly dependent methods without causing merge,

build or test conflicts. This way, we can improve the precision of the conflict detection strategy. Conversely, we also want to check if semantic conflicts are being missed by the projects test suites. As mentioned before, this is expected given the limitations of how we establish test conflicts ground truth.

While analyzing false positive instances we try to understand if the contributions clearly do not interfere with each other or if there is a possibility of interference— when there are semantic conflicts. To this end, we rely on a broader notion of interference defined by Horwitz et al. (HORWITZ; PRINS; REPS, 1989) and used in our previous work (CAVALCANTI; BORBA; ACCIOLY, 2017). This definition states that "two contributions (changes) to a base program interfere when the specifications they are individually supposed to satisfy are not jointly satisfied by the program that integrates them; this often happens when there is, in the integrated program, data or control flow between the contributions. We then say that two contributions to a base program are conflicting when there is no valid program that integrates them and has no unplanned interference".

The challenge associated with such a more comprehensive comparison criteria is that it is not computable in our context (BERZINS, 1986; HORWITZ; PRINS; REPS, 1989). Therefore, we use Horwitz et al.'s definition in a conservative way during the manual analysis. For example, if one contribution edits a variable assignment used by a command that was added/edited by the second contribution, then there is a possibility of interference, and we classify this predictor as a conflict. In addition, if one contribution edits an if statement condition, while the other adds/edits commands inside this if statement body, we also consider this to be a conflict. We consider that such cases should be reported to developers involved as a potential conflicts.

Alternatively, if one of the contributions does not alter the program semantics— refactors a command or edits comments— we consider that there is not an interference. In addition, if the contributions edit unrelated local variables inside the same method, we consider that they do not interfere with each other, and could be merged together without further problems. In the results section we report what happened in each manually analyzed case. Moreover, we also report the source code of these cases in our Appendix.

4.1.4 Research Question 4 (RQ4): What other change patterns are associated with conflicts?

We answer this question by manually analyzing all false negative instances depicted in Figure 23. Our aim is to learn about what other types of change patterns are associated with conflicts in our sample. This way, new conflict predictors could arise, increasing the recall of our results.

4.2 STUDY SETUP

To explain how we answer the research question described in the last section, here we present our study setup and describe the selection, mining and analysis of our data. All the scripts and data used in this study are available in the Appendix.

Like our previous study, we focus our analysis on Java projects hosted on GitHub. In addition, we select projects using Travis CI and Maven as build manager. We select projects using Travis CI because, besides being the most used CI service (ZHAO et al., 2017), it provides all build information associated with a commit.[3] We use this information to compute our metrics as we further detail. Furthermore, we focus on projects using Maven because we use its log report information for filtering conflicts without human effort.

Figure 24 illustrates the study design, which is divided in the three following phases: in the first phase, we select Java projects from GitHub and Travis CI using Maven as build manager to filter those containing at least one build or test conflict. Section 4.2.1 describes this phase in more detail. Then, in the second phase, we use the same infrastructure from the previous study to reproduce merge scenarios from each selected project. Only this time, besides collecting merge conflicts, we enhance the Conflict Analyzer tool to collect conflict predictor instances as well. In this phase we compute the metrics used to answer **RQ1** and **RQ2**. We explain how we do so in Section 4.2.2. Finally, in the third phase, we perform a manual analysis on a sub sample of reported false positives and false negatives so that we can answer **RQ3** and **RQ4**. Section 4.2.3 explains this analysis in further detail.

4.2.1 Phase 1: Filtering Projects Containing Build and Test Conflicts

We start selecting our sample on GitHub by filtering Java projects containing at least 40 stars and 50 forks. We choose a minimum number of stars and forks to avoid selecting toy and personal projects. With the list of selected projects, for each project we check if the repository contains both Travis CI and Maven configuration files —*.travis.yml* and *pom.xml*. We also check the project current status on Travis (active or not). This way we ensure we select only projects using Maven as build manager, and having data available on Travis CI.

For projects meeting those requirements, we execute a script that clones each project locally and retrieves their merge commit list, just like we did in our previous study. However, as most projects adopted Travis CI later in its life cycle, we filter project merge commits dated after the first finished build on Travis.

For each selected merge commit we use its build status on Travis CI, together with its Maven build log report, and its parent commits build status, to identify build and test conflicts. As explained in Chapter 2, when a developer pushes new commits to the remote

[3] <https://docs.travis-ci.com/api>

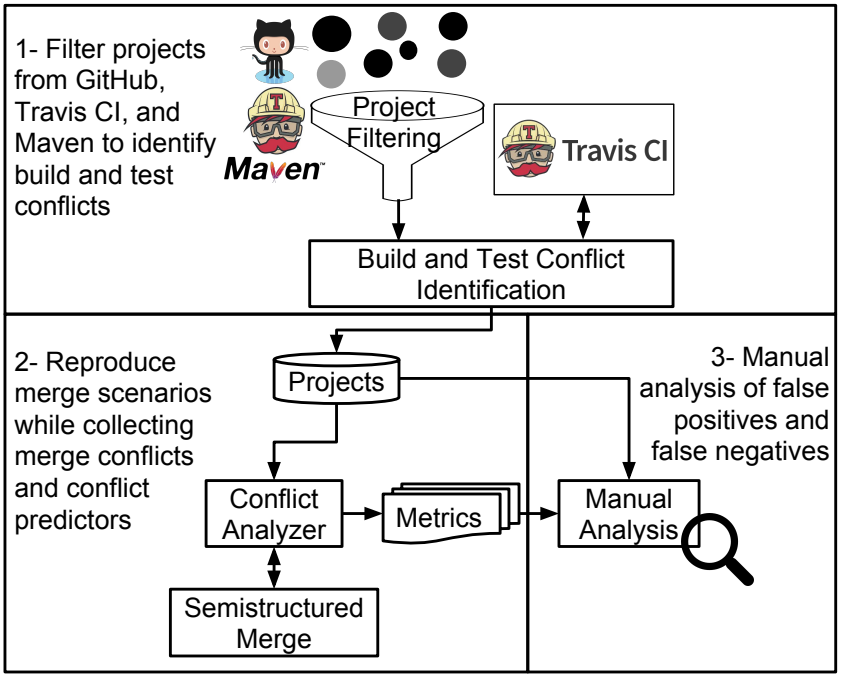

Figure 24 – Study design.

repository in GitHub, or when a pull request is merged, Travis CI gets notified and starts to run the build process. However, Travis CI builds only the latest commit state in the push command or pull request to run the analysis, so not all commits have an associated build status on Travis CI. Because of that, we use a script that forces the commit build creation when there is no build yet. Basically, we create a project fork, activate it on Travis CI, and clone it locally. Then, every push to the remote fork creates a new build on Travis CI. So, for each merge commit, or merge commit parent, without an associated build on Travis CI, we reset the fork repository head to this commit and force push it to the remote fork.

If the merge commit build status on Travis CI is *passed* it means that there is no build error, and none of the tests fails. For these merge commits, we consider that there are no build or test conflicts. In contrast, if the merge commit build status is *errored*— when the build is broken— or *failed*— the build is ok, but one of the tests failed—, we consider it to be a build or a test conflict **candidate**, respectively. However, there are some conditions that must be satisfied first.

It is possible that a build breaks or a test fails due to external configuration problems such as trying to download a dependency that is no longer available, or exceeding the time to execute tests. To eliminate these cases, we analyze for each build its Maven log report seeking for specific message errors (SILVA, 2018). Basically, there are two external causes responsible for interrupting a build process:

- Remote constraints: the build fails because Travis or another external service required by a build process was temporarily unavailable. We discard these scenarios

because they do not reflect issues caused by developers changes, therefore, not characterizing a conflict;

- Environment configuration: the build process fails due to unsolvable or wrong project dependencies. We only discard such scenarios when no changes were made to configuration files. This restriction ensures an external problem is responsible for the build failure.

After discarding those cases, we check the merge commit parents' status to eliminate cases where the build was already broken or with failing tests before the merge. If this is the case, we consider the merge commit broken build or failed test was carried over from its broken parents, instead of being caused by conflicting contributions.

Therefore, we consider that a merge commit with an *errored* build status has a build conflict if its parents have a *passed* or *failed* build status, which means that the build breaks only after code integration. Likewise, we consider that a merge commit with a *failed* build status has a test conflict if its parents have a *passed* build status.

By the end of this phase, we select only projects containing build or test conflicts to proceed to the second phase of the study, where we collect merge, and conflict predictors occurrences. Moreover, each merge scenario selected to the second phase has its associated commits— merge commit and its parents— built on Travis CI.

4.2.2 Phase 2: Collecting Merge Conflicts and Conflict Predictors

In this phase we use FSTMerge to reproduce all merge scenarios dated after Travis CI first finished build from the projects selected in Phase 1 while collecting information about merge conflicts using the conflict pattern catalog. However, we implement some changes to our infrastructure so that we can collect instances of conflict predictors —EditSameMC and EditDepMC. We discuss such changes in more detail over the next sections.

4.2.2.1 Collecting EditSameMC predictors

Collecting EditSameMC instances is straightforward because we already detected EditSameMC instances associated with merge conflicts in the previous study. As explained in Chapter 3, FSTMerge uses *diff3* algorithm to merge the content inside methods. If *diff3* returned a text containing conflict markers, we collected an EditSameMC instance. For this study, we extended this algorithm so that, if there are no conflict markers after executing *diff3*, we compare the parents version to the base version. If both parents differ from the base version, we collect an EditSameMC instance where there was no merge conflicts. Later, at the end of the merge scenario replication, we check if this predictor occurrence is associated with a build or a test conflict. If there is no conflict, we classify this merge scenario as a false positive instance.

In addition, we use the same logic described in Chapter 3 to analyze EditSameMC predictors and check if they are different spacing false positives. This happens when one of the contributions only made changes related to code spacing, which is irrelevant for Java code syntax. This way we can compute and compare our metrics in both ways, considering all EditSameMC predictors, and filtering the different spacing ones, since they do not represent conflicting contributions.

4.2.2.2 Collecting EditDepMC predictors

To detect EditDepMC predictors, while reproducing the merge, we collect all method instances with non-spacing changes made by at least one parent— we discard changes related to different spacing. We also keep the information about which parent was responsible for editing each method. By the end of the merge process, we have a list of all methods changed by one of the parents.

Then, for each method changed by parent 1, we check if any other method changed by parent 2 has a method call to it. Similarly, we do the same inverting parent 1 and parent 2 in the description above. Figure 25 illustrates this approach. Suppose that, by the end of the merge process, we have three methods in our list. Methods m and n from class A, edited by parent 1, and method o from class B, edited by parent 2. In this example, we need to check if m calls o, if n calls o, if o calls m, and if o calls n. Note that there is no need to check if m calls n, or if n calls m since these methods were edited by the same parent. If there are reflexive calls inside those methods, we do not identify them.

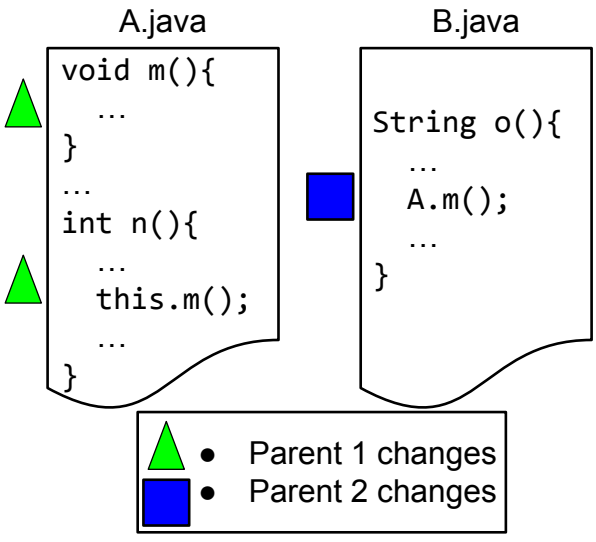

Figure 25 – Looking for EditDepMC predictor instances.

To check if one method calls another one, for performance reasons, we use a two step approach. For example, consider that we need to check if method o calls method m. First, we perform a simple textual search to see if the name of the method m is inside method o

body declaration. In case it does, we use Eclipse JDT library[4] to parse class B and build its AST. Then, we visit B AST nodes until we get to method o declaration. There, we list all the method invocations, and check if any of them matches with method m from class A.

In Figure 25 example, after performing all necessary method reference checks, we note that there is one EditDepMC predictor instance involving methods o and m. As we do for EditSameMC instances not associated with merge conflicts, we check if the EditDepMC instances are associated with a build or a test conflict collected in Phase 1.

4.2.3 Phase 3: false positives and false negatives analysis

After computing the conflict predictors precision and recall, we conduct an analysis to understand the causes of the false positives, and false negatives from our sample. We start by randomly selecting a sub sample of the false positive instances to manually analyze them. Because we reproduce the merge scenarios locally, we keep a copy of the files containing EditSameMC and EditDepMC instances that are not associated with conflicts.

During these files' analysis, we check the changes made by each developer trying to understand if there is a possibility of interference between their contributions. If one of the contributions performs changes that does not change program semantics, such as renaming a local variable, or removing an extra pair of parenthesis inside an *if* statement, or if the developers edit variables that are not related, we consider that there is no interference. Conversely, if both developers change the program semantics, and they edit related variables, for example, when one developer changes an *if* statement condition, while the other edit commands inside that *if* statement, then we consider that there is an interference.

As for the false negatives, that is, a conflict that is not associated with a predictor, we have two different strategies to analyze them. Because we use our previous study infrastructure to reproduce the merge scenarios, we automatically collect the merge false negative conflicts causes using the conflict pattern catalog. In contrast, in order to understand what caused the build and test false negative conflicts, we analyze the Maven log reports associated with the *errored* and *failed* builds. These logs contain information about the cause of these failures.

4.3 RESULTS

In the first phase of this study we analyze a total of 64,445 merge scenarios from 422 Java projects from GitHub using Travis CI and Maven. From this total, 551 merge scenarios meet our build conflict criteria —the merge commit build status is *errored* while

[4] <https://www.eclipse.org/jdt/>

the parents' status are either *passed* or *failed*. However, after performing the Travis log report analysis, we eliminate 467 of those merge scenarios because their builds fail dues to external reasons not related to the contributions being merged. In such cases, we cannot be sure that there is a build conflict due to conflicting contributions. Therefore, we only consider the remaining 84 merge scenarios to have build conflicts. Alternatively, we only found 5 merge scenarios meeting the test conflict criteria —merge commit build fails while parents' builds passes. This time, the log report analysis did not eliminate any of the test conflict instances. In summary, by the end of this study first phase, we select a total of 45 projects containing 89 merge scenarios with build or test conflicts.

Like our first study, here we do not target representativeness or diversity (NAGAPPAN; ZIMMERMANN; BIRD, 2013). Nonetheless, we consider that our sample contains substantial and active software systems with some degree of diversity with respect to dimensions such as size, domain, and number of collaborators. For example, Cloudify, a cloud infrastructure platform, has 408 KLOCs, and 23 active collaborators, Java Jwt, a library for creating and verifying JSON Web Tokens on the JVM has only 8 KLOCs and 8 collaborators, and OkHttp, an HTTP client for Java and Android applications has 57 KLOCs and 128 collaborators. For further information on our sample, we provide a complete subject list in the Appendix. In contrast, we did not find many merge scenarios containing build, and especially, test conflicts. In Section 4.5 we further comment about this situation, but we believe this happens because developers might be executing the projects build and test script and resolving these conflicts locally before committing the integrated code version.

4.3.1 Conflict predictors' precision and recall

In the second phase of this study, we take as input the list of 45 projects and use FSTMerge to reproduce a total of 5,647 merge scenarios dated after each project first finished build on Travis CI. In this sample, a total of 290 merge scenarios have merge conflicts, and 508 have EditSameMC and EditDepMC conflict predictors. If we remove different spacing instances, the total number of merge scenarios containing merge conflicts drops to 251, while the number of merge scenarios containing predictors drops to 469.

Moving on with the analysis, we cross information about merge scenarios containing predictors associated with merge, build, and test conflicts. In total, there are 286 merge scenarios containing at least one predictor occurrence associated with a conflict occurrence. If we remove the different spacing predictors and conflicts, this number drops to 272. Moreover, there are 282 merge scenarios containing EditSameMC instances associated with conflicts. By removing the different spacing occurrences, this number drops to 266. Finally, there are 45 merge scenarios containing EditDepMC instances associated with conflicts. As explained in Section 4.2.2, we do not collect EditDepMC different spacing instances.

Using the data described above we compute precision and recall considering the conflict

predictors together and individually. Moreover, we also measure these metrics considering all predictor and conflict instances, and filtering the different spacing instances. Table 7 summarizes the results related to **RQ1** and **RQ2**.

Table 7 – Precision and recall results according to the predictors considered. WDS means without different spacing.

	Both Predictors	**EditSameMC**	**EditDepMC**	**Both Predictors WDS**	**EditSameMC WDS**
Precision	56.29%	55.51%	8.85%	57.99%	56.71%
Recall	83.62%	82.45%	13.15%	82.67%	80.85%

4.3.2 False positive Manual Analysis

In the third phase of this study, we answer **RQ3** by conducting a manual analysis of the false positives from our sample, that is, merge scenarios containing EditSameMC or EditDepMC predictor instances that are not associated with conflicts. In total, our sample has 222 merge scenarios containing predictors not associated with conflicts. If we remove the different spacing instances, this number drops to 197. From this sample, we randomly select 10 EditSameMC, and 10 EditDepMC instances to conduct the manual analysis. Table 8 and Table 9 summarize EditSameMC and EditDepMC false positives analysis, respectively. In summary, we consider that 8 predictor instances have the possibility of interference, while 12 do not. All the false positives and false negatives manually analyzed are available in the Appendix.

4.3.3 False Negative Analysis

To answer **RQ4**, we analyze false negatives conflict causes to learn what types of changes— besides the predictors— are associated with conflicts. In our sample there are 56 conflicting merge scenarios where no conflict predictor was involved. From this total, 20 merge scenarios have merge conflicts (35.71%), 33 have build conflicts (35.71%), and 3 have test conflicts (5.35%). Because we use our previous study infrastructure to reproduce merge scenarios, we automatically collect merge conflict causes using the conflict pattern catalog. Alternatively, we conduct a manual analysis to understand false positive causes for build and test conflicts.

Among the 20 merge scenarios containing merge conflicts, 11 scenarios have EditSameFD conflicts (55%), 7 scenarios have SameSignatureMC conflicts (35%), 1 scenario has the ModifierList conflict (2.5%), and 1 scenario has one ImplementList conflict (2.5%).

Furthermore, in our sample, 33 merge scenarios have build conflicts not associated with the predictors. In this sample, the most frequent situation in build conflicts— a total of

Table 8 – EditSameMC false positive analysis.

Project	**edits Summary**	**Interference**
JavaPoet	Parents change unrelated variables	No
OpenGrok	One parent changes a variable assignment passed as an argument in a method call edited by the other parent	Yes
Jackson Databind	One parent changes a variable assignment used by the other developer to change an if statement condition	Yes
CorfuDB	One parent changes a variable assignment, while the other parent changes this same variable method call	Yes
Swagger Core	Parents change unrelated variables	No
Wire	One parent changes a variable assignment used in a for statement condition changed by the other parent	Yes
Jackson Databind	Parents edit unrelated variables	No
OkHttp	One parent refactors	No
Restheart	One parent refactors	No
Web Magic	One parent removes commands inside an if body declaration while the other parent changes the if statement condition	Yes

20 conflicts (60.61%)— happens when one developer adds a new reference to a program element— such as a class, a method, or a variable— while the other developer deletes or renames that element. For example, in one of project Blueprints merge scenarios, one developer adds a new method calling another method that was removed by the other developer. Consequently, after the merge, the compiler could not build the file containing the reference to the removed method. In such cases, Crystal would be able to correctly detect them, since it performs the build of the merged artifacts. Moreover, we do not identify such cases as EditDepMC because there is more than one level of dependency in the methods call graph.

In contrast, the second most frequent cause for build conflicts in the false negatives sample are syntactic malformed programs after the merge. More specifically, 10 merge scenarios (30.31%) from projects Java Driver, Cloud Slang, and Hdiv have broken builds because some of the files did not have the expected license header, causing a compilation error on Travis CI.

Table 9 – EditDepMC false positive analysis.

Project	**edits Summary**	**Interference**
OkHttp	One parent changes one method while the other parent changes this method call inside the other method	Yes
Jackson Databind	One parent refactors	No
Cloudify	One parent changes an if statement condition inside of which there is a call to the method edited by the other parent. The other parent adds a new return command to the second method	Yes
Jackson Databind	One parent edits comments	No
Wire	Both parents change the same variable assignment which is passed as an argument from one method to the other	Yes
Truth	One parent refactors	No
Moshi	One parent refactors	No
JavaParser	One parent refactors	No
Retrofit	One parent refactors	No
Singularity	One parent refactors	No

The remaining three merge scenarios in the false negatives sample have different conflict causes. In one merge scenario from project ScribeJava, while one contribution adds a new class implementing an existing interface, the other developer adds a new method to this interface. After the merge, there is a compilation error because the newly added class does not implement all interface methods.

One build conflict from project Blueprints has an occurrence of the SameSignatureMC pattern where both contributions copy and paste the same method across different repositories and one of them edits the method indentation. Because of that, the line-based merge tool reported a conflict and the developer responsible for the integration tried to fix the conflict by copying and pasting the two versions of the same method to the resulting file. As a result, there is a compilation error due to duplicate method declarations. Conversely, if one had used a semistructured merge tool in this merge scenario, there would not be a merge nor a build conflict because FSTMerge only reports SameSignatureMC conflicts when one of the contributions edits the code content, ignoring spacing changes.

The last merge scenario containing a build conflict in the false negatives sample comes

from the Jackson-core project. Figure 26 depicts the differences between the merge commit parents. The left side parent removed the line containing the local declaration of variable *f* and added it as a parameter to the method. Meanwhile, the right side parent edited *INPUT* variable content.

```
private void _testSymbolsWithNull(JsonFactory f, boolean useBytes)
    throws Exception
{
    final String INPUT = "{\"\\u0000abc\" : 1, \"abc\" : 2}";
    JsonParser parser = useBytes ? f.createParser(INPUT.getBytes("U1
```

```
private void _testSymbolsWithNull(boolean useBytes)
    throws Exception
{
    final JsonFactory f = new JsonFactory();
    final String INPUT = "{\"\\u0000abc\" : 1, \"abc\":2}";
```

Figure 26 – Merge scenario from Jackson-core project.

Because the contributions edited consecutive lines of the same method —variables *f* and *INPUT* declarations— the line-based tool reports a conflict involving these lines. The developer tried to resolve this conflict by copying and pasting both local variable declarations to the resulting file. However, he did not notice the new parameter added by one of the parents. Consequently, variable *f* has two local declarations, causing a compilation error.

Note that this last build conflict is actually an EditSameMC instance missed by FSTMerge. This happens because, as explained in Chapter 3, FSTMerge cannot match methods when their signature is changed. Therefore, this merge scenario has an EditSameMC instance associated with a conflict, which makes it a true positive in our sample that the actual infrastructure is not able to detect.

As for the false negative test conflicts, our sample has three instances coming from projects Jedis and Wire. Two of these conflicts happened because not directly dependent methods were edited, and the third one happened because one developer updated a test case executing a method that was edited by the other developer.

4.4 HOW EFFECTIVE ARE THE CONFLICT PREDICTORS?

The precision and recall metrics gives a notion of how effective an awareness tool considering EditSameMC and EditDepMC predictor would be if it was used during the development of the 45 projects from our sample. Such evidence can guide better decisions regarding a awareness tool conflict awareness strategy.

The precision indicates that for over half (57.99%, after removing the different spacing cases) of the merge scenarios where the tool triggers an alarm, it is indeed alerting developers about changes associated with merge, build and test conflicts. Meanwhile, the recall indicates that we capture 82.67% of the merge scenarios containing merge, build and test conflicts by using an awareness tool considering both predictors.

However, to establish the ground truth for build and test conflicts we rely on the status of building and testing processes executed by Travis CI. Whereas this provides quite

precise guarantees for build conflicts, the guarantees for test conflicts are as good as the project test suites. So, even for projects with strong test suites, actual semantic conflicts might be missed by the existing tests. As a matter of fact, during the false positive analysis we find that 8 out of 20 (40%) false positive instances are missed semantic conflicts. This evidence suggests that if we had better test cases our precision would increase. In contrast, better test cases would find more semantic conflicts not caused by the predictors as well. Consequently, the precision results we report represent lower bounds of actual semantic conflicts, whereas the recall results are upper bounds of semantic conflicts.

Furthermore, we note a significant difference between the two predictors' precision and recall when we analyze them individually. While EditSameMC precision is 56.71%, EditDepMC precision is only 8.85%. Likewise, while EditSameMC recall is 80.85%, EditDepMC recall is 13.15%. Because there is not much difference between the measured precision and recall considering the predictors together and EditSameMC individually, there is significant evidence that EditSameMC instances dominate our measurements. This is due to the fact that EditDepMC instances are not associated with merge conflicts, which are the most numerous in our sample. Nonetheless, we consider that a solution containing both predictors would still be advisable.

Our precision and recall results provide evidence to guide different conflict detection strategies depending on each team preferences. For instance, if one particular team prefers to be conservative and alert developers about a large part of the conflicts at the cost of dealing with some false positives, then detecting EditSameMC and EditDepMC as we do in this study is a reasonable strategy.

In contrast, if a team aims at precision, even at the cost of loosing some conflicts, then it could alert developers about EditSameMC instances only when the contributions edit the same or consecutive lines of the same method, which necessarily leads to merge conflicts. This approach is similar to Crystal (BRUN et al., 2013), a tool that proactively integrates commits from different developer repositories with the purpose of warning developers of merge, build and test conflicts.

Finally, there is also the possibility of using other methods to further increase the predictors' precision without compromising their recall as we further detail in the next section.

4.4.1 Strategies to improve the precision and recall of the conflict predictors

The false positives and false negatives analysis provides insights of opportunities to further increase an awareness tool precision and recall. In this section we discuss such results and the actions they support.

During the false positives analysis, we find 8 cases— 5 EditSameMC, and 3 EditDepMC— where, even though there are no conflicts associated, it would still be advisable to alert developers about such changes because there are interferences. For example, in one Edit-

SameMC instance from project Web Magic, while one developer removes code inside an *if* statement, the other developer changes the *if* statement condition.

Conversely, we also find 9 false positive cases— 3 EditSameMC and 6 EditDepMC— where there is clearly no interference because one of the contributions does not alter program semantics. For instance, in one EditDepMC instance from project Jackson Databind, one of the contributions renames a local variable, while in an EditSameMC case from project RESTHeart one of the contributions simply removes an extra semicolon.

By observing the types of changes that were made, we learn about strategies that might improve the conflict predictors precision. We can divide them in the following two categories:

1. Identify and ignore cases where clearly there is no interference;

2. Identify possible interferences.

Regarding the first category, we already use in this study a strategy to detect cases where one of the contributions changes only code spacing. Considering our sample, this strategy improves the overall precision in 2.93%. Another simple strategy would be detecting when one of the contributions edits only comments, as it happens in one case from project Jackson Databind.

Perhaps, a possible strategy to detect and ignore cases with clearly no interference would be to run refactoring detection tools, such as the one proposed by Nikolaos et al. (TSANTALIS et al., 2018), to detect and ignore predictor cases where one of the contributions performs solely refactoring edits without changing program semantics. However, some refactorings, such as renamings, might cause build conflicts. For example, when inside the same method one developer renames a local variable while the other developer adds code using this same variable. In such cases, after detecting the refactoring, the tool could replay the refactoring over the other side. This could fix the problem of variable rename, for instance, because the new uses of the variable would also be renamed.

An alternative way to use this strategy would be in the context of a tool such as Crystal. While running its integration routine it identifies refactorings causing build conflicts. Moreover, Crystal would be able to provide more comprehensive alerts by adding a refactoring detection algorithm to its integration routine. For example, if no conflicts are detected, but both contributions change the program semantics it could alert developers to be more cautious about this integration scenario. Conversely, if no conflicts are detected and one of the contributions performs only refactorings, then Crystal could report that there is no interference in this scenario. We suggest this analysis as a future work in Chapter 6. Nonetheless, in an environment with many developers committing often, Crystal's speculative analysis might become to expensive.

For the second category of strategies, in Section 4.1, we mention that interferences often happens when there is data or control flow between the contributions. Therefore,

one possible mechanism to identify possible interferences would be to check the existence of information control flow between the contributions as an approximation for computing interference. This is exactly what Filho (FILHO, 2017) investigates. He analyzed a total of 157 merge scenarios from 52 Java projects containing EditSameMC predictors. He finds information control flow in 64% of the merge scenarios. Then, after a manual analysis, he reports that there was indeed interference in 42.86% of the merge scenarios with information flow between contributions. He also describes improvements to increase the precision of his technique.

Alternatively, another strategy to identify interference would be using a similar approach to Böhme et al. (BöHME; OLIVEIRA; ROYCHOUDHURY, 2013), which proposes to generate regression tests that expose change interaction errors. They do that by generating a graph called Change Dependence Graph (CDG) to summarize the control flow and dependencies across changes. The CDG is then used as a guide during program path exploration via symbolic execution— thereby producing test cases which witness change interaction errors. An extension of this strategy, generating test cases exercising commands changed by both developers might be able to identify more semantic conflicts. We suggest this increment as a future study in Chapter 6.

Regarding the false negatives analysis from our sample, we consider that it would not be hard to detect most part of them. For example, all merge conflicts from the false negatives sample would be detected by the conflict pattern we describe in Chapter 3. Syde (HATTORI; LANZA, 2010) would be able to detect such cases as well. Furthermore, except for the build false negatives caused by the missing license headers from specific project rules, the other build conflicts related to one contribution adding a reference to a program element which was renamed, moved, or deleted by the other contribution are already detected by existing awareness tools such as Palantír, Syde, and Crystal (SARMA; REDMILES; HOEK, 2012; HATTORI; LANZA, 2010; BRUN et al., 2013). Such evidence suggests that practices used by these tools are feasible as well.

4.5 ARE BUILD AND TEST CONFLICTS NOT THAT FREQUENT AFTER ALL?

Although this study does not aim to measure build and test conflicts frequency, we could not help to notice that despite analyzing a considerable amount of merge scenarios and projects in the first phase of our study, we did not find many build and test conflicts. This becomes more evident when we compare our results to previous studies assessing the frequency of build and conflicts. Kasi and Sarma (KASI; SARMA, 2013), for example, reports build conflicts occurring in ranges between 2-15% and test conflicts occurring in ranges between 5-35%, while Brun et al. (BRUN et al., 2013) describes both kinds of conflicts ranging around 33%. In this section we discuss some of the reasons why our numbers are so different compared to other studies.

We believe there are mainly two reasons for such contrasting numbers and they are both related to differences between how we collect test and build conflicts. First, the previous studies rely only on the merge commit build status. They do not consider parents commit build status. This way, false positives might have been introduced. For example, the build might have been already broken or with failing tests before the merge. In such cases, we consider the merge commit broken build or failed test was carried over from its broken parents, instead of being caused by conflicting contributions. 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.

In our study we mitigate both threats since we analyze Travis CI log report to filter builds with errors caused by external problems, and we also check the merge commit parents status. This way we increase confidence that the merge commit build problems are caused by conflicting contributions.

Perhaps the decision of analyzing merge commits that occurred after the project has adopted Travis CI might have impacted the conflicts frequency. According to previous studies (ZHAO et al., 2017), the adoption of CI practices helps to maintain the code quality. This is so because, when a project adopts CI practices 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 (MUYLAERT; ROOVER, 2017) that broken builds occur more frequently in regular commits than in merge commits. Finally, to better analyze this hypothesis, for future works, we suggest a comparative study between projects using CI platforms and projects that do not use them (but use automatic build scripts).

In conclusion, we believe that build and test conflicts occur more frequently than what we report here. As we narrow our numbers while trying to increase the soundness of our results, we might be loosing build and test true positives as well. However, we need further studies to understand the impact of our methods in our results. A better way for evaluating conflicts— not only build and test, but also merge conflicts— would be by having access to developers private workspaces instantaneously evaluating the cases without any external influences. In Chapter 6 we propose such analysis as future work.

4.6 THREATS TO VALIDITY

Our empirical analyses and evaluation naturally leave open a set of potential threats to validity. We discuss such threats in this section, which is organized according to the types of threats.

4.6.1 Construct Validity

One of the threats concerning our metric for detecting test conflicts is that we rely on the projects existing test suites to detect them. This means that part of the semantic conflicts might escape. As mentioned before, better test cases would probably increase the precision, and decrease the recall reported in this study. Nonetheless, because the notion of interference we use is not computable, it is impossible to detect all semantic conflicts.

In addition, we answer **RQ3** by conducting a manual analysis on 20 false positive instances randomly chosen from our original sample of 203 false positives. Due to the size of this sub sample, our results might not be representative of the entire sample. Therefore, although it was not our intention, we cannot drive conclusions about the proportions between false positives that we could avoid and those that we could not. In order to do so, one would have to choose a statistically representative sample, which is out of the scope of this work.

4.6.2 Internal Validity

As we reuse part of our previous study infrastructure to reproduce merge scenarios, we inherit part of its threats as well (ACCIOLY; BORBA; CAVALCANTI, 2017). In particular, because FSTmerge fails to identify renaming changes, we miss EditSameMC instances where one of the contributions changes the method signature. In fact, one of the build false negatives we analyze is actually a true positive that FSTMerge misses because of method renaming.

In addition, on Travis a build can be composed of a set of jobs; each job varies itself in some way. For example, different jobs can be used to simulate the same project with different environment configurations. Therefore, it is possible to declare which jobs should not be considered for the final build status. Thus, if a build conflict happens on a non-valid job, we do not detect it.

For future work, we could edit *travis.yml* file aiming to consider all jobs for the final build status. However, non-valid jobs are used only to verify how the project behaves on a specific configuration. Therefore, problems in these scenarios possibly would not lead developers to spend time with them.

Finally, we used a manual analysis to analyze the false positive instances from our sample. Because it was a manual analysis, we could have committed mistakes. To mitigate this threat, we used the interference definition in a conservative way. Whenever we considered that the contributions could have interfered with each other, we considered it as a conflict. So, there might be cases where there is no interference and we they should be classified as false positives. In future analysis we could use more proper interference analysis such as Filho used in his master thesis (FILHO, 2017).

4.6.3 External Validity

In this study we focus on open-source Java projects hosted on GitHub, using Travis CI and Maven. Thus, results generalizability to other platforms and programming languages is limited. Such requirements were necessary to reduce the influence of confounds, increasing internal validity. We need subsequent studies to further understand the precision and recall of the predictors for other programming languages. Nevertheless, we are confident that we have analyzed active and substantial systems from various domains.

4.7 CONCLUSIONS

This second study to understand if alerting developers whenever the edit the same method or directly dependent methods would be an efficient strategy to avoid collaboration conflicts in terms of raising few false positives and without too many false negatives. To this end, we reproduce 5,647 merge scenarios from 45 Java-maven-travis projects from GitHub to measure the precision and recall of our considered conflict predictors. Our results indicate that, considering both conflict predictors together, we achieve a precision of 57.99% and a recall of 82.67%. We believe that such results are useful to guide different early conflict detection strategies.

Moreover, based on our manual analysis results, we provide further insights about what we could do to provide more precise results. Our intuition is that if we implement some of those changes our precision would increase because we would be able to discard cases where the involved contributions clearly do not interfere with each other. In contrast, by implementing better test cases, our recall would likely decrease since we would be able to catch other types of test conflicts that were not caused by our predictors.

Finally, in this chapter we conclude the discussion regarding the core work of this thesis. In the next chapter we present previous related work and how they compare with our work.

5 RELATED WORK

In this chapter we describe some of the previous studies that we use as base evidence for our study, and related work divided by their different topics.

5.1 PREVIOUS STUDIES INVESTIGATING DIFFERENT ASPECTS OF COLLABORATION CONFLICTS

A number of empirical studies provide evidence about collaborative development issues. In previous chapters we have already mentioned and discussed some of them. In this section we present a summary of the key points presented previously.

In Chapter 2 we mention Kasi and Sarma (KASI; SARMA, 2013) and Brun et al. (BRUN et al., 2013) studies that reproduced merge scenarios from different GitHub systems with the purpose of measuring the frequency of merge scenarios that resulted in merge, build and test conflicts. These studies respectively show average conflicting scenarios rates for merge conflicts of 14.38%, and 17%, while the median of our conflicting scenarios rate was 6.64%. Compared to those studies, in Chapter 3, we also assess the conflicting scenarios rate for merge conflicts. The median of our conflicting scenarios rate was 6.64%.

Compared to those studies, we use a much larger sample which makes our results more representative. We also used a semistructured merge tool that avoids a large number of spurious conflicts often reported by typical line-based tools which were used in those studies. Because of that, we concluded that sophisticated merge tools reduce the number of merge conflicts, and might decrease merge resolution effort.

In addition, we go further by deriving a conflict pattern catalog and measuring how frequently those patterns occur, and the probability of having a merge conflict while editing different language syntax elements. We also bring evidence about other problems that developers often face while working collaboratively. For example, the conclusion that developers often need to copy pieces of code or rename methods across different repositories. Therefore we go beyond the analysis provided in those studies and complement their results.

Besides analyzing merge conflicts frequency, Kasi and Sarma, and Brun et al. also provide evidence about build and test conflicts frequency. Kasi and Sarma reports build conflicts occurring in ranges between 2-15% and test conflicts occurring in ranges between 5-35%, while Brun et al. (BRUN et al., 2013) describes both kinds of conflicts ranging around 33%. In Chapter 4, although we do not aim at measuring build and test conflicts frequency, we discuss some of the reasons why our results were so different from theirs. After all, our average conflicting rate for build and test conflicts was much smaller (less than 1%).

First, these studies rely solely on merge commit build status to compute build and test conflicts. This means that the build or tests might have been failing before the merge and the failures were not caused by code integration. Moreover, they do not analyze the reason behind the build failure which might have been cause by external factors such as missing dependencies and not by build or test conflicts. These factors, as discussed may add false positives to the results. Therefore, the analysis we provide is probably more accurate in the sense that is has less false positives. However, a drawback from our study is that, while trying to filter false positives from our sample, we might have lost build and test conflicts as well. Thus, we would need a more careful analysis to generalize our results regarding build and test frequency.

Zimmermann (ZIMMERMANN, 2007), in contrast, assessed the number of merge conflicts using a different metric as the author reproduced files integration— and not merge scenarios— from 4 CVS hosted projects. Thus, we cannot compare our results with his. However, we believe that running our study on a centralized version control system would show an increased number of conflicts and conflicting merge scenarios. We leave this comparison as future work.

Still considering the frequency of collaboration conflicts, Perry et al. (PERRY; SIY; VOTTA, 2001) made an observational case study to analyze the effect of parallel changes on a large-scale industrial software system. They reported that, although 90% of the files could be merged without problems, the degree of parallel changes is high— merge conflicts involved between 2 to up to 16 parallel changes. We found similar results in Chapter 3, since most part of merge conflicts (91.24%) involved more than two developers. In addition, we used open-source projects from varying sizes and domains which contributes to generalize Perry et al.'s conclusions.

Regarding software merging techniques, Mens (MENS, 2002) describes a comprehensive overview of the field, and suggested directions for future research. Among them, he claimed for the need of a detailed but language independent taxonomy of the kinds of changes, and corresponding conflicts, that can be made to software. In this work we provide a catalog of merge conflict patterns in terms of the kinds of changes leading to conflicts. Although our catalog is not language independent, some of our patterns, including the most frequent ones such as EditSameMC could be extend to different programming languages. We leave the study of deriving conflict patterns using different programming languages as future work.

Concerning the cost of resolving conflicts, previous studies have tried to estimate it. For example, Kasi and Sarma (KASI; SARMA, 2013) estimated conflict resolution effort as the time interval between when a conflict first occurred and when it was resolved. In other words, they compute the number of days that the conflict persisted in the master repository. They reported that resolving merge conflicts took substantial effort, typically spanning multiple days. However, their metric assumed that the computed time intervals

reflected the efforts of developers working exclusively to resolve the conflict, which is not always the case. This means that this metric is an over-approximation.

We believe that a main challenge for estimating conflict resolution effort is that different conflicts might demand different resolution effort. In this sense, Cavalcanti et al. (CAVALCANTI; BORBA; ACCIOLY, 2017), while comparing different merge approaches (unstructured and semistructured), estimated the effort to resolve different types of conflicts by evaluating the strategy used by developers while resolving them. They assumed that resolutions including only changes from the merged contributions (without new code, nor combination of contributed code) probably demand less effort. While this estimation is a fair approximation of the time needed to fix the code— which is part of the total integration effort—, it does not consider the time needed to understand the changes, reason about the conflict and then decide how to fix it.

In contrast, other studies did not quantitatively measure the cost of resolving conflicts, but they reported, based on experimental observations, that resolving merge conflicts is not so trivial. It might take considerable time, and is an error-prone activity. For example, Sarma et al. (SARMA; REDMILES; HOEK, 2012) reported that developers commonly rush to commit their tasks before others so they would not have to deal with conflicts while pushing their changes to the shared repository. In addition, Bird and Zimmermann (BIRD; ZIMMERMANN, 2012) report that a frequent cause for integration errors are merge conflicts that were not resolved correctly.

Alternatively, McKee et al. (MCKEE et al., 2017) conducted a series of interviews and surveys to understand developers perceptions of merge conflicts. They reported that if developers perceive a conflict as too complex or if they do not have much knowledge in the code area of the conflict, they might feel the need to alter their resolution strategy, such as reverting conflicting changes, and in some cases delaying the task of resolving conflicts.

Those studies trying to estimated the cost of resolving conflicts do relate directly to our main goal since we investigate merge conflicts frequency, the structure of conflicts, and conflict predictors precision and recall. However, in Chapter 3 we report that merge conflicts usually involve contributions from more than two developers. Thus, although such analysis does not measure directly the effort to resolve them, we believe that resolving conflicts involving a single developer is probably easier than resolving conflicts involving different developers. Also, in Chapter 4, we present examples from real projects where the developer performing the integration introduced build problems while trying to resolve merge conflicts. Thus, our work complements previous findings and reinforces the claim that resolving conflicts is an error-prone activity.

Alternatively, other studies have tried to analyze different technical and organizational aspects that might have an impact on the occurrence of collaboration conflicts. For example Cataldo and Herbsleb (CATALDO; HERBSLEB, 2011) tried to understand different

aspects leading to conflicts. They presented an empirical analysis of a large-scale project where they examined the impact that software architecture characteristics, and organizational 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 organizational factors such as the geographic dispersion of development teams, can lead to higher integration failure rates.

Likewise, Leßenich et al. (LESSENICH et al., 2017) performed an empirical study analyzing how different factors, such as the size of changes, the number of files changed, and the location of changes could be related to a higher numbers of merge conflicts. However, none of the factors analyzed in their study had a predictive power concerning the frequency of merge conflicts.

Furthermore, Shihab et al. (SHIHAB; BIRD; ZIMMERMANN, 2012) 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 reported that, indeed, the branching strategy does have an effect on software quality and that misalignment of branching structure and organizational structure is associated with higher post-release failure rates.

Finally, Estler et al. (ESTLER et al., 2014), investigated the impact of awareness information in the context of globally distributed software development. Among their findings, they concluded that insufficient awareness information affects more negatively developers' performance than actual merge conflicts.

Our work complements these works because we also examine factors that relate to integration failures on collaborative development environments. However, we analyze different factors. While Cataldo and Herbsleb (CATALDO; HERBSLEB, 2011) and Leßenich et al. (LESSENICH et al., 2017) analyzed architecture level and organizational factors that lead to integration failures, and Shihab et al. (SHIHAB; BIRD; ZIMMERMANN, 2012) analyzed branching strategies that have an impact on software quality, we analyze which code changes often lead to merge conflicts, and the effectiveness of code changes as conflict predictors. Conversely, like Estler et al. (ESTLER et al., 2014), our results reinforce the importance of using and improving awareness tools.

Regarding developer's coordination dependencies while working collaboratively, Blincoe et al. (BLINCOE; VALETTO; DAMIAN, 2013) conducts a study to analyze what is the reduced set of essential task properties that are indeed indicative of coordination needs between a pair of developers working independently. Their purpose is to optimize the process of tasks coordination without the risk of overwhelming developers with a large list of recommendations for coordination awareness. With a similar purpose, in our work we try to understand what changes most likely cause collaboration conflicts so that we do not have to raise so many false alarms to developers like existing tools such as Palantír does.

Moreover, Xuan and Filkov (XUAN; FILKOV, 2014), quantitatively analyses the phenomenon of synchronous development manifested when file commits by two developers are close together in time and modify the same files. They report a strong correlation between synchronous development and communication, that is, for pairs of developers, more co-commit bursts are accompanied with more communication bursts, and their relationship follows closely a linear model. Such evidence complements our work since they provide evidence that synchronous development occurs often, and suggest different requirements for coordination awareness than the recommendations we make.

5.2 TOOLS AND STRATEGIES FOR CONFLICT DETECTION AND RESOLUTION

Tools and strategies to support collaborative development environments use different strategies to both decrease integration effort, and improve correctness during task integration. Throughout this work we have mentioned most of them. Cassandra (KASI; SARMA, 2013), for example, is a tool that analyzes task constraints to recommend an optimum order of tasks execution so that conflicts can be avoided. While the tasks are being developed, Palantír (SARMA; REDMILES; HOEK, 2012) is an awareness tool that informs developers of ongoing parallel changes, and Crystal (BRUN et al., 2013), proactively integrates commits from developer repositories with the purpose of warning them if their changes conflict. In contrast, other awareness tools, such as Syde (HATTORI; LANZA, 2010), build code artifact ASTs to make the analysis of changes more precise. WeCode (aES; SILVA, 2012) continuously merges uncommitted and committed changes to detect conflicts on behalf of developers before they check-in their changes. Moreover, Bellevue (GUZZI et al., 2015), is an IDE extension to make committed changes always visible, and code history accessible inside developers' workspaces.

Regarding such awareness and early conflict detection tools, in Chapter 3 we discuss how implementing some strategies considering our conflicts patterns could help Palantír (SARMA; REDMILES; HOEK, 2012) and Syde to provide more precise results while alerting developers on concurrent changes. In addition, Crystal could postpone its speculative analysis when our conflict patterns are not found.

Moreover, in Chapter 4, we argue that Crystal would be able to provide more comprehensive alerts by adding a refactoring detection algorithm to its integration routine. For example, if no conflicts are detected, but both contributions change the program semantics it could alert developers to be more cautious about this integration scenario. Conversely, if no conflicts are detected and one of the contributions performs only refactorings, then Crystal could report that there is no interference in this scenario.

Thus the evidence we report in this work might help to increase the precision and recall of existing awareness and early conflict detection tools. We leave the implementation and evaluation of such improvements as future work.

Alternatively, when the performed tasks are ready for being merged, TIPMerge (COSTA et al., 2016) has an algorithm that recommends developers who are best suited to perform merges considering different metrics such as developers' past experience in the project, their changes in the involved branches, and dependencies among modified files. Our work supports the need of such tool since one of our findings from Chapter 3 is that most merge conflicts involve more than two developers, and choosing a more suitable person to resolve such conflicts could be useful.

Finally, given that it is not always possible to detect conflicts before code integration, tools like FSTMerge (APEL et al., 2011), and JDime (APEL; LESSENICH; LENGAUER, 2012) offer solutions to reduce integration effort by automating the resolution of some types of conflict, such as the ordering conflicts.

Our work brings evidence that reinforces the need of using more sophisticated merge tools to decrease collaboration conflicts resolution effort. In addition, we also provide small improvements to FSTMerge algorithm, together with empirical evidence that they indeed improve FSTMerge's results.

Conversely, MergeHelper (NISHIMURA; MARUYAMA, 2016) 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. Similar to MergeHelper, MolhadoRef (DIG et al., 2008) is also an operation-based approach, but it records refactoring operations used to produce one version and replays them when merging different contributions. In Chapter 3, we have shown SameSignatureMC conflict instances caused by methods being renamed in different branches and then leading to merge conflicts. In such cases tools like MolhadoRef could help to understand and resolve them.

6 CONCLUSIONS

In this work we conduct different empirical studies to learn about how conflicts happen, and the effectiveness of two conflict predictors. When working in a collaborative development environment, developers implement different tasks in an independent way. Consequently, during the integration, one might have to deal with collaboration conflicts. Previous studies indicate that conflicts occur frequently, and impair developers' productivity. In this context, the study described in Chapter 3 aims at understanding the structure of the changes that lead to conflicts. First, we derived a conflict catalog with 9 conflict patterns expressed in terms of the performed kinds of changes considering involved syntactic language structures. To assess the occurrence of such patterns in open-source systems, we conducted an empirical study reproducing 70,047 merge scenarios from 123 GitHub Java projects. Furthermore, we focused on conflicts reported by a semistructured merge tool, avoiding a large number of spurious conflicts often reported by typical line-based merge tools.

Our results show that 84.57% of merge conflicts happen because developers edit the same lines, or consecutive lines of the same method. However, editing methods, class fields, or modifier lists have similar probabilities of leading to merge conflicts. This means that, if we improve awareness tools to alert developers in those cases, we might avoid most merge conflicts. In addition, merge conflicts occur in a total of 9.38% of the analyzed merge scenarios. Moreover, by slightly improving the merge algorithm to better handle spacing and consecutive line edit conflicts, we got statistically significant lower numbers. Compared to previous studies, our results show that using more sophisticated merge tools reduces the number of conflicting merge scenarios.

We also found that developers often copy methods, or even entire files across repositories, which is evidence of the need for tools that enable partial merges. Finally, as a complementary result, our data indicates that merge scenarios, conflicting merge scenarios, and merge conflicts usually involve more than two developers. This result suggests that integrating different branches is not often an easy task since one needs to understand and merge contributions made by different developers.

Therefore, the work described in Chapter 3 was a first exploration into semistructured merge conflicts' structure. Like most empirical studies, our work has limitations which could be improved in many ways. For example, regarding internal validity, FSTMerge has false positives such as the renaming method problem— when one contribution renames a method and the second contribution edit lines inside it. Likewise, FSTMerge also has false negatives such as the imports list situation— both contributions adding imports declaration to classes with the same name leading to type ambiguity errors. Although we believe that those problems did not have a significant impact in our main conclusions, it

would be important to implement strategies to mitigate them. We describe such strategies as future work.

Despite such limitations, from that study we learned that editing the same lines of the same method is, by far, the most common cause for merge conflicts. However, we needed further studies to investigate if editing the same method would indeed be an effective conflict predictor, considering not only merge, but also build and test conflicts. To this end, in Chapter 4 we describe an empirical study where we reproduce 5,647 merge scenarios from 45 Java-maven-travis projects from GitHub to measure the precision and recall of the following conflict predictors: edits to the same method, and edits to directly dependent methods. Our results indicate that, considering both conflict predictors together, we achieve a precision of 57.99% and a recall of 82.67%. Such results are useful to guide different strategies for early conflict detection, depending on how teams perceive the occurrence of conflicts.

Moreover, based on our manual analysis results, we learned about strategies that could further improve our results. For example, by detecting refactorings we would be able to discard cases where the involved contributions clearly do not interfere with each other. In contrast, by implementing better test cases, our recall would likely decrease since we would be able to catch other types of test conflicts that were not caused by our predictors.

There are two main limitations regarding our second empirical study. The first one is that we rely on the projects existing test suites to detect them. This means that part of the semantic conflicts might escape. Although detecting all semantic conflicts is not possible, improving the quality of the tests, in particular, trying to explore contributions interactions might be helpful to detect more semantic conflicts. Back in Chapter 4 we discuss other strategies to improve precision.

The second limitation is how we filter merge commit build status to identify build conflicts. While trying to achieve more precise results by removing potential false positives, we might be missing actual build conflicts. However, we need further studies to understand the impact of our methods in such results. We present these studies as future work.

Finally, both of our empirical studies suffer from the same threat to the external validity since both samples consist solely of Java open-source projects hosted on GitHub. This means that we cannot generalize our results considering systems in other programming languages or version control systems. Moreover, some of Git commands such as *rebase*, *squash*, and *cherry-pick* might erase from the project development history actual merge commits. Therefore, our empirical results represent a lower bound for the actual number of merge conflicts. To have a better notion of the frequency of merge conflicts one would have to analyze developers' local repositories logs to check how frequently these commands were used. We leave this analysis as future work.

6.1 FUTURE WORK

As the proposed studies described here are part of a broader context, a set of related aspects were left out of the scope of this thesis. Thus, in this section we suggest them as future work.

There are different possible directions to enhance our first empirical study, described in Chapter 3. For example, although we analyzed a large number of merge commits, our results could benefit from replications analyzing other projects, including projects in centralized version control system such as SVN or CVS. Likewise, it would be interesting to replicate this study by deriving a new conflict pattern catalog for a different programming language, or even for a different merge tool. Moreover, one could answer additional questions with our data. For example, what are the conflict patterns inside method bodies? What percentage of those conflicts involve method signatures or just statements inside the method bodies? To answer this question, one would have to use a diff tool such as GumTree (FALLERI et al., 2014), which builds the full AST. This would replicate Menezes (MENEZES, 2016) work, which reports that most conflicts involve method invocations, method declarations, variable declarations, commentaries, and if statements.

Moreover, in this study we analyzed merge conflicts' frequency. Another important aspect to analyze is the cost associated to solving merge conflicts. We noticed that previous studies that tried to estimate conflict resolution effort have either used experimental observations, proxies, or over-approximations (KASI; SARMA, 2013; CAVALCANTI; ACCIOLY; BORBA, 2015; SARMA; REDMILES; HOEK, 2012; BIRD; ZIMMERMANN, 2012). We believe that a solid way to estimate the effort to resolve different types of conflicts would be to conduct controlled experiments where developers have to resolve conflicts while time and other metrics are being measured.

Alternatively one could make a study to analyze our results on a per-project basis, understanding, for example, why some projects have more false positives than others, why some projects have more SameSignatureMC conflicts than others, etc. Other interesting research questions were left outside of scope of this paper, mainly the ones involving technical, organizational, and developers behavioral factors that might influence the presence/severity of conflicts.

Finally, an important process-related question is who is responsible for integrating the merges. Such a decision is likely to influence the merge conflict resolution process. For some of the projects we analyzed, such as Generator-jhipster, the integrator information is not easily available at the project description pages and files. One could maybe try to infer that by making a historical analysis of merge commit authors. This would likely require a rigorous manual analysis to derive heuristics that could be used to answer this question. Another option would be to interview developers.

Chapter 4's study presents different directions for future works as well. First, our results could benefit from replications analyzing different projects, using different pro-

gramming languages, different version control systems, and different CI tools. Conversely, we could generate tests to expose more test conflicts. One possibility would be to extend Böhme et al. (BöHME; OLIVEIRA; ROYCHOUDHURY, 2013) approach to generate test cases exercising both developers' contributions. We could also replicate this study monitoring developers private repositories. This study would provide a better notion about the real frequency of conflicts. Finally, as discussed in Section 4.4, we could add refactorings detection algorithms to Crystal and compare if this would actually increase this tool's effectiveness.

REFERENCES

ACCIOLY, P.; BORBA, P.; CAVALCANTI, G. Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Software Engineering*, Springer Link, 2017.

ACCIOLY, P.; BORBA, P.; SILVA, L.; CAVALCANTI, G. Analyzing conflict predictors in open-source java projects. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. [S.l.]: ACM, 2018. (MSR '18).

aES, M. L. G.; SILVA, A. R. Improving early detection of software merge conflicts. In: *Proceedings of the 34th International Conference on Software Engineering*. [S.l.]: IEEE Press, 2012. (ICSE '12).

APACHE. *Apache Subversion*. 2015. <https://subversion.apache.org/>. Accessed: 2014-11-14.

APACHE. *Maven*. 2018. <https://maven.apache.org/>. Accessed: 2018-01-25.

APEL, S.; LESSENICH, O.; LENGAUER, C. Structured merge with auto-tuning: Balancing precision and performance. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: ACM, 2012. (ASE 2012).

APEL, S.; LIEBIG, J.; BRANDL, B.; LENGAUER, C.; KäSTNER, C. Semistructured merge: Rethinking merge in revision control systems. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. [S.l.]: ACM, 2011. (ESEC/FSE '11).

APPENDIX. *Chapter 3 Online Appendix*. 2018. <http://goo.gl/jmVJW7>. Accessed: 2018-01-23.

APPENDIX. *Chapter 4 Online Appendix*. 2018. <https://conflictpredictor.github.io/onlineAppendix/>. Accessed: 2018-01-23.

BARIK, T.; LUBICK, K.; MURPHY-HILL, E. Commit Bubbles. In: *Proceedings of the International Conference on Software Engineering, New Ideas and Emerging Results Track*. [S.l.]: ACM, 2015. (ICSE 2015).

BERZINS, V. On merging software extensions. *Acta Informatica*, v. 23, n. 6, 1986.

BIRD, C.; RIGBY, P. C.; BARR, E. T.; HAMILTON, D. J.; GERMAN, D. M.; DEVANBU, P. The promises and perils of mining git. In: *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*. [S.l.]: IEEE Computer Society, 2009. (MSR '09).

BIRD, C.; ZIMMERMANN, T. Assessing the value of branches with what-if analysis. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. [S.l.]: ACM, 2012. (FSE '12).

BLINCOE, K.; VALETTO, G.; DAMIAN, D. Do all task dependencies require coordination? the role of task properties in identifying critical coordination needs in software projects. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. [S.l.]: ACM, 2013. (ESEC/FSE 2013).

BöHME, M.; OLIVEIRA, B.; ROYCHOUDHURY, A. Regression tests to expose change interaction errors. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2013. (ESEC/FSE 2013).

BONFERRONI, C. E. *Teoria statistica delle classi e calcolo delle probabilità*. [S.l.]: Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commerciali di Firenze, 1936.

BRUN, Y.; HOLMES, R.; ERNST, M.; NOTKIN, D. Early detection of collaboration conflicts and risks. *Software Engineering, IEEE Transactions on*, IEEE Computer Society, 2013.

CATALDO, M.; HERBSLEB, J. D. Factors leading to integration failures in global feature-oriented development: An empirical analysis. In: *Proceedings of the 33rd International Conference on Software Engineering*. [S.l.]: ACM, 2011. (ICSE '11).

CAVALCANTI, G.; ACCIOLY, P.; BORBA, P. Assessing semistructured merge in version control systems: A replicated experiment. In: *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement*. [S.l.]: ACM, 2015. (ESEM'15).

CAVALCANTI, G.; BORBA, P.; ACCIOLY, P. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, ACM, 2017.

COSTA, C.; FIGUEIREDO, J.; MURTA, L.; SARMA, A. Tipmerge: Recommending experts for integrating changes across branches. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.]: ACM, 2016. (FSE 2016).

DIAS, M.; BACCHELLI, A.; GOUSIOS, G.; CASSOU, D.; DUCASSE, S. Untangling fine-grained code changes. In: *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. [S.l.]: IEEE Computer Society, 2015. (SANER 2015).

DIG, D.; JOHNSON, R. The role of refactorings in api evolution. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*. [S.l.]: IEEE Computer Society, 2005. (ICSM '05).

DIG, D.; MANZOOR, K.; JOHNSON, R. E.; NGUYEN, T. N. Effective software merging in the presence of object-oriented refactorings. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 34, n. 3, p. 321–335, 2008.

ECLIPSE. *JGit User Guide*. 2015. <http://wiki.eclipse.org/JGit/User_Guide>. Accessed: 2018-01-25.

ESTLER, H. C.; NORDIO, M.; FURIA, C.; MEYER, B. et al. Awareness and merge conflicts in distributed software development. In: *Proceedings of the IEEE 9th International Conference on Global Software Engineering*. [S.l.]: IEEE Computer Society, 2014. (ICGSE'14).

FALLERI, J.; MORANDAT, F.; BLANC, X.; MARTINEZ, M.; MONPERRUS, M. Fine-grained and accurate source code differencing. In: *ACM/IEEE International Conference on Automated Software Engineering*. [S.l.: s.n.], 2014. (ASE'14).

FILHO, R. *Using Information Flow to Estimate Interference Between Developers Same-Method Contributions*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, 2017. Accessed: 2018-01-22.

FOUNDATION, F. S. *Concurrent Versions System*. 2015. <http://www.nongnu.org/cvs/>. Accessed: 2014-11-14.

FOWLER, M. *Continuous Integration*. 2006. <https://www.martinfowler.com/articles/continuousIntegration.html>. Accessed: 2018-01-27.

Free Software Foundation. *Diff utils user's manual*. 2017. <https://www.gnu.org/software/diffutils/manual/diffutils.html>. Accessed: 2018-01-25.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

GIT. 2018. <https://git-scm.com/>. Accessed: 2018-01-25.

GITHUB. 2018. <https://github.com/>. Accessed: 2018-01-25.

GNU. *GNU Merge*. 2015. <http://www.gnu.org/software/diffutils/manual/html_node/>. Accessed: 2015-02-09.

GOUSIOS, G.; PINZGER, M.; DEURSEN, A. v. An exploratory study of the pull-based software development model. In: *Proceedings of the 36th International Conference on Software Engineering*. [S.l.]: ACM, 2014. (ICSE 2014).

GUZZI, A.; BACCHELLI, A.; RICHE, Y.; DEURSEN, A. van. Supporting developers' coordination in the ide. In: *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing*. [S.l.]: ACM, 2015. (CSCW '15).

HATTORI, L.; LANZA, M. Syde: A tool for collaborative software development. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*. [S.l.]: ACM, 2010. (ICSE '10).

HENKEL, J.; DIWAN, A. Catchup!: Capturing and replaying refactorings to support api evolution. In: *Proceedings of the 27th International Conference on Software Engineering*. [S.l.]: ACM, 2005. (ICSE '05).

HORWITZ, S.; PRINS, J.; REPS, T. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, ACM, New York, NY, USA, v. 11, n. 3, 1989.

JACKSON, D.; LADD, D. A. Semantic diff: A tool for summarizing the effects of modifications. In: *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1994. (ICSM '94), p. 243–252. ISBN 0-8186-6330-8. Disponível em: <http://dl.acm.org/citation.cfm?id=645543.655704>.

KALLIAMVAKOU, E.; DAMIAN, D.; BLINCOE, K.; SINGER, L.; GERMAN, D. M. Open source-style collaborative development practices in commercial projects using github. In: *Proceedings of the 37th International Conference on Software Engineering*. [S.l.]: ACM, 2015. (ICSE '15).

KASI, B. K.; SARMA, A. Cassandra: Proactive conflict minimization through optimized task scheduling. In: *Proceedings of the 2013 International Conference on Software Engineering*. [S.l.]: IEEE Press, 2013. (ICSE '13).

KHANNA, S.; KUNAL, K.; PIERCE, B. C. A formal investigation of diff3. In: *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*. [S.l.]: Springer-Verlag, 2007. (FSTTCS'07).

LESSENICH, O.; SIEGMUND, J.; APEL, S.; KÄSTNER, C.; HUNSEN, C. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering*, 2017.

LEVENSHTEIN, V. I. *Binary Codes Capable of Correcting Deletions, Insertions and Reversals*. [S.l.], 1966.

LIMA, G. *Uma Abordagem para Evolução e Reconciliação de Linhas de Produtos de Software Clonadas*. Tese (Doutorado) — Universidade Federal do Rio Grande do Norte, 2014. Accessed: 2018-01-12.

MCKEE, S.; NELSON, N.; SARMA, A.; DIG, D. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.]: IEEE Computer Society, 2017. (ICSME '17).

MENEZES, G. *On the Nature of Software Merge Conflicts*. Tese (Doutorado) — Federal Fluminense University, 2016. Accessed: 2017-06-16.

MENS, T. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, IEEE Press, 2002.

MERCURIAL. *Mercurial SCM*. 2018. <https://www.mercurial-scm.org/>. Accessed: 2018-01-25.

MUSLU, K.; SWART, L.; BRUN, Y.; ERNST, M. D. Development history granularity transformations (N). In: *30th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: IEEE Computer Society, 2015. (ASE '15).

MUYLAERT, W.; ROOVER, C. D. Prevalence of botched code integrations. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. [S.l.]: IEEE Press, 2017. (MSR '17).

NAGAPPAN, M.; ZIMMERMANN, T.; BIRD, C. Diversity in software engineering research. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. [S.l.]: ACM, 2013. (ESEC/FSE 2013).

NISHIMURA, Y.; MARUYAMA, K. Supporting Merge Conflict Resolution by Using Fine-Grained Code Change History. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

PERRY, D. E.; SIY, H. P.; VOTTA, L. G. Parallel changes in large-scale software development: An observational case study. *ACM Transactions on Software Engineering and Methodology*, ACM, 2001.

POTVIN, R.; LEVENBERG, J. Why google stores billions of lines of code in a single repository. *Commun. ACM*, ACM, 2016.

ROSENTHAL, R. *Parametric measures of effect size.* [S.l.]: Russell Sage Foundation., 1994.

SARMA, A.; REDMILES, D. F.; HOEK, A. van der. Palantír: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, IEEE Computer Society, 2012.

SHIHAB, E.; BIRD, C.; ZIMMERMANN, T. The effect of branching strategies on software quality. In: *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.]: ACM, 2012. (ESEM '12).

SILVA, L. M. P. *Build and Test Conflicts in the Wild*. Dissertação (Mestrado) — Universidade Federal de Pernambuco, 2018. Accessed: 2018-01-22.

SVAJLENKO, J.; ISLAM, J. F.; KEIVANLOO, I.; ROY, C. K.; MIA, M. M. Towards a big data curated benchmark of inter-project code clones. In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. [S.l.]: IEEE Computer Society, 2014. (ICSME '14).

TRAVIS. *Travis CI*. 2018. <https://travis-ci.org/>. Accessed: 2018-01-25.

TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L. M.; MAZINANIAN, D.; DIG, D. Accurate and efficient refactoring detection in commit history. In: *Proceedings of the 2018 International Conference on Software Engineering*. [S.l.]: IEEE Press, 2018.

TUFFERY, S. *Data Mining and Statistics for Decision Making*. 1st. ed. [S.l.]: Wiley Publishing, 2011.

WILCOXON, F.; WILCOX, R. A. *Some rapid approximate statistical procedures*. [S.l.]: Lederle Laboratories, 1964.

XUAN, Q.; FILKOV, V. Building it together: Synchronous development in oss. In: *Proceedings of the 36th International Conference on Software Engineering*. [S.l.]: ACM, 2014. (ICSE 2014).

ZHAO, Y.; SEREBRENIK, A.; ZHOU, Y.; FILKOV, V.; VASILESCU, B. The impact of continuous integration on other software development practices: A large-scale empirical study. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017. (ASE 2017).

ZIMMERMANN, T. Mining workspace updates in CVS. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. [S.l.]: IEEE Computer Society, 2007. (MSR '07).