Pós-Graduação em Ciência da Computação

Klissiomara Lopes Dias

Towards Requirements for Merge Conflict Avoidance Strategies

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2020

Klissiomara Lopes Dias

Towards Requirements for Merge Conflict Avoidance Strategies

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

Área de Concentração: Engenharia de Software
Orientador: Prof. Dr. Paulo Henrique Monteiro Borba

Recife
2020

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

D541t Dias, Klissiomara Lopes
Towards requirements for merge conflict avoidance strategies / Klissiomara Lopes Dias. – 2020.
94 f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências.

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

005.1 CDD (23. ed.) UFPE - CCEN 2020 - 74

Klissiomara Lopes Dias

“**Towards Requirements for Merge Conflict Avoidance Strategies**”

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: 20/02/2020.

Orientador: Prof. Dr. Paulo Henrique Monteiro Borba

I dedicate this work to everyone who gave me the necessary support to get here.

ACKNOWLEDGEMENTS

Agradeço à Deus, por tudo.

Aos meus pais pelo amor, dedicação, incentivo e por nunca terem medido esforços para oportunizar direta e indiretamente todas as conquistas a aprendizados ao longo da minha vida.

Aos meus irmãos por todo amor, incentivo e apoio, os quais homenageio através do meu irmão Kelvin que me acolheu com sua família nos primeiros meses do Doutorado e ao longo de toda a jornada, dando o conforto e orientações necessários, sempre atento.

Ao meu orientador, professor Paulo Borba, pela inquestionável competência com a qual orienta seus alunos e pelas oportunidades oferecidas. Para além de um mentor acadêmico, também me oportunizou grandes aprendizados que me fizeram e farão crescer não apenas academicamente, mas como ser humano.

Aos amigos e irmãos de pesquisa (Filhos de Borba) que o Doutorado trouxe, Léuson, Paola, Thais e Guilherme e aos amigos do Labes/SPG Waldemar (Neto), Wylliams, Jorge e Karine que tornaram o dia a dia mais leve na rotina do Labes, no convívio e conversas durante os almoços e paradas para os cafezinhos, que fizeram toda a diferença para o suporte não apenas emocional, mas também discussões acadêmicas. A todos estes amigos e demais colegas que não citei, homenageio-os através de Léuson e Neto que, assim como eu, estiveram distante de suas famílias e com os quais convivi mais rotineiramente nos anos que estive em Recife, pelos muitos momentos de descontração e suporte sempre célere e atentos para ajuda e discussão acadêmica, que me salvaram em muitos momentos e foram fundamentais durante o processo.

Aos demais membros do SPG pelas experiências e conhecimentos compartilhados. Aos alunos de graduação Marcos Barreto, Luis Delgado, ao pesquisador Waldemar Ferreira e aos profissionais voluntários entrevistados, pela fundamental colaboração nos estudos que deram origem a este trabalho.

Ao CIn/UFPE e aos seus funcionários pela estrutura e recursos fornecidos, assim como ao INES — Instituto Nacional de Ciência e Tecnologia para Engenharia de Software. Além desses, também agradeço ao CNPQ por financiar a minha pesquisa e à Administração Superior da Universidade Federal Rural da Amazônia (UFRA) por permitir a minha liberação para cursar o Doutorado. Aos colegas e amigos Pedro Campos, Fabio Bezerra e Orlando Ohashi por terem sido fundamentais para a viabilidade administrativa e prática da minha liberação.

Aos membros da banca professores Leonardo Murta, Eduardo Almeida, Rodrigo Bonifácio, Kiev Gama e Vinicius Garcia, pela paciência, disponibilidade e comentários valiosos que contribuíram para este trabalho e contribuirão para trabalhos futuros.

Obrigada a todos.

ABSTRACT

Merge conflicts often occur when developers change the same code artifacts. Such conflicts might be frequent in practice, and resolving them might be costly and is an error-prone activity. To minimize these problems by reducing merge conflicts, it is important to better understand how merge conflicts are affected by technical and organizational factors, so we can derive a set of factors development teams need to pay attention to avoid conflicts. With that aim, we conducted two empirical studies. First, we quantitatively investigate seven factors related to modularity, size, and timing of developers' contributions by reproducing and analyzing 73504 merge scenarios in GitHub repositories of Ruby and Python MVC projects. Then, to qualitatively evaluate our findings and explore other ways to avoid merge conflicts, we conducted 16 exploratory semi-structured interviews with practitioners across 13 companies, among them 4 global companies. Additionally, we also investigate how developers could be assisted in conflict avoidance by exploring the usefulness of a hypothetical tool for alerting merge conflict risk based on the factors we investigate. We bring quantitative evidence of 6 out of 7 modularity, size, and timing factors that are more likely associated with merge conflicts. We found most interviewees perceive an association between merge conflict occurrence and each factor we investigate in the previous study. Overall, we new 15 merge conflict factors considered important by the interviewees and 11 practices they use to avoid conflicts. Additionally, interviewees say that most merge conflicts are straightforward; exceptions, to mention a few, involve conflicts caused by global refactorings, when changes are related in non-trivial ways, demanding semantic understanding of the changes, and depending on the project phase. Finally, most interviewees consider a tool for alerting merge conflict risk useful for distinct purposes (e.g., to help the integrator or DevOps to early monitoring conflict issues) and give us insight into contexts in which this type of tool would not apply (e.g., projects that follow strict policies to avoid conflicts). Our results bring quantitative and qualitative evidence related to merge conflict reduction hypotheses and advice raised but not empirically evaluated by previous works, and we also go further by exploring new hypotheses related to contribution modularity and conclusion delay. By finding 15 new merge conflict factors in relation to our previous quantitative study and understanding how interviewees perceive their importance, we derive a set of more robust requirements for the development of more accurate conflict prediction models. Such models could then be used as a basis of project management and assistive tools that could help teams better deal with merge conflicts. Development teams can benefit from these results by improving the recommendations and guidelines to merge conflict avoidance. We also provide ideas on the usefulness and desirable features for the development of a tool for alerting conflict risk.

Keywords: Code integration. Merge conflict. Modularity. Collaborative development. Empirical study.

RESUMO

Conflitos de integração frequentemente ocorrem quando os desenvolvedores mudam os mesmos artefatos de código. Tais conflitos podem ser comuns na prática, e resolvê-los pode ser uma atividade custosa e propensa a erros. Para minimizar esses problemas reduzindo os conflitos de integração, é importante entender melhor como os conflitos de integração são afetados por fatores técnicos e organizacionais. Dessa forma, poderemos derivar um conjunto de fatores que as equipes de desenvolvimento poderão ficar atentas para evitar conflitos. Com esse objetivo, investigamos sete fatores relacionados à modularidade, tamanho e tempo das contribuições dos desenvolvedores. Para isso conduzimos dois estudos empíricos. Primeiro, reproduzimos e analisamos 73504 cenários de integração de código em repositórios GitHub de projetos Ruby e Python desenvolvidos usando o framework MVC. Segundo, para avaliar qualitativamente nossos resultados e explorar outras formas de evitar conflitos de integração, realizamos 16 entrevistas exploratórias semiestruturadas com profissionais de 13 empresas, entre elas 4 empresas globais. Além disso, também investigamos como os desenvolvedores poderiam ser assistidos na prevenção de conflitos, explorando a utilidade de uma ferramenta hipotética para alertar o risco de conflito integração com base nos fatores que investigamos. Nossos resultados trazem evidências quantitativas de que 6 dentre os 7 fatores de modularidade, tamanho e tempo que investigamos estão mais associados a conflitos de integração de código. Observamos que a maioria dos entrevistados percebe uma associação entre a ocorrência de conflitos de integração e cada fator que investigamos no estudo anterior. Ao todo, observamos 15 novos fatores de conflitos de integração que são considerados importantes pelos entrevistados e 11 práticas que eles sutilizam para evitar conflitos. Além disso, os entrevistados dizem que a maioria dos conflitos integração são simples para resolver, mas existem exceções, tais como quando envolvem conflitos causados por refatorações globais. Também observamos que a maioria dos entrevistados considera uma ferramenta para alerta do risco de conflito de integração útil, por exemplo, para facilitar o trabalho do integrador ou DevOps no monitoramento precoce de potenciais problemas de integração do código. Contudo, há contextos onde esse tipo de ferramenta não se aplicaria, por exemplo em projetos que seguem políticas rigorosas para evitar conflitos. Nossos resultados trazem evidências relacionadas à hipóteses e recomendações para a redução de conflitos levantados, mas não empiricamente avaliados por trabalhos anteriores, e também vamos além explorando novas hipóteses relacionadas à modularidade da contribuição e atraso de conclusão. Ao encontrar 15 novos fatores de conflito de mesclagem em relação ao estudo quantitativo realizado e entender como os entrevistados percebem sua importância, derivamos um conjunto de requisitos mais robusto para o desenvolvimento de modelos de previsão de conflitos mais precisos. Tais modelos poderiam então ser usados como base para o gerenciamento de projetos e ferramentas assistivas que poderia ajudar as equipes a lidar melhor com conflitos de fusão. As equipes de desenvolvimento poderiam se beneficiar desses resultados, melhorando as recomendações e diretrizes para evitar conflitos de integração. Também fornecemos ideias sobre a utilidade e características desejáveis para o desenvolvimento de uma ferramenta para alertar o risco de conflito.

Palavras-chaves: Integração de código. Conflitos de integração. Modularidade. Desenvolvimento colaborativo. Estudos empíricos.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In a collaborative software development context, developers make changes in software artifacts often without full awareness of changes being made by other developers. Version control systems (SUBVERSION, 2019; MERCURIAL, 2019; GIT, 2019) play a key role in collaborative software development. They help developers coordinate parallel changes in software products in a safe way by keeping track of project artifacts and provide a shared repository where developers can work together. That way, developers can individually work in their local copies of software artifacts, isolating their changes from others. In those scenarios, developers' parallel work can end up in hidden and undesirable interferences among the developers' tasks. In turn, when developers decide to update their changes to the shared repository, conflicts might occur. Merge conflicts, in particular, often occur when developers change the same code artifacts. Despite the advances of version control systems, the state of practice of merge tools rely on purely textual analysis to detect and resolve conflicts (MENS, 2002; ZIMMERMANN, 2007). That way, the merge tools currently adopted by the software development industry are not able to automatically resolve developers' overlapping textual changes.

Previous studies bring quantitative evidence that conflicts often occur and impair developers' productivity since they require developers to dedicate time and effort to solve them. Besides, understanding conflicting changes often is a demanding and error-prone task since developers might even introduce new code issues while attempting to fix conflicts (ZIMMERMANN, 2007; BIRD; ZIMMERMANN, 2012; KASI; SARMA, 2013; BRUN et al., 2013). As a matter of fact, a recent study conducted by Brindescu and others (BRINDESCU et al., 2020) brings evidence that code changes associated with merge conflict resolution are twice more likely related to the insertion of bugs in the code. This way, software quality might also be impacted since test conflicts might not be detected during the test phase, escaping to production releases and compromising correctness. Thus, merge conflicts might impair both development productivity and software quality.

Such evidence has guided and motivated the development of different tools and strategies that try to avoid or reduce conflicts. For example, by recommending an optimum order of task execution per developer (KASI; SARMA, 2013) or by providing more team awareness (GUIMARãES; SILVA, 2012; SARMA; REDMILES; HOEK, 2012; BRUN et al., 2013). The first strategy, based on the recommendation of task execution order, does not prevent developers from making decisions during the task execution that could yet end up in conflicts. While the remaining strategies generate many false positives and miss important false negatives since they are based on simple factors to predict conflicts, such as the existence of parallel changes made by two developers in the same class.

Additionally, in an attempt to avoid dealing with conflicts, developers adopt risky practices such as rushing to finish changes first (GRINTER, 1996; SARMA; REDMILES; HOEK, 2012; COSTA et al., 2014), and partial check-ins (SOUZA; REDMILES; DOURISH, 2003). Similarly, partially motivated by the need to reduce conflicts, development teams have been adopting techniques such as trunk-based development (ADAMS; MCINTOSH, 2016; HENDERSON, 2017; POTVIN; LEVENBERG, 2016) and feature toggles (BASS; WEBER; ZHU, 2016; ADAMS; MCINTOSH, 2016; FOWLER, 2017; HODGSON, 2017).

It is worth notice that the state of practice highlights the importance given by practitioners to the merge conflict problem. However, based on common beliefs, one can argue that merge conflict resolution is often a simple task, or even that they occur with low frequency. Maybe this perception is only possible, in practice, because the developers already adopt strategies to avoid conflicts, knowing that once a merge conflict occurs, it can end up in developer productivity interruption and software quality problems. For instance, Brindescu and others (BRINDESCU et al., 2020) bring evidence that the most common type of merge conflict is semantic conflicts, those that require the understanding of the logic of the changes involved to resolve the merge conflict successfully. By mining 143 open-source repositories, they found 59.46% of all merge conflicts belong to the semantic conflict type. Besides, they also found that the code involved in a conflict has twice more likely chances of being buggy. Furthermore, as we before mentioned, some practices adopted by developers are risky, such as partial check-ins, since it can also introduce buggy code. That context motivates the need for studies to investigate safe ways to better support practitioners in dealing with conflicts.

As studies in the area (MENS, 2002; ZIMMERMANN, 2007; BIRD; ZIMMERMANN, 2012; KASI; SARMA, 2013; BRUN et al., 2013) observe largely varying merge conflict rates among projects, and others (ESTLER et al., 2014) have observed a reduced negative merge conflict impact, it is important to find out which factors related to the developer's contributions are more associated with merge conflicts and are responsible for these variations. That way, to minimize these problems and better support team developers to prevent or reduce merge conflicts, it is important to understand how conflict occurrence is affected by more elaborate technical and organizational factors that are related to characteristics of the developer's contributions. By learning about these factors, we can derive a more solid set of conflict factors to the development of more accurate strategies to avoid or reduce merge conflicts.

With that aim, we conducted two empirical studies. First, we investigate seven factors related to modularity (*Changes to a common slice*), size (*Number of developers*, *Number of commits*, *Number of changed files*, and *Numberof changed lines*), and timing (*Duration* and *Conclusion delay*) of developers' contributions (DIAS; BORBA; BARRETO, 2020) by reproducing and analyzing 73504 merge scenarios in GitHub repositories of Ruby and Python MVC projects. We find evidence that the likelihood of merge conflict occurrence significantly increases when contributions to be merged are not modular in the sense that they involve files from the same MVC slice (related model, view, and controller files). We also find bigger contributions involving more developers, commits, and changed files are more likely associated with merge conflicts. Regarding the timing factors, we observe contributions developed over longer periods of time are more likely associated with conflicts. No evaluated factor shows predictive power concerning both the number of merge conflicts and the number of files with conflicts. Our results bring evidence related to merge conflict reduction hypotheses and advice raised but not empirically evaluated by previous studies (MCKEE et al., 2017; ADAMS; MCINTOSH, 2016; CATALDO; HERBSLEB, 2013). Still, we also go further by exploring new hypotheses related to contribution modularity and conclusion delay. Besides, we derive a set of more elaborated requirements for the development of merge conflict prediction models compared to the used by existing strategies (GUIMARãES; SILVA, 2012; SARMA; REDMILES; HOEK, 2012; BRUN et al., 2013) to drive the development and improvement of awareness and conflict detection tools.

Although we found quantitative evidence of modularity, size, and timing factors that are more associated with merge conflict occurrence, we want to refine our results to understand the issues involved. For instance, we want to better understand how the factors we investigate in the first study could be observed in different software development contexts that could possibly use other languages and frameworks, for example. Also, we want to learn about other associated factors, we did not consider before and their impact on merge conflicts. Additionally, we also want to go further by investigating different ways to avoid conflicts and investigate the practical feasibility of implementing a tool for alerting of merge conflict based on the modularity, size, and timing factors we investigate in the first study. So, in a second empirical study, we conducted 16 exploratory semi-structured interviews with practitioners across 13 companies, among them 4 global companies and 7 Brazilian companies (three companies hosted at Porto Digital technological innovation park). We found most interviewees perceive an association between merge conflict occurrence and each factor we investigate in the previous study. They gave us a comprehensive view of other factors that can influence the modularity, size, and timing of developers' contributions and, in turn, could impact positively or negatively in merge conflict occurrence. Overall, we observe 20 merge conflict factors (5 of them we had already investigate in our prior work) considered important by the interviewees and 11 practices they use to avoid conflicts.

Additionally, merge conflicts are observed with varying frequency rates, similar to previous studies in the area, including our first study. However, contrary to these studies, we go further and identify possible project contextual factors such as code integration frequency, team size, and task size, that could be responsible for the variations found in the different studies in the area. Moreover, interviewees also report that most merge conflicts are straightforward to solve, for instance, due to the small size of conflicting changes. However, there are exceptions; for example, they said merge become complex when it involves conflicts caused by global refactorings. Another example is when the conflicting changes are related in non-trivial ways, demanding semantic understanding of the changes. Interviewees also reported that merge conflicts become more complex to solve depending on the project phase. Finally, most interviewees consider a tool for alerting merge conflict risk useful for distinct purposes. For example, to assist less experienced developers and help the integrator or DevOps to early monitoring conflict issues. They also give us insight into contexts in which this type of tool would not apply, for instance, in projects that follow strict policies to avoid conflicts.

By finding 15 new merge conflict factors in relation to our previous study and understanding how interviewees perceive their importance, we derive a set of more robust requirements for the development of more accurate conflict prediction models. Such models could then be used as a basis of project management and assistive tools that could help teams better deal with merge conflicts. Development teams can benefit from these results by improving the recommendations and guidelines to merge conflict avoidance. We also provide ideas on the usefulness and desirable features of a tool for alerting conflict risk.

The studies we conducted during the execution of this thesis advances the state of research by providing a better understanding of existing factors more associated with merge conflict occurrence and how to prevent it. First, we conducted an empirical quantitative study to investigate the effect of 7 modularity, size, and timing of developers' contributions on merge conflicts. Then, we conducted a second study to qualitatively evaluate our findings and go further by exploring other existing merge conflict factors and ways to avoid merge conflicts. We did that based on the perceptions of practitioners working in different project contexts, with different team size, domain, programming languages and frameworks not limited to Ruby and Python, as we did in the first study. Based on our findings, a variety of research studies, strategies, recommendations, and tools can be derived, aiming at warning team members (i.e., developers, project managers, etc.) about code changes and more risky decisions during the execution of the developer's tasks that can lead to merge conflicts. Thus, software development teams can be better assisted and become more aware of the risks to make better decisions for preventing merge conflicts.

The studies and results presented in this thesis were first reported as research papers. This way, each study is presented in a chapter that includes the corresponding article manuscript format and content. The remainder of this document is organized as follows:

- In Chapter 2, we present the essential concepts used throughout this work;
- In Chapter 3, we describe our first empirical study that quantitatively investigates the association between merge conflicts and modularity, size, and timing factors. This chapter is published in (DIAS; BORBA; BARRETO, 2020), with the co-authoring of Paulo Borba and Marcos Barreto. Paulo Borba reviewed and guided this work. Marcos Barreto helped with the elaboration of the scripts to extract the modularity factor for our python sample and replicate the study for that sample.

- In Chapter 4, we describe our second empirical study that qualitatively assesses the findings of our prior study and investigates other ways do help in strategies for conflict avoidance. This study was performed in collaboration with Paulo Borba, Waldemar Ferreira, Luis Delgado, and Marcos Barreto. Paulo Borba reviewed, guided this work, and participated as the third researcher responsible for driving consensus during the interview analysis process. Waldemar Ferreira participated as a member of the INES support group for empirical studies and helped during the coding and analysis of interviews as the second researcher. Luis Delgado helped by participating in some interviews (by taking notes when necessary) and transcribed the interviews they participated in. Marcos helped with the transcription of some interviews.

- In Chapter 5, we present our concluding remarks and future work.

2 BACKGROUND

2.1 COLLABORATIVE DEVELOPMENT ENVIRONMENT AND CONFLICTS

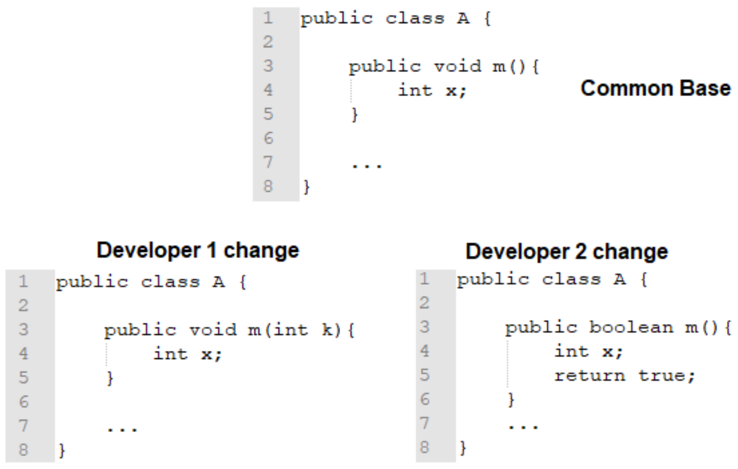
In a collaborative development environment, tasks are assigned to developers that work separately using individual copies of project files. As a result, while trying to integrate different contributions, one might have to deal with conflicting changes. Conflicting changes might occur when the code changes implemented by a given developer while developing his task can interfere in the changes made by other developers in their corresponding tasks. Conflicts might be detected during the merge step (due to *textual* conflicts), while building the system (due to *build* conflicts), or when running tests (due to *test* conflicts) (BRUN et al., 2013).

During code integration, for example, a merge conflict emerges when it is not possible to automatically integrate development contributions. A typical merge conflict happens when a developer tries to update the central repository by publishing changes he has made in a file and someone already edited a same area of the same file (the same text line or consecutive lines).

From now on, consider in the following examples that two developers always pull the most recent revision of the project code from the main remote (*master branch*) repository before starting working in their corresponding tasks. Thus, the code snippet on the top of each figure represents the common *Base* revision of a given class made available for both developers after they update their local workspace.

Figure 1 illustrates a merge scenario that leads to merge conflict. Developer 1 adds a parameter in the method signature (left code snippet), whereas Developer 2 edits the return method type (right code snippet). Despite its compatibility, most VCSs requires manual intervention to integrate these code changes, because Unix diff3- the default diff tool- does not consider language syntax. Besides, in that specific example, as both developers made changes to the signature of the same method, there is a need for understanding which change should be considered before solving the conflict.

Other conflicts might emerge after code integration. For instance, in a build conflict, correct individual contributions can introduce syntax inconsistencies into the system code when combined, even if changes affect different files. Figure 2 illustrates an example of merge scenario that leads to build conflict. The *Base* revision of class B contains a method m2() that return value 10. Developer 1 adds a method m1() in class A that calls the method m2() of class B. In the meantime, Developer 2 changes class B by removing the method m2() of this class. Developer 2 push their changes to the master repository before Developer 1 concludes his task. When Developer 1 integrates his code, no merge conflict occurs, since the two developers made changes in different artifacts. However, the method



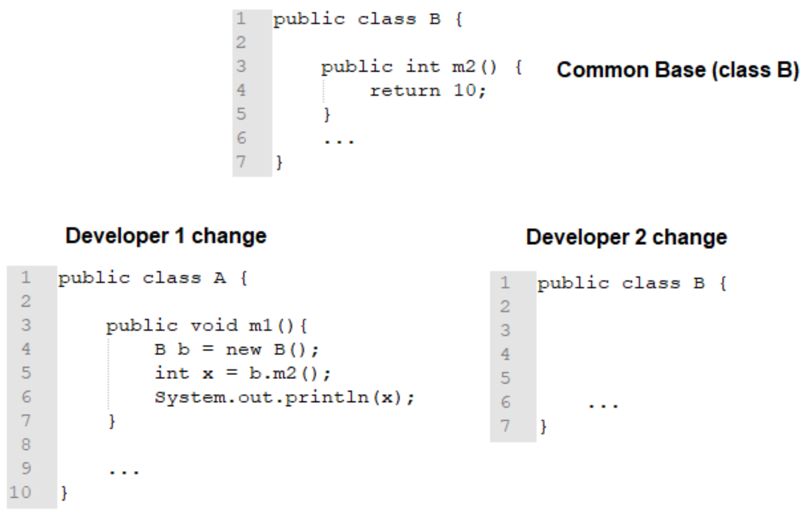
```
public class A {

    public void m(){
        int x;
    }

    ...
}
```

```
public class A {

    public void m(int k){
        int x;
    }

    ...
}
```

```
public class A {

    public boolean m(){
        int x;
        return true;
    }
    ...
}
```

Figure 1 – Example of conflicting merge scenario

call (left code snippet) in class A implemented by Developer 1 breaks the code because Developer 2 removed the method definition m2() in class B (right code snippet).



```
public class B {

    public int m2() {
        return 10;
    }
    ...
}
```

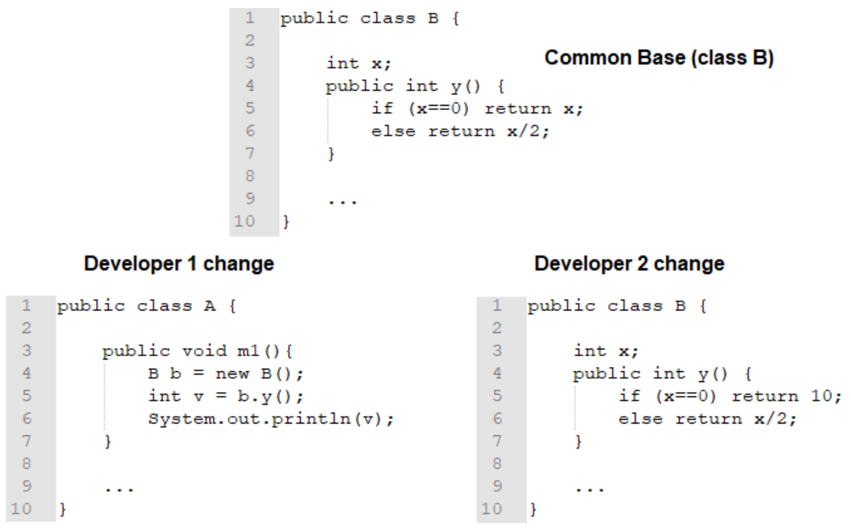
```
public class A {

    public void m1(){
        B b = new B();
        int x = b.m2();
        System.out.println(x);
    }

    ...
}
```

```
public class B {




    ...
}
```

Figure 2 – Example of conflicting build scenario

Test failure is another possible symptom of conflicting changes. In that case, there is no merge conflict neither build conflict, but some test case fails- at this point, we consider any kind of automatic test (unit test, functional test, acceptance test, among others). Similarly to *build* conflict, the problem might occur even if changes affect different files, but at this time, the key concern is the existence of a logical relationship among code elements. Figure 3 illustrates a situation that might cause a *test* conflict. The changes in class A made by Developer 1 (left code snippet) expect 0 as a result of call b.y(), given that method m1() does not explicitly assign a value for b.x. In contrast, Developer 2 changes the method y() in class B by defining that this method outputs value 10 in that case (right code snippet). As a result, a test that verifies method m1() fails after integration.

Finally, conflicts still might remain undetected as bug after code integration, as Figure



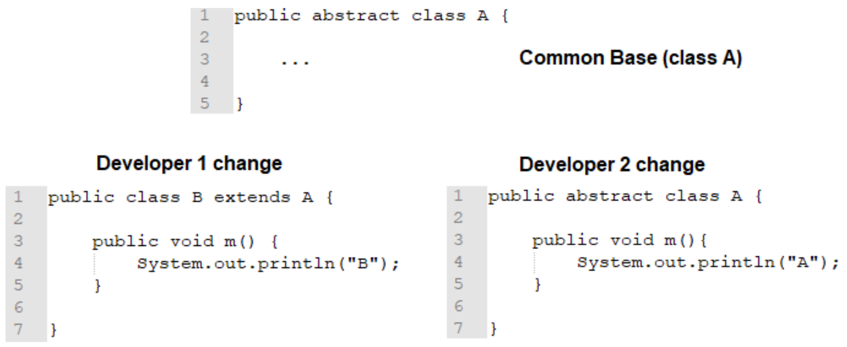
```java
public class B {

    int x;
    public int y() {
        if (x==0) return x;
        else return x/2;
    }

    ...
}
```

```java
public class A {

    public void m1(){
        B b = new B();
        int v = b.y();
        System.out.println(v);
    }

    ...
}
```

```java
public class B {

    int x;
    public int y() {
        if (x==0) return 10;
        else return x/2;
    }

    ...
}
```

Figure 3 – Example of test conflict

4 illustrates this case. The method m() added by Developer 1 in class B (left code snippet) causes an unexpected method overriding since Developer 2 concurrently added a method with the same signature in the superclass A (right code snippet). Suppose the code integration and the build succeed, the unit test related to the superclass also succeeds because it does not consider the new subclass. In turn, the unit test related to the subclass succeeds because it produces the expected result. The bug only emerges when someone updates the unit test related to the superclass to verify all the class hierarchy.



```java
public abstract class A {

    ...

}
```

```java
public class B extends A {

    public void m() {
        System.out.println("B");
    }

}
```

```java
public abstract class A {

    public void m(){
        System.out.println("A");
    }

}
```

Figure 4 – Example of test conflict that remains as bug

2.1.1 Conflicts in Practice

Some empirical studies bring quantitative evidence that conflicts occur frequently. For example, (ZIMMERMANN, 2007) analysed 4 large open-source systems and reported that between 23% and 47% of all merges had conflicts. (BRUN et al., 2013) analysed 9 of the most active open-source systems in GitHub repositories and reported that 17% of all merges had conflicts. To compute build and test conflicts, this same study analysed 3 projects and reported that 76% of merges completed cleanly, 16% of merges had conflicts, 1% of merges resulted in a build conflict, 6% of merges resulted in a test conflict, and

33% of all clean merges caused a build or test conflict. (KASI; SARMA, 2013) analysed 4 open source projects hosted on GitHub and reported that each project exhibited different distribution of each type of conflict as follows: Merge conflicts ranged from 4.2% to 19.3%, build conflicts ranged from 2.1% to 14.7%, and test conflicts ranged from 5.6% to 35%. Furthermore, conflicts occur irrespective of the size of the project (KLOC) or the numbers of developers.

In addition, these studies also present evidence that conflicts are costly to solving, regardless of their nature. In fact, in any case, developers dedicate time and effort to detecting, understanding, and effectively tracing the cause and seeking a solution for these problems. As an illustration, Bird and Zimmermann (BIRD; ZIMMERMANN, 2012) investigated how developers use branches in a large industrial project. They conclude integrating changes from multiple branches can be difficult and error-prone. As a consequence, branches incur an overhead in both developer effort and time. (BRUN et al., 2013) reported conflicts persist 10 days average (1.6 days median).

In turn, (KASI; SARMA, 2013) reported the resolution times for conflicts vary across 4 analysed projects. Developers needed between 6 days average (2 days median) and 22.93 days average (10 days median) to solve merge conflicts. In case of build conflicts, the resolution time varied between 0.75 days average (0.75 days median) and 5 days average (8 days median). Test conflicts required between 6.01 days average (4 days median) and 30.5 days average (14 days median) to be solved. These numbers are a proxy for the effort of conflict resolution. Given version control history only provide information about the time that the conflict emerges, and the time it no longer exists, the study assumes if a developer faced a conflict, then she exclusively worked to resolve that in subsequent merges, which is unrealistic. Even so, the results corroborate with evidence reported by other studies.

Finally, fear of conflicts might affect developers behavior (BRUN et al., 2012). Some developers avoid working in parallel to ensure not running into conflicts. Others share their code hurriedly in an attempt to avoid responsability for conflicts, while others postpone the incorporation of teammates work because they fear tricky conflicts. These deviant behaviors is one more evidence that conflicts negatively impact on software development.

2.2 RUBY ON RAIL APPLICATIONS

We execute our study using Ruby on Rails applications.[1] Ruby on Rails is a Model-View-Controller (MVC) framework for web applications. This means every Rails application organizes code into three distinct layers that we briefly describe as follows. The model layer is responsible for business logic and manages the interaction with elements in a database, including data validation. The view layer represents the user interface as HTML files (views) with embedded Ruby code and can provide content in different formats, such

[1] http://rubyonrails.org/

as HTML, PDF, XML, and so on. The controller layer interacts with models and views, receiving requests from views, processing data from models, and transferring data back to views.

Also, Ruby on Rails provides a router mechanism to assist controllers for dispatching incoming requests. That is, a router receives each incoming request, parses it and sends it to a controller class as a method call (action). To illustrate, suppose a user clicks on the **update profile** button in her prole page. Then, the application receives a URL request **http://my.app/profile/update/1**. The routing component translates the received request to a method call for **update** in the controller class **ProfileController** using 1 as an argument that identies the prole ID. The method **update** finds the profile, organizes the view to show the current content of the prole for edition, and invokes the view code.

Ruby on Rails became a popular framework, especially in the agile software development community since it improves developers productivity. Relying on naming conventions, this framework automatically configures code elements, enabling teams to release web applications faster than when using other technologies.

3 UNDERSTANDING PREDICTIVE FACTORS FOR MERGE CONFLICTS

In typical collaborative development environments, each developer has a private workspace and share contributions through a central repository, isolating changes from others. Although, in principle, this allows developers to work more efficiently by promoting parallel development, conflicts might emerge when integrating code, since changes made by one developer are hidden from others until one decides to update the central repository. Merge conflicts, in particular, often occur when developers change the same code artifacts, and resolving them might be costly and is an error-prone activity (MENS, 2002; ZIMMERMANN, 2007; BIRD; ZIMMERMANN, 2012; KASI; SARMA, 2013; BRUN et al., 2013).

To minimize these problems by reducing merge conflicts, it is important to better understand how conflict occurrence is affected by technical and organizational factors. So in this paper[1] we investigate seven factors related to three different aspects of developers contributions: *modularity* (contributions to be merged do not change a common application slice—MVC slice in this study, that is, related model, view, and controller files from web projects based on MVC frameworks), *size* (number of developers, commits, changed files, and lines in the contributions to be merged), and *timing* (contributions duration and conclusion delays). We evaluate the factors effect on the following variables: conflict occurrence, number of conflicts, and number of files with conflicts. This way we answer a number of research questions and can understand the impact the factors have on merge conflicts.

To answer these questions, we analyze 73504 merge scenarios from 100 Ruby (61759 merge scenarios) and 25 Python (11745 merge scenarios) projects hosted on GitHub. Given our interest in studying the modularity aspect, all projects are based on popular MVC frameworks available for the two languages: Rails for Ruby projects, and Django for Python projects. Our sample is composed only by merge scenarios resulted from a git merge command. For each merge scenario, we collect data about the mentioned factors, and reproduce the merge operation, observing conflict occurrence, number of conflicts, and number of files with conflicts. We then explore a number of regression models to assess the factors effect on these variables, and manually analyze scenarios to better understand the involved issues.

Our findings reveal that the likelihood of merge conflict occurrence significantly increases (6.13 times for the Ruby sample, and 4.39 times for the Python sample) when contributions to be merged are not modular in the sense that they involve files from the same MVC slice. We also find that bigger contributions involving more developers, commits, or changed files are more likely associated with merge conflicts for the Ruby sample. The same also

[1] The term paper is used throughout the text since this chapter includes the format and content of the research paper correspondent to the study, as highlighted in Chapter 1.

applies for the Python sample. Contributions involving more changed code lines are more likely associated with merge conflicts only for the Python sample.

By contrast, our sample reveals that the conclusion delay between merged contributions has no effect on merge conflict occurrence. Moreover, no evaluated factor shows predictive power concerning both the number of merge conflicts and the number of files with conflicts. So whereas the factors can predict conflict occurrence and the associated damage, they cannot predict the extent of the damage. Besides these main findings, we bring a number of extra observations and relate them to previous work. For example, conflicts happen in 13.4% of merge scenarios in our Ruby aggregated sample, with project rates ranging from 0.9% up to 54.5%. For the Python sample, conflicts happen in 10.0%, with project rates ranging from 2.1% up to 37.5%.

Our findings suggest that managers of MVC projects should consider aligning the structure of development tasks with the structure of the associated application MVC slices, and avoid the parallel execution of tasks that focus on common slices. Artificially aligning task structure with the structure of the underlying programming language modules or packages, could lead to highly coupled tasks that compromise parallel development. Our results could also be used to derive merge conflict prediction models. Project management and assistive tools could benefit from these models by supporting earlier decisions about when to integrate contributions to mitigate or reduce conflicts.

The remaining of this paper is organized as follows. Section 3.1 motivates our study and presents our research questions. Section 3.2 describes our study setup and how we collect the investigated predictors.[2] In Sections 3.3 and 3.4, we respectively discuss findings and implications. Section 3.5 presents the threats to the validity of our study. Related work is discussed in Section 3.6.

3.1 MERGE CONFLICTS IN PRACTICE

High degrees of parallel changes, and merge conflicts, have been observed in a number of industrial and open-source projects that use different kinds of version control systems (PERRY; SIY; VOTTA, 2001; ZIMMERMANN, 2007; BRUN et al., 2013; KASI; SARMA, 2013). This is observed even when using advanced merge tools (APEL et al., 2011; APEL; LESSENICH; LENGAUER, 2012; CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY; BORBA; CAVALCANTI, 2018) that avoid common spurious conflicts identified by state of the practice tools. Resolving such conflicts might be time consuming and is an error-prone activity (SARMA; REDMILES; HOEK, 2012; BIRD; ZIMMERMANN, 2012; MCKEE et al., 2017). So, to avoid dealing with conflicts, developers adopt risky practices such as rushing to finish changes first (GRINTER, 1996; SARMA; REDMILES; HOEK, 2012), and partial check-

[2] The term predictors used throughout the text refers to the independent variables we investigate and is not used to imply causality. Variations such as *predict* and *predictive* are used throughout the text with the meaning of “potential” to predict.

ins (SOUZA; REDMILES; DOURISH, 2003). Similarly, partially motivated by the need to reduce conflicts, development teams have been adopting techniques such as trunk-based development (ADAMS; MCINTOSH, 2016; HENDERSON, 2017; POTVIN; LEVENBERG, 2016) and feature toggles (BASS; WEBER; ZHU, 2016; ADAMS; MCINTOSH, 2016; FOWLER, 2017; HODGSON, 2017).

As these studies observe largely varying merge conflict rates among projects, and others (ESTLER et al., 2014) have observed a reduced negative merge conflict impact, it is important to find out technical and organizational factors that are associated with merge conflicts and are responsible for these variations. This way we can better propose strategies to prevent or reduce merge conflicts.

With that aim, we initially carried out informal exploratory interviews with five experts from four software companies, two of them with offices around the world. One of the experts is an independent consultant with experience in a number of countries. The others have experienced one or more roles as developer, technical leader, software architect, and manager. As the idea was to understand how these experts experience merge conflicts in practice, questions were mostly aimed at understanding how they define, plan, organize, allocate, and execute development tasks, and how this could impact on conflicts.

Although most experts acknowledged different degrees of merge conflict occurrence associated to parallel development and the lack of task modularity, one of them surprisingly reported merge conflicts were not an issue even under the presence of parallel development and distributed teams. He attributed the lack of conflicts to the use of Rails, a Ruby MVC framework.

Influenced by foundational modularity work (PARNAS, 1972; BALDWIN, 2000; CATALDO; HERBSLEB, 2013), we hypothesized the lack of conflicts resulted not from the simple use of Rails, but from matching the structure of development tasks with the structure of key framework concepts: *slices*, that is, groups of model, view and controller files related to a particular domain object, as illustrated in Figure 5 for the Post slice in a Ruby on Rails application. The model layer is responsible for business logic and manages the interaction with elements in a database, including data validation. The view layer represents the user interface as `.erb` files containing HTML with embedded Ruby code. The controller layer interacts with models and views. Figure 5 shows, surrounded by red boxes, a slice named Post and its related files: `posts_controller.rb` (controller layer), `post.rb` (model layer), `_table.html.erb`, `archived.html.erb`, `edit.html.erb`, `index.html.erb`, and `show.html.erb` (view layer). This is aligned with current development practices (BOTTCHER, 2017), which propose so called *vertical slices* as an "strategy to empower developers so they can deliver valuable outcomes with short feedback cycles".

For MVC projects, evaluating this hypothesis is especially important because, in addition to the structure defined by vertical slices, one could also consider the structured defined by the three model, view, and controller layers or horizontal slices, not to mention

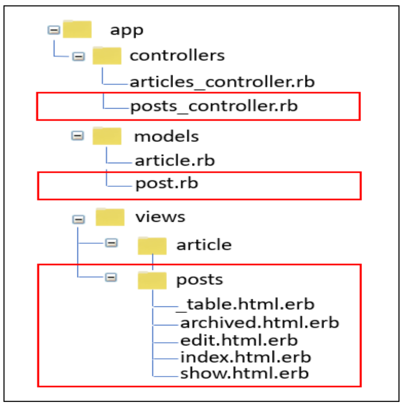


Figure 5 – Post slice files

the underlying structure of the programming language modules and packages. So, to avoid conflicts, we would not only have to avoid parallel tasks that focus on common modules, but we would have to decide which modular structure to consider for task allocation in the first place.

With that motivation, and considering the importance of context to Software Engineering research (BRIAND et al., 2017), we decided to investigate whether contribution *modularity* influences merge conflicts in Ruby and Python MVC projects. We assume contributions are the net result of development tasks; in general, each contribution consists of a graph of commits, but they often can be seen simply as a sequence of commits. We consider that two contributions to be merged are modular when they do not involve files from a common MVC slice. So we ask the following research question.

RQ1: What is the effect of contribution modularity on merge conflicts?

We use changes to a common MVC slice as a measure of non modularity. We analyze the files modified by each contribution in a merge scenario, and relate them to their slices. A merge scenario is a triple formed by left and right commits[3] to be merged, and a base commit that is a common ancestor to left and right. The left (right) contribution starts with the left (right) commit in a scenario, and includes all reachable commits up to, but no including, the base commit.

Motivated by previous work, interview feedback, and anecdotal evidence, we go beyond the modularity aspect and investigate merge conflict factors related to contribution *size*. McKee and others (MCKEE et al., 2017), for example, informally claim that practitioners should attempt to make smaller commits, and commit often to prevent and alleviate the severity of merge conflicts. However, they do not formally investigate whether contribution

[3] In git repositories, every merge commit has two parents. We arbitrarily named them as left and right commits.

size metrics are related to merge conflict occurrence. To assess that, we ask the following question.

> **RQ2: What is the effect of contribution size on merge conflicts?**
>
> For answering this question, we compute the *number of developers*, *commits*, *changed files*, and *changed lines* in each contribution to be merged. Then, for each factor, we compute the geometric mean ($\sqrt{a.b}$) of the values obtained in both contributions (say a and b). This way we normalize the data since a and b might substantially differ.

Finally, we also consider factors related to the contribution *timing* aspect. Adams and McIntosh (ADAMS; MCINTOSH, 2016) suggest that the best way for reducing conflicts is to keep branches short-lived and merge often. Bird and Zimmerman (BIRD; ZIMMERMANN, 2012) interview developers that suggest that problems are caused by long delays in integrating contributions and moving them between teams. To better assess that, we ask the following question.

> **RQ3: What is the effect of contribution timing on merge conflicts?**
>
> For answering this question, we use contribution *duration* and *conclusion delay.* For each contribution to be merged, we determine duration by computing the number of days between the last commit in the contribution (the one just before merging) and the common ancestor with the other contribution. Then we compute the geometric mean as RQ2. To determine conclusion delay, we compute the difference, in days, between the dates of the last commit of each contribution.

Besides these three main questions, we later introduce and investigate a number of derived and related questions that more deeply explore the issues involved.

3.2 STUDY SETUP

To answer the research questions presented in the previous section, we analyze a number of merge scenarios, collecting data about the mentioned conflict factors, and reproducing the merge operation in each scenario to observe conflict occurrence, number of conflicts, and number of files with conflicts. To support that, we implement an infrastructure that involves two steps, as illustrated in Figure 6. The first focuses on mining merge scenarios (triple formed by a base commit and the two parent commits associated with a merge commit) from GitHub projects, reproducing them, and collecting information about the dependent variables. The second collects, for each scenario, information about the conflict factors, and explores regression models to assess the factors effect on these variables. Later in this section we discuss how we obtain our sample.

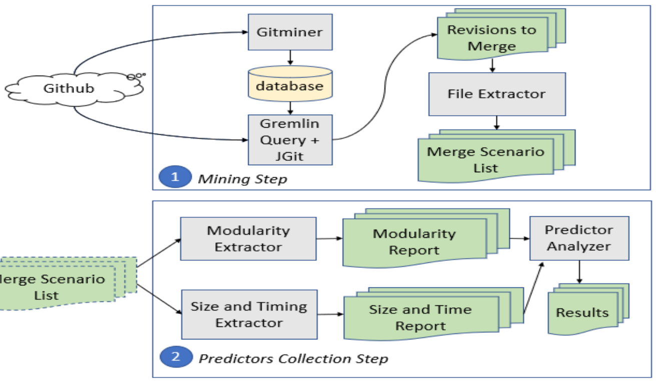


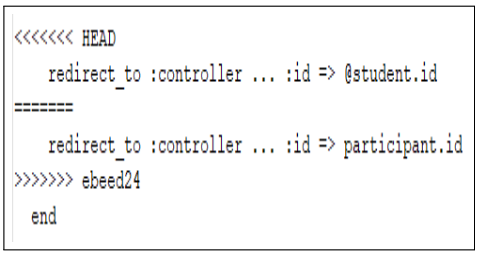
Figure 6 – Study Setup

We characterize a merge conflict from three different perspectives (*merge conflict occurrence*, *number of merge conflicts*, and *number of files with merge conflicts*), so we could investigate whether the factors we evaluate are related with not only conflict occurrence and the associated damage, but also with the extent of the merge conflict damage. We measure the extent of the damage in two different ways, by counting the number of merge conflicts, and the number of files with merge conflict.

3.2.1 Mining Step

To mine merge scenarios, we implemented a script[4] to retrieve the full version control history of each evaluated project. To do so, we locally clone the project and query the project history to retrieve a list of all merge commit ids— commits resulting from a *git merge* command. This list is ordered by merge date. For each merge commit, our script also collects the ids of their parent commits and common ancestor. We then, for each merge commit, checkout the revisions involved in the merge scenario: base, and left and right parent revisions, as explained in the previous section. After that we reproduce the merge scenario and check whether the merge resulted in conflicts. When that is the case, we set the *merge conflict occurrence* metric to 1; otherwise we set it to 0. As the *git merge* command result lists the names of all files involved in conflicts, we easily compute the *number of files with merge conflicts* by counting the number of files in the resulting list. Also, as a result of a *git merge* command, all chunks of code involved in a merge conflict (*conflicting chunks* (GHIOTTO et al., 2018)), are delimited by lines containing specific markers ("«««<" and "»»»>", with the separator "======="). Figure 7 illustrates how a typical merge conflict looks like. So, to compute the number of merge conflicts, we look for these patterns in all files involved in a merge scenario that resulted in a conflict. This way, the *number of merge conflitcs* of a merge scenario is the sum of all conflicting chunks in the files in the scenario. These data constitute the dependent variables used in this study and are summarized in Table 1 (see *Dependent Variables* category), with the

[4] Available in our online appendix (APPENDIX, 2018).



```
<<<<<<< HEAD
    redirect_to :controller ... :id => @student.id
=======
    redirect_to :controller ... :id => participant.id
>>>>>>> ebeed24
  end
```

Figure 7 – A conflicting chunk example of merge 96df565 from Expertiza project

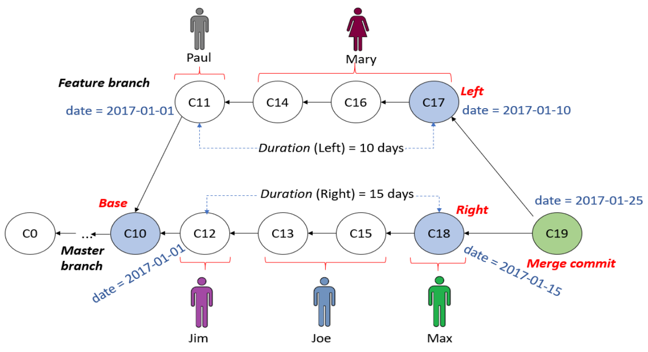


Figure 8 – Example of Merging Contributions

metric name on the left, and its description on the right.

Subsequently, our scripts extract the list of file names changed (edited, added, or removed) by all revisions between a parent (left or right) and a base revision of a merge scenario. As explained in the previous section, the changes associated to these files is what constitutes a contribution. To better understand that, consider Figure 8, which illustrates such a merge scenario. Let us consider a hypothetical project, which adopts a feature branch model for implementing development tasks in parallel to the master branch. Figure 8 presents part of the commit history that includes these two branches. In this picture, let us assume the *Master* branch has already integrated with the branch that contains changes made by three developers (Jim, Joe, and Max), while two developers (Paul and Mary) work in the *Feature* branch. As the *Master* branch has changed since the *Feature* branch creation, when Mary tries to integrate the *Feature* branch by invoking git merge a new merge commit, C19, is created. The corresponding merge scenario is depicted in blue. Its base, left, and right revisions are respectively C10, C17, and C18. The merge commit is not actually part of the scenario. The left contribution is composed by four commits (C11 to C17), and the right contribution is also composed by four commits (C12 to C18). Two developers (Paul and Mary) worked on the left contribution, while three developers (Jim, Joe, and Max) worked on the right contribution. Other details in the figure are later explained.

As our mining step is driven by the merge commits we find on GitHub repositories, we do not collect all integration scenarios that actually occurred in the project, as a number of git commands (rebase, cherry-pick, stash apply, etc.) locally integrate code but do not leave public traces of the integration, and some others, like squash, might even rewrite project history and erase traces that would eventually appear in public repositories (KALLIAMVAKOU et al., 2014; BIRD et al., 2009; LESSENICH et al., 2018).

3.2.2 Predictors Collecting step

With the scenarios and associated extra information collected in the previous step, in this step our scripts collect contributions, slices, and information about the conflict factors so that we can answer the research questions. Table 1 summarizes the metrics according to each investigated factor category (see *Modularity Metric*, *Size* and *Timing Metrics*).

Table 1 – Merge conflict factors.

Metric	Description
Number of merge conflicts	Sum of all conflicting chunks (reported by the git line-based merge tool when merging the associated contributions) in the files in the scenario.
Merge conflict occurrence	Binary variable (assuming boolean values in this study), with 1 indicating that there was at least one merge conflict when merging the associated contributions. Otherwise the variable is set to 0.
Number of files with merge conflicts	Number of files with at least one merge conflict reported by the git line-based merge tool when merging the associated contributions.
Changes to a common slice	Binary variable (assuming boolean values in this study), with 1 indicating that each contribution changed at least one file belonging to a common slice. Otherwise the variable is set to 0.
Number of developers	The geometric mean of the number of commit authors in each contribution. We choose commit *author* over *committer* because the first refers to the person who originally wrote the contribution, whereas the second refers to who last applied the contribution (CHACON; STRAUB, 2014).
Number of commits	The geometric mean of the number of commits in each contribution.
Number of changed files	The geometric mean of the number of changed files in each contribution. A change applied to a file in a commit and later reverted is considered to change the file according to this metric.
Number of changed lines	The geometric mean of the number of added and removed lines in each contribution. Modified lines are counted as first removed and then added (ZHAO et al., 2017). Adding a line to a file in a commit and later deleting it in the same contribution leads to two changed lines according to this metric.
Duration	The geometric mean of the number of days between the common ancestor and the last commit in each contribution.
Conclusion delay	The difference, in days, between the dates of the last commit of each contribution.

Except for the *Modularity Extractor* component, the same infrastructure was used to study Ruby and Python projects. A language and framework specific component is needed to extract the *modularity* factor, since each framework relies on specific naming conventions and directory structures to represent slices. For example, Rails adopts an specific directory structure for separately storing models, views, and controllers files. By contrast, Django supports a variety of similar structures and conventions.

In Rails, classes inside the model directory represent models, which are named in the singular such as `post.rb`. The views are stored in subfolders of the views directory, named as the plural of the corresponding model names. So the posts folder contains all view files related to the post model. The controller class is prefixed with the plural of its related model name. These conventions are depicted in Figure 5, which illustrates a common way to organize Rails code.

The module notion (called slice or MVC slice) used in this study relies on framework conventions for organizing code. It consists of a group of related model, view, and controller files that can be traced and matched by combining both the naming conventions and standard directory structure established by each MVC framework.

So, looking for these patterns, our scripts identify slices (and their associated files) such as the *Post* slice illustrated in Figure 5. The changes to a *common slice* metric is then easily computed by analyzing the files modified by each contribution of a merge scenario, and relating them to the previously identified slices.

As we want to detect whether merge scenario contributions changed files from the same slice, the *changes to a common slice* metric is set to 1 when both contributions changed at least one file related to the same slice, and it is set to 0 otherwise. Considering the example in Figure 8, suppose that Joe and Mary changed the `posts_controller.rb` class (which is related to the Post slice in Figure 5) as part of their respective contributions. In this case, *changes to a common slice* is set to 1.

To better understand how we compute some of the metrics, consider the example in Figure 8. All metrics are collected per merge scenario. As the values obtained in each contribution related to a given merge scenario might substantially differ, we normalize these metrics by using geometric mean. Other ways to normalize these values could be used. We decide to use geometric mean by following a closely related study (LESSENICH et al., 2018) to ours. So, we can compare both studies more accurately.

The number of developers is 2 (Paul and Mary) in the left contribution, and 3 (Jim, Joe, and Max) in the right. Thus the variable *number of developers* is set to 2.4 ($\sqrt{2.3}$) for this merge scenario. We compute *duration* first by determining the duration of each contribution (left/right), which in this study means the number of days between the *base* (common ancestor) and the last commit in a contribution (the one just before merging). Second, we compute the geometric mean of these values to determine *duration*. As the left contribution duration time is 10 days (C11 author date[5] is 2017-01-01 and C17 author date is 2017-01-10), and the right contribution is 15 days (C12 author date is 2017-01-01 and C18 author date is 2017-01-15), the contributions *duration* is set to 12.2 ($\sqrt{10.15}$) for this scenario.

Finally, we compute *conclusion delay* by determining the difference, in days, between the dates of the last commit (the one just before merging) of each contribution. This way, in the merge scenario depicted in Figure 8, the *conclusion delay* for the contributions in this scenario is set to 5, given that C17 author date is 2017-01-10, and C18 author date is 2017-01-15, and these are the last commits of each contribution.

3.2.3 Sample

To select our Ruby sample, we first used GitHub's advanced search page[6] to return only Ruby[7] projects with more than 500 stars, and ordered the list by recent project activity, as a start point to filter meaningful projects. We apply the same criteria for the Python sample.

After that, we manually analyzed the resulting lists of Ruby and Python projects discarding those that do not use the Rails and Django MVC frameworks, given GitHub does not provide a mechanism to query projects according to the used MVC framework. As the module notion (called slice or MVC slice) used in this study relies on Frameworks conventions for organizing code, we also exclude projects that use one of these frameworks but do not fully comply with standard framework conventions, such as having a specific directory structure. For example, we only considered Rails projects whose project's directory structure contains an *app* folder with the following subfolders: *models*, *views* and *controllers*. This is needed both for the sample relevance and for conformance with our scripts that rely on framework conventions. To discard less relevant projects, we manually check the available project documents such as Readme file, official web site, and wikis. If these files are not available or do not provide evidence of relevance, we discard the project.

Due to the filtering limitations of GitHub's advanced search page, we initially obtained a large number of non-MVC projects, forcing us to gradually decrease the stars threshold we initially adopted. Despite that, we highlight the number of stars was used only as an initial filter, in an attempt to use a well-known Github proxy to get popular projects first. We further apply additional criteria to identify active and real development projects (KALLIAMVAKOU et al., 2014). Besides, the final project list conforms to existing recommendations to avoid personal projects, since all projects have at least three commiters and 92% of them has more than 10 commiters. As a result, we consider the first 100 Ruby projects and 25 Python projects in their respective lists.

[5] The date when the author made the original commit.
[6] <https://github.com/search/advanced>
[7] GitHub search page allows only filtering repositories based on programming language, not on frameworks.

Although we have not systematically targeted representativeness or even diversity (NAGAPPAN; ZIMMERMANN; BIRD, 2013), by inspecting our sample we observe some degree of diversity concerning the following dimensions: size, domain, number of collaborators, number of commits, and number of merge scenarios. Our sample contains projects corresponding to applications in different domains such as Development, System Administration, and Communications (SOUZA; MAIA, 2013). For example, we categorize applications that support in some way the software development into the Development category. We summarize the domain diversity of our sample in Table 2. They also have varying sizes, as shown in Table 3. For example, Refinery CMS News, a plugin for Refinery CMS, has only 2.3 KLOCs, while Discourse, a platform for community discussion, has approximately 716 KLOCs. Moreover, Sapos has 11 collaborators, while Whitehall has 154 collaborators.

Table 2 – Distribution of projects by Domain

Domain	Ruby	Python
Development	28%	40%
System Administration	19%	24%
Communications	15%	16%
Business & Enterprise	14%	12%
Home & Education	11%	-
Security & Utilities	6%	8%
Graphics	6%	-
Audio & Video	1%	

Table 3 – Projects Data

Sample	KLOC	Contributors	Stars
Ruby	7 – 445	3 – 629	11 – 2433
Python	46 – 88	12 – 378	711 – 4731

The first and last revisions of evaluated projects range from 20 September 2007 to 30 November 2017, for the Ruby sample, and range from 20 October 2007 to 30 November 2017 for the Python sample. For further information on our sample, we provide a complete subject list in our online appendix (APPENDIX, 2018).

3.3 RESULTS

In this section we present our study results. For brevity, we mostly report and discuss the results of the Ruby sample, referring to the Python results when they diverge from the Ruby results.

3.3.1 RQ1: What is the effect of contribution modularity on merge conflicts?

Conflicts occur even when merging modular contributions

To answer **RQ1**, we first analyze the frequency of conflicting merges taking into account the modularity of the related contributions. Although most scenarios with conflicts (57.3%) in the Ruby sample are not modular in the sense that their contributions involve files from the same MVC slice, we observe that merge conflicts also occur with modular merge scenarios, which change disjoint sets of slices (42.7%). So aligning slice and task structure by defining tasks that focus on specific slices, and only executing in parallel tasks that focus on disjoint slices, gives no guarantees of conflict avoidance. The results of a per project analysis reinforce the trend of the aggregated sample. Just 7.0% of the projects have conflicts only in non modular scenarios, and just 7.0% have conflicts exclusively in modular scenarios.

Surprisingly, this goes against the expectation raised by one of the experts we interviewed (see Section 4.1), and does not confirm our motivating hypothesis. So we try to better understand the issues involved. By manually inspecting the files of a few conflicting modular contributions, we observe that conflicts are caused because of parallel changes to files that are not part of the slice structure; this includes configuration files, and files that define classes reused across slices, as later detailed in the paper. So the structure of the slices covers most, but not all, application files. This invalidates the motivating hypothesis. Nevertheless, this does not preclude weaker relations between conflicts and contribution modularity.

Likelihood of conflict occurrence significantly increases when contributions to be merged are not modular

We then further investigate the relation between the *changes to a common slice* and *merge conflict occurrence* metrics. Knowing that modular contributions do not prevent merge conflicts, we now investigate whether non modular contributions increase the likelihood of merge conflict occurrence. With that aim, following the Principle of Parsimony (CRAWLEY, 2014), we apply Logistic Regression (JR; LEMESHOW; STURDIVANT, 2013) models to estimate the probability of merge conflict occurrence. Logistic regression is the simplest technique that applies to our context: a binary dependent variable (*merge conflict occurrence*, (say c), and continuous[8] and categorical independent variables. For example, when we consider *changes to a common slice* (say cs) as the categorical independent variable, our model (say m) is expressed as follows:

$$m = glm(c \sim cs, family = binomial, data = dataSample)$$

[8] We also apply logistic regression to assess the *size* and *timing* metrics, as shown in Sections 3.3.2 and 3.3.3.

where *glm* is the R (PROJECT, 2017) function used to run a logistic regression. This function receives as first parameter the dependent variable to the left of the $\sim$ (in this example, c) and the independent variable after the $\sim$ (in this example, cs[9]). After the comma, we specify that the distribution is binomial, as the outcome variable c is binary. Finally, we inform the sample dataset.

To assess the fit of a model, we report the deviance, and the percentage of the deviance explained by the model. Lower deviance values are associated with better fit of the model to the data. To simplify interpretation, we report the odds ratio associated with our measure, instead of reporting the regression coefficients. An odds ratio is a relative measure of effect size, which allows the comparison of groups regarding the occurrence of a specific event. In this case, this measure allows evaluating the effect size of modular contributions compared to non modular contributions on merge conflict occurrence. Odds ratio larger than 1 indicate a positive relationship between the independent and dependent variables, whereas odds ratio less than 1 indicate a negative relationship.

Table 4 – Odds Ratios from Regression Assessing Merge Conflicts Factors.

	Model I	Model II	Model III	Model IV	Model V	Model VI	Model VII
Changes to a common slice	6.13^{***}	4.28^{***}	3.80^{***}	5.18^{***}	4.28^{***}	3.78^{***}	5.11^{***}
Number of commits		3.55^{***}			3.53^{***}		
Number of developers			1.17^{***}	1.56^{***}		1.15^{***}	1.51^{***}
Number of changed files			3.27^{***}			3.24^{***}	
Number of changed lines				0.99			0.99
Duration						1.04^{***}	1.09^{***}
Conclusion delay					1.01	1.01	1.01
Deviance Explained	10.9%	17.5%	19.0%	14.3%	17.5%	19.0%	14.5%

($^{***}p < 0.001$; $^{**}p < 0.01$; $^{*}p < 0.05$)

Table 4 presents the logistic regression results for all metrics investigated in this study. A model in column i indicates that the dependent variable *merge conflict occurrence* is expressed in terms of the variables in the first column that have a corresponding value in column i.

The superscript labels associated with the values indicate the statistical significance of results, as described just below the table. Thus, Model I in Table 4 reveals a statistically significant ($p < 0.001$) relationship between merge conflict occurrence and the non modularity of contributions that change common slices. So we can say that non modular contributions have a significant effect on merge conflict occurrence. More precisely, when contributions change files from the same slice, the likelihood of merge conflict occurrence is 6.13 times higher than when contributions change files from different slices. Although

[9] To run a logistic regression with more than one independent variable, each new variable should be listed separated by +.

not shown in the table, for Python, we observe a 4.39 times higher likelihood. In Model I, the modularity factor explains 10.9% of the deviance in the data. When we look at individual projects, the modularity factor can explain up to 33.9% of the deviance.

These results better explain the initial expectation discussed in Section 4.1. Managers of MVC projects should consider aligning the structure of development tasks with the structure of the associated application MVC slices, and avoid the parallel execution of tasks that focus on common slices. Whereas this does not eliminate merge conflicts as initially anticipated, it can help to significantly reduce occurrence. Tasks that focus on different slices have reduced chances of changing the same files, and therefore of leading to conflicts.

Contribution modularity is not associated with the extent of merge damage

Given the strong effect of *changes to a common slice* on *merge conflict occurrence*, we explore whether a similar effect applies to the other conflict metrics. As our variable *changes to a common slice* is binary, we compute the point-biserial correlation coefficient. First with *number of merge conflicts*, and then with *number of files with merge conflicts*. We consider the effect size based on the correlation's strength and significance ($p < 0.05$), adopting a minimal superior threshold of 0.6 for strong correlations (ANDERSON; FINN, 1996). We observe only weak correlations (0.07 to 0.13).

The absence of strong correlation is also observed with a corresponding per project analysis. This suggests contribution non modularity has no predictive power concerning the number of merge conflicts or the number of files with conflicts. So whereas non modularity can be used to predict conflict occurrence and the associated damage, it cannot predict the extent of the damage.

Conflicts in non modular scenarios often occur in model, view, and controller files

In an attempt to investigate the overlap of changes to a common slice, and to better understand the results described so far, we manually analyze a number of merge scenarios, especially the ones with both conflicts and modular contributions (change files in disjoint slices), and the ones with no conflicts but also non modular contributions (change files in at least a common slice).

In scenarios with conflicts and modular contributions, we find most conflicts occur in configuration files, and files that define infrastructure or reusable classes that are used across slices. For example, in the Fat Free CRM project[10] both contributions modify the *config/deploy.rb* file, which automates the deployment process. We observe one recurrently affected configuration file is Gemfile.lock. Analyzing individual projects, conflicts in this file range from 0% up to 100% of all conflicting contributions.

[10] fatfreecrm/fat_free_crm; commit 4d1e3e4.

In scenarios with no conflicts but non modular contributions, we often observe each contribution applies only small and non scattered changes to the files that belong to the common slice. For example, again in the Fat Free project[11], one contribution simply deletes one line while the other edits a different and separated line of the same controller method.

Going further, we analyze conflicting scenarios with non modular contributions. In these cases, conflicts could occur not only in the common slice files (model, view, and controller files), but also in reusable or configuration files. To confirm the conclusions so far, we have to make sure conflicts often occur in the slice files. So we ask How frequently do slice files conflict? With further automatic analysis, we find that most scenarios (62.8%) in our aggregated subsample (of conflicting scenarios with non modular contributions) contain at least one slice file with at least one conflict. A per project analysis reveals the same applies for most (77.0%) projects, with rates ranging from 50.0% up to 100.0% of the scenarios in their respective subsamples. Small but significant part of the projects do not contain conflicts in slice files (3.0%), or contain in less than half of the scenarios (13.0%). This outcome brings evidence that not only the chances of merge conflicts is higher when contributions change files from the same slice, but also that overlapping changes in slice files are expressive, reinforcing the relevance of structuring developer tasks in a modular way to reduce merge conflicts.

Most contributions involve changes to more than one MVC module

To assess the potential of obvious alternatives to the promising slice structure we discussed so far, we follow previous work (BALDWIN, 2000) that analyzes task modularity. In such work, the notion of modularity adopted considers the system packages as module unit. Differently, our notion of modularity is based on both files purposes and underlying system directory structure, which we refer in this work as MVC module. Thus, we group files from a Rails project into five different MVC modules: Model (contains all application model files), View (contains all view files), Controller (contains all controller files), Config (contains configuration files containing routes, database schema, logs, Rakefile, Gemfile, etc.), and App (contains global reusable files such as CSS files, templates, static pages, and Javascript files). In this way, we investigate the potential of the MVC module structure for defining tasks and avoiding conflicts. In particular, we first ask How are contributions spread along the MVC modules? We found most contributions (65.3%) affect more than one MVC module (**Mixed**). Moreover, when contributions focus on a single MVC module, the **Config** module is the most affected (23.6%). This suggest that contributions rarely focus on a single MVC module. In the context of the analyzed projects, tasks often have to change more than one module. It is then hard to support parallel tasks that focus on disjoint modules and, therefore, have lower chances of merge conflicts. Artificially changing task focus to support changes to a single module could lead to highly coupled

[11] fatfreecrm/fat_free_crm; commit 72eae62.

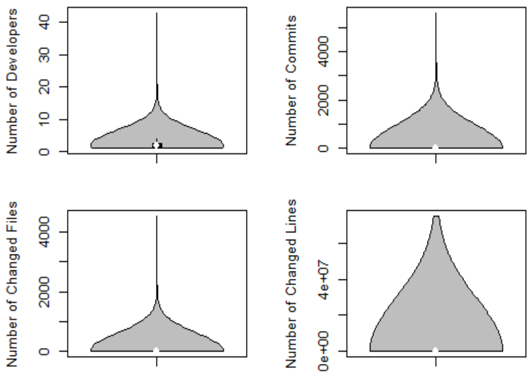


Figure 9 – Size Factors Descriptive Statistics

tasks that compromise parallel development. So aligning task structure with the MVC module structured just explained is not supported by our sample. This partially contrasts with previous work (ORAM; WILSON, 2010), which finds that most changes focus on a single module. However, they have studied three non MVC projects in different programming languages, and they use a finer notion of change.

3.3.2 RQ2: What is the effect of contribution size on merge conflicts?

Going beyond the modularity aspect, we now investigate the effect of the size factors on merge conflicts. Most integrated contributions involve small geometric mean values for the four size metrics, but values vary widely across merge scenarios, as we present in Figure 9. For instance, we observe the number of commits was of 9.5 ± 32.5 (*average* $\pm$ *standard deviation*), and the number of changed files was 16.1 ± 49.1 in the Ruby sample. So, we curate the data and eliminate outliers by converting the size metrics to standardized scores using Z-score (LARSEN; MARX, 2017).

Likelihood of conflict occurrence increases when contributions to be merged have more developers, commits, and changed files

Following common practice on evaluating logistic regression models, like the one adopted by Cataldo and Herbsleb (CATALDO; HERBSLEB, 2011), in the next models, we add the various independent metrics associated with the different research questions. This approach allows us to explore the independent and relative impact of different sets of factors. So, to evaluate the effect of the four *size* metrics on merge conflict occurrence, we explore a number of models by adding the four size factors to the model used in the previous section to evaluate the modularity factor. However, given the presence of continuous variables in the new models, and due to additional assumptions for logistic regression with more than one variable, we first check for collinearity[12] (KUTNER; NACHTSHEIM; NETER, 2004), as it may impair prediction. So, before executing the models, we performed a pair-wise

[12] Collinearity occurs when two or more independent variables are highly correlated.

correlation analysis. This way, we avoid models with high correlation (multicollinearity) among the independent variables.[13] We found most size factors are strongly correlated (Spearman's correlation coefficient greater than 0.6). For example, *number of commits* shows a high level of correlation with *number of developers* ($\rho = 0.73$). For brevity, we leave the details of the collinearity diagnostic to the online appendix (APPENDIX, 2018).

Considering the collinearity results, we then run three new logistic regression models that combine only factors with weak (0.2 to 0.39) and moderate (0.4 to 0.59) correlation. These are Models II, III, and IV in Table 4. Perhaps not surprisingly, number of *developers*, *commits*, and *changed files* are significantly associated with higher probability of merge conflict occurrence. For instance, according to Model II, as the geometric mean of the number of commits in contributions to be merged increases, the chances of merge conflict occurrence also increase 3.55 times (and 2.16 in the Python model omitted here). With lower but still relevant intensity, we observe effects for the other size factors. For example, according to Model III, increasing the geometric mean of the number of changed files also increases the likelihood of merge conflict occurrence by 227% (odds ratio equal to 3.27). For Python, the results follow a similar trend except for *number of changed lines*. By contrast the Ruby results, we find evidence that the number of changed lines has an effect on merge conflict occurrence for our Python sample (odds ratio range from 1.49 up to 1.64, $p < 0.001$). We confirmed this different outcome for the Ruby sample during our manual analyses. We found conflicting scenarios occurred independently of the *number of changed lines* in the Ruby sample.

Note the odds ratio value varies among models depending on the combined factors. This variation also occurs in the percentage of the deviance explained. The size factors are responsible for 3.4% to 8.1% of the deviance in the data (the difference in the deviance explained between model I and each of the other models, II–IV). Modularity and size factors together explain from 14.3% to 19.0% of the deviance. When we look at individual projects, these factors can explain up to 53.5% of the deviance.

Contribution size is not associated with the extent of merge damage

Given the effects of the size factors on *merge conflict occurrence*, we assess whether a similar effect applies to the other conflict metrics. First with *number of merge conflicts*, and then with *number of files with merge conflicts*, we use the Spearman correlation since it is based on rank data and does not assume a linear relationship. Adopting the same correlation threshold used in Section 3.3.1, we observe only weak correlations (0.2 to 0.39). For instance, *number of commits* does not strongly correlate with *number of conflicts* ($\rho = 0.32, p < 2.2e16$) nor with *number of files with conflicts* ($\rho = 0.32, p < 2.2e16$).

[13] During the study execution, we test whether our dependent variables could be related, also performing a logistic regression model. The results showed they are not related ($p > 0.9$).

Our online appendix (APPENDIX, 2018) brings the detailed results; numbers are slightly different after the third decimal.

To better understand the absence of correlation, we individually run the same analysis for each project in our sample. Similarly to the results of the aggregated sample, we also do not observe strong and significant correlations. So no size factor shows a predictive power concerning the number of merge conflicts. The same holds for the number of files with conflicts. Again, whereas contributions size can be used to predict conflict occurrence and the associated damage, it cannot precisely predict the extent of the damage in terms of the exact number of conflicts or files with conflicts. These numbers seem to be much more sensitive to other aspects of the changes in contributions, as explored next.

The size metrics are not definitive

To better understand the issues involved, we conduct a manual analysis of a number of merge scenarios that either support or contradict the discussed results. We then observe conflicting scenarios with a single developer per contribution. What really matters is that both developers simultaneously changed the same method area. For example, one contribution indented the method body while the other edited the same method.[14] When observing non conflicting scenarios with a high number of developers, we find cases with a small number of commonly changed files. In one contribution[15] the changes are small and not scattered, although the other contribution has 26 developers.

Conflicting scenarios with a low number of changed files often had contributions changing the same slice file. For instance, in one merge scenario[16] with on average 3 files, the contributions changed the same view file in adjacent regions. Non conflicting scenarios with a high number of files often had contributions that change different application slices. For example, in one merge scenario[17] with around 37 files per contribution, few files were changed by both contributions. Moreover, the changes were small and not scattered.

When analyzing conflicting scenarios with a low number of changed lines, conflicts often occurred due to changes to configuration and global reusable files. For instance, in one merge scenario,[18] the contributions changed the same Javascript file. In this example, each contribution changed a single line. Non conflicting scenarios with a high number of changed lines usually occurred when contributions changed different slices, with a low number of common files. Moreover, we often observed a high number of added and deleted files increasing the number of lines involved.[19]

[14] For example expertiza/expertiza; commit 96df565.
[15] instructure/canvas-lms; commit e8f15f7.
[16] Katello/katello; commit 902d867.
[17] nasa/earthdata-search; commit e86e243.
[18] nasa/earthdata-search; commit 49fd722.
[19] nasa/earthdata-search; commit 7a25dfc.

3.3.3 RQ3: What is the effect of contribution timing on merge conflicts?

We finally investigate the effect of the timing factors on merge conflicts. Contribution *duration*, and *conclusion delay*, are often small but widely vary, as we present in Figure 10. So, similarly to the size metrics, we also standardized the timing metrics.

Contributions developed over longer periods of time are more likely associated with conflicts

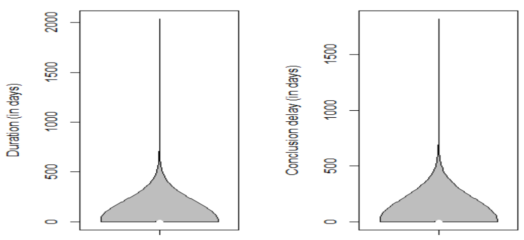


Figure 10 – Timing Factors Descriptive Statistics

We run a collinearity analysis following the same process described in the previous section. The pair-wise correlation analysis shows strong correlation between the *duration* and *number of commits* factors ($\rho = 0.75$ with $p = 0$). By contrast, *conclusion delay* showed weak correlation with all size factors (varying from 0.24 to 0.32, $p = 0$), and moderate correlation with *duration* ($\rho = 0.41$, with $p = 0$).

We then add the *timing* factors to the previous models (II, III, and IV) and run the logistic models V, VI, and VII in Table 4. Our findings show the higher the contribution duration, the higher the chances of merge conflict occurrence (odds ratio vary from 1.04 to 1.09, Models VI and VII, and 1.18 to 1.23 in the Python model omitted here). On the other hand, we found no evidence that the conclusion delay increases merge conflict occurrence ($p > 0.1$, Models V, VI, and VII).

These outcomes show that only one of the timing factors, contribution *duration*, should be considered by managers and development team as a relevant variable to reduce or even avoid merge conflicts. However, its effect is not as relevant as the ones observed for the modularity and size factors.

Contribution duration and conclusion delay are not associated with the extent of merge damage

Conforming to what was observed for the modularity and size factors, we find no effect of the timing factors on the number of conflicts and the number of files of conflicts. Our analysis is identical to the one explained in the previous section.

3.4 IMPLICATIONS

Conflict reduction by defining and allocating tasks

To reduce conflict occurrence, our findings suggest that managers of MVC projects should consider aligning the structure of development tasks with the structure of the associated application MVC slices, and avoid the parallel execution of tasks that focus on common slices. As MVC and agile projects most often have feature or user-story based change requests, the suggested strategy shall bring modularity and productivity benefits. In particular, directly associating requests to tasks, and tasks to slice modules, supports working on requests in parallel with reduced risks of integration problems due to conflict occurrence. This idea applies not only to enhancements and the development of new features but also for bug-fixing tasks since many bug fixes are related to a given feature. This way, based on the feature is possible to infer the corresponding slice.

By contrast, artificially aligning change request and task structure with the structure of the underlying programming language modules, packages, or MVC layers could lead to highly coupled tasks that compromise parallel development. Moreover, this way we would likely have more parallel tasks that affect non disjoint sets of modules, consequently increasing the chances of conflict occurrence. Naturally, change requests that focus on the need to develop an specific component, instead of developing a feature or user story, are more easily aligned with component specific tasks. These, in turn, better align with the structure of language modules. But this kind of change request is not frequent in MVC and agile projects, especially after the initial project phase.

In summary, as MVC projects support two alternative modular structures, managers should define tasks so that they align with the structure they fit best. Managers should also avoid the parallel execution of tasks with either different guiding structures or focus on common modules. Although not enough for eliminating conflicts as initially expected by one of the developers we interviewed, the advice we give might still bring benefits due to conflict reduction.

Conflict prediction models and tools

Our regression models and the associated data collection, processing, and analysis infrastructure we developed for our study could be used to derive more advanced conflict prediction models. Project management and assistive tools could then benefit from these models. In particular, merge conflict prediction models could help managers to better plan and manage development tasks and resources. By monitoring repository information, management tools could support early decisions about when to integrate contributions to mitigate or reduce conflicts. For instance, managers could monitor metrics related to the variables we consider here, with appropriate thresholds derived from our models and live project data, and demand long-living branches, or branches with many developers and commits, to more frequently pull from master, or merge with it.

The manual analyses we performed on conflicting modular contributions reveal that a significant part of the conflicts occur in configuration files, especially *Gemfile.lock* files, which are automatically updated whenever a new gem is installed; these files record the installed gem versions, and keeping them under version control is mandatory practice in many teams. Our analysis also reveals that *Gemfile.lock* conflicts could be automatically resolved by simply discarding local changes of the *Gemfile.lock* with conflicts, that can be done by doing a checkout[20] in such file, and then rebuilding the project based on the local Gemfile file. These steps could be automated, for instance, by defining a custom merge driver[21]. By exploring the structure of other configuration files such as `Gemfile`, we believe further conflict reduction could be achieved by development teams that adopt semi-structured merge tools (CAVALCANTI; BORBA; ACCIOLY, 2017; APEL et al., 2011), since possible conflicting dependency declarations could be treated as different nodes in the merge tree. The same might apply for JavaScript and Cascading Style Sheets files, to name a few that we observed. Adopting advanced merge tools that explore the knowledge derived from our analysis could further reduce the risk of conflicts when developers work on separate slices. Additionally, these findings shed light on a lack of modularity in the Rails framework. The need for changes in common files, despite the modularity existing among developers' tasks concurrently performed, derives implications for the Rails development team. For instance, the developers could develop a new strategy for the configuration of shareable resources in Rails application to improve framework modularity. So, the framework could be less dependent on external strategies for merge conflict reduction, such as those we mentioned before.

Our findings can also drive the development and improvement of awareness and conflict detection tools. Awareness tools like Palantir (SARMA; REDMILES; HOEK, 2012), for instance, could also exploit our results by warning developers of MVC projects when they change the same application slice, instead of just warning when they change the same file. This could encourage necessary coordination, as developers might eventually interfere with each other if they change files from the same slice. Crystal (BRUN et al., 2013), a conflict detection tool, applies speculative merge to automatically and transparently integrate, build, and test contributions in the background so that conflict occurrence can be anticipated, and developers warned accordingly. The merge conflict factors we study here could be used as an initial filter to reduce the effort of Crystal's speculative merging infrastructure.

[20] `git checkout - Gemfile.lock`
[21] <https://git-scm.com/docsgitattributes#_defining_a_custom_merge_driver>

Theories and other studies

Our results bring evidence related to merge conflict reduction hypotheses and advice raised but not empirically evaluated by previous studies (CATALDO; HERBSLEB, 2013; ADAMS; MCINTOSH, 2016; MCKEE et al., 2017), but we also go further by exploring new hypotheses related to contribution modularity and conclusion delay. Current state of the practice merge tools report conflicts when contributions to be integrated change overlapping textual areas of the same file. This fact, however, is not actionable since it is hard to precisely guess which files and specific areas will be changed by performing a programming task, even knowing the developer responsible for the task. It helps, however, to explain our observations, since changing common slices increases the chances of changing the same MVC file, possibly leading to overlapping areas and conflicts. Similarly, the longer the duration and size of contributions to be integrated, often the greater is the area of the program text changed by the contributions, and consequently the greater the changes of overlapping. By contrast, conclusion delay does not increase such area.

Our observations seem to be consistent with the socio-technical theory of coordination (HERBSLEB, 2016) proposed by James Herbsleb, Audris Mockus, and Jeff Robertson. This theory postulates that aligning technical dependencies with coordination activities should lead to higher quality and better productivity. We do not study coordination activities here, but it seems plausible to assume that open-source developers quite often are not co-located, and that they rarely coordinate to avoid merge conflicts. So reducing technical dependencies would also reduce coordination needs, helping to achieve the alignment proposed by the theory. But technical dependence reduction is precisely what we obtain by asking developers to focus on disjoint slices, since developers will less likely work on the same files. Moreover, we observe in our study that the resulting contribution modularity is associated with merge conflict reduction. This, in turn, often leads to productivity by reducing conflict resolution effort, and might lead to better quality by reducing the introduction of bugs when resolving conflicts.

Our results also indicate the need for further studies to investigate the extent of the merge damage, since the investigated factors can predict conflict occurrence and the associated damage, but not the extent of the damage. The value of predicting conflicts is to avoid the effort of conflict resolution, and the negative consequences of not properly fixing the conflict and, consequently, introducing bugs. By predicting the extent of the conflict damage we could give improved advice, under the assumption that if the damage is small, then so is the effort to resolve it. So avoiding conflicts, in case of smaller damage, could not be critical.

Based on our findings and infrastructure, other studies could explore the modularity hypothesis in other domains. For the Android application domain, for example, one could study whether integrating changes to the same Android component would be associated with conflict occurrence. In Section 3.3.1 we explore a modularity notion based on MVC modules and contrast that with the MVC slices results. This notion of module and others based on project directory structure could be used in other domains and languages such as C and Java.

3.5 THREATS TO VALIDITY

Our evaluation naturally leaves open a set of potential threats to validity, which we explain in this section.

3.5.1 Construct

Our motivation partially assumes that contributions correspond to development tasks, but we actually do not check whether the commits in a contribution result from the execution of the same task. For projects that systematically refer to task ids in commit messages, we could proceed with further confirmation. However, most projects we analyze do not conform to that. Nevertheless, contributions as defined here ultimately correspond to code that was independently developed and has to be integrated. If each contribution resulted from the execution of one or more tasks becomes more a question of task granularity.

Similarly, we do not actually assess whether a large number of commits is associated with large or complex tasks. The large number of commits could be due to either large tasks or many small tasks executed in parallel and integrated in a contribution. The same general idea is valid for the other numeric factors. So contributions do not directly correspond to the concept of a developer task, but we believe this only affects one possible interpretation of our results.

The module notion used in this study relies on Frameworks conventions for organizing code. Knowing that any convention can be violated, we do not consider in our sample projects that do not fully comply with basic standard framework conventions, such as having specific directory structures. This way our scripts can identify slices looking for file and directory patterns and naming conventions to identify slices (and their associated files). We also conducted a manual analysis on the scripts results by selecting random merge scenarios to check for script accuracy. Although not observed, our scripts could miss slices that do not follow the assumed patterns.

Our attempt to measure the extent of the merge conflict damage by the number of conflicts and the number of files with conflicts is not accurate because a single conflict might be harder to resolve than a number of simple conflicts. Other metrics should be considered. However, this does not impose risks on our results related to these two dependent variables since none were positive.

3.5.2 Internal

A potential threat to internal validity is the use of public Git repositories for collecting merge scenarios. As these projects might make use of mechanisms such as *rebase*,[22] which rewrites project history (KALLIAMVAKOU et al., 2014; BIRD et al., 2009; LESSENICH et al., 2018), and we do not have access to private repositories in the developers machines, needed to trace and collect all integration scenarios,[23] we might have missed a number of scenarios and conflicts that have not reached the public repositories we analyze. Aranda and Venolia (ARANDA; VENOLIA, 2009) report a similar difficulty in extracting complete information about developer's activities based only on electronic repositories history. As a result, the number of merge scenarios we analyze is a lower bound of the total number of integration scenarios. Although *rebases* performed when accepting a pull request through GitHub GUI might be possible to detect automatically, the GitHub support for integration of pull requests via rebase is relatively new, covering less than one year of commits in the projects in our sample. Only 3.2% of the projects in our sample have a concentration of commits in the last two years of our sample period. The fact remains, however, that the impact of an increased sample on the study results is hard to predict because this mostly depends on the development practices of each project, but we are not aware of factors that could make the missed conflicts different from the ones we analyzed.

Although our analysis does not discriminate between pull request and push scenarios, the use of pull request does not affect our timing metrics since we do not consider the pull request date for computing those metrics. As we explained in Section 3.2.2, the last commit made in the contribution (the one just before merging, which could be the last new commit added as a result of a pull request update) is the one considered to establish the date (*author date*) when a developer contribution was concluded.

3.5.3 External

Our sample contains Ruby and Python MVC projects, which aligns with our purpose of investigating factors that are associated with merge conflicts in the context of web applications developed by using MVC frameworks. The limited scope of our study also aligns with Briand and others (BRIAND et al., 2017) claim regarding the importance of context to Software Engineering research. Although we can not generalize our results for projects that use other frameworks and languages, we highlight the only language and framework dependent part of our infrastructure is the *Modularity Extractor*, given that different programming languages and frameworks differently support modularity. So only this module should be changed to replicate our study with other technologies.

[22] <https://git-scm.com/docs/git-rebase>

[23] *Rebases* performed when accepting a Pull Request through GitHub GUI, might be possible to automatically detect with some heuristics, by relating information in the pull request log with information in the target repository.

Furthermore, we are not aware of any factor that would not make our results generalize to other languages with similar MVC frameworks, but we have no supporting evidence.

3.6 RELATED WORK

Our work is mostly related to the recent empirical study conducted by Lebenich et al. (LESSENICH et al., 2018). In this study, the authors survey 41 developers to understand factors that have the potential to predict the number of merge conflicts resulting from code integration. Like in our work, the authors also conduct an empirical study to investigate potential effects the identified factors might have on merge conflicts. However, we go further by analyzing the effect not only over the number of conflicts, but also over conflict occurrence and the number of files with conflicts. We also investigate a different set of factors, with only two in common, both related to contribution size: number of commits and number of changed lines. Although both works measure the number of changed files, the metrics we use are slightly different since we consider all files changed by at least one of the contributions, while they consider only the files changed by both contributions. Nevertheless, one could argue that the way they compute the number of changed files (changed files in common) generalizes the notion of modularity (Changes to common slice) we use in our study. But that is not the case. Although both factors are similar, they did not find the number of common changed files as a factor with positive predictive power. Besides, the way we compute modularity (*changes to a common slice*) is better than theirs in the sense that it helps project managers before conflicting changes be made. For example, it is easier for a project manager to recommend a given developer A does not change the same slice that a given developer B than recommending developer A does not change the same files that developer B. The second scenario requires developer A to know which files developer B will change during the execution of his task, which is not practical. So, our modularity factor is useful for project planning since it helps task distribution to avoid merge conflicts, contrasting the number of common changed files. This way, *Changes to a common slice* can be seen as a refinement of the changed common files metric.

Thus, we consider *modularity* and *timing* factors, which are not covered by Lebenich et al. The results concerning the just two commonly analyzed factors and the single dependent variable (*Number of merge conflicts*) are compatible. In fact, they do not find any effect between the factors they analyze and the number of conflicts. Our work is the first study that confirms their results. Furthermore, they do not have any results regarding the two other dependent variables we investigate (*Merge conflict occurrence* and *Number of files with merge conflicts*), and thus they do not derive the same conclusions. Our sample has 38 fewer projects, but we analyze more than 3 times (3.42) merge scenarios, that include projects in two languages and MVC frameworks (Rails and Django), whereas their sample focus only on Java projects not limited to the MVC context.

Another related work is the study conducted by Ahmed and others (AHMED et al., 2017).

They investigate the effect of code smells on merge conflicts, and the code smells impact on code quality. They found, among others, conflicting merge scenarios are associated with more code smells than those merge scenarios not involved in a merge conflict. Our study brings evidence of factors related to the modularity, size, and timing of developers contributions that are associated with merge conflicts. As a code smell is an indication of bad design, their study result is in line with ours since some code smells could be related to a modularity problem. However, we investigate a different dimension of modularity (*Changes to a common slice*) while they investigate bad design issues that are more associated with merge conflicts.

McKee and others (MCKEE et al., 2017) investigate developers perceptions about merge conflicts resolution difficulty. They show developers consider the number of conflicting files, conflicting lines, and the size of changes (based on developers perception, not on metrics) as important factors for resolution difficulty. Although we do not investigate merge resolution difficulty, our results show no evidence that the number of files with conflicts correlates with the size of changes.

Nelson and others (NELSON et al., 2019), in a recent study, extended the work discussed in the previous paragraph by presenting a model of developers' processes for managing merge conflicts, based on the developer perceptions. They found, among others, that developers pointed out the complexity of the project structure as one of the perceived factors that affect their ability to resolve the merge conflicts. Furthermore, the dependencies of conflicting code were mentioned by developers as one of the difficulty factors of merge conflicts. As these aspects are related to different dimensions of modularity, that prior study reinforces the importance of our findings since we bring evidence that the modularity of developers contribution is a driving factor for merge conflicts. Besides, we further bring evidence of factors related to the size and timing of developers contributions that are associated with merge conflicts.

Ghiotto and others (GHIOTTO et al., 2018) investigate 2,731 GitHub projects to understand how merge conflicts characteristics affect the strategies developers use to resolve them. They found, among others, that the number of merge conflicts (named number of conflicting chunks in their work) has influential on developers perception about merge difficulty and affects ttheir conflict resolution strategy. Although we do not aim at evaluating the merge difficulty and the strategy developers use to solve conflicts, our study relates to theirs since we also consider the *number of merge conflicts* as one of the conflict characteristics we investigate. However, our interest is to understand how factors of modularity, size, and timing of developers contributions can affect merge conflicts, while they are aiming at understanding how merge conflicts characteristics can affect developers' conflict resolution strategy.

Another empirical study by Cataldo and Herbsleb (CATALDO; HERBSLEB, 2011) investigate the impact of technical and organizational factors on the failure of integration tests in a large-scale project. Although we focus on different metrics, our study complements Cataldo and Herbsleb since we also examine technical and organizational factors. However, we focus on merge conflicts since they happen first.

Accioly et al. (ACCIOLY; BORBA; CAVALCANTI, 2018) performs an empirical study aiming to investigate the structure of code changes that lead to merge conflicts. While they investigate the structure of code changes that lead to merge conflicts, we investigate other technical and organizational contribution factors that are associated with merge conflicts.

Other works propose new tools and strategies to both decrease integration effort and improve correctness during task integration. Cassandra, proposed by Kasi and Sarma (KASI; SARMA, 2013) is a tool that analyzes task constraints to recommend an optimum order of task execution. Although the strategy of recommending an optimum order of task execution per developer helps on task planning aiming at avoiding merge conflicts, it does not prevent developers from making decisions during the task execution that could end up in conflicts.

Palantir (SARMA; REDMILES; HOEK, 2012) informs developers of ongoing parallel changes by looking at the artifacts in the developer's workspace. It monitors the code change status by comparing the local copy of a given artifact (in each developer workspace) to the same corresponding artifact copy in the main repository. This approach raises many false positives and misses important false negatives since they are based on simple factors to predict conflicts, such as the existence of parallel changes made by two developers in the same class. Lighthouse (SILVA et al., 2006), an Eclipse plug-in, also proposes a strategy for improving awareness. However, contrary to Palantir, it applies a concept named *Emerging Design*. In this case, the tool track all changes being made to the code in the developers' workspaces and automatically builds an Emerging Design view, which provides all developers the same view of the current design. This approach can be disturbing for developers since they are enabled to view the whole code design updates instead of a more specific view that could better assist them in preventing or reducing conflicts considering the changes involving their tasks.

Crystal, proposed by Brun et al. (BRUN et al., 2013), proactively integrates commits from developers' repositories with the purpose of warning them if their changes conflict. WeCode (GUIMARãES; SILVA, 2012) and Safe-Commit (WLOKA et al., 2009) employ a similar approach with the difference that they also integrate and run tests considering uncommitted code of developers' local workspace. Such kind of speculative merge, such as the used by Crystal, WeCode, and Safe-Commit, early detects conflicts and consequently reduces resolution effort, despite their solutions rely on the quality of the test suite available for a given project. By contrast, conflict prediction with our metrics would avoid conflicts, and eliminate resolution effort in such cases. On the other hand, prediction might have larger costs associated to false positives and false negatives; speculative merge is free of false negatives, but false positives are reported when at least one of the conflicting changes is temporary and supposed to be reverted before task completion. We are not aware of supporting evidence in favor of one of these approaches. Furthermore, tools like those we cited do not aim at predicting merge conflicts, given that they were developed for awareness purposes during collaborative development.

4 HOW MERGE CONFLICTS HAPPEN AND HOW DEVELOPERS TRY TO AVOID THEM?

Although version control systems (ZIMMERMANN, 2007; KASI; SARMA, 2013; BRUN et al., 2013; PERRY; SIY; VOTTA, 2001) assist developers in performing coordinated changes to software artifacts for supporting collaborative development, and the use of advanced merge tools (APEL et al., 2011; APEL; LESSENICH; LENGAUER, 2012; CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY; BORBA; CAVALCANTI, 2018) that avoid common spurious conflicts, studies show that merge conflicts frequently occur. This impairs developers' productivity, since merging conflicting contributions often is a demanding and error-prone task (BIRD; ZIMMERMANN, 2012; SARMA; REDMILES; HOEK, 2012; MCKEE et al., 2017).

In an attempt to minimize these problems by reducing merge conflicts, in a prior study (DIAS; BORBA; BARRETO, 2020), we investigate how conflict occurrence is affected by technical and organizational factors. By considering 7 modularity, size, and timing factors of developers' contributions, we analyze 73504 merge scenarios from 100 Ruby and 25 Python MVC projects hosted on GitHub. We found evidence that 6 out of 7 factors we investigate are more likely associated with merge conflicts. That prior work helps the development of conflict avoidance strategies (e.g., guidelines, processes, prediction models, tools, etc.) through the learning of modularity, size, and timing factors that cause conflicts.

In the current study, we qualitatively evaluate our findings to better understand possible associated factors we did not consider before and their impact on merge conflicts and go further try to understand different aspects that can help with conflict avoidance. This way, we can refine our results by confronting them with the interviewees' perceptions and better propose strategies to prevent or reduce merge conflicts. First, we try to learn about contexts where merge conflict avoidance is more critical by exploring how frequently merge conflicts occur and in which kind of merge scenarios the conflicts become more complex. This way, we can define a set of factors focusing on those that have more relevance in the conflict occurrence variation among development projects and those more demanding in terms of conflict effort resolution. Then, we try to find out other factors interviewees consider relevant to cause merge conflicts. This way, we can enrich the learning of ways to avoid merge conflicts by deriving a more robust set of conflict factors, and the possible relationships among them. This can help development teams define conflict avoidance strategies more precisely, based on more contextual project characteristics. Additionally, we try to understand which practices developers most use to deal with merge conflicts as another way to learning about how to avoid conflicts. Finally, we also explore the usefulness of a hypothetical tool for alerting merge conflict risk. This way, we can learn more about how tools like that can assist development teams in preventing conflicts.

With that aim, we conducted 16 exploratory semi-structured interviews with practitioners (males and females) across 13 companies, among them 4 global companies and 7 Brazilian companies (among them, 3 hosted at Porto Digital Technological innovation park). The companies belong to private and government organizations, with practitioners with different levels of experience, profiles (most software developers), working on software development projects in varied team sizes.

We found most interviewees perceive an association between merge conflict occurrence and each factor we investigate in our previous study (DIAS; BORBA; BARRETO, 2020). That reinforces the results of our prior study related to the 6 significant factors we observed and raises doubt about the factor (*conclusion delay*) for which we have not found a significant effect in our sample. The interviewees gave us insight into other factors that can influence the modularity, size, and timing of developers' contributions and, in turn, could impact positively or negatively in merge conflict occurrence. For example, there was a minor group of interviewees that do not perceive the association between the conclusion delay of developers' contributions and merge conflict occurrence (a negative impact). For some of them, among others, the potential of long conclusion delays become a factor for merge conflicts (a positive impact) is most related to how frequently developers update their local workspace with the ongoing changes in the master repository.

Overall, we observe 20 merge conflict factors reported by interviewees (5 of them we have already investigate in our prior study). That way, by finding new 16 merge conflict factors that interviewees considered important, we can conduct new quantitative studies and develop more accurate prediction models. We also find 11 practices developers most use to avoid conflicts. For each of these practices, we found at least one merge conflict factor for which a given practice could be helpful. For instance, the practice *Split a big task into smaller independent tasks* helps to avoid or reduce conflicts caused by changes involving big tasks (*Task size* factor). Conversely, we found no practice correspondence for two factors, *Changes in God class* and *Code review timing.* That suggests the need for new studies to explore and define strategies to help the development team with these factors.

Additionally, interviewees report that merge conflicts occurrence varies among different projects. Some of them say that the variation depends on project contextual factors, such as code integration frequency, team size, and task size. Concerning the complexity, interviewees report that most merge conflicts are straightforward to solve. For example, some of them briefly mentioned that the commit message usually informs the changes made in each contribution to be merged, and that makes easy developers understand what is needed to solve the conflict. Others said that as the code is often integrated, the conflicts are usually small. One more classic example of simple conflict involves accepting both conflicting code changes. Developers also report there are cases in which merge conflicts become complex. For example, when involving conflicts caused by global refactorings, mostly because current merge tools lack in assist developers in the resolution of this type of merge conflict. Another mentioned example is when changes are related in non-trivial ways, demanding semantic understanding of the changes. Besides, interviewees also report merge conflict become more complex depending on the project phase.

Finally, most interviewees consider a tool for alerting merge conflict risk useful for distinct purposes. For example, to assist less experienced developers in avoiding conflicts. As another example, a conflict risk alert tool could help the integrator or DevOps to early monitoring conflict issues and warning developers to reduce integration problems. One interviewee gave us insight into contexts in which this type of tool would not apply. These contexts usually involve projects developing under a large codebase, with large team sizes that follow strict policies to avoid conflicts, such as trunk-based development, with continuous integration and modular task distribution.

By finding 15 new merge conflict factors in relation to our previous work and understanding how interviewees perceive their importance, we can conduct new quantitative studies and develop more accurate prediction models. Project management and assistive tools could then benefit from these models that better fit the individual projects' characteristics. Practitioners can benefit from these results by improving the recommendations and guidelines to merge conflict avoidance. Tools builders also could benefit from the results since the practitioners indicate that current merge toolsets lack in assist developers in some types of complex merge conflicts. Also, we provide ideas on the usefulness, and desirable features interviewees suggest for a tool for alerting conflict risk.

The remaining of this paper[1] is organized as follows. Section 4.1 presents our research questions. Section 4.2 describes the research method we use to collect the practitioners' perceptions about the questions. In Sections 4.3 and 4.4, we respectively discuss findings and implications. Section 4.5 presents threats to the validity of our study. Related work is discussed in Section 4.6.

4.1 RESEARCH QUESTIONS

With the motivation presented in the previous section, we present seven research questions that guided our study:

- RQ1: How do practitioners perceive the frequency and complexity of merge conflicts? A number of empirical studies provide evidence on conflict occurrence (ZIMMERMANN, 2007; KASI; SARMA, 2013; BRUN et al., 2013; PERRY; SIY; VOTTA, 2001; CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY; BORBA; CAVALCANTI, 2018; BIRD; ZIMMERMANN, 2012; DIAS; BORBA; BARRETO, 2020; MENS, 2002), but they show large conflict rate variation across projects. So it is important to know if developers perceive such variation and the underlying causes. Others (AHMED et al., 2017; GHIOTTO et al., 2018) investigate varied issues in collaborative development by categorizing

[1] The term paper is used throughout the text since this chapter includes the format and content of the research paper correspondent to the study, as highlighted in Chapter 1.

merge conflicts according to conflict resolution difficulty. To identify factors that could be most related to conflict resolution complexity, we explore how practitioners experience merge conflicts.

- RQ2: How do practitioners perceive the modularity, size, and timing factors?
  We want qualitatively explore how practitioners perceive the modularity (changes to a common slice), size (the number of changed files, number of changed lines, number of developers, and number of commits), and timing (duration and conclusion delay) factors we investigate in the first study (DIAS; BORBA; BARRETO, 2020) and their influence on merge conflict occurrence.

- RQ3: Are there other factors that are associated with merge conflicts?
  Although our previous study considers factors that have been informally associated with conflicts in the literature, it is likely not a comprehensive list of factors. We are not even sure if it includes the most relevant factors. So, to understand if we did not consider factors that are perceived as important by developers, we ask practitioners for factors they consider relevant for causing merge conflicts.

- RQ4: Do practitioners use some practice to avoid or reduce merge conflicts?
  We also want to explore which practices practitioners use to prevent or reduce merge conflicts.

- RQ5: Could a tool to alert merge conflict risk be useful?
  We want to investigate whether a hypothetical tool for alerting of conflict risk based on the factors we investigate could be useful in the context of the practitioners and desirable features.

4.2 RESEARCH METHOD

To answer the research questions we present in the previous section, we carried out semi-structured interviews with 16 project practitioners. Table 5 summarizes practitioners' characteristics.

4.2.1 Participants selection

The interviewees were selected by advertising on social networking sites (Linkedin and Facebook) and by directly contacting project team leaders via email to ask them for sending a message for their teams. As a result, 21 practitioners contacted us interested in participating in our research. We ended up confirming and interviewing 16 practitioners. Following the core principle guidelines for research ethics (STRANDBERG, 2019), we guarantee to keep any sensitive data confidential to researchers, and the right of the

Table 5 – Interviewed practitioners.

ID	Gender	Company	Current Ocupation	Overall Experience	Team size (# developers)
I1	Female	Government	Configuration manager	10 to 15 years	2 to 5
I2	Male	Government	Technical leader	10 to 15 years	10 to 20
I3	Male	Private	Developer	<2 years	10 to 20
I4	Male	Private	Developer/Configuration Manager/ Technical leader/Software Architect	5 to 10 years	2 to 5
I5	Male	Private	Technical leader/DevOps	>15 years	>20
I6	Male	Government	Configuration Manager/DevOps	>15 years	10 to 20
I7	Male	Private	Developer	10 to 15 years	10 to 20
I8	Female	Government	Developer	5 to 10 years	2 to 5
I9	Female	Private	Developer/Software Architect	10 to 15 years	5 to 10
I10	Male	Private	Developer	10 to 15 years	5 to 10
I11	Male	Private	Developer	5 to 10 years	5 to 10
I12	Male	Private	Technical leader/Developer/ Software Architect	5 to 10 years	5 to 10
I13	Male	Private	Developer	5 to 10 years	5 to 10
I14	Male	Private	Developer	<2 years	10 to 20
I15	Male	Private	Technical leader/Developer	5 to 10 years	2 to 5
I16	Male	Private	Developer	5 to 10 years	5 to 10

interviewees to withdraw from the study at any moment (MERRIAM; TISDELL, 2016). Also, the interviewees agreed to the interview's recordings and anonymity by signing an informed consent term. Additionally, we contacted the interviewees in advance by email or videoconference software to schedule the interview and asked them to fill an online demographics form.

4.2.2 Interview Process

The interview script was composed of 14 open-ended questions to explore how practitioners perceive merge conflict factors. The script begins with general questions and moves toward more specific ones, as we made available in (ATTACHMENTS, 2020). We assessed the quality of the interview script by conducting two pilot interviews with two practitioners, not in the interviewees' sample. Based on their feedback, we could identify which questions led to useless data, which questions were not clearly formulated, and which questions, suggested by them, we have thought to include.

Before starting each interview, we explained to the interviewee about the study context and presented the interview structure. Also, before asking about the more specific questions (modularity, size, and timing factors), we explained to the interviewees the definition of each of these factors. Interviews were conducted mostly via Skype or Google Hangouts (68.75%) due to the geographical location of the interviewees. Four interviews we made face-to-face (25%) and another by a phone call (6.25%). All interviews were audio-taped and transcribed, totaling 10 hours and 50 minutes of audio time.

4.2.3 Interviews analysis

We followed the guidelines provided by Saldana (SALDANA, 2015) and Glaser and others (GLASER, 1992) to code, categorize, and synthesize data. We did that in two rounds. In the first, we only considered the transcribed interviews of the first five interviewed. This way, after familiarizing ourselves with the data, by reading the transcripts several times, we began coding the transcripts. The codes arising from each interview were constantly compared to codes of other interviews. This way, we could look for similar codes in the data and grouped them into categories related to the research questions we investigate in this study. We repeated the process through the second round for the remaining interview transcripts. As a result, we derived 390 codes and 15 categories.

Two researchers participated in the analysis process. This way, all analysis steps were executed in pairs, followed by a conflict resolution meeting at the final of each step. A third researcher was also available during the whole process for guidance and discussion of results. Furthermore, when needed, the third researcher was also responsible for driving consensus.

4.3 RESULTS

In this section, we present the main study results organized by the research questions.

4.3.1 How do practitioners perceive the frequency and complexity of merge conflicts?

Merge conflicts are observed by all interviewees, but not in the same way

All interviewees have observed merge conflicts in their development context. But the perception of merge conflict frequency varies, ranging from low to a daily rate frequency. The interviewees that informed that merge conflicts occur but in low frequency, could not accurately state how frequently they occur. For other interviewees, the variation in the merge conflict rate depends on other factors related to the development process. The interviewees' perception of merge conflict frequency is illustrated in Table 6.

For example, one interviewee mentioned that merge conflict frequency is proportional to duration (DIAS; BORBA; BARRETO, 2020) of task development. According to that interviewee, a big task usually increases the chances of a developer working on it for long periods without integrating his changes (pull the other developers' changes from the master branch or push his changes to the master branch), leading to conflicts. So, for this interviewee, task size and duration seem to be directly related factors. Also, the interviewee pointed out that the frequency in which developers integrate their contributions has an

Table 6 – Interviewees' perception of merge conflict frequency.

# interviewees occurrence	Merge conflict frequency
7	Low frequency
3	Monthly
2	Weekly
2	Daily
2	Depends on other factors: Duration, Task size, Integration frequency, Project size, Team size.

impact on conflict frequency, in the sense that low integration frequency is more associated with merge conflict occurrence. Another interviewee reported the merge conflict frequency mainly depends on the project size and the team size.

RQ1 Summary (Merge conflict frequency): All interviewees affirmed merge conflicts occur in their development context. But the perception of merge conflict frequency varies. For two interviewees, the variation in the merge conflict rate depends on factors such as contribution duration, task size, project size, team size, and time pressure.

Most merge conflicts the interviewees face are trivial or straightforward to solve, but there are exceptions

Most interviewees reported that the majority of merge conflicts they face are simple or straightforward to solve. For example, some interviewees briefly mentioned that the commit message usually informs the changes made in each contribution to be merged, and that makes easy developers understand what is needed to solve the conflict. Another said that as the code is often integrated, the conflicting changes are usually small. Besides, the own knowledge in the area of the conflict was also reported as an aspect that simplifies conflict resolution. The two last examples align with what Nelson and others (NELSON et al., 2019) observe in their study in which they investigate, among others, how developers plan, perform, and evaluate merge conflict resolutions. They found that the size of conflicting code is one of the difficulty measures developers take into consideration to decide whether to work on the conflict resolution or defer it. They also found developers consider their own knowledge in the area of the conflict as one of the top factors when estimating the difficulty of merge conflict resolution. Although most interviewees reported that the majority of merge conflicts they face are simple or straightforward to solve, some interviewees gave us insights about scenarios where merge conflict resolution becomes complex, as shown in Table 7.

Table 7 – Insights on when merge conflict resolution becomes complex.

Merge conflict resolution complexity	When involves architectural changes
	When involves Rename refactorings
	When demands understanding the semantics of changes
	When developers postpone the integration of finished tasks
	When developers do not follow recommended branch policies
	Depends on the project phase
	Depends on the contribution duration
	Lack of readability of SCM tools in aiding for merge conflict resolution

For example, according to some interviewees, when the conflict solution demands the understanding of the semantics of the involved changes, merge conflict resolution becomes more complex. Semantic errors usually require developers to perform a deeper and more time-consuming analysis to understand their cause (GUZZI et al., 2015). This perception is aligned with a recent work (AHMED et al., 2017) that classifies as semantic conflicts any merge conflict that affects the programs' functionality (e.g., the variable name changed) to differentiate the difficulty of merge conflicts. Another recent study (GHIOTTO et al., 2018) categorizes the difficulty of merge conflicts in a more fine-grained level based on the choices a developer makes in resolving a kind of conflict. This way, they provide a more detailed insight on, among others, the complexity of semantic conflicts, for instance, a resolution involving the choice of one of the conflict versions is less complex than a resolution involving a combination of two conflicting versions.

Interviewees also reported that merge complexity increases when developers do not follow the recommended branch policies. For instance, a interviewee mentioned that conflicts get worse if, for instance, developers create new branches starting from a wrong revision (e.g., of some parent branch behind in time). Although branch-based development allows developers to focus on their tasks without worrying about being affected by the other parallel developers' changes, the effort involved in integrating the developers' branch is usually dependent on how much work went on in the branches to be merged (BIRD; ZIMMERMANN; TETEREV, 2011). So, keeping branches short-lived and merge often reduces the size of developers' contributions. In turn, small contributions reduce the contribution duration (DIAS; BORBA; BARRETO, 2020) and, this way, the period during which other developers could have made conflicting code changes (ADAMS; MCINTOSH, 2016).

For some interviewees, some types of code changes are more associated with merge conflict complexity. For instance, a interviewee reported that when the conflicting code involves architectural changes (e.g., some component rearchitecting), conflict resolution becomes more complex. As changes related to software architecture usually crosscutting many files, this could increase not only the likelihood of overlapping changes but also the extent of conflicting points in terms of the number conflicting files and code regions involved in each file involved in conflicts. In turn, this could lead to more complex conflicts (GHIOTTO et al., 2018).

Another interviewee reported another example of a type of code change more associated with merge conflict complexity. The interviewee reported that merge conflict complexity increases when the conflicting code involves some types of code refactorings (e.g., change package, rename class). This perception reinforces a recent work (MAHMOUDI; NADI; TSANTALIS, 2019) that is the first to empirically observes that conflicts involving code refactoring are usually more complex, compared to conflicts with no refactoring changes. Additionally, according to interviewee perception, this type of conflict often gets worse when inside a given file involved in the conflict, already has a merge conflict to solve. Moreover, that interviewee also considers SCM tools lack in provide a more supportive aid for this type of merge conflict resolution.

Interviewees also reported that the period a task takes to be developed is a driver for increased merge conflict complexity. According to those interviewees, the longer the code takes to be developed without integrated their changes, the more the chances of many changes be made in parallel that will be integrated, increasing merge complexity. So, for those interviewees, long contributions duration (DIAS; BORBA; BARRETO, 2020) are more associated with merge conflict complexity.

The interviewees perceive that, when developers postpone code integration of finished tasks, it increases the merge conflict complexity. Although one could think in a variety of reasons for that (e.g., change in the priority of which features will be delivered to the client), it gets worse when it happens in a non-planned way. For instance, merge conflicts become complex when a developer misses integrating his finished task, for no reason, and removes the task from the to-do task list, as mentioned by the interviewee. This way, other developers can inadvertently make changes in files corresponding to the same feature, assuming no ongoing task is changing common files. Besides, the longer a postponed task takes to be integrated into the mainline development, the higher the merge complexity due to the main branch has advanced in time with other developers' changes. Although in our previous work (DIAS; BORBA; BARRETO, 2020), we do not investigate merge conflict complexity, we did not observe the conclusion delay of developers' contributions as associated with a merge conflict, as we before mentioned (see Section 4.3.1). However, we now have the insight, as we present in Section 4.3.2, that this could be influenced by other factors, such as the frequency developers update their local workspace with other developers' changes.

To conclude, the stage of the project is also perceived as a factor that influences merge conflict complexity. Depending on the project phase, for instance, in the final phase of delivery, more parallel changes are usually made, increasing the chances of getting a higher number of inconsistent changes and so the merge conflict complexity (GHIOTTO et al., 2018). Although this perception seems obvious, prior work (LESSENICH et al., 2018) found no evidence that the commit density (measured as the number of commits in the last week of development of a branch) is associated with the number of merge conflicts. This result sheds light on the need for defining better metrics to assess how merge conflicts occur depending on the project phase since, according to the interviewee perception, the stage of the project influences merge conflict complexity.

RQ1 Summary (Merge conflict complexity): Most merge conflicts the interviewees face are trivial or straightforward to solve, but there are exceptions. Eight interviewees highlighted scenarios when conflict resolution gets complex. Such scenarios usually involve conflict resolution that demands to understand the semantics of the changes, when the recommended branch policies are not followed by developers, when the resolution involves architectural changes, or in cases of Rename refactorings. Also, the lack of readability of SCM tools in aiding for a resolution involving some types of conflicts, such as global refactorings, was mentioned. Besides, longer contribution duration, the postponement of the integration of finished tasks, and the project phase may contribute to increasing the merge conflict resolution complexity, according to tho these interviewees.

4.3.2 How do practitioners perceive the modularity, size, and timing factors?

Most interviewees observe changes to a common slice as associated with merge conflict occurrence

As we want to explore how the interviewees perceive the modularity factor *changes to a common slice* that we investigate in our prior study (DIAS; BORBA; BARRETO, 2020), we only asked that specific question for those interviewees that reported to align their development tasks conform to the modularity definition we use in this study. So, the results presented in this section consider the answers of 10 out of 16 interviewees.

The interviewees' perception confirms the result of our previous study (DIAS; BORBA; BARRETO, 2020), in which we analyzed 125 Github repositories and found that the likelihood of conflict occurrence significantly increases when contributions to be merged are not modular in the sense that they involve files from the same MVC slice. Besides, we also found that the layer division task seems to be more conflict-prone compared to the MVC division task (vertical slices), as reported by one interviewee as follows:

"*In verticals slices, we practically do not have merge conflicts. For instance, we have new modules that are short-duration projects. For these projects, we try not to use the layer model to distribute tasks and, when we use the vertical model, we do not have conflicts.*" (I4_58, Developer/Configuration Manager/Technical leader/Software Architect)

Three interviewees confirm another result of our previous study, in which we found that conflicts occur even when merging modular contributions and that they are caused because of parallel changes to files that are not part of the slice structure; this includes configuration files and files that define classes reused across slices. For instance, as reported by one of the interviewees:

"*[...] lately we are developing an Angular application. So, there is a file that must be updated whenever you create a new module. For instance, you have to add the module to that file and to configure the route to the corresponding page. So, the conflicts often occur in that file because everybody needs to make changes to it.*" (I4_58, Developer/Configuration Manager/Technical leader/Software Architect)

Most interviewees observe the number of changed files as associated with merge conflict occurrence

Most interviewees (14 out of 16) reported they observe that contributions with many changed files are more likely associated with merge conflict occurrence, such as the reported by one of these interviewees that:

"*Yes. I think that the more the number of files, the more likely the conflict chance. I am an anguished person. If I made a functionality that I presume I finished, I integrate the changes right away. So I liberate myself of any responsibility.*" (I8_26, Developer)

Some of these interviewees gave us insights on confounding factors that can influence the number of changed files in a contribution and, in turn, could affect positively or negatively the merge conflict occurrence, as illustrated in Table 8.

Table 8 – Confounding factors that can influence the number of changed files in contributions.

Factor	**Confounding factors**
Number of Changed Files	Code change Nature (bug fix, new feature, refactoring, etc.)
	Task size
	Duration
	Integration Frequency

According to interviewees' perception, the type of code changes can influence the number of files; for instance, a new feature development often results in contributions with more changed files (mostly new files) than a bug fixing, and the integration of new features hardly result in conflicts. So, the nature of code changes can result in a higher number of changed files in developers' contributions without necessarily lead to merge conflicts. Conversely, crosscutting changes such as the resulting from some types of code refactorings usually involve changes in many files, and this code change nature increases the likelihood of merge conflict occurrence (MAHMOUDI; NADI; TSANTALIS, 2019).

The interviewees also perceive that when the code integration that resulted in conflicts involves too many changed files, typically, the contribution was developed during a long period without being merged (pull the other developers' changes or push their changes to the main branch). So, the task size, contribution duration, and the frequency of developers integrate their contributions can influence the number of changed files in contributions and, in turn, its impact on merge conflicts. However, it is worth noting that the relationship between the number of changed files in contributions, and these three confounding factors are not always directly proportional. Again, we can use the nature of code changes as the basis for that discussion. One could be involved in a task that took long to be developed because the developer needed to spend more time thinking in the solution than changing the code. In that scenario, the code changes could be affected only one file (or few files) and also no merge conflicts during its integration. In other words, a long contribution duration not necessarily results in many changed files. As an additional example, if the parallel tasks under development do not have technical dependencies among them, the number of changed files hardly would impact on merge conflicts and, this way, how long they take to be developed and how often they are integrated is not an issue that could lead to merge conflicts. So, the relationship among these factors can go in both ways (direct and inverse association) and impacts positively or negatively in merge conflicts, depending on the code change nature.

To conclude, not surprisingly, interviewees observe an inverse association between the factors duration and integration frequency in the sense that the lower the integration frequency, the higher the duration of a given contribution. Also, the interviewees' perception confirmed another evidence we bring in our previous study (DIAS; BORBA; BARRETO, 2020) with concerning the positive relationship we found between the number of changed files and contribution duration, the reason of why they are not put together in the same model we assess due to be confounding factors. Besides, we now have a better understanding related to the results of the manual analysis we conducted in the previous study, in which we found clean merge scenarios with too many changed files and also conflicting scenarios with only one or a few files, that represent scenarios that contrary the main result. Based on the interviewees' perception, we now understand the reason why this happen and alert us to the need for considering the observed confounding factors when investigating merge conflicts.

As a final remark, two interviewees do not observe the number of changed files as associated with merge conflict occurrence. In an in-depth analysis of their development context, we found they both are from the same company and have a structure of development tasks quite isolated, based on MVC slices. Besides, development teams use mechanisms (e.g., JIRA Tickets) for helping team awareness of the ongoing changes.

Most interviewees observe the number of changed lines as associated with merge conflict occurrence

Most interviewees (14 out of 16) reported they observe that contributions with many changed lines are more likely associated with merge conflict occurrence, such as the reported by one of them that:

"*When I see a lot of changes, and even some refactorings, those end up generate more conflicts because they change more code lines. The LOC is bigger.*" (I2_46, Technical leader)

Some of these interviewees gave us insights on confounding factors that can influence the number of changed lines in a contribution and, in turn, could affect positively or negatively the merge conflict occurrence, as illustrated in Table 9.

Table 9 – Confounding factors that can influence the number of changed lines in contributions.

Factor	Confounding factors
Number of Changed Lines	Code change nature
	Number of changed files
	Task size
	Duration
	Changes in God Class

For some interviewees, global non-semantic changes can influence the number of changed lines in a contribution. For instance, if a given developer inadvertently resolves to format an entire class, this ends up in a contribution with too many changed lines not related to the target task. According to the interviewee, this type of changes often resulting in contributions with more changed lines that leads to spurious (and timing-consuming) conflicts, since the conflicts are not related to the developed task.

According to the interviewees, the number of changed lines in contributions can also be influenced by other types of code change nature, for the same reasons we already presented in the previous section (see Section 4.3.2). For instance, changes made in existing code (e.g., bug fixing) often result in a few changed lines compared to changes involving new code (e.g., new feature development). The interviewees also gave us a more in-depth vision of the code change nature from the perspective of change location in the sense that scattered changes throughout different locations in the file are more conflict-prone than those that involve contiguous lines. This information confirms the results of the manual analyses we conducted in our previous work (DIAS; BORBA; BARRETO, 2020), in which we found that in clean merge scenarios involving not modular contributions, the developers' changes usually affected not scattered lines. Also, another result is concerning the relationship between the number of changed files and the number of changed lines. According to interviewees, the number of changed lines is inversely proportional to the number of changed files in the sense that, in contributions involving too many files, the changes usually affect a few code lines in each file. We observe the same, during the manual analysis we conducted in the previous study to better understand the characteristics of some clean scenarios with a high number of changed files.

Another confounding factor reported as associated with the number of changed lines concerning the size of developers' tasks. According to the interviewee, if a developer needs to add a lot of code into an existing file, his task is probably big, and that more likely increases the chances of other developers making overlapping changes to that file. Conversely, if a developer adding a lot of code in a new file, the task size not necessarily become an issue and so less likely the merge conflict occurrence.

Another interviewee reported that he observes that changes in God class (FOWLER et al., 2012) usually lead to merge conflicts. This result confirms the outcomes of a recent study (AHMED et al., 2017) that found changes involving God class is one of the three code smells more strongly associated with merge conflicts. As a God class concentrates many functionalities shared by other classes, other parallel developers changes have a high likelihood of touching the God class, and this can lead to merge conflicts.

To conclude the list of confounding factors related to the number of changed lines, the contribution duration was mentioned for some interviewees for the same reason we presented in the previous section (Section 4.3.2). According to the interviewees, conflicts involving too many changed lines is often a symptom that the contribution was developed during a long period without being merged (pull the other developers' changes or push their changes to the main branch), leading to merge conflicts.

As a final remark, one interviewee said he does not observe that the number of changed lines influences the merge conflict occurrence due to the team developer follow the recommended guidelines to prevent interferences among tasks, such as modular task distribution.

Most interviewees observe the number of developers in contributions to be merged as associated with merge conflict occurrence

The number of developers in contributions to be integrated is perceived by most interviewees (13 out of 16) as a factor that influences merge conflict occurrence. Some of them highlighted other factors that could influence the number of developers and could impact positively or negatively the merge conflict occurrence, as illustrated in Table 10

Table 10 – Confounding factors that can influence the number of developers in contributions.

Factor	Confounding factors
Number of Developers	Integration frequency
	Project size
	Team size
	Communication
	Inter-team development
	Task division strategy
	Software architecture
	Branch policy

According to interviewees, the frequency in which a developer integrate changes (from or into the mainline branch) is inversely proportional to the number of other developers' contributions they will get during his contribution integration. This happens since low integration frequency more likely increases the number of developers' contributions made in the meantime and, in turn, also more chance of overlapping changes. Moreover, this scenario gets worse in projects with large team sizes since there are a high number of developers contributing to the code, and this increases the conflict chance.

We also observed interviewees perceive that the size of the projects can influence the number of developers in contributions to be merged in the sense that in large projects, developers often have more flexibility to choose to make parallel changes that affect different regions of the codebase.

Other interviewees perceive team communication, more specifically the lack of communication, as a factor that could influence the number of developers in contributions to be integrated. It makes sense because team communication can be critical in case of changes that could affect many different code regions (e.g., some types of code refactorings (MAHMOUDI; NADI; TSANTALIS, 2019)) and, in turn, might affect other developers' contributions. This gets worse, depending on the team size. According to interviewees' perception, the bigger the number of developers involved in projects, the higher the difficulty of communication among them.

Inter-team development is another mentioned factor that can influence the number of developers. According to interviewees' perception, when more than one team is changing the same codebase, this often increases the number of developers involved in contributions that affect common code regions. Also, in this kind of context, especially if the teams are not co-located, communication becomes more difficult.

The task division strategy adopted by the teams is also mentioned as a factor that could influence the number of developers. For example, the interviewee perceives that projects involving the layer task division often involve more developers in contributions to be merged than projects using MVC slice task division. It makes sense, since, layer task division could increase the chances of more developers making parallel changes to the same feature and, in turn, the changes could affect common files, increasing the likelihood of overlapping changes. A more in-depth analysis, which was confirmed by the interviewee, showed us that the interviewee has a specific context in the sense that the projects involving layer task division have more developers in the team than the projects involving the MVC layer task division. So, a higher merge conflict occurrence in layers tasks maybe being influenced not only by the adopted task division strategy but also the team size (as a confounding factor to the task division strategy), since more developers in a team often increase the chances of more parallel overlapping changes. This way, the influence of the team size in the number of developers involved in contributions to be merged can play differently, depending on the adopted task division strategy. But this needs further investigation.

The software architecture influences the number of developers involved in contributions in the sense that it drives the adopted task division strategy. So, depending on the architecture, the developers' task can be distributed in a modular way to prevent or, at least, reduce interferences. Still, according to developer perception, a well-defined architecture helps developers to identify the impact of changes during merge conflict resolution fast. In any case, the number of developers involved in contributions to be merged does not become an issue when the software architecture helps with modular task division, according to the interviewee.

Also, a interviewee mentioned that the conflict influence due to the number of developers in contributions to be merged is mostly dependent on whether the team follows the recommended branch policies. For example, if developers create new branches starting from a wrong revision (e.g., of some parent branch behind in time), they will get a lot of other developers' contributions while integrating his code.

To conclude, only one interviewee reported does not observe the influence of the number of developers in merge conflicts. According to her perception, as she works in a small team (four developers), the number of developers contributing in parallel is also low. It is worth noting; however, we observe in a more in-depth analysis that they work based on modular tasks that are quite isolated from each other, and all developers often integrate their changes aiming at avoiding conflicts. Another one did not know how to answer the question because they usually are not aware of how many contributions of different developers his changes need to be merged with while integrating his contributions. We missed questioning one interviewee.

Half of the interviewees observe the number of commits as associated with merge conflict occurrence

The number of commits is observed by half of the interviewees (8 out of 16) as associated with merge conflict occurrence, as mentioned by one of them as following:

"*[...] the number of commits is a problem because you always have to think in merges and rollbacks. If you do a lot of unnecessary commits, any merge or rollback is much more complicated. The ideal solution, which not always happen, is to do atomic commits. [...] So, contributions with many commits are complicated, especially if you have many developers changing the code.*" (I6_90, Configuration Manager/DevOps)

This perception is an interesting finding because it confirms the result in our previous work (DIAS; BORBA; BARRETO, 2020), in which we found a strong positive correlation between the number of commits and the number of developers involved in contributions.

Additional factors (see Table 11) that could influence the number of commits in contributions and, this way, its effect on merge conflicts, were already mentioned as confounding factors in previous sections: the contribution duration and code integration frequency. According to developers' interviewees, the number of commits in a contribution increases as the time spent developing a given task without integrating (from or into the main branch) the changes also increases. In our previous work, we also found a strong positive correlation between the number os commits and contribution duration factors.

Table 11 – Confounding factors that can influence the number of commits in contributions.

Factor	**Confounding factors**
Number of Commits	Duration
	Integration frequency
	Number of developers

To conclude, six interviewees do not observe the influence of the number of commits in contributions and merge conflicts. According to these interviewees, in the projects they participate in, developers are warned for making atomic commits.

Most interviewees observe the contribution Duration as associated with merge conflict occurrence

Most interviewees (13 out of 16) answered they observe that contributions with many days influence conflict occurrence. Some of them gave us insights on associated factors that can affect the contribution duration timing and, this way could contribute positively or negatively to merge conflict occurrence, as shown in Table 12)

Table 12 – Confounding factors that can influence the contribution duration.

Factor	Confounding factors
Duration	Code change nature
	Change Size
	Task distribution

For instance, according to interviewee perception, contributions developed over longer periods of time is not necessarily a problem because it mostly depends on the type of code changes (code change nature) involved. For example, if there are technical dependencies among the contributions to be integrated, he mentioned that, in this case, task synchronization is what would be made the difference to avoid conflicts. That information seems to be consistent with the socio-technical theory of coordination (HERBSLEB, 2016) proposed by James Herbsleb, Audris Mockus, and Jeff Robertson. This theory postulates that aligning technical dependencies with coordination, activities should lead to higher quality and better productivity. So reducing technical dependencies would also reduce coordination needs, helping to achieve the alignment proposed by the theory. But technical dependence reduction is precisely what we obtain by asking developers to focus on disjoint slices since developers will less likely work on the same files. This way, task distribution has also an impact on the effects of contribution duration in merge conflict occurrence.

Two interviewees did not observe the contribution duration as a factor associated with merge conflicts. However, one of them reinforces the discussion we present in a previous section (see Section 4.3.2) that the size of changes is a factor that must be considered when investigating whether contribution duration has a negative or positive impact on merge conflicts, as follows:

"*[...] I can spend eight hours a day, as sometimes happens, to solve a single function that ends up with only two lines of code changed, because I've spent all day thinking over the solution. So it is more about the amount of change you do than the time you spent doing it.*" (I3_93, Developer)

In a more in-depth analysis, the two interviewees that do not perceive the contribution duration as associated with merge conflicts work on modular tasks that are planned to prevent conflicts. So, the contribution duration does not become an issue for them. As a final remark, another interviewee did not know how to answer the question.

Most interviewees observe the Conclusion Delay (of contributions to be integrated) as associated with merge conflict occurrence

Most interviewees (11 out of 16) observe that contributions with longer conclusion delays (DIAS; BORBA; BARRETO, 2020) impacts on merge conflict occurrence. We discuss some of the reasons (see Section 4.3.1) developers postpone the integration of their contributions for long periods, increasing their conclusion delay. For instance, a longer conclusion delay can occur due to a task priority change. Or, as reported by one interviewee, a given developer could intentionally postpone integrating his changes to avoid dealing with conflicts in the case of large contributions, for instance, the development of a big feature as following:

"*[...] I usually intentionally held to integrate the changes because I knew that the headache I will face to have to make these changes constantly. The sum of them will be bigger than if I expected at the end to integrate my changes. So that was not unusual.*" (I11_41, Developer)

Some interviewees gave us insights on associated factors that can increase the conclusion delay, and this way could contribute to merge conflict occurrence, as shown in Table 13.

Table 13 – Confounding factors that can influence the conclusion delay of contributions.

Factor	**Confounding factors**
Conclusion Delay	Change size
	Code review timing
	Integration Frequency

For the same reason we discuss in the previous section, according to interviewees' perception, the size of the changes is an associated factor that could influence positively or negatively the likelihood of merge conflicts due to the conclusion delay of contributions. Moreover, the potential of long conclusion delays become a factor for merge conflicts is most related to the frequency developers update their local workspace with the ongoing changes in the master repository. So, the integration frequency is a factor associated with conclusion delay that also could positively or negatively impacts the likelihood of merge conflict occurrence. Yet the code review timing was also mentioned as a factor associated with longer conclusion delays, as he reported:

"*[...] sometimes, the code review process takes longer or is not prioritized, depending on the project phase or daily demands [...], and that usually ends up in conflicts since, at the time when the contribution is integrated, the main branch has already advanced a lot. So, I think the conclusion delay must be taken into consideration by the code review process because the longer it takes to integrate what you have developed to the main branch,* *the more potential for generating conflict.*" (I15_21, Technical leader/Developer)

To conclude, two interviewees do not observe the conclusion delay as a factor associated with conflict occurrence. This result is aligned with our findings in the prior study (DIAS; BORBA; BARRETO, 2020). According to the interviewee, there is no long conclusion delay between developers' contributions given the developers do not postpone the integration of finished tasks. The other interviewee attributed to the modular task division among developers and the team awareness of ongoing changes the reasons why he does not observe the conclusion delay as a factor for merge conflict.

Based on that information, we now have a better understanding of why the perceptions of most interviewees diverge from the result of our prior study regarding the conclusion delay factor. Some projects' contextual characteristics seem directly affects the impact of the conclusion delay of contributions on merge conflict. But, a more in-depth investigation is needed to conclude that.

Most of our finding in the prior study is aligned with the interviewees' perceptions. However, this second study allows us to refine our results by understanding the confounding factors related to modularity, size, and timing factors and the effect in merge conflict occurrence depending on a given combination between these factors. To the best of our knowledge, this is the first study that brings empirical evidence on confounding factors related to modularity, size, and timing of developers' contributions and their possible impact on merge conflict occurrence. We summarize the derived set of confounding factors in Table 14.

Table 14 – Modularity, size, and timing confounding factors.

Merge conflict Factors	**Confounding factors**
Modularity factor	
Changes to a common slice	-
Size factors	
Number of developers	Communication, Integration frequency, Project size, Team size, Inter-team development, Branch policy, Software architecture, and Task division strategy
Number of commits	Integration frequency, Duration, and Number of developers
Number of changed files	Task size, Integration frequency, Duration, and Code change nature
Number of changed lines	Task size, Duration, Code change nature, Number of changed files, and Changes in God Class
Timing factors	
Duration	Code change nature, Change size, Task distribution, Number of changed files, Number of changed lines, and Number of commits
Conclusion delay	Integration frequency, Change size, and Code review timing

RQ2 Summary: Most interviewees observe that the modularity (*changes to a common slice*), size (*number of developers, commits, changed files*, and *changed lines*), and timing (*duration* and *conclusion delay*) factors as associated with merge conflict occurrence. That confirms the majority of the findings of the first quantitative study we conducted, except for the *conclusion delay* and *number of changed lines* factors. In the earlier study, we do not observe the contribution *conclusion delay* as associated with the merge conflict occurrence in our sample. The same applies tho the *number of changed lines*, except for the python sample. The interviewees also reported a set of confounding factors that can influence how the size and timing factors can impact on merge conflict occurrence. For instance, for the *number of changed lines* factor, some interviewees observe that the type of impact (positive or negative) of this factor on merge conflict occurrence depends on other factors, such as the task size, and the code change nature, to mention a few.

4.3.3 Are there other factors that are associated with merge conflicts?

We go further to get more detailed information and asked interviewees about the factors they consider relevant to cause merge conflicts. Interviewees mentioned 12 factors as the top causes of merge conflicts. Some interviewees had already reported most factors (10 out of 12) as confounding factors while answering about the size and timing metrics (RQ2). This way, we highlight in this section only the two other new factors identified.

According to one interviewee, depending on the *project phase*, merge conflict occur more frequently. As an example, he mentioned that in the early stages of the project, when the team is composed, and nobody has the system domain knowledge, for instance, in projects developed from scratch, the chance of conflicts increases. That makes sense, given that in the very initial stages of software development can be hard to predict how a given task could impact another without the domain knowledge of the system. The same could happen, for instance, if the team does not know enough about the software architecture used in the project. It is worth noting that in Section 4.3.1, another interviewee mentioned the project phase as a factor associated with merge conflict complexity but related to a different stage, concerning the approaching of deadlines. Another interviewee also mentioned the lack of awareness of other developers' changes as a factor that influences merge conflict occurrence. When the developers do not know how the ongoing tasks can interfere with each other, this could increase the chances of overlapping changes and therefore merge conflict can occur.

Among the remaining ten merge conflict factors observed by interviewees, it is worth notice that the contribution *conclusion delay* was also mentioned by one interviewee. We first investigate this factor in our quantitative study (DIAS; BORBA; BARRETO, 2020) and did not find evidence of its association with merge conflict occurrence, as we present in Chapter 3. However, we can now refine our initial understanding of the impact of the *conclusin delay* in merge conflicts, since based on the interviewees' perceptions, there are cases in which this factor has a positive effect on conflict occurrence. According to that interviewee, sometimes, for instance, due to task priority change, the code integration of a given task that was already underway in a given developer branch is postponed. In that case, when the time to integrate that postponed task arrives, conflicts usually happen since the master branch has already advanced in time with other developers' changes.

The other nine merge conflict factors mentioned by the interviews were: Task size, Integration frequency, Project size, Tem size, Code change nature, Communication, Inter-team development, Task distribution, and Software Architecture. All these factors and the reasons why some interviewees observe their association with merge conflict occurrence also were discussed throughout Section 4.3.2, the reason why we do not present them in this section.

Considering the confounding factors found and presented in the previous section, we can identify 20 merge conflict factors that are considered relevant by developers as associated with merge conflict occurrence. Among those, we already investigate 5 of them in our first study (DIAS; BORBA; BARRETO, 2020) presented in the previous chapter. We illustrate this comparison and the common factors between the two studies in Figure 11. This way, based on this second qualitative study, we identified 15 new merge conflict factors that the interviewees consider more associated with merge conflict occurrence.

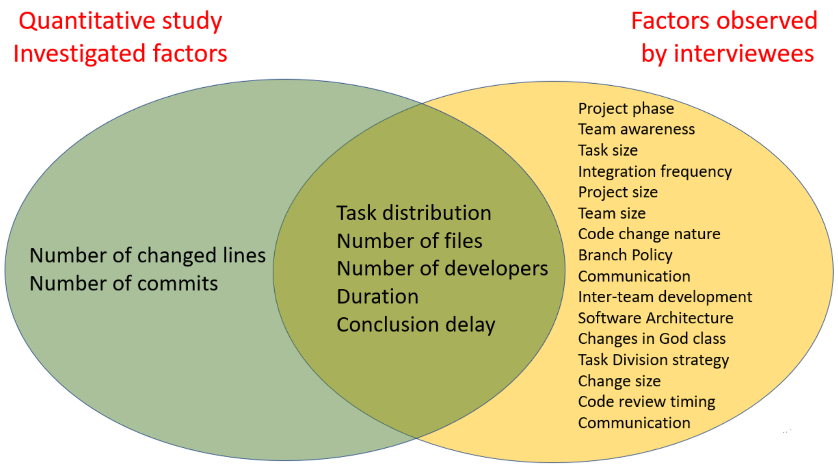


Figure 11 – Merge conflict factors comparison between quantitative and qualitative studies

As a final remark, interviewees reported more 5 factors they consider relevant for causing merge conflicts, as follows: such as Developer Experience, Developer Training, Task Cohesion, High daily task flow, and Programming language characteristics. However, based on what the interviewees report, they refer to indirect factors that lead to more code generation (more changed lines and/ou changed files). For instance, some interviewees consider the developer seniority as associated with merge conflict occurrence because, according to their perception, more experienced developers usually generate less code while developing a task compared to the less experienced ones. In other words, less experienced developers generate more code, and that increases the chances of other developers making changes in common code regions (files, lines). So, we can not consider Developer experience and the remaining 4 factors as merge conflict factors, but possible reasons for an increase in the size of developers' contributions in terms of the number of changed lines and/or changed files. Size factors, including the number of change files and changed lines, we investigate in our previous study (DIAS; BORBA; BARRETO, 2020) and present further results in Section 4.3.2.

RQ3 Summary: Overall, we identified two new factors the interviewees observe as associated with merge conflict occurrence: the *project phase* and *team awareness*. The remaining ten factors reported by the interviewees had already been mentioned as confounding factors by others (or were reinforced while answering this question by the same interviewee), related to size and timing metrics (RQ2).

4.3.4 Do practitioners use some practice or strategy to avoid or reduce merge conflicts?

We found 11 practices interviewees reported they use to avoid or reduce merge conflicts. The most-reported practice used by the interviewees is to perform code integration often (pull/push) (BIRD; ZIMMERMANN, 2012). The frequency established to code integration varies among interviewees. For 12 out of 16 interviewees stated they observe that integrate the code often reduces the potential for merge conflicts. This observation aligns with some industry recommending rule of thumbs (ADAMS; MCINTOSH, 2016) for dealing with code integration problems since periodic code integration increases team awareness by making ongoing changes to the system available (CATALDO et al., 2007).

As a subcategory of the *Integrate often* practice, two interviewees also reported that continuous integration (CI) (ADAMS; MCINTOSH, 2016) helps to reduce both the potential for conflicts and their complexity. By adopting CI, developers are frequently encouraged to update their local workspace with the changes made by other developers, by performing a *pull* operation, before starting the development of a new task. After finish a given task, they made the corresponding changes available to others by performing a push operation to the main remote repository (master branch). This process more likely increases the chances of a given developer receiving the changes made by others in the files he will change, before starting their task, which can prevent merge conflicts. Besides, as developers often integrate their changes (pull/push often), any merge conflict can be early detected and can be solved on time when the conflicting changes are small and still fresh in the developers' minds before it becomes more complex. We highlight that although only two interviewees mentioned using continuous integration as a practice to reduce merge conflicts, all interviewees reported the adoption of CI process in their development projects.

The second most reported practice is the use of modular task distribution to avoid merge conflicts. 10 out of 16 interviewees informed the way how developers' tasks are developed play an important role in avoiding merge conflicts (DIAS; BORBA; BARRETO, 2020; PARNAS, 1972; BALDWIN, 2000; CATALDO; HERBSLEB, 2013). So, whenever it is possible, the task distribution focus on assuring that developers make parallel changes on disjoint modules to avoid merge conflicts.

To follow the recommended branch policies appears in the third position among the most mentioned practice. Branch-based development (BIRD; ZIMMERMANN, 2012; ADAMS; MCINTOSH, 2016) is the branch model used for most interviewees. According to 3 out of 16 interviewees, the developers are warned to adopt the defined branch merging policies accordingly. As an example, as one said, at the beginning of a sprint, all developer branches are created starting from the same revision of some parent branch (e.g., master branch) to grant merge conflicts reduction either occurrence or complexity. In that case, potential merge conflicts will encompass only those regarding the new contributions generated in the current sprint.

Another practice that is in the third position of most reported practices involves not postpone the integration of finished tasks. 3 out of 16 interviewees cited that developers should integrate finished tasks as soon as possible to reduce conflict risk. For instance, one interviewee mentioned that there are scenarios of some finished tasks not integrated due to task priority change, resulting in long contributions conclusion delay (DIAS; BORBA; BARRETO, 2020). So when those tasks go to be integrated into the main repository, conflicts usually happen. For the same reason, do not postpone the integration of finished tasks also applies inversely, as reported by another interviewee. For example, when a bug is fixed and updated into the main repository, it should also be updated into the developers' branches during the sprint cycle instead of waiting to integrate at the final of the sprint. These both scenarios demand more controlled planning to prevent keeping a list of finished tasks for long periods without integration.

In addition, focusing on communication is also among the practices in the third position most mentioned by the interviews. 3 out of 16 interviewees affirmed that the quality of team communication (GUZZI et al., 2015) plays an important role among the practices to avoid or reduce conflicts. One highlighted that team communication must be intensively worked because they could reflect on other strategies to avoid conflicts, for instance, on how often developers integrate their changes and the level of the team awareness (CATALDO; HERBSLEB, 2013) of ongoing changes. Besides, another interviewee stated that a more experienced team also communicates more to each other, which ends up reducing conflict occurrence. Also, one interviewee observes that the team size can impact team communication quality, reinforcing the need for mechanisms or channels to better promote communication among developers.

The fourth most reported practice aiming at reducing the potential of merge conflicts is the adoption of mechanisms for improving team awareness. 2 out of 16 interviewees mentioned that, for instance, in complex projects, the team adopts mechanisms such as design review to improve team awareness of the ongoing changes. Tools like Lighthouse (SILVA et al., 2006), an Eclipse plug-in, can help on the use of design review strategy since this tool aiming at providing team awareness by tracking all ongoing changes in the developers' workspaces and automatically enables all developers the same view of the current design. Also, another interviewee mentioned the adoption of daily meetings for team awareness is a common practice for preventing or reduces merge conflicts. This makes sense given as improved team awareness allows the identification of relevant dependencies (CATALDO; HERBSLEB, 2013) and, in turn, also helps to reduce conflicts.

Another practice in the fourth position, the lock of a given file before making critical changes, is reported by 2 out of 16 interviewees as a practice to avoid merge conflicts. According to these interviewees, depending on the task (e.g., refactorings, change of API version, etc.), the best way to avoid conflicts is to lock the target file or files. This way, developers are prevented from making changes in the same file, avoid to deal with more complex conflicts or at least burdensome type of conflicts to solve. One of these interviewees also mentioned this practice demands a high level of communication among developers, and this could be made verbally or electronically aiding by tools.

The remaining four practices had only one mention. For example, one interviewee reported that the developers of his team are recommended to follow policies to avoid spurious conflicts caused by global non-semantic changes. For instance, when necessary, any code formatting is made in a controlled way, preventing both this type of conflict arises, and developers from wasting time to solve it. Although this type of conflict is not complex, its resolution is usually time-consuming.

Another interviewee reported splitting a big task into smaller independent tasks to avoid or reduce conflicts. According to this interviewee, the adoption of this practice results in small contributions that are constantly integrated until the development of the entire task be finished. This way, if adopted in a systematic way, each developer also will receive small changes concerning the ongoing changes, and so this potentially reduces conflicts (DIAS; BORBA; BARRETO, 2020) or, at least, allows dealing with conflicts while they are small and less complex. So, this practice also positively reflects in the Integrate often practice. Also, the adoption of pair-programming for similar tasks was mentioned by the same interviewee as another practice recommended in his company to reduce conflicts. In that case, when more than one developer needs to work in the same feature or very similar tasks concurrently, the corresponding tasks are developed employing pair-programming instead of being distributed to different developers. Moreover, according to the perception of this interviewee, it also promotes often code integration. The reason is that the other person not directly involved in coding could be responsible for alerting the other developer the best time to integrate the code, so helping to avoid long periods of coding without update the changes.

Finally, one interviewee reported that the distribution of similar tasks to the same developer is used as a practice to avoid merge conflicts is the projects he works. In this case, when there are similar tasks or tasks that, if developed in parallel, could cause overlapping changes, he affirmed to adopt as a practice to avoid conflicts to allocate the corresponding tasks for the same developer instead of distributing them to different ones.

By considering all the information reported by interviewees and summarized trough the main results we presented in this study, we can easily infer or relate the practices that apply to one or more merge conflict factors. For instance, the *Split a big task into smaller independent tasks* practice is a clear recommendation to avoid or reduce merge conflicts caused due to big tasks (Task size factor). For all 11 practices, we found at least one conflict factor for which the practice could be helpful. However, we could not find a correspondent practice for two merge conflict factors: *Changes in God Class* and *Code review timing*. We summarized the practices and the factors they apply in Table 15.

As a final remark, we highlight that, among the practices reported by the interviews, some are well-knowing by the industry as recommended rules of thumbs in an attempt to reduce the potential of code integration problems, such as integrate code often, modular task distribution, and branch policies. Other practices, for instance, adopting pair-programming for similar tasks, are not so commonly observed in the wild. In any case, there is still a lack of studies that empirically investigate the real impact of the use of these practices on merge conflict occurrence. This shed light on the need for further studies focusing on this topic.

Table 15 – Practices and Factors.

# Interviewees occurrence	Practices	Merge conflict factor assisted by the practice
12	Integrate Often	Duration, Task size, Integration frequency, Project phase, Conclusion delay, Team awareness, Number of changed files, Number of developers, Change size.
10	Modular task distribution	Task distribution, Software architecture.
3	Follow the recommended branch policies	Branch policy, Team awareness.
3	Do not postpone the integration of finished tasks	Conclusion delay, Team awareness.
3	Focus on Communication	Integration frequency, Team size, Communication, Inter-team development.
2	Provide mechanisms for team awareness	Team awareness
2	Lock file before making critical changes	Code change nature, Team awareness.
1	Follow policies for global non-semantic changes	Code change nature
1	Split a big task into smaller independent tasks	Duration, Task size, Integration frequency, Number of changed files, Change size.
1	Pair-programming for similar tasks	Integration frequency, Project size, Task distribution, Software architecture.
1	Allocate similar tasks to the same developer	Project size, Task distribution, Software architecture.

RQ4 Summary: We identified 11 practices the interviewees use to avoid or reduce merge conflicts. Two of these practices, *Integrate often* and *Modular task distribution*

were the most common reported by the interviewees. We relate each practice to one or more merge conflict factors that a given practice could be useful to avoid or reduce merge conflicts. We could not relate a practice that could apply to only 2 out of 20 merge conflict factors: *Changes in God Class* and *Code review timing.*

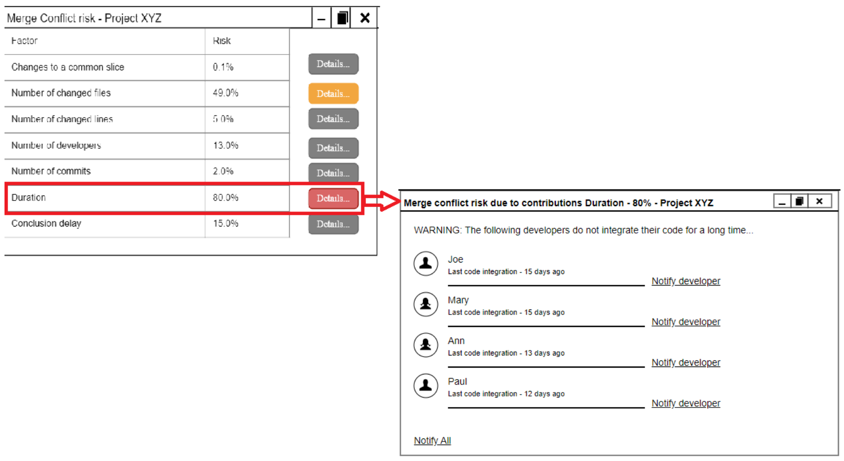
4.3.5 Could a tool to alert of merge conflict risk be useful?

We investigate whether a hypothetical tool for alerting the risk of merge conflict could be useful in the interviewees' software project development context. Our main goal is to evaluate the practical feasibility of implementing a conflict risk alert tool in the future by considering the modularity, size, and timing factors we investigate in our first study (DIAS; BORBA; BARRETO, 2020). Also, we want to gather information on how a tool like that could better assist the interviewees based on their opinion.

Before asking interviewees whether they would consider a tool for alerting of merge conflict risk useful in their context, we first contextualize them by providing examples on how we realize the use of a tool like that. Then, we asked the corresponding question.

For instance, as an example of a use case, let's consider the tool used by a given project manager (or any team member responsible for the project management status). The project manager opens the tool and sees a panel that shows the status of a given (or a set of factors) factor of interest with regards to the risk of merge conflicts. The factors available by the tool can be previously configured by the user according to his needs. The tool constantly monitors the developers' repositories by computing in the background, whether some of the factors have achieved the risk threshold previously configured. For instance, one could set the contribution *duration* factor in alerting the risk of conflict whenever its threshold achieves, for example, 80% of conflict risk. In this case, the user could be visually warned (the red button in Figure 12) that the factor achieved the risk threshold. Then, the user can explore more detailed information about the risk, such as the name of the developer (s) involved, and more detailed information about the corresponding factor. For instance, in the example of the risk of merge conflicts due to the contribution duration factor, the warning message could inform the number of days a given developer does not integrate his code (pull or push). So, the project manager can be aware of the factor motivating the risk and could use this information to make better decisions on how to deal with such a given risk. The same general idea applies to remaining size, timing, and modularity factors. Also, this tool could be integrated into the project management tool or developer IDE. In the example in Figure 12, the message warns the user that the developer Joe and Mary did not integrate their changes for the last 15 days. Similarly, Ann and Paul did not integrate their code for the last 13 and 12 days, respectively.

A tool for alerting the merge conflict risk was considered useful for most (13 out of 16) interviewees for different purposes. For instance, according to four interviewees, a risk conflict alert tool could assist the project management by providing useful process



Merge Conflict risk - Project XYZ

Factor	Risk	
Changes to a common slice	0.1%	Details...
Number of changed files	49.0%	Details...
Number of changed lines	5.0%	Details...
Number of developers	13.0%	Details...
Number of commits	2.0%	Details...
Duration	80.0%	Details...
Conclusion delay	15.0%	Details...

Merge conflict risk due to contributions Duration - 80% - Project XYZ

WARNING: The following developers do not integrate their code for a long time...

Joe
Last code integration - 15 days ago
Notify developer

Mary
Last code integration - 15 days ago
Notify developer

Ann
Last code integration - 13 days ago
Notify developer

Paul
Last code integration - 12 days ago
Notify developer

Notify All

Figure 12 – Use case example of the conflict risk alert tool for the duration factor

metrics to project managers based on real project status. This way, project managers could become more aware of developers' daily practices and, in turn, could make more safe decisions or adjustments concerning the development process. As an example, they could identify negligent developers regarding the use of recommended practices and improve the guidance related to that.

Two other interviewees mentioned that a risk alert tool could be most useful for the member team responsible for working on the integration of developers' contributions, configuration management, or DevOps, mainly if used integrated to the continuous integration process. His or her work would be reduced or becomes less tiresome, especially in cases where there is no systematic way of early monitoring conflict issues. Just having a tool to assist the integrator would be of great help. Moreover, they could warn developers of conflict rates and their causes. This warning would guide developers to make decisions for reducing the integration problems.

A conflict risk alert tool also could be useful to improve team awareness, as reported by another two interviewees. According to these interviewees, in their context, most merge conflicts occur when they do not know what other developers are doing. Moreover, sometimes although they know what tasks other developers are involved in, they not always have the notion of whether they will change files in common. Such a tool like this would help them better manage this problem.

Some other uses of a tool for alerting merge conflict risk mentioned by other interviewees involve, for instance, its use to assist less experienced developers, so they would be advised before making conflicting changes. Aligned with that, the use of a tool like that could reduce merge conflicts, according to another interviewee, since it would aid developers to make better decisions on how to deal with the potential of conflicts, possibly save effort with resolution.

Another example mentioned by interviewees is the use of a tool like that in complex projects, those with many code changes per day or that are observed to be of high complexity. According to the interviewee, as a certain noise level is expected in this type of project, a tool like this would avoid that the code integration problems increase in an uncontrolled level.

Software development projects with scenarios where long-lived branches are needed could also benefit from a conflict risk tool, according to one interviewee. This type of tool could be useful for monitoring long-lived branches since these scenarios demand more control to prevent conflicts. Aligned to the main idea of monitoring the developers' branches aiming at evaluating a safe time to integrate their contributions, Santos and others (SANTOS; MURTA, 2012) proposes a tool that computes the complexity of merging branches based on ten metrics. Three metrics among those they use are similar to some we investigate in our previous study (DIAS; BORBA; BARRETO, 2020): the number of changed files changed lines and number of merge (textual) conflicts. A conflict risk alert tool based on the metrics of both studies could benefit since it could combine strategies to provide more precise information on when to integrate developers' contributions based on both the complexity and conflict risk of integration branches.

Aligned with the previous idea, another interviewee affirmed that this type of tool could be useful for alerting developers about the repository update status. This way, a conflict risk tool could monitor the frequency in which developers integrate their contributions to the main repository or update their local workspace with the other developers' changes. To support this idea, we could benefit from the infrastructure already available of the DyeVC tool (CESARIO; INTERIAN; MURTA, 2017) since it assists team members, not only developers, in visualizing the project evolution by identifying the ongoing changes in all developers repositories and the dependencies among those repositories. As DyeCV also gathers information about existing commits in each developer repository, it can notify whether a given repository is behind oh ahead of another. This type of information gathered by DyeCV could serve as input to a conflict risk alert tool for computing the risk of merge conflicts. As DyeVC does not focus on detecting merge conflicts, it can be used combined with a conflict risk alert. So, developers would be systematically assisted in managing the integration frequency of their contributions, which could prevent potential conflicts.

Two interviewees consider a conflict risk alert tool useful with conditionals, for instance, in the context of more complex projects with more developers since, in simpler projects involving a small team, the conflicts are usually not complex to solve. Besides, one mentioned the SCM support available in the used IDE meets his needs when facing conflicts. Both agreed that, in any case, a tool like the asked could serve for aiding process metrics, for instance, to help the adoption of best practices to avoid conflicts, such as mentioned by other interviewees.

One interviewee did not consider a tool for alerting of conflict risk useful for developers at all, especially in his development context. In a more in-depth analysis of his development context, based on his reports thought the interview, we found the interviewee belongs to one of the four global companies we mention in that study. He works in projects that are developed under a large shared codebase, with large team sizes making concurrent parallel changes. That obligates the developers to follow strict policies to avoid conflicts, such as trunk-based development, with continuous integration and modular task distribution. Besides, according to the interviewee, when conflicts happen in the projects he works in, they are trivial to solve, and a tool like that could disturb the development team (ESTLER et al., 2014). However, he sees usefulness in this type of tool for collecting quality metrics to conduct a retrospective analysis of the development process for improvement purposes.

Some interviewees (11 out of 16) also gave us insight on desirable features they would like to see in a tool for alerting of conflict risk, such as when to alert the users of potential conflicts. The answers vary, as shown in Table 16. For instance, some interviewees informed the alert could occur any time during the task development. In contrast, others would prefer the alerts to be configurable according to the team and project needs.

Table 16 – Merge conflict risk alert tool desirable features.

Practices	**Desirable Features**
When to alert	At the beginning of task development
	During the task development
	At the final of task development
	Daily
	Before integrating the local changes
	Configurable according to the team and project needs
Tool integration	Developer IDE
	Continuous Integration
Additional metrics	Team size
	Project size (file size, number of lines per file)
	Team organization (pair-programming, etc.)
	Other Task Division Strategies (Back-end/Front-end, etc.)

Other features were mentioned, such as the need for a tool with this purpose to be integrated into the developer IDE. Moreover, other metrics were suggested for predicting conflicts and would provide improvements on the conflict prediction by reflecting more contextual projects characteristics, such as the team size, the team organization for developing tasks (e.g., pair-programming), the project size (e.g., file size, number of lines per file, etc.), and other task division strategies (e.g., Back-end/front-end, etc.).

RQ5 Summary: A tool for alerting merge conflict risk was considered useful for most interviewees, especially for preventing less experienced developers from making conflicting changes, and for help project managers, integrators, and DevOps for monitoring project status and make decisions to better guide developers to prevent or reduce merge conflicts. However, for other interviewees, there are contexts where a conflict risk alert tool could disturb the development team. For instance, in software development contexts involving a large shared codebase, with large team sizes working on trunk-based development and by using strict policies to avoid merge conflicts, such as modular task distribution or practices to early detect code integration problems, such as continuous integration. Some of the interviewees also gave us insight on desirable features, such as IDE integration, as well as by suggesting additional factors to consider, for instance, the team size, project size, task division strategy used, among others.

4.4 IMPLICATIONS

Our findings are aligned with most of the results of our prior study (DIAS; BORBA; BARRETO, 2020) concerning the influence of modularity, size, and timing factors in merge conflicts, reinforcing the implications we discuss there. However, we go further and bring qualitative empirical evidence of confounding factors and also by better understanding other ways to avoid or reduce merge conflicts. The results derive further implications and bring insights for future work, as follows.

4.4.1 For Researchers

Our findings advance the state of research by providing a better understanding of existing factors more associated with merge conflict occurrence. This work complements our first study (DIAS; BORBA; BARRETO, 2020) since we went further by investigating the relevance of the modularity, size, and timing factors, not only limited to Ruby and Python languages and frameworks, as the first study. We found these factors also apply in different software development contexts with varied team size, organizational culture, and programming languages, to mention a few.

The results of this second study indicate a set of 20 factors that practitioners perceive as relevant for causing merge conflicts. We also provide a comprehensive view of how these factors can influence each other and the impact of a given association among them in merge conflicts. We evaluate 5 out of 20 factors in our first study (DIAS; BORBA; BARRETO, 2020). Researchers could benefit from the results of this second study to investigate the remaining 15 factors. So, researches could observe whether the practitioners' perceptions conform to what happens in developers' repositories in the wild, for instance, by conducting quantitative studies in those repositories.

For instance, a possible merge conflict factor we do not investigate is the programming language. Some interviewees indicated the programming language as a factor associated with merge conflicts. We do not categorize it as a factor since we analyze the interviewees mentioned that based on measures of size, such as the fact there are programming languages and frameworks more verbose than others and, this way, more lines of code are usually written for those.

Other factors could also be investigated, such as the developer experience, team size, and project phase, to mention a few. For instance, our study brings evidence that merge conflict will occur more frequently depending on the project phase, according to perceptions of some interviewees. Although it seems obvious that the project phase can be associated with merge conflict occurrence, the reality is that there is a lack of studies that empirically confirms that. However, it is known that the execution of developers' tasks is more bug-prone in specific days of the week (ZIMMERMANN; ZELLER, 2005). Also, the code changes associated with the resolution of merge conflicts have more chances to become buggy (BRINDESCU et al., 2020). Based on those two prior studies, one could define measures to compute the project evolution in different stages to investigate whether the project phase or a given day of the week impacts on merge conflict occurrence and which type of impact it could be.

We did not find variations on the results of the interviewees' perceptions in terms of project domain, occupation, and social aspects, such as gender. However, we highlight that it is worth considering a deeper investigation of these factors and others, such as the level of communication of team members, occupation, etc., in further studies to evaluate whether these factors can play differently in other contexts and sample sizes.

Based on our findings, researchers can derive requirements for the conduction of new research studies in the area and the improvement or development of more accurate conflict prediction models, for instance, by providing more contextual project characteristics.

4.4.2 For Practitioners

The interviewees indicated that, although there are rules of thumb for avoiding merge conflicts, not all developers educate themselves on development processes that prevent and alleviate the complexity of merge conflicts. Based on the interviewees' perceptions, we present 11 practices most used to avoid or reduce conflicts. For each practice, we found at least one of the reported merge conflict factors that can be supported for it. Thus, developers could benefit that as advice on how to prevent merge conflicts. Also, managers and technical leaders can benefit from that result by improving the recommendations and guidelines to merge conflict avoidance in their development teams.

4.4.3 For Tool Builders

Our findings also provide insights into the development and improvement of toolsets that better fit developers' needs during merge conflict resolution. For example, the interviewees indicate the lack of current merge toolsets for handling merge conflicts more complex, for instance, those involving some types of global refactorings. This reinforces the need for tools and strategies that can assist developers in the merging process in the presence of refactorings (MAHMOUDI; NADI; TSANTALIS, 2019). A different kind of complex merge conflict that interviewees report is when developers need to understand the code involved in the conflicting changes. Prior study (COSTA et al., 2016) has focused on recommending developers best suited to perform a merge based on the developers' experience across branches and project history, reinforcing the relevance of tools like that. Furthermore, the improvement and development of project management and assistive tools could benefit from our findings by using prediction models based on the factors we derive and, in turn, could better fit the individual projects' characteristics. Finally, the development of a conflict alert tool could also benefit from the prediction models based on our findings and from the insights we bring concerning how the interviewees perceive the usefulness of a tool for merge conflict risk alert.

4.5 THREATS TO VALIDITY

Our study is limited by the number of practitioners that voluntarily agreed to participate. Despite the number of interviewees, we observe some degree of diversity since we interviewed 16 practitioners (4 females) across 13 different companies (most private), 4 of them with offices around the world. Besides, we observe diversity among the interviewees concerning the experience time (in years) working on software development projects, the size of teams they are used to work in, and the team organization. Although we can not generalize the results only based on interviewees' diversity, this varying helps for reducing the bias due to the reduced sample.

This study is based on practitioners' perceptions gathered from the interviews we conducted. To reduce the subjectivity of our findings, each interview was analyzed in pairs, followed by a conflict resolution meeting. When needed, a third researcher was responsible for driving consensus. Still, to reduce subjectivity and increase the confidence (Seaman, 1999) of our findings, we compare the results with existing literature.

To reduce bias concerning the questions, we follow well-know guidelines (MERRIAM; TISDELL, 2016) to properly formulate interview questions aiming at avoiding to induce the interviewee's answers. Besides, to reduce problems concerning questions misunderstanding, we assess the quality of the interview questions by conducting pilot interviews with two practitioners, whose answers were not considered in the results. Based on their feedback, we could identify which questions led to useless data, which questions are not clearly formulated, and which questions, suggested by them, we have thought to include.

During the pilot execution, we were also worried about evaluating the questions corresponding to the investigation of the merge conflict factors of our prior quantitative study results (DIAS; BORBA; BARRETO, 2020). To reduce bias, we intentionally do not inform the existence of this previous study and, in turn, its results. Also, before asking the question corresponding to each factor, we explain his definition by providing practical examples to assure the interviewee's understanding of the factor definition.

We do not provide interviewees with a prototype that could better show them how a tool for alerting of merge conflicts risk could work. Aware that the absence of a prototype represents a clear risk concerning the findings derived from this specific question of this study, we try to reduce the risks by giving interviewees examples of the scenarios of use of a tool like that before asking them the specific question. As our main goal with that specific question was to investigate the practical feasibility of implementing a hypothetical tool for alerting of merge conflict risk, we found the interviewee's feedback expressive in the sense we get valuable information on scenarios of use where a tool like that could be useful or not.

To conclude, all practitioners voluntarily agreed to participate in this study. Besides, following the core principle guidelines for research ethics (STRANDBERG, 2019), we guarantee to keep any sensitive data confidential to researchers. This way, all data were anonymously analyzed.

4.6 RELATED WORK

We relate our findings to existing literature throughout the manuscript. This way, in this section, we make additional remarks.

Our findings reinforce the results of our prior study (DIAS; BORBA; BARRETO, 2020) since most interviewees perceive modularity, size, and timing factors as associated with merge conflicts. Besides, they provide us a comprehensive view of confounding factors, their relationships, and the impact of a given relationship on merge conflict occurrence. We also go further by investigating other ways to help in conflict avoidance strategies, trough new factors, practices used to prevent or reduce conflicts, and by exploring the usefulness of a tool for merge conflict risk alert.

Another recent study related to ours is the one conducted by Lebenich et al. (LESSENICH et al., 2018) that survey 41 developers to understand factors that have the potential to predict the number of merge conflicts resulting from code integration. Based on the survey results, they define a set of factors, with only two in common to ours. Then, by mining GitHub repositories, they do not find any effect between the factors they analyze and the number of conflicts. Still, their results are not comparable to ours since they investigate the potential to predict the number of conflicts while our study, we investigate the potential to predict conflict occurrence. Besides, we go further and investigate other potential merge conflict factors.

McKee and others (MCKEE et al., 2017) investigate developers' perceptions about merge conflicts resolution difficulty. They show developers consider, among others, the size of changes as an important factor for resolution difficulty. Our results complement theirs since we found interviewees perceive that contributions developed over long periods usually increase merge complexity because this often results in bigger contributions that are more associated with merge conflict occurrence.

Mahmoud and others (MAHMOUDI; NADI; TSANTALIS, 2019) recently investigated the magnitude of the impact that refactorings have on merge conflicts by analyzing almost 3,000 GitHub java repositories. Among their finds, they show that refactoring operations are involved in 22% of merge conflicts and identify that conflicts involving code refactorings are usually more complex compared to conflicts with no refactoring changes. Their work is the first large-scale empirical study to understand the relationship between refactorings and merge conflicts. Our work reinforces their findings since, according to the practitioners' perception, conflicts involving code refactoring figure among the worst conflicts to solve.

A recent study conducted by Vale and others (VALE et al., 2020) brings the most comprehensive investigation about the relationship between *communication* and merge conflict occurrence. They investigate the history of 30 popular open-source projects hosted in GitHub, in nine different programming languages, among them Java, C++, Python, and Ruby. To measure the communication factor, they consider mechanisms of communication provided by GitHub utilities based on awareness, pull-request, and changes in artifacts. Among their main results, they do not found a significant association between communication and merge conflict occurrence. This result contradicts the common belief and the interviewees' perceptions we found in our study, in the sense that the interviewees report the communication factor among those more associated with merge conflicts. This shed light on the need for further studies since the authors of this prior study also consider that the approaches they use as communication measures, as well as project specifics characteristics, can be contributed to their results, reinforcing the need for additional research. These implications align with our results, given we also find that the communication factor can be influenced by other factors, which can impact positively or negatively the merge conflict occurrence, such as the number of developers, as reported by some interviewees.

Our work is also related to the Guzzi and others (GUZZI et al., 2015) study that conducted an investigation to explore how to support developers' collaboration. Their focus was on identifying requirements for the design of a tool to enable developers to better coordinate. They interviewed 11 practitioners from 9 different companies. Although their study did not investigating merge conflict factors, our findings reinforce their results since the interviewees also observe Communication and Team awareness as important factors that should be considered to prevent problems and react fast during collaborative development. Besides, the interviewees also report benefiting from software modularity to avoid inefficient task assignments or merge conflicts. Additionally, we interviewed more practitioners from more different companies, most but not limited to software developers, so we could explore merge conflicts in a broader perspective.

According to the interviewees' perceptions we present in our study, changes involving

God class usually increases the likelihood of merge conflicts. This confirms a recent study (AHMED et al., 2017) that investigates which code smells are more associated with merge conflicts by mining 143 GitHub repositories. They found that changes involving God class belong to the top 5 list of code smells more strongly associated with merge conflicts.

Costa and others (COSTA et al., 2014) investigate the problem of developers' assignment for merging branches by conducting a multi-method approach. By mining eight GitHub repositories corresponding to projects in a varying set of programming languages, they collected information about developers, commits, and merge scenarios for learning how the branch merging occurs. Then, they refine their findings by surveying 164 developers to investigate the developers' perceptions of how merge conflicts are solved in practice. Among their main results, they found most developers (80%) try to commit before other developers to avoid dealing with merge conflicts. Also, they identify that 76% of developers adopt, in some measure, policies to avoid merge conflicts, such as task allocation based on features, small and frequent commits, to mention a few. That previous work is similar to ours since we also adopt a multi-method approach. Besides, they reinforce the motivation of our research showing that a considerable proportion of developers adopt risky practices, such as rushing to finish their tasks before others to avoid dealing with conflicts. Moreover, we complement their findings since we bring quantitative and qualitative evidence that the modularity of tasks and the size of developers' contributions are relevant drivers to guide policies to avoid merge conflicts.

A number of empirical studies provide evidence about collaborative development issues and variations on conflict occurrence among projects analyzed (ZIMMERMANN, 2007; KASI; SARMA, 2013; BRUN et al., 2013; PERRY; SIY; VOTTA, 2001; CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY; BORBA; CAVALCANTI, 2018; BIRD; ZIMMERMANN, 2012; DIAS; BORBA; BARRETO, 2020; MENS, 2002). Based on interviewees' perceptions, we bring insights on factors related to contextual project characteristics that could be responsible for these variations among projects.

5 CONCLUSIONS

In this work, our main goal was to help developers on how to avoid or reduce merge conflicts. First, we conducted an empirical quantitative study to investigate the effect of 7 modularity, size, and timing of developers' contributions on merge conflicts. Then, we conducted a second study to qualitatively evaluate our findings and go further by exploring other ways to avoid merge conflicts.

We bring quantitative evidence of 6 out of 7 modularity, size, and timing factors that are more likely associated with merge conflicts. Our results bring evidence related to merge conflict reduction hypotheses and advice raised but not empirically evaluated by previous studies, but we also go further by exploring new hypotheses related to contribution modularity and conclusion delay. Furthermore, the infrastructure we developed for the quantitative study could be used, for instance, to derive more advanced conflict prediction models.

Concerning the second qualitative study, we find that most interviewees consider all 7 modularity, size, and timing factors we investigated in the first study as more associated with merge conflicts. Also, we identify 20 factors (5 of them we have already observed in the first study) the interviewees also consider relevant for causing merge conflicts and 11 practices developers use to avoid them. We present a comprehensive view of how other factors can influence the modularity, size, and timing factors we investigate in the previous study and the possible impact of a given association between these factors in merge conflict occurrence. Additionally, we learn about contexts where merge conflict avoidance is more critical and factors that are more associated with merge complexity. Finally, most interviewees consider a tool for alerting merge conflict risk useful for distinct purposes and provide us ideas on desirable features to assist development teams in avoiding or reducing conflicts. Conversely, one interviewee gives us insight into the contexts in which this type of tool does not apply.

By finding 15 new merge conflict factors in relation to the first study and understanding how interviewees perceive their importance; we derive a set of more robust requirements for the development of more accurate conflict prediction models. In turn, project management and assistive tools could then benefit from models that better fit the individual projects' characteristics. Practitioners can benefit from these results by improving the recommendations and guidelines to merge conflict avoidance. Tools builders also could benefit from the results since the practitioners indicate that current merge toolsets are lacking in support developers to handle complex merge conflicts.

5.1 CONTRIBUTIONS

We summarize our contributions as follows:

- Quantitative empirical evidence of 6 out of 7 modularity, size, and timing factors of developers' contributions that are more likely associated with merge conflict occurrence;
- Qualitative empirical evidence of the relevance of all 7 factors investigated in the first quantitative study. Also, a list of confounding factors related to the modularity, size, and timing of developers' contributions, with a comprehensive discussion about their possible impact on merge conflict occurrence.
- Qualitative empirical evidence of 15 new merge conflict factors, in relation to our first study that interviewees considered relevant for causing merge conflicts;
- A list of 11 practices developers most use to avoid merge conflicts with a mapping between merge conflict factors and each identified practice that could assist team members in preventing or reducing conflicts;
- Requirements for the development of more accurate merge conflict prediction models;
- Requirements for the improvement and development of tools for assisting developers in conflict avoidance or reduction;
- We provide the infrastructure we developed for the quantitative study in our online appendix (APPENDIX, 2018), allowing its replication;
- We made available all the material that was produced for the qualitative empirical study in (ATTACHMENTS, 2020).

5.2 FUTURE WORK

As the presented studies are part of a broader context, a set of related aspects will be left out of scope. Thus, the following topics are not directly addressed in this thesis, but we suggest them as future work:

- Based on our infrastructure, other studies could replicate our empirical quantitative study using proprietary projects;
- Based on our findings and infrastructure, other studies could explore the modularity hypothesis in other domains. For the Android application domain, for example, one could study whether integrating changes to the same Android component would be associated with conflict occurrence;

- We bring a comprehensive set of new 15 factors that can drive the execution of new quantitative studies for evaluating whether the practitioners' perceptions of this study conform with what happens in developers' repositories in the wild;
- As we present a set of more elaborate factors, other studies can use them as requirements for the development of more accurate conflict prediction models;
- Other studies could benefit from our findings to the development of a conflict risk alert tool since we provide a list of requirements practitioners consider important or desirable in a tool like that;
- We also provide insights on factors more associated with merge complexity that could benefit studies focusing on the development of tools to assist developers during the resolution of more complex conflicts.

REFERENCES

ACCIOLY, P.; BORBA, P.; CAVALCANTI, G. Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Software Engineering*, Springer US, v. 23, n. 4, p. 2051–2085, 2018.

ADAMS, B.; MCINTOSH, S. Modern release engineering in a nutshell–why researchers should care. In: *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. [S.l.]: IEEE, 2016. p. 78–90.

AHMED, I.; BRINDESCU, C.; MANNAN, U. A.; JENSEN, C.; SARMA, A. An empirical examination of the relationship between code smells and merge conflicts. In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.]: IEEE, 2017. p. 58–67.

ANDERSON, T.; FINN, J. D. *The New Statistical Analysis of Data*. [S.l.]: Springer, 1996.

APEL, S.; LESSENICH, O.; LENGAUER, C. Structured merge with auto-tuning: balancing precision and performance. In: ACM. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.], 2012. p. 120–129.

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. p. 190–200.

APPENDIX, O. 2018. URL: https://merge-conflict-factors.github.io/merge-conflict-factors/.

ARANDA, J.; VENOLIA, G. The secret life of bugs: Going past the errors and omissions in software repositories. In: *2009 IEEE 31st International Conference on Software Engineering*. [S.l.]: IEEE, 2009. p. 298–308.

ATTACHMENTS, T. 2020. URL: https://bit.ly/2VuOQGs.

BALDWIN, C. Y. *Design Rules: The Power of Modularity*. [S.l.]: The MIT Press, 2000.

BASS, L.; WEBER, I.; ZHU, L. *DevOps: A Software Architect's Perspective*. [S.l.]: Addison-Wesley Professional, 2016.

BIRD, C.; RIGBYY, P. C.; BARR, E. T.; HAMILTON, D. J.; GERMANY, D. M.; DEVANBU, P. The promises and perils of mining git. In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. [S.l.]: IEEE, 2009. p. 1–10.

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. p. 45:1–45:11.

BIRD, C.; ZIMMERMANN, T.; TETEREV, A. A theory of branches as goals and virtual teams. In: *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*. [S.l.]: Association for Computing Machinery, 2011. p. 53–56.

BOTTCHER, E. *What are our core values and practices for building software?* 2017. URL: https://goo.gl/6QonqY.

BRIAND, L.; BIANCULLI, D.; NEJATI, S.; PASTORE, F.; SABETZADEH, M. The case for context-driven software engineering research: Generalizability is overrated. *IEEE Software*, IEEE, v. 34, n. 5, p. 72–75, 2017.

BRINDESCU, C.; AHMED, I.; JENSEN, C.; SARMA, A. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering*, Springer US, v. 25, n. 1, 2020.

BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, IEEE, v. 39, n. 10, p. 1358–1375, 2013.

BRUN, Y.; KIVANç, M.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Predicting development trajectories to prevent collaboration conflicts. *FutureCSD 2012: The Future of Collaborative Software Development*, 2012.

CATALDO, M.; BASS, M.; HERBSLEB, J. D.; BASS, L. On coordination mechanisms in global software development. In: *Proceedings of the International Conference on Global Software Engineering*. [S.l.]: IEEE Computer Society, 2007. p. 71–80. ISBN 0769529208.

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. p. 161–170.

CATALDO, M.; HERBSLEB, J. D. Coordination breakdowns and their impact on development productivity and software failures. *IEEE Transactions on Software Engineering*, IEEE, v. 39, n. 3, p. 343–360, 2013.

CAVALCANTI, G.; BORBA, P.; ACCIOLY, P. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, ACM, v. 1, n. OOPSLA, p. 59:1–59:27, 2017.

CESARIO, C.; INTERIAN, R.; MURTA, L. Dyevc: an approach for monitoring and visualizing distributed repositories. *Journal of Software Engineering Research and Development*, v. 5, 2017.

CHACON, S.; STRAUB, B. *Pro Git*. [S.l.]: Apress, 2014.

COSTA, C.; FIGUEIREDO, J.; MENEZES, G. Ghiotto lima de; MURTA, L. Characterizing the problem of developers' assignment for merging branches. *International Journal of Software Engineering and Knowledge Engineering*, v. 24, 2014.

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.]: Association for Computing Machinery, 2016. p. 523–534.

CRAWLEY, M. J. *Statistics: An Introduction Using R*. [S.l.]: Wiley, 2014.

DIAS, K.; BORBA, P.; BARRETO, M. Understanding predictive factors for merge conflicts. *Information and Software Technology*, v. 121, p. 106256, 2020. ISSN 0950-5849.

ESTLER, H. C.; NORDIO, M.; FURIA, C. A.; MEYER, B. Awareness and merge conflicts in distributed software development. In: *Proceedings of the 2014 IEEE 9th International Conference on Global Software Engineering*. [S.l.]: IEEE Computer Society, 2014. p. 26–35.

FOWLER, M. *Feature Toggle*. 2017. URL: https://goo.gl/QfJ6mM.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D.; GAMMA, E. *Refactoring: Improving the Design of Existing Code*. [S.l.]: Addison-Wesley, 2012.

GHIOTTO, G.; MURTA, L.; BARROS, M.; HOEK, A. v. d. On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github. *IEEE Transactions on Software Engineering*, IEEE, v. 39, n. 3, p. 1–1, 2018.

GIT. 2019. URL: https://git-scm.com/.

GLASER, B. G. *Basics of Grounded Theory Analysis: Emergence Vs. Forcing*. [S.l.]: Sociology Pr, 1992.

GRINTER, R. E. Supporting articulation work using software configuration management systems. *Comput. Supported Coop. Work*, Kluwer Academic Publishers, v. 5, n. 4, p. 447–465, 1996.

GUIMARãES, M. L.; 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. p. 342–352. ISBN 9781467310673.

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 Social Computing*. [S.l.]: Association for Computing Machinery, 2015. p. 518–532.

HENDERSON, F. *Software Engineering at Google*. 2017. URL: https://arxiv.org/abs/1702.01715.

HERBSLEB, J. Building a socio-technical theory of coordination: Why and how (outstanding research award). In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.]: ACM, 2016. p. 2–10.

HODGSON, P. *Feature Branching vs. Feature Flags: What's the Right Tool for the Job?* 2017. URL: https://goo.gl/4D2AMv.

JR, D. W. H.; LEMESHOW, S.; STURDIVANT, R. X. *Applied logistic regression*. [S.l.]: John Wiley & Sons, 2013.

KALLIAMVAKOU, E.; GOUSIOS, G.; BLINCOE, K.; SINGER, L.; GERMAN, D. M.; DAMIAN, D. The promises and perils of mining github. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.]: ACM, 2014. p. 92–101.

KASI, B. K.; SARMA, A. Cassandra: proactive conflict minimization through optimized task scheduling. In: *Proceedings of the 2013 International Conference on Software Engineering. ICSE '13*. [S.l.]: IEEE Press, 2013. p. 732–741.

KUTNER, M. H.; NACHTSHEIM, C.; NETER, J. *Applied linear regression models*. [S.l.]: McGraw-Hill/Irwin, 2004.

LARSEN, R. J.; MARX, M. L. *An Introduction to Mathematical Statistics and Its Applications*. 6. ed. [S.l.]: Pearson, 2017.

LESSENICH, O.; SIEGMUND, J.; APEL, S.; KÄSTNER, C.; HUNSEN, C. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engg.*, Kluwer Academic Publishers, v. 25, n. 2, p. 279–313, 2018.

MAHMOUDI, M.; NADI, S.; TSANTALIS, N. Are refactorings to blame? an empirical study of refactorings in merge conflicts. In: . [S.l.: s.n.], 2019. p. 151–162.

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, 2017. p. 467–478.

MENS, T. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, IEEE, v. 28, n. 5, p. 449–462, 2002.

MERCURIAL. 2019. URL: https://www.mercurial-scm.org/.

MERRIAM, S. B.; TISDELL, E. J. *Qualitative Research: A Guide to Design and Implementation*. [S.l.]: Jossey Bass, 2016.

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. p. 466–476.

NELSON, N.; BRINDESCU, C.; MCKEE, S.; SARMA, A.; DIG, D. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empir Software Eng*, Springer US, v. 24, n. 5, p. 2863–2906, 2019.

ORAM, A.; WILSON, G. *Making software: What really works, and why we believe it*. [S.l.]: O'Reilly Media, Inc., 2010.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, ACM, v. 15, n. 12, p. 1053–1058, 1972.

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 (TOSEM)*, ACM, v. 10, n. 3, p. 308–337, 2001.

POTVIN, R.; LEVENBERG, J. Why google stores billions of lines of code in a single repository. *Commun. ACM*, ACM, v. 59, n. 7, p. 78–87, 2016.

PROJECT, R. *The R Project for Statistical Computing*. 2017.

SALDANA, J. *The Coding Manual for Qualitative Researchers*. [S.l.]: SAGE Publications Ltd, 2015.

SANTOS, R. d. S.; MURTA, L. G. P. Evaluating the branch merging effort in version control systems. In: *26th Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2012. p. 151–160.

SARMA, A.; REDMILES, D. F.; HOEK, A. V. D. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, IEEE, v. 38, n. 4, p. 889–908, 2012.

Seaman, C. B. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, v. 25, n. 4, p. 557–572, 1999.

SILVA, I. A. d.; CHEN, P. H.; WESTHUIZEN, C. V. d.; RIPLEY, R. M.; HOEK, A. v. d. Lighthouse: Coordination through emerging design. In: *eclipse '06: Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology EXchange*. [S.l.]: Association for Computing Machinery, 2006.

SOUZA, C. R. B. de; REDMILES, D.; DOURISH, P. Breaking the code, moving between private and public work in collaborative software development. In: *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*. [S.l.]: ACM, 2003. p. 105–114.

SOUZA, L. de; MAIA, M. Do software categories impact coupling metrics? In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. [S.l.]: IEEE, 2013. p. 217–220.

STRANDBERG, P. E. Ethical interviews in software engineering. *CoRR*, abs/1906.07993, 2019.

SUBVERSION. 2019. URL: http://subversion.apache.org/.

VALE, G.; SCHMID, A.; SANTOS, A. R.; ALMEIDA, E. S. de; APEL, S. On the relation between github communication activity and merge conflicts. *Empirical Software Engineering*, Springer US, v. 25, n. 1, 2020.

WLOKA, J.; RYDER, B.; TIP, F.; REN, X. Safe-commit analysis to facilitate team software development. In: *IEEE 31st International Conference on Software Engineering*. [S.l.: s.n.], 2009. p. 507–517.

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: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.]: IEEE Press, 2017. p. 60–71.

ZIMMERMANN, T. Mining workspace updates in cvs. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories. MSR'07*. [S.l.]: IEEE Computer Society, 2007. p. 11–.

ZIMMERMANN, T.; ZELLER, A. When do changes induce fixes? on fridays. In: *In Proc. International Workshop on Mining Software Repositories (MSR),St.* [S.l.: s.n.], 2005.