Information and Software Technology 121 (2020) 106256

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Understanding predictive factors for merge conflicts

Klissiomara Dias[a,b,*], Paulo Borba[a], Marcos Barreto[a]

[a] Informatics Center, Federal University of Pernambuco, Av. Jornalista Anibal Fernandes, s/n 50740-560 Recife, PE, Brazil
[b] Cyberspatial Institute, Federal Rural University of Amazonia, Av. Presidente Tancredo Neves, n° 2501, 66.077-830 Belém, PA, Brazil

ARTICLE INFO

Keywords:
code integration
merge conflict
modularity
collaborative development
empirical study

ABSTRACT

Context: 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.

Objective: To minimize these problems by reducing merge conflicts, it is important to better understand how conflict occurrence is affected by technical and organizational factors.

Method: With that aim, we investigate seven factors related to modularity, size, and timing of developers contributions. To do so, we reproduce and analyze 73504 merge scenarios in GitHub repositories of Ruby and Python MVC projects.

Results: 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.

Conclusion: Our results could be used to derive recommendations for development teams and merge conflict prediction models. Project management and assistive tools could benefit from these models.

1. Introduction

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 [1–5].

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 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

* Corresponding author at: Informatics Center Av. Jornalista Anibal Fernandes, s/n 50740-560 Recife, Brazil.
E-mail addresses: kld2@cin.ufpe.br (K. Dias), phmb@cin.ufpe.br (P. Borba), msb5@cin.ufpe.br (M. Barreto).

https://doi.org/10.1016/j.infsof.2020.106256
Received 6 March 2019; Received in revised form 12 December 2019; Accepted 6 January 2020
Available online 11 January 2020
0950-5849/

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 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 2 motivates our study and presents our research questions. Section 3 describes our study setup and how we collect the investigated predictors.[1] In Sections 4 and 5, we respectively discuss findings and implications. Section 6 presents the threats to the validity of our study. Related work is discussed in Section 7. Finally, in Section 8, we conclude.

2. 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 [2,4–6]. This is observed even when using advanced merge tools [7–10] 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 [3,11,12]. So, to avoid dealing with conflicts, developers adopt risky practices such as rushing to finish changes first [11,13], and partial check-ins [14]. Similarly, partially motivated by the need to reduce conflicts, development teams have been adopting techniques such as trunk-based development [15–17] and feature toggles [15,18–20].

As these studies observe largely varying merge conflict rates among projects, and others [21] 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.

[1] 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.

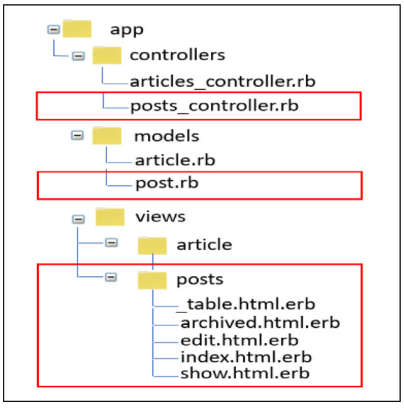

Fig. 1. Post slice files.

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 [22–24], 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 Fig. 1 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. Fig. 1 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 [25], 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 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 [26], 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[2] 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 [12], 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 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 [15] suggest that the best way for reducing conflicts is to keep branches short-lived and merge often. Bird and Zimmerman [3] 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. 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. 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.

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.

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

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

Fig. 2. A conflicting chunk example of merge 96df565 from Expertiza project.

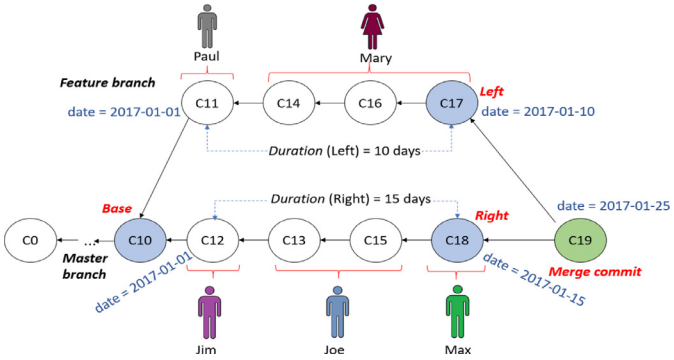

Fig. 3. Example of Merging Contributions.

3.1. Mining Step

To mine merge scenarios, we implemented a script[3] 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* [28]), are delimited by lines containing specific markers ("<<<<<<<" and ">>>>>>>", with the separator "======="). Fig. 2 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 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 Fig. 3, which illustrates such a merge scenario. Let us consider a hypothetical project,

[3] Available in our online appendix [27].

Table 1
Merge conflict factors.

Metric	Description
Dependent Variables	
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.
Modularity Metric	
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.
Size Metrics	
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 [32].
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 [33]. 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.
Timing Metrics	
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.

which adopts a feature branch model for implementing development tasks in parallel to the master branch. Fig. 3 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 [29–31].

3.2. 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*).

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 Fig. 1, 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 Fig. 1. 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 Fig. 3, suppose that Joe and Mary changed the `posts_controller.rb` class (which is related to the Post slice in Fig. 1) 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 Fig. 3. All metrics are collected per merge scenario. This way, we normalize these metrics by using geometric mean since the values regarding each contribution might substantially differ.

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[4] is 2017-01-01 and C17 author date is 2017-01-10),

[4] The date when the author made the original commit.

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 Fig. 3, 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.

3.3. Sample

To select our Ruby sample, we first used GitHub's advanced search page[5] to return only Ruby[6] 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 [29]. 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.

Although we have not systematically targeted representativeness or even diversity [34], 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 [35]. 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.

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 [27].

[5] https://github.com/search/advanced.

[6] GitHub search page allows only filtering repositories based on programming language, not on frameworks.

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

4. 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.

4.1. 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 2), 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 [36], we apply Logistic Regression [37] models to estimate the probability of merge conflict occurrence. Logistic regression is the simplest technique that applies to our context: a binary dependent variable

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%

(***$p < 0.001$; **$p < 0.01$; *$p < 0.05$).

(*merge conflict occurrence*, (say *c*), and continuous[7] 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:

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

where *glm* is the R [38] 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*[8]). 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 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 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 2. 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 [39]. 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[9] 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.

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[10], 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

[7] We also apply logistic regression to assess the *size* and *timing* metrics, as shown in Sections 4.2 and 4.3.

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

[9] fatfreecrm/fat_free_crm; commit 4d1e3e4.

[10] fatfreecrm/fat_free_crm; commit 72eae62.

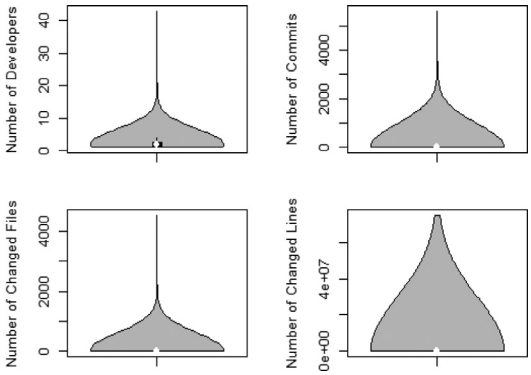

Fig. 4. Size Factors Descriptive Statistics.

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 [34] 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 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 [40], 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.

4.2. 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 Fig. 4. For instance, we observe the number of commits was of 9.5 ± 32.5 (*average* ± *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 [41].

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 [42], 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[11] [43], as it may impair prediction. So, before executing the models, we performed a pair-wise correlation analysis. This way, we avoid models with high correlation (multicollinearity) among the independent variables.[12] 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 [27].

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

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

[12] 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 ($p > 0.9$).

threshold used in Section 4.1, 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* ($\rho = 0.32, p < 2.2e16$). Our online appendix [27] 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.[13] 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[14] 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[15] 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[16] 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,[17] 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.[18]

4.3. 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 Fig. 5. 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

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 $p = 0$).

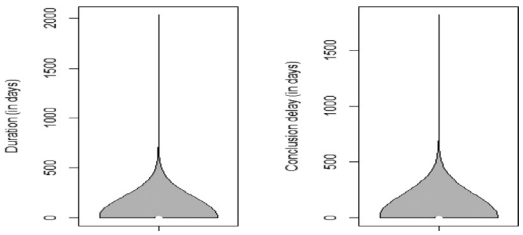

Fig. 5. Timing factors descriptive statistics.

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.

5. 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.

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

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

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[19] 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[20]. 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 [7,9], 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.

Our findings can also drive the development and improvement of awareness and conflict detection tools. Awareness tools like Palantir [11], 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 [5], 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.

Theories and other studies

Our results bring evidence related to merge conflict reduction hypotheses and advice raised but not empirically evaluated by previous studies [12,15,24], 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 [44] 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 4.1 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.

6. Threats to validity

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

6.1. 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

[19] `git checkout -- Gemfile.lock`

[20] https://git-scm.com/docsgitattributes#_defining_a_custom_merge_driver.

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.

6.2. 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*,[21] which rewrites project history [29–31], and we do not have access to private repositories in the developers machines, needed to trace and collect all integration scenarios,[22] we might have missed a number of scenarios and conflicts that have not reached the public repositories we analyze. Aranda and Venolia [45] 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, 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.

6.3. 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 [26] 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. 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.

7. Related work

Our work is mostly related to the recent empirical study conducted by Lebenich et al. [31]. 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. Moreover, 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 [46]. 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 [12] 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 [47], 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 [28] 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

[21] https://git-scm.com/docs/git-rebase

[22] *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.

merge conflicts (named number of conflicting chunks in their work) has influential on developers perception about merge difficulty and affects their 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 [42] 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. [10] 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 work propose new tools and strategies to both decrease integration effort and improve correctness during task integration. Cassandra, proposed by Kasi and Sarma [4] is a tool that analyzes task constraints to recommend an optimum order of task execution. Palantir [11] informs developers of ongoing parallel changes, and Crystal, proposed by Brun et al. [5], proactively integrates commits from developers repositories with the purpose of warning them if their changes conflict. Such kind of speculative merge early detects conflicts and consequently reduces resolution effort. 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.

8. Conclusions

We conducted an empirical study to investigate the effect of modularity, size, and timing of developers contributions on merge conflicts. Our results indicate 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. We also observe that bigger contributions involving more developers, commits, and changed files are more likely associated with merge conflicts. The same applies to the contribution duration.

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 this study could be used, for instance, to derive more advanced conflict prediction models.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Klissiomara Dias: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing. **Paulo Borba:** Conceptualization, Methodology, Validation, Writing - review & editing, Supervision, Funding acquisition. **Marcos Barreto:** Software, Data curation.

Acknowledgments

We thank the anonymous reviewers and members of the Software Productivity Group for the quite pertinent suggestions that contributed to improve this work. This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, and CNPq grant 465614/2014-0. It was also supported by CNPq projects 309172/2017-9 and 408516/2018-6.

References

[1] T. Mens, A state-of-the-art survey on software merging, IEEE Trans. Softw. Eng. 28 (5) (2002) 449–462.
[2] T. Zimmermann, Mining workspace updates in cvs, in: Proceedings of the Fourth International Workshop on Mining Software Repositories. MSR'07, IEEE Computer Society, 2007, p. 11.
[3] C. Bird, T. Zimmermann, Assessing the value of branches with what-if analysis, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, 2012, pp. 45:1–45:11.
[4] B.K. Kasi, A. Sarma, Cassandra: proactive conflict minimization through optimized task scheduling, in: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13, IEEE Press, 2013, pp. 732–741.
[5] Y. Brun, R. Holmes, M.D. Ernst, D. Notkin, Early detection of collaboration conflicts and risks, IEEE Trans. Softw. Eng. 39 (10) (2013) 1358–1375.
[6] D.E. Perry, H.P. Siy, L.G. Votta, Parallel changes in large-scale software development: an observational case study, ACM Trans. Softw. Eng. Methodol. (TOSEM) 10 (3) (2001) 308–337.
[7] S. Apel, J. Liebig, B. Brandl, C. Lengauer, C. Kästner, 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, ACM, 2011, pp. 190–200.
[8] S. Apel, O. Leßenich, C. Lengauer, Structured merge with auto-tuning: balancing precision and performance, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 120–129.
[9] G. Cavalcanti, P. Borba, P. Accioly, Evaluating and improving semistructured merge, Proc. ACM Programm. Lang. 1 (OOPSLA) (2017) 59:1–59:27.
[10] P. Accioly, P. Borba, G. Cavalcanti, Understanding semi-structured merge conflict characteristics in open-source java projects, Empir. Softw. Eng. 23 (4) (2018) 2051–2085.
[11] A. Sarma, D.F. Redmiles, A. Van Der Hoek, Palantir: early detection of development conflicts arising from parallel code changes, IEEE Trans. Softw. Eng. 38 (4) (2012) 889–908.
[12] S. McKee, N. Nelson, A. Sarma, D. Dig, Software practitioner perspectives on merge conflicts and resolutions, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 467–478.
[13] R.E. Grinter, Supporting articulation work using software configuration management systems, Comput. Support. Coop. Work 5 (4) (1996) 447–465.
[14] C.R.B. de Souza, D. Redmiles, P. Dourish, Breaking the code, moving between private and public work in collaborative software development, in: Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work, ACM, 2003, pp. 105–114.
[15] B. Adams, S. McIntosh, Modern release engineering in a nutshell–why researchers should care, in: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, IEEE, 2016, pp. 78–90.
[16] F. Henderson, Software engineering at google. arXiv:1702.01715. Accessed: December 2017
[17] R. Potvin, J. Levenberg, Why google stores billions of lines of code in a single repository, Commun. ACM 59 (7) (2016) 78–87.
[18] L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2016.
[19] M. Fowler, Feature toggle, Accessed: December URL https://goo.gl/QfJ6mM2017.
[20] P. Hodgson, Feature branching vs. feature flags: What's the right tool for the job?, Accessed: December URL https://goo.gl/4D2AMv2017.
[21] H.C. Estler, M. Nordio, C.A. Furia, B. Meyer, Awareness and merge conflicts in distributed software development, in: Proceedings of the 2014 IEEE 9th International Conference on Global Software Engineering, IEEE Computer Society, 2014, pp. 26–35.
[22] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (12) (1972) 1053–1058.
[23] C.Y. Baldwin, Design Rules: The Power of Modularity, The MIT Press, 2000.

[24] M. Cataldo, J.D. Herbsleb, Coordination breakdowns and their impact on development productivity and software failures, IEEE Trans. Softw. Eng. 39 (3) (2013) 343–360.
[25] E. Bottcher, What are our core values and practices for building software?, 2017Accessed: October URL https://goo.gl/6QonqY.
[26] L. Briand, D. Bianculli, S. Nejati, F. Pastore, M. Sabetzadeh, The case for context–driven software engineering research: generalizability is overrated, IEEE Softw. 34 (5) (2017) 72–75.
[27] Online Appendix,2018. Accessed: March URL https://merge-conflict-factors.github.io/merge-conflict-factors/
[28] G. Ghiotto, L. Murta, M. Barros, A.v.d. Hoek, On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github, IEEE Trans. Softw. Eng. 39 (3) (2018). 1–1
[29] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D.M. German, D. Damian, The promises and perils of mining github, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 92–101.
[30] C. Bird, P.C. Rigbyy, E.T. Barr, D.J. Hamilton, D.M. Germany, P. Devanbu, The promises and perils of mining git, in: 2009 6th IEEE International Working Conference on Mining Software Repositories, IEEE, 2009, pp. 1–10.
[31] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, C. Hunsen, Indicators for merge conflicts in the wild: survey and empirical study, Automat. Softw. Engg. 25 (2) (2018) 279–313.
[32] S. Chacon, B. Straub, Pro Git, Apress, 2014.
[33] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, B. Vasilescu, The impact of continuous integration on other software development practices: A large-scale empirical study, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE Press, 2017, pp. 60–71.
[34] M. Nagappan, T. Zimmermann, C. Bird, Diversity in software engineering research, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 466–476.
[35] L. de Souza, M. Maia, Do software categories impact coupling metrics? in: 2013 10th Working Conference on Mining Software Repositories (MSR), IEEE, 2013, pp. 217–220.
[36] M.J. Crawley, Statistics: An Introduction Using R, Wiley, 2014.
[37] D.W. Hosmer Jr, S. Lemeshow, R.X. Sturdivant, Applied Logistic Regression, John Wiley & Sons, 2013.
[38] The r project for statistical computing, 2017Accessed: December. URL https://www.r-project.org/.
[39] T. Anderson, J.D. Finn, The New Statistical Analysis of Data, Springer, 1996.
[40] A. Oram, G. Wilson, Making software: What really works, and why we believe it, O'Reilly Media, Inc., 2010.
[41] R.J. Larsen, M.L. Marx, An Introduction to Mathematical Statistics and Its Applications, 6, Pearson, 2017.
[42] M. Cataldo, J.D. Herbsleb, Factors leading to integration failures in global feature-oriented development: an empirical analysis, in: Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, pp. 161–170.
[43] M.H. Kutner, C. Nachtsheim, J. Neter, Applied linear regression models, McGraw-Hill/Irwin, 2004.
[44] J. Herbsleb, 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, ACM, 2016, pp. 2–10.
[45] J. Aranda, G. Venolia, The secret life of bugs: going past the errors and omissions in software repositories, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE, 2009, pp. 298–308.
[46] I. Ahmed, C. Brindescu, U.A. Mannan, C. Jensen, A. Sarma, An empirical examination of the relationship between code smells and merge conflicts, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2017, pp. 58–67.
[47] N. Nelson, C. Brindescu, S. McKee, A. Sarma, D. Dig, The life-cycle of merge conflicts: processes, barriers, and strategies, Empir. Softw. Eng. 24 (5) (2019) 2863–2906.