The Private Life of Merge Conflicts

Marcela Cunha
Centro de Informática
Universidade Federal de Pernambuco
Brasil
mbc3@cin.ufpe.br

Paola Accioly
Universidade Federal do Cariri
Brasil
paola.accioly@ufca.edu.br

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco
Brasil
phmb@cin.ufpe.br

ABSTRACT

Collaborative development is an essential practice for the success of most nontrivial software projects. However, merge conflicts might occur when a developer integrates, through a remote shared repository, their changes with the changes from other developers. Such conflicts may impair developers' productivity and introduce unexpected defects. Previous empirical studies have analyzed such conflict characteristics and proposed different approaches to avoid or resolve them. However, these studies are limited to the analysis of code shared in public repositories. This way they ignore local (developer private) repository actions and, consequently, code integration scenarios that are often omitted from the history of remote shared repositories due to the use of commands such as `git rebase`, which rewrite Git commit history. These studies might then be examining only part of the actual code integration scenarios and conflicts. To assess that, in this paper we aim to shed light on this issue by bringing evidence from an empirical study that analyzes git command history data extracted from the local repositories of a number of developers. This way we can access hidden integration scenarios that cannot be accessed by analyzing public repository data as in GitHub based studies. We analyze 95 git reflog files from 61 different developers. Our results indicate that hidden code integration scenarios are more frequent than the visible ones. We also find higher conflict rates than previous studies. Our evidence suggests that studies that consider only remote shared repositories might loose integration conflict data by not considering the developer's local repository actions.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**.

KEYWORDS

collaborative software development, merge conflicts, empirical software engineering, repository mining

ACM Reference Format:
Marcela Cunha, Paola Accioly, and Paulo Borba. 2022. The Private Life of Merge Conflicts. In *XXXVI Brazilian Symposium on Software Engineering (SBES 2022), October 5–7, 2022, Virtual Event, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3555228.3555240

SBES 2022, October 5–7, 2022, Virtual Event, Brazil

ACM ISBN 978-1-4503-9735-3/22/10…$15.00
https://doi.org/10.1145/3555228.3555240

1 INTRODUCTION

In a software development environment, team members often work collaboratively through a shared remote repository. It's common for developers to work on tasks independently of each other. Each one uses their own local copies of a remote project repository. When a developer concludes a task, it is time to integrate the associated contributions and conflicts might then occur if developers changed overlapping areas in a common file. These are called merge conflicts [Bird and Zimmermann 2012; Brun et al. 2013; Kasi and Sarma 2013; Mahmood et al. 2020; Perry et al. 1998; Zimmermann 2007], as opposed to other kind of conflicts that might be detected during building [Da Silva et al. 2022], testing [Silva et al. 2020], or even at system production time.

Although many merge conflicts are easy to fix, some may demand significant effort and system knowledge before they can be resolved. Besides that, there is a risk of a developer incorrectly resolving a conflict. When this happens, the consequence is the introduction of unexpected defects in the system [Bird and Zimmermann 2012; McKee et al. 2017]. In fact, such conflicts may impair developers' productivity and compromise system quality [Bird and Zimmermann 2012; McKee et al. 2017; Sarma et al. 2012]. Because of these negative consequences, and as merge conflicts might often occur [Bird and Zimmermann 2012; Brun et al. 2013; Kasi and Sarma 2013; Mahmood et al. 2020; Mens 2002; Perry et al. 1998; Zimmermann 2007], a number of studies focus on understanding conflict characteristics, examining different mechanisms for proactive conflict detection [Brun et al. 2011; Guimarães and Silva 2012; van der Hoek and Sarma 2008], and proposing tools to more effectively resolve conflicts [Cavalcanti et al. 2017; Clementino et al. 2021; Mens 2002; Nishimura and Maruyama 2016; Tavares et al. 2019]. This topic has been deeply studied in the literature, with the aim of seeking proper technical and organizational support to avoid the negative impact of conflicts on development productivity and software quality. However, evidence in the literature is limited to detecting and analyzing merge conflicts [Accioly et al. 2018a; Apel et al. 2011; Cavalcanti et al. 2015; Zimmermann 2007] in shared remote repositories such as the ones available in GitHub, many of them considered to be the main project repositories.

By focusing only on merge scenarios visible in *shared remote* repositories, these studies ignore *local (private)* repository actions and, consequently, integration scenarios that are often omitted from the history of remote shared repositories due to the use of commands such as `git rebase` and `squash`, which rewrite Git commit history. This way integration scenarios and conflicts that locally occur and are resolved in the developers' private environment become hidden, that is, are not visible in remote repositories and, therefore, are not analyzed by these studies. As a consequence,

these studies do not assess the influence that local actions might have on the occurrence of merge conflicts in remote repositories. In addition, these studies also miss integration scenarios and conflicts originated from the use of Git commands (rebase, cherry-pick, etc.) that perform code integration but leave no trace in remote shared repository histories. In the case of rebase, for example, it integrates changes from two branches but rewrites the history so that it's linear. As it does not explicitly signal that integration has occurred through a special (merge) commit, such integrations are not considered in previous studies. The failure to detect these integration scenarios might negatively impact research results, as, for instance, integration and conflict frequency could be higher than what is actually observed by considering only data from shared remote repositories.

To assess that, we aim to shed light on this issue by bringing evidence from an empirical study that analyzes git command history data extracted from the local (private) repositories of a number of developers. In particular, this paper contributes with a new study that differs from what has been presented in the literature in two main ways: (a) examining developer local repository logs instead of remote shared repositories, and (b) going beyond `git merge` and analyzing additional git code integration commands and the conflicts they generate. The core idea is to evaluate developer's actions in local private repositories to understand their daily work integration practices, and how these differ from the `git merge` only actions visible in shared remote repositories. This way, we analyze here code integration scenarios that previous merge conflict studies were not able to analyze. We are then able to reveal here the *private* life of merge conflicts,[1] and contrast it with the more widely known *public* life of merge conflicts, which has often been exposed by GitHub based studies that appear in the literature [Accioly et al. 2018a,c; Cavalcanti et al. 2017, 2019; Ghiotto et al. 2020; Nguyen and Ignat 2018].

We do that by applying quantitative and qualitative techniques to answer our research questions. First, using tools that we developed, we collect and analyze the local git history reference logs, or reflogs)[2] of 95 private repositories owned by 61 developers. Our tools are able to identify the different kinds of integration scenarios mentioned before, and calculate their frequency and other derived measurements. Second, to understand which factors could influence the choice of git code integration commands and approaches, we interviewed 9 (of the 61) developers. Finally, the first author acted as participant observant [Sedano et al. 2017] in a professional git based development context where the author works. The author could observe the developer's routine related to the Git commands and the company policy towards the `git merge`.

According to our sample, *hidden* code integrations (performed by commands such as rebase and cherry-pick, or even git merges that were reverted or erased by subsequent squashes) are much more (approximately 6 times) frequent than potentially *visible* code integrations (performed by git merge and that might reach remote repositories). We have also observed conflict rates of up to 37% resulting from `git merge`, revealing developers that often have to locally deal with conflicts when invoking `git merge` in their private repositories. Since many merge conflicts are resolved locally before the developers synchronize their contributions to shared remote repositories, we can see no trace of them when analyzing only remote repositories. The other git code integration commands such as rebase and cherry-pick also lead to conflicts, with a lower bound rate of 10% in one private repository in our sample. Such conflicts are also missed by analyses that focus only on remote repositories, as rebase and cherry-pick scenarios are not visible at all with remote only history information. In summary, these results bring evidence that studies that focus only on GitHub project history might be losing integration conflict data by not considering the information in local repositories, reinforcing the need to consider both the *public* and the *private* life of merge conflicts.

2 MOTIVATION AND BACKGROUND

Each code change made by a developer has to be integrated into a project remote (possibly main) repository to be visible to the rest of the contributors. However, not all integration attempts are successful. Conflicts can arise when merging the code changes. Studies have reported that from 10% to 20% of all integration scenarios fail [Brun et al. 2011; Kasi and Sarma 2013], with some projects achieving rates of almost 50% [Brun et al. 2011; Zimmermann 2007], but this might vary depending on project practices and other factors [Accioly et al. 2018a; Apel et al. 2012], with some studies observing no conflicts resulting from invoking commands like `git merge` [Accioly et al. 2018a].

All of these empirical studies collect evidence by analyzing public Github repositories [Accioly et al. 2018a,b; Ahmed et al. 2017; Brun et al. 2011; Cavalcanti et al. 2015, 2017; Ghiotto et al. 2020; Kasi and Sarma 2013; Nguyen and Ignat 2018]. Despite bringing significant evidence for software engineering researchers and practitioners, these studies suffer from a common threat to validity: all the integration scenarios they consider come from shared remote project repositories. These repositories only track integration scenarios created explicitly by the `git merge` command. However, there are other ways to integrate code using Git [Chacon and Straub 2014], and they do not leave traces in the history of remote repositories. Besides that, other Git commands might rewrite repository history, which could then even erase `git merge` integration scenarios from the history of remote repositories. So remote repositories reflect only part of the code integration scenarios and merge conflicts that developers had to face during a project.

In fact, unlike the merge command, the following commands do not leave, or can erase, traces of code integration in the history of remote project repositories: (a) rebase, (b) cherry-pick, (c) squash, and (d) stash apply, as detailed next. The (a) rebase command is used to integrate changes from a branch by reapplying them on top of the committed changes from another branch, which then represents the new base for the integrated changes. Different from the merge command, which creates an additional (merge) with the integrated code, rebase creates new "clone" commits for each commit in the original rebased branch. This way the rebase command changes repository history keeping it linear and, therefore, easier to analyze, as we can see in Figure 1. The drawback of using it is not being able to identify, in the illustrated case, that Feature was actually independently developed and only later integrated to Master, as

[1] Hereafter we use the term *merge conflict* to denote any conflict reported during code integration time, no matter if using `git merge` or other git code integration commands.
[2] https://git-scm.com/docs/git-reflog

the light grey parts of the figure are not visible by inspecting the repository.

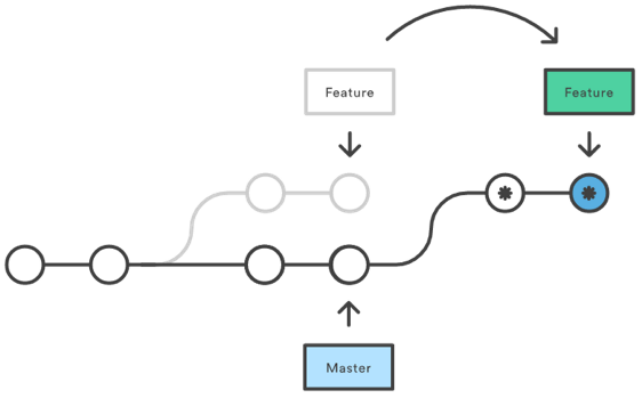

Figure 1: Rebase scenario, via Atlassian. (https://tinyurl.com/rebasegitcommand).

The (b) cherry-pick command is used to reapply selected changes (commits) from one branch into another branch. This is similar to rebase, but in a more constrained way, dealing with commits instead of branches. The result is also a linear history, leaving no trace that the reapplied commits are, in fact, copies of commits independently developed in another branch and only later integrated to the target branch.

The third command that can hide integration scenarios is (c) squash, which can, for instance, turn a sequence of commits into a single commit, joining their changes, but leaving no trace of the commits in the sequence. The specific scenario in which squash can hide code integration is when a merge commit is among the squashed commits. The result is the loss of the merge commit record, that is, the code integration trace that was left in the original history.

Finally, the Git stash feature allows one to temporarily save their changes in a stack, and later apply them to a branch with the command (d) stash apply, effectively integrating code that were independently developed.

All of these commands represent different code integration situations that could happen without relying on the merge command; none of them is visible in remote repositories. Consequently, by ignoring them, previous studies might be evaluating only a fraction of the code integration scenarios and the resulting merge conflicts. Besides that, since previous empirical studies only collect data from remote repositories, they might even miss conflicts that were detected and solved in a developer's machine, and never reached a remote repository.

For this reason, besides remote repositories, studies should also analyze local repositories of the developers involved in the project. Such data would enable the investigation of which integration scenarios developers use more frequently. This way, we would be able to understand the practices carried out by the development teams, and their motivations for the approaches chosen to solve the integration problems. Finally, there would be a possibility to study the understanding of the factors that may or may not influence current conflicts and create new solutions to avoid them.

3 METHODOLOGY

The main goal of this paper is to shed some light on the *private* life of merge conflicts. We focus on studying developers' local repositories to understand a diverse group of code integration scenarios, as some of them cannot be detected by analyzing only remote repositories. To achieve this goal, we develop tools to mine local code integration scenarios using Git, evaluate the frequency of each identified command and investigate the possible reasons that lead developers to choose particular code integration commands and strategies. We answer the following research questions:

- **RQ1**: How frequently developers integrate code using the merge command?
- **RQ2**: How frequently developers use commands that hide integration from the development history?
- **RQ3**: How frequently merge scenarios end up in conflicts? And how about the hidden integration scenarios?
- **RQ4**: Which factors can influence developer choice of Git integration commands and strategies?

To answer RQ1, RQ2, and RQ3, we measure the frequency of the integration scenarios we find in local repositories, and how often they lead to conflicts. We conduct a *quantitative* and *retrospective* study by collecting developers' local logs, and analyzing them using scripts that measure the frequency of different Git commands and the successful and failure code integrations scenarios. For answering RQ4, and understanding which project characteristics lead to a more significant difference between the public and private lives of conflicts, we carry on a *qualitative* study by performing semi-structured interviews with 9 developers, besides conducting a participant-observation process [Sedano et al. 2017] including the software companies that contributed to this research. In this qualitative study, the first author observed the git usage in the development environment of one participant company since the author has worked there since 2018. Also, this author exchanged emails with a developer from another software company participant asking about which git command is most used to integrate the code in the team and if there is company guidance related to this to the developers. Meanwhile, the second author collected the same information from a third software company participant during an informal conversation with the team leader while visiting.

3.1 Study Setup

Sample. To answer the research questions, we use a sample of 95 log files (reference logs, or reflogs)[3] belonging to 61 developers. We collected these files mainly through a website[4] that we developed for this study and that participants used to upload their files. We used e-mail and social media (mainly LinkedIn, Facebook, and Twitter) to propagate our website and collect voluntary submissions. Attempting to reach more participants, the authors use their institutional e-mail to contact managers from companies partners from the authors' university and from projects from others universities. The authors use their professional and personal profiles on social media to engage independent volunteers by sharing our research and our website.

[3]https://git-scm.com/docs/git-reflog
[4]https://tinyurl.com/privatelifemergeconflictsws

In the end, 4 software companies agreed to participate in this research, with their developers also submitting files using our website. One company is an open-source software company, where the first author has worked since 2018. Two companies have projects and research in partnership with the authors' university. Finally, the last software company is specialized in publishing game development for computer and mobile devices, and the research participant agreement was made through one employee who was a student of the third author.

Of 61 developers, 17 were independent volunteers, and 44 were volunteers from the software companies we contacted.

Scripts for log analysis. To automatically analyze our dataset, we implemented a script to identify and count the Git commands of interest to this study, as discussed in the previous section. The script takes reflog files as input and generates tables with the occurrences of the commands and related metrics. Since a developer could have worked on more than one project or repository, the script aggregates all the log information related to each developer. We decided to aggregate the data by the developer because it facilitates the evaluation from an individual perspective view.

Interviews. To complement the log analyses, we conducted semi-structured interviews with 9 developers from the same software company. This software company has a project in partnership with the authors' university. The interviews were conducted in one afternoon, and we created an interview script containing 19 questions to explore the developer experience using Git.

Although this is, in general, considered a small panel, it is adequate for our simpler goal of using the interviews to better understand the results of the log analysis. The interview script begins with questions to determine whether the developer knows and uses each Git command analyzed in the study and why they use (or not) the commands. It then moves forward to questions about how they usually integrate their code with the shared remote repository and if they had to deal with merge conflicts.

The last questions ask whether there is any configuration management and git usage policy or standard in the daily work of the software teams to integrate code using Git. The interviews were carried out individually, and we recorded each one of them. The following is an example of some of the questions we asked in the interview.

- Do you have experience with or know the rebase command?
- When using it, did you have to deal with any conflicts?
- Have you ever had to deal with conflicts resulting from the merge command? How did you solve them?
- Is there a workflow pattern in your company to work with Git?

The study sample, the script code, and the interview script are available online.[5]

Observations. Concerning the participant observation analysis, the first author collected notes while working as an engineer on one of the software company participants of this study and collected information by exchanging emails with one developer from another software company participant. The second author collected information from a different software company by having informal conversations with the team leader while visiting. All the information collected was related to analyzing if any company has a policy or guidance toward using git integration commands.

[5]https://tinyurl.com/privatelifemergeconflictsgh

3.2 Git Log Analysis

After studying the Git commands that hide code integration scenarios, we know that to answer our research questions we need to automatically identify these commands using data from local private repositories.

Command identification. Our strategy is to analyze the local history of the project, as captured by the local repository reflog file. This file contains information of all the reference related git actions performed locally by a developer in its local repository. This file is automatically saved and updated by Git as the user performs Git commands.

It is located in the repository's Git directory and can be found through the following directory path: ".git/logs/HEAD". It represents the reflog of the Git repository, and it is a record of all commits that were referenced in the repository. This file has an expiry subcommand that cleans up old or unreachable reflog entries. The reflog expiration date is set to 90 days by default, but this date can be configurable. Also, developers can access the reflog through the command `git reflog` in the terminal.

For each commit performed by the developer in a local repository, Git creates a unique key to reference it, known as an SHA-1 hash [Chacon and Straub 2014]. That way, Git does not lose track of the repository history. To inform the most recent commit, Git maintains a pointer known as "HEAD" [Chacon and Straub 2014]. This pointer indicates which branch the developer is in, and the last performed git command. HEAD is continuously updated automatically by Git, without manual intervention, and this information is captured in the reflog.

The reflog file consists of a sequence of lines, each representing a different git action performed by the developer. In each line, it is possible to observe the following data, as Figure 2 depicts: (1) the hash of the previous HEAD commit (first number sequence), (2) the hash of the current HEAD commit (second number sequence), (3) developer's Git name, (4) developer's Git e-mail, (5) the command performed and (6) linked message.

```
 (1) (2)  (3)            (4)                          (5)    (6)
0000 94c3 developer_name <developer_email> 153 -0300 clone: clone from git@github.com
94c3 ee7a developer_name <developer_email> 153 -0300 commit: First commit
ee7a 49cc developer_name <developer_email> 153 -0300 checkout: moving from master to test
49cc c046 developer_name <developer_email> 153 -0300 commit: Add class
```

Figure 2: Reflog file lines.

In Figure 2, we can see that the developer first cloned the repository in their local machine. After that, supposing this is the beginning of the project history, they created a first commit. Finally, a new branch was created and then the developer committed changes to that secondary branch.

As not all integration commands are clearly represented in this kind of log file, we manually analyzed a number of files to understand how to obtain the integration scenarios, including those that do not appear in the main project history. The commands explored

were the following: (a) merge, (b) rebase, (c) cherry-pick, (d) squash, and (e) stash apply. The first attempt at identification was made by reading the log files to differentiate the various types of Git commands. Soon after, tests were carried out in local repositories to reproduce the commands mentioned above to recognize how Git records them in reflog files.

After finishing this analysis, it was possible to identify the occurrence of the commands: (a) merge, (b) rebase, (c) cherry-pick, and (d) squash. The merge command and the cherry-pick command are identified as shown in Figure 3. The rebase command can be recorded in two ways: the direct form (Figure 3) and in an interactive (Figure 4). The rebase can also be used together with the pull action.

Regarding the squash command, Git does not record the squash with a single line command, but we can identify it through the interactive rebase. At the interactive rebasing, the developer can choose a list of commits to perform the squash action.

```
7047 b7a7 developer_name <developer_email> 153 -0300  merge featureA: Merge made by the 'recursive' strategy
b7a7 3914 developer_name <developer_email> 153 -0300  rebase: refactor project structure
3914 3f38 developer_name <developer_email> 153 -0300  cherry-pick: Add method
```

Figure 3: Reflog lines examples.

```
173f 9119 developer_name <developer_email> 153 -0300 rebase -i (start): checkout 91194
9119 e26d developer_name <developer_email> 153 -0300 rebase -i (reword): updating HEAD
e26d 928d developer_name <developer_email> 153 -0300 rebase -i (reword): Adding parameter
928d 54f5 developer_name <developer_email> 153 -0300 rebase -i (pick): Regrouping test
54f5 6938 developer_name <developer_email> 153 -0300 rebase -i (pick): Adding test
6938 6938 developer_name <developer_email> 153 -0300 rebase -i (finish): returning to head
```

Figure 4: Log line using interactive rebase.

The concrete cases illustrated in the figures follow an abstract and consistent pattern that applies to other instances of the invoked commands, allowing us to uniformly process reflog patterns by matching these patterns.

Regarding the stash apply command, we could not recognize it in the log file history. No trace of this command is left in reflogs. So we do not consider here all possible integration scenarios, but the ones we consider are still sufficient to support our conclusions.

Integration identification. Since all the identified Git commands were described in the previous section, we must discuss which integration scenarios we could extract from them.

The scenario that we are seeking is the integration one. Sometimes, the integration can act as a fast-forward action when we run an integration command. A fast-forward integration is when Git moves the pointer of the branch forward. An example is when you try to merge one commit with a commit that can be reached by following the first commit's history, and Git moves the pointer forward because there is no divergent work to merge [Chacon and Straub 2014]. This scenario is not interesting for our research because it is not about a merge scenario but a branch update. This means that there is nothing to integrate, and the branch pointer will move straight forward, resulting in a linear history. Since this scenario does not create a merge commit in the repository history, we don't count this scenario in our study, as desired.

When integrating code, it can be a successful or failed action. If the integration happens without conflict and it was not a fast-forward action, we count it as a successful scenario. When merge conflicts occur, the user has three options: try to fix the conflicts and continue the integration process; skip, meaning that you will bypass the commit that caused the failure; and abort, meaning that you will undo the operation. Regardless of the decision made in such a situation, the conflict makes us count this as a failed integration scenario. For each integration scenario recognized in the Git log analysis, we identify the action results to count how many times the integration was successful or not.

Concerning the successful integration, there are scenarios in which we could not differentiate them from the fast-forward because the log line of both scenarios had the same characteristics. It means that our script is counting both scenarios as successful integrations. This happened with the cherry-pick and the squash commands. The consequence is an overestimated result for these commands when measuring the successful integration.

Regarding the failed integration, there are scenarios in which we could not identify the abort or skip scenarios in the studied commands because the log file does not record this information. This happened with all the analyzed commands except for the rebase command done interactively. It means that our script is losing merge conflict data since it only identifies one of the three pieces of evidence representing that the command failed by a merge conflict. The consequence is underestimated result for most of the commands when measuring the failed integration.

A failure integration scenario of the merge and the cherry-pick commands can be identified with a single line. However, it is different with the rebase and squash commands because is identified in a block of commands. Figure 5 shows an example of these commands.

```
6d7c c802 developer_name <developer_email> 159 -0300  rebase -i (start): checkout c802
c802 445a developer_name <developer_email> 159 -0300  rebase -i (pick): Fix test and update snapshot
445a b593 developer_name <developer_email> 159 -0300  rebase -i (pick): Create tests
b593 34ae developer_name <developer_email> 159 -0300  rebase -i (squash): Create tests
34ae 4a2d developer_name <developer_email> 159 -0300  rebase -i (continue): Create Component
4a2d 4a2d developer_name <developer_email> 159 -0300  rebase -i (finish): returning to refs/heads/branchA
8754 7744 developer_name <developer_email> 159 -0300  rebase -i (start): checkout 7744
7744 8864 developer_name <developer_email> 159 -0300  rebase -i (continue): Add method
8864 0652 developer_name <developer_email> 159 -0300  rebase -i (continue): Fix condition
0652 8754 developer_name <developer_email> 159 -0300  rebase -i (abort): updating HEAD
```

Figure 5: Failed squash and rebase scenario.

Since all integration scenarios could not be identified correctly, some of our identified scenarios indicate an approximated result. Some results are overestimated, meaning that the actual result could be smaller. This happens with the successful integration scenarios. Meanwhile, others results are underestimated, meaning that the actual result could be bigger. This occurs with the failed integration scenarios.

Therefore, the number of occurrences will not be exact for some commands and analyses, which we can only provide estimations in this study. We can observe the summary of the estimations in Table 1.

4 RESULTS

Following the study design presented in the previous section, we analyze 95 Git log files from 61 developers to investigate the frequency of integration commands in their local repositories. From 61 developers, we collected a total of 34504 Git commands. We now present the results of our analysis for each research question.

Table 1: The identified integration scenarios of the Git commands. Arrows indicate whether the number is underestimated (↑, meaning the numbers should be bigger in practice) or overestimated (↓)

Git command	Successful	Failed
Merge	=	↑
Cherry-Pick	↓	↑
Squash	↓	↑
Rebase		
- Normal	=	↑
- Interactive	=	=
- Pull –rebase	=	↑

4.1 RQ1: How frequently developers integrate code using the merge command?

To answer RQ1, our script identifies merge occurrences for each developer. This frequency is the addition of the successful and the failed scenarios. Since our script could not identify the abort or skip scenarios in the studied commands (see Section Methodology), our result represents the underestimated frequency of merge occurrences compared to the developer's actual numbers.

From 34504 Git commands, 4131 of them were Git integration scenarios, representing 12% of all actions. Regarding the merge command, 548 scenarios were identified, representing only 13.3% of the 4131 integration scenarios. The major part of the integration scenarios, that is 86.7%, is carried out by git code integration commands such as `git rebase` and `squash`, which leave no trace on remote shared repositories.

We also observe that the percentage of merge commands substantially varies depending on the developer, as can be seen in the top part of Figure 6, which shows the distribution per developer. This graph presents the distribution of the percentage of merge command scenarios in relation to the total of integration commands, which includes both the visible integration commands (merge) and the hidden integration command (rebase, cherry-pick, and squash).

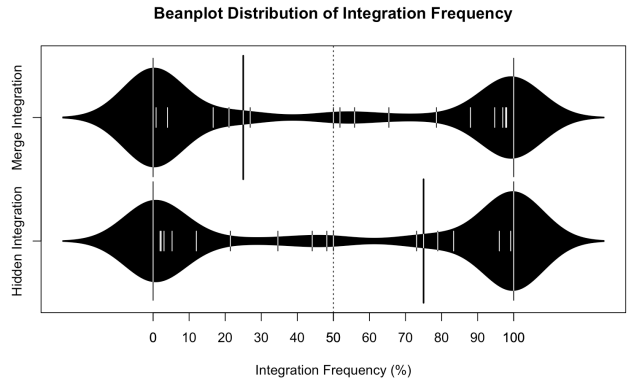

Figure 6: Beanplot Distribution of Integration Frequency.

In the top part of Figure 6, we can see that the merge action is more dispersed in the extremes of the graph, meaning that there are developers that often use this integration command, whereas others rarely use it. There are only a few points in the middle of the graph, showing that these developers do not seem to have a well-defined preference for visible or hidden integration commands.

Result 1: The frequency of code integration scenarios created by the git merge command is significantly smaller than the number of integration scenarios created by other git code integration commands, representing only 13.3% of the captured integration scenarios in our sample. However, it can substantially vary per developer, with some developers heavily relying on this command for code integration. This result represents an underestimated frequency of the merge occurrences.

4.2 RQ2: How frequently developers use commands that hide integrations from the development history?

Using the same method described to answer RQ1, we compute the frequency of the commands that hide integrations. In this question, our script considers the occurrence of the following commands: rebase, cherry-pick and squash. To answer this question, we must count the successful and failed integrations scenarios for these commands.

Since our script could not identify some scenarios, such as fast-forward integration, skip or abort scenarios, for all the hidden integration commands (see Section Methodology), our result represents an estimated frequency of hidden integration commands when compared to the developer's actual numbers.

From 34504 Git commands, 4131 of them were Git integration scenarios, representing 12% of all actions. If we analyze just the integration scenarios, the hidden scenarios represent 86.7%. Concerning only hidden integration scenarios identified, rebase was the most used command among the developers (65.7%), followed by cherry-pick (30.5%) and squash (3.8%). According to our sample, hidden integration is more frequent than the visible ones (13.3%) in the private repositories.

Although these commands frequencies are estimated numbers, our results maintain the same direction. Because, even in the worst scenario, the results would follow the same direction because the frequency of the rebase command is four times bigger than the frequency of the merge command in our sample. So the use of approximations, in this case, might affect the exact reported numbers, but not the overall conclusion and direction of the result.

This result varies per developer, having a similar behavior as the merge. As shown in the bottom part of Figure 6, the hidden integration distribution varies, showing more occurrences in the extremes sides. Comparing both distributions in the previous figure, we can see that they complement each other.

Result 2: According to our sample, hidden integration scenarios are more frequent than the visible ones (13.3%) in the private repositories, representing 86.7%. However, it can substantially vary per developer. The most used command among the hidden integrations is the rebase.

4.3 RQ3: How frequently merge scenarios end up in conflicts? And how about the hidden integration scenarios?

According to existing studies [Accioly et al. 2018a; Brun et al. 2013, 2011; Cavalcanti et al. 2015; Kasi and Sarma 2013] on merge conflicts and resolutions in public repositories, it has been reported that from 5% to 20% of all merges fail, with some projects achieving rates of almost 50% [Brun et al. 2011; Zimmermann 2007]. But, what about the merge conflict rate in the developer's local repositories?

Examining our sample, we have a total of 548 merges identified. Among these merges, 204 (37.2%) failed, that is, ended up having at least one merge conflict. Let us analyze the distribution of failed merges frequency by developer, as Figure 7 shows.

Analyzing Figure 7, we notice that the median of this graph is almost on axis 0 and many occurrences of failure appear throughout the distribution, meaning that developers had failed when merging in different proportions compared to each other. Remember that we do not count the merge conflict scenarios followed by the abort action due to our script limitation for identifying this action (see Section Methodology). Because of this, this result represents a lower bound number of failed merge scenarios.

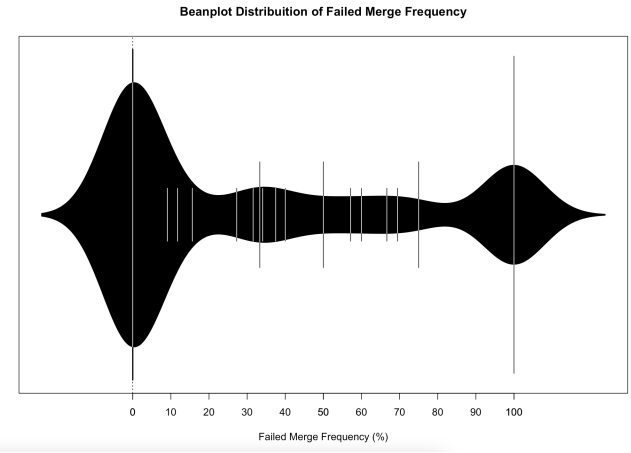

Figure 7: Beanplot Distribution of Failed Merge Frequency.

Our results show a higher merge conflict rate in the developer's local repository (37.2%) than studies that analyzed merge scenarios from the shared remote repository (5–20%). Still, this result should be interpreted as a complement to future analyses because we are just demonstrating that merge conflicts are happening locally and maybe with a higher frequency.

At the same time that merge is less frequent in local repositories, it tends to fail more often. Thus, this evidence suggests that studies might be losing merge conflict data by not analyzing the developer's private repository. Such merge commits might not appear in the shared remote repository history due to the use of Git commands that rewrites history, such as rebase and squash.

In addition, we have identified 3583 hidden integration scenarios in our sample. 405 of them ended up in conflicts, representing 11.3% of the total. Because of the script limitation, this result also represents the lower bound number of conflict scenarios. Hence, we can say that the hidden scenarios' situation is the opposite of the merge because it is more frequent in local repositories, but it tends to fail less.

This result represents an underestimated frequency of failed integration commands. But, since one of our research goals is to inform that studies are losing merge conflict data by not analyzing the developer repository, reporting a minimum frequency of code integration conflict would not invalidate our result. No matter how imprecise the approximation is in this case, it's still solid evidence that studies are losing merge conflict data by not analyzing the developer repository. The exact percentage of loss, however, is not accurate.

Result 3: Our results show that *hidden* and *visible* code integrations happen in the developers' local repository and both could end up in merge conflicts (hidden integrations with lower bound of 11.3% and visible integrations with lower bound of 37.2%). These conflicts are resolved locally before the developers synchronize their contributions to shared remote repositories, so we can see no trace of them when analyzing only remote repositories. In resume, these results bring evidence that studies that focus only on GitHub project history might be losing integration conflict data by not considering the information in local repositories, reinforcing the need to consider both the *public* and the *private* life of merge conflicts.

4.4 RQ4: Which factors can influence developer choice of Git integration commands and strategies??

Since the choice of git code integration commands and strategies can significantly impact the degree of integration scenarios and merge conflicts that are missed by code integration studies that focus only on shared remote repositories, it's important to investigate why developers choose a particular integration command or strategy in their work routine. In particular, it's important to identify which factors influence their choice. To understand that and answer this research question, we conducted semi-structured interviews with nine developers from the same company, collected additional information by exchanging emails with other developers, and performed a participant observation study with one software company.

We also performed an additional Git reflog analysis to complement the interviews by investigating secondary research questions that were derived from the analysis of the interview transcripts. This analysis groups the developers into two categories: those with a high visible integration frequency, and those with a high hidden integration frequency. We grouped only developers that use the same code integration strategy (visible or hidden) in more than 70% of the situations in which they have to integrate code. This way we leave out of the analysis only four developers that adopt a less unbalanced mix of strategies. After the grouping, we analyzed which company each developer works for, comparing it with the information collected from interviews and the participant observations notes. This way we can shed light on which aspects are more likely to affect code integration command or strategy choice.

Among the 61 developers from our sample, 25 have visible integration frequency ranging from 79% to 100%, constituting one of

the groups just mentioned. Of those 25, 19 developers only invoked visible integration commands, whereas 6 developers invoked both merge commands and hidden integration commands. We find that all employees of 2 specific software companies belong to this group. All 9 interviewed developers belong to this group, and all work at the same company. The interviews showed that many of them are not sufficiently familiar with the git integration commands that lead to hidden integration. They are also not aware of any company guidance for using specific git integration commands in their work routine. Regarding the other software company representing this group, we learn from the emails exchange that there is a company-wide recommendation for using the merge command, instead of rebase, for example. This is a light recommendation, though, not being mandatory to use the git merge command.

The second group consists of 32 developers having hidden integration frequency ranging from 73% to 100%. From this total, 26 developers only invoked hidden integration commands, and six invoked also the merge command besides the hidden integration commands. We find that all employees of 2 specific software companies belong to this group.

Based on our conversation with the team leaders, and experience from working as an engineer at one of the software companies participating in this study, we know that one of the companies has strict guidelines for not allowing the use of the git merge command. In contrast, the other company recommends using rebase instead of merge, but not in a mandatory way.

Outside the two polarized groups, only four developers performed both kinds of integration with balance. Those developers do not belong to any software company described before. Due to lack of information, we cannot inform the factors that influence their choice of balanced integration commands.

Result 4: Git experience and company guidance might influence which Git integration command or strategy a developer chooses. For example, a developer less fluent in Git may prefer to use the merge command, as it is often considered simpler.

5 DISCUSSION

Before this study, our main questions were whether hidden integration scenarios happened frequently, and why developers preferred to use such commands instead of `git merge`. Based on our results, hidden integration scenarios might occur even more frequently than visible ones; 6 times more frequently in our sample. This is an alarming difference. We expect that similar or even greater differences can be observed in contexts that recommend or demand the use of commands such as git rebase. We also bring evidence that hidden integration scenarios lead to conflicts as well, although in a smaller rate than can be observed from the visible scenarios in our sample.

Such expectations are reinforced by the analysis of the performed interviews, which show that the motivation to adopt different code integration strategies might come from personal Git experiences or simply from company guidance. This way, our study brings implications for both software engineering researchers and practitioners, as we detail further.

Previous empirical studies analyze merge conflicts characteristics based only on the history of the main GitHub repository [Accioly et al. 2018a,b; Ahmed et al. 2017; Cavalcanti et al. 2015, 2017; Ghiotto et al. 2020; Nguyen and Ignat 2018], that is, a public shared remote repository. Our results suggest that these studies might be missing both integration scenarios and merge conflicts. By not including the developer's local repositories, these studies are analyzing and reporting just a fraction of the actual integration scenarios and conflicts that were faced by developers working in those projects. So at least part of the results in those studies should be interpreted as lower bounds, for example, of how often integration conflicts occur in practice. A few studies even list that as a possible threat, but does not bring further evidence of the problem nor give an idea of the dimension of the threat, as we do here.

Detailed further work is then necessary to understand how precisely each previous work is affected by the findings we present here. Some might even not be affected at all, if their sample consists only of projects that demand developers to use only git merge for code integration; but, given how projects are often selected for these studies, we would expect such situation to be rare. Others might not be affected because their outcomes are not dependent on the existence of hidden integration scenarios; this is certainly not the case of outcomes that are related to conflict frequencies or causes, for example. To understand how each work is affected, a researcher has to analyze carefully the study outcomes and how they could be potentially impacted by the existence of hidden integration scenarios and conflict. To measure that with precision, in the cases of risk, one should have access to the reflog files from the developers of the analyzed repositories.

6 THREATS TO VALIDITY

Construct Validity. In order to detect hidden merge scenarios, we decided to analyze the developer's Git reflog files. Git creates the reflog file locally when the developer clones or starts a new repository using Git. However, the reflog file gets deleted when the developer deletes the project from the local machine. Even if the developer clones again the previously deleted project, the reflog file old content cannot be recovered. Besides that, Git limits the size of the reflog file. Meaning that it erases the reflog content at some point so that future Git actions can be saved. All those situations we describe mean that we might not be analyzing the full history of the developer's action in a repository. This, nevertheless, does not compromise our results showing that studies that focus only on shared remote repositories might be missing code integration scenarios and conflicts. In fact, since we might be considering only part of the local repository history, we know that the studies could be missing more than we report here, but not less.

Since our chosen strategy to identify hidden integration scenarios depends on reflog files, we created a script responsible for detecting these scenarios. Still, we could not detect all the successful and failed scenarios. To minimize the threat, we specify whether the numbers associated with a particular command are underestimated or overestimated. Nevertheless, even with estimated results, we could inform the ranges of command occurrence and therefore analyze potential worst and best situations.

Internal Validity. A potential threat is our approach to identifying hidden integration scenarios. We implemented a script responsible for determining the occurrence of those actions. The script

reads each reflog file line and detects which Git command that line corresponds to. We implemented this script based on our manual analysis while we were identifying the syntactic patterns of the Git commands in the Git log files. As a result, the script may not be counting the exact number of occurrences of the Git commands that hide integration. The reason can be a specific scenario that does not appear in the log file or a new pattern of a particular Git command that we did not implement in the script. We made manual analysis in different log files before running the study to reduce this threat.

External Validity. In this research, we collect our data via a website. This website was available for voluntary submission. This aspect could lead our study to be impacted by self-selection bias, which would only represent a particular subset of people who were comfortable participating. However, more than half of our sample came from developers teams from software companies we contacted with, and some of these developers had the option to self decide if they would participate because their manager asked them to participate. Therefore, we reduce the threat of self-selection bias. Another threat is that our semi-structured interviews were with developers from the same software company; so our answer for RQ4 is strongly associated with this company context.

Most reflog files collected in this study came from 4 different software companies described in the previous sections. Although our research does not restrict any Git projects from participating in the study, it was not possible to collect a significant number of log files. The reason provided by other companies was the sensitive information that could be in the commit message or even in the branch name provided by the developer. Even though our research would not analyze this kind of information, some companies refused to share their log files. Thus, our sample might not be large and representative enough to generalize our results. For future work, we could work on increasing the sample size.

7 RELATED WORK

Previous studies provide evidence about collaborative software development. As mentioned earlier, those studies focus on analyzing merge conflicts and their resolution. For example, Ghiotto et al. [Ghiotto et al. 2020] focus on analyzing merge conflicts of 2731 open-source Java projects that resulted in recommendations for future merge techniques that could help resolve certain types of conflicts. Likewise, other studies [Ahmed et al. 2017; Brun et al. 2011; Kasi and Sarma 2013; Le Nguyen and Ignat 2017] also studied large open-source projects with a similar purpose.

In contrast, Ji et al. [Ji et al. 2020] investigate the invisible integrations performed by the rebase command. This study examines how developers rebase their working branches in the pull requests. They collect 82 Java repositories from GitHub and identify 51,183 rebase scenarios from the pull requests of these repositories. According to this study, rebasing is widely used in pull requests because it can relieve the burden of reviewing changes and keep the commit history clean. Their result shows that conflicts arise in 24.3%-26.2% of rebases. However, they claim no significant difference between the possibilities of textual conflicts arising in rebases and merges. Furthermore, one of their main contributions is that developers adopt similar strategies shown in existing studies on merges when resolving textual conflicts on rebases.

Ji et al.'s study was the only one we found so far that analyzed hidden integration scenarios. In comparison with our research, our work goes further by identifying other hidden integrations scenarios beyond the rebase: cherry-pick and squash. Besides that, we collect our data through the developer's local history. Compared to our results, we collected and compared merge scenarios involving three different hidden integration scenarios. While the prior study only collects the rebase scenarios and compares their results with selected studies. One of their findings suggests no difference in the conflict rate between rebases and merges, showing that textual conflicts arise in 24.3–26.2% of rebases. This result diverges from ours because merge conflict (lower bound of 37% / ↑37%) is more frequent than the conflicts in the hidden integration scenarios (lower bound of 10% / ↑10%) in our sample. In this way, our results complement theirs since conflicts that developers resolve locally may not appear in the pull request history.

Related to the influence from choosing between different Git integration commands, our study found that company guidance can be an essential factor of influence because it can lead or prohibit the usage of a specific command. The other factor was personal Git experience, making less experienced developers choose easier commands, such as merge, instead of rebase. Meanwhile, Ji et al. investigate when and why developers decide to rebase branches in pull requests with a somewhat different focus.

Even though we have different approaches, we could be one of the first studies to research other ways to integrate code besides merge and enhance their importance to provide comprehensive insights on software merging.

8 CONCLUSIONS

In this paper, we report a quantitative and a qualitative study to measure the frequency of the integration scenarios in developer's local repositories. The main difference from previous merge studies was the detection of hidden integration scenarios, that is, scenarios that do not appear in the shared remote repository history. In order to do so, we collected a total of 95 logs from 61 different developers and analyzed the actions recorded in their local repository log files. We implemented a script that counts and computes the metrics used to answer our research questions. Additionally, we conducted semi-structured interviews with nine developers to learn more about their habits while using Git to merge code.

Our results indicate that *hidden* code integrations are much more (approximately 6 times) frequent than potentially *visible* code integrations (performed by git merge and that might reach remote repositories). Additionally, we observed conflict rates of up to 37% resulting from `git merge`, revealing developers that often have to locally deal with conflicts when invoking `git merge` in their private repositories. Since many merge conflicts are resolved locally before the developers synchronize their contributions to shared remote repositories, we can see no trace of them when analyzing only remote repositories. The other git code integration commands also lead to conflicts, with a lower bound rate of 10% in one private repository in our sample. Such conflicts are also missed by analyses that focus only on remote repositories. In resume, these results bring

evidence that studies that focus only on GitHub project history might be losing integration conflict data by not considering the information in local repositories, reinforcing the need to consider both the *public* and the *private* life of merge conflicts.

Regarding the factors that could influence the choice between the integration ways of code, we find that Git experience and company guidance can affect which Git integration command a developer chooses. We plan to explore how the developer's local actions could influence the conflicts from the project's shared remote repository in future work.

ARTIFACT AVAILABILITY

Our research data is available in a GitHub repository.[6]

ACKNOWLEDGMENTS

We thank the practitioners for participating in the study and the anonymous reviewers. We also thank INES (National Institute of Software Engineering), FACEPE (IBPG-28-1.3/20 and IBPG-567-1.03/22) and CNPq (309235/2021-9).

REFERENCES

Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. 2018a. *Understanding Semi-Structured Merge Conflict Characteristics in Open-Source Java Projects (Journal-First Abstract).* Association for Computing Machinery, New York, NY, USA, 955. https://doi.org/10.1145/3238147.3241983

Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. 2018b. *Understanding Semi-Structured Merge Conflict Characteristics in Open-Source Java Projects (Journal-First Abstract).* Association for Computing Machinery, New York, NY, USA, 955. https://doi.org/10.1145/3238147.3241983

Paola Accioly, Paulo Borba, Léuson Silva, and Guilherme Cavalcanti. 2018c. Analyzing Conflict Predictors in Open-Source Java Projects. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 576–586. https://doi.org/10.1145/3196398.3196437

Iftekhar Ahmed, Caius Brindescu, Umme Ayda Mannan, Carlos Jensen, and Anita Sarma. 2017. An Empirical Examination of the Relationship between Code Smells and Merge Conflicts. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Markham, Ontario, Canada) *(ESEM '17)*. IEEE Press, 58–67. https://doi.org/10.1109/ESEM.2017.12

Sven Apel, Olaf Leßenich, and Christian Lengauer. 2012. Structured merge with auto-tuning: balancing precision and performance. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 120–129. https://doi.org/10.1145/2351676.2351694

Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured merge. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. ACM Press. https://doi.org/10.1145/2025113.2025141

Christian Bird and Thomas Zimmermann. 2012. Assessing the Value of Branches with What-If Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 45, 11 pages. https://doi.org/10.1145/2393596.2393648

Yuriy Brun, Reid Holmes, M.D. Ernst, and David Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *Software Engineering, IEEE Transactions on* 39 (10 2013), 1358–1375. https://doi.org/10.1109/TSE.2013.28

Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 168–178. https://doi.org/10.1145/2025113.2025139

Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. 2015. Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–10. https://doi.org/10.1109/ESEM.2015.7321191

Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (oct 2017), 27 pages. https://doi.org/10.1145/3133883

Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. 2019. The Impact of Structure on Software Merging: Semistructured Versus Structured Merge. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1002–1013. https://doi.org/10.1109/ASE.2019.00097

Scott Chacon and Ben Straub. 2014. *Pro Git* (2nd edition ed.). Apress.

Jônatas Clementino, Paulo Borba, and Guilherme Cavalcanti. 2021. *Textual Merge Based on Language-Specific Syntactic Separators*. Association for Computing Machinery, New York, NY, USA, 243–252. https://doi.org/10.1145/3474624.3474646

Léuson Da Silva, Paulo Borba, and Arthur Pires. 2022. Build Conflicts in the Wild. *J. Softw. Evol. Process* 34, 4 (apr 2022), 28 pages. https://doi.org/10.1002/smr.2441

Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André van der Hoek. 2020. On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub. *IEEE Transactions on Software Engineering* 46, 8 (2020), 892–915. https://doi.org/10.1109/TSE.2018.2871083

Mário Luís Guimarães and António Rito Silva. 2012. Improving Early Detection of Software Merge Conflicts. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 342–352.

Tao Ji, Liqian Chen, Xin Yi, and Xiaoguang Mao. 2020. Understanding Merge Conflicts and Resolutions in Git Rebases. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 70–80. https://doi.org/10.1109/ISSRE5003.2020.00016

Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive Conflict Minimization through Optimized Task Scheduling. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 732–741.

Hoai Le Nguyen and Claudia-Lavinia Ignat. 2017. Parallelism and conflicting changes in Git version control systems. In *IWCES'17 - The Fifteenth International Workshop on Collaborative Editing Systems*. Portland, Oregon, United States. https://hal.inria.fr/hal-01588482

Wardah Mahmood, Moses Chagama, Thorsten Berger, and Regina Hebig. 2020. Causes of merge conflicts: a case study of ElasticSearch. 1–9. https://doi.org/10.1145/3377024.3377047

Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. 2017. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 467–478. https://doi.org/10.1109/ICSME.2017.53

T. Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462. https://doi.org/10.1109/TSE.2002.1000449

Hoai Le Nguyen and Claudia-Lavinia Ignat. 2018. An Analysis of Merge Conflicts and Resolutions in Git-Based Open Source Projects. *Computer Supported Cooperative Work (CSCW)* 27 (12 2018). https://doi.org/10.1007/s10606-018-9323-3

Yuichi Nishimura and Katsuhisa Maruyama. 2016. Supporting Merge Conflict Resolution by Using Fine-Grained Code Change History. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 661–664. https://doi.org/10.1109/SANER.2016.46

D.E. Perry, H.P. Siy, and L.G. Votta. 1998. Parallel changes in large scale software development: an observational case study. In *Proceedings of the 20th International Conference on Software Engineering*. 251–260. https://doi.org/10.1109/ICSE.1998.671134

Anita Sarma, David F. Redmiles, and André van der Hoek. 2012. Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes. *IEEE Transactions on Software Engineering* 38, 4 (2012), 889–908. https://doi.org/10.1109/TSE.2011.64

Todd Sedano, Paul Ralph, and Cécile Péraire. 2017. Software Development Waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 130–140. https://doi.org/10.1109/ICSE.2017.20

Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. 2020. Detecting Semantic Conflicts via Automated Behavior Change Detection. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 174–184. https://doi.org/10.1109/ICSME46990.2020.00026

Alberto Tavares, Paulo Borba, Guilherme Cavalcanti, and Sergio Soares. 2019. Semistructured Merge in JavaScript Systems. 1014–1025. https://doi.org/10.1109/ASE.2019.00098

André van der Hoek and Anita Sarma. 2008. Palantir: enhancing configuration management systems with workspace awareness to detect and resolve emerging conflicts.

Thomas Zimmermann. 2007. Mining Workspace Updates in CVS. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. 11–11. https://doi.org/10.1109/MSR.2007.22

[6] https://tinyurl.com/privatelifemergeconflictsgh