# Merge Nature: a tool to support research about merge conflicts

Luan Reis Ciribelli
João Pedro Lima*
luanreisciribelli@ice.ufjf.br
joao.carvalho.lima@ice.ufjf.br
Universidade Federal de Juiz de Fora
Juiz de Fora, Minas Gerais, Brasil

Heleno de S. Campos Junior
helenocampos@id.uff.br
Universidade Federal Fluminense
Niterói, Rio De Janeiro, Brasil

Márcio de Oliveira Barros
marcio.barros@uniriotec.br
Universidade Federal do Estado do Rio de Janeiro
Rio de Janeiro, Brazil

André van der Hoek
andre@ics.uci.edu
University of California, Irvine
Irvine, U.S.A.

Leonardo Gresta Paulino Murta
leomurta@ic.uff.br
Universidade Federal Fluminense
Niterói, Brazil

Gleiph Ghiotto
gleiph@ice.ufjf.br
Universidade Federal de Juiz de Fora
Juiz de Fora, Minas Gerais, Brasil

## ABSTRACT

The modern software development model requires concurrent programming to create more complex programs. As such, existing approaches are able to track and merge the changes made on a code repository, allowing more developers to work collaboratively. Despite these solutions, 10% to 20% of all merges result in conflicts, such as when changes are made on the same file region. Existing studies have analyzed, prevented, predicted, and even automatically resolved merge conflicts. However, they are usually focused on a specific language or on language-independent information from the source-code lines involved in the conflict. This situation hinders the generalization of results. To enrich the information to be used by further studies about merges, we present the Merge Nature Tool, which recreates all project merges and extracts characteristics from conflicts, such as the resolutions adopted by developers, and the language constructs involved in the conflict for Java, C++, and Python files. The tool was evaluated regarding reproducibility and generalization. It was able to reproduce 87.75% of the results of a previous study. The differences can be explained by newly introduced features. On the other hand, regarding generalization, the tool was used to analyze a set of 100 merges from Python projects. The merges were analyzed manually to ensure the reliability of the results. We found that 100% of the data matched the expected result. We expect this tool to facilitate future work regarding merge conflicts. For instance, it may allow comparing the characteristics of merge conflicts from different programming languages.

## KEYWORDS

Git, version control systems, software merge, merge conflicts

## 1 INTRODUCTION

The complexity of software projects has been increasing over the years, requiring an increasing number of developers coding over the same set of artifacts in parallel. In this context, version control systems (VCS) allow multiple developers to work simultaneously through branches, which can receive contributions from different developers and have specific goals. However, their contributions must be combined using the merge feature provided by the VCS.

A merge can report conflict when it is impossible to combine the contributions performed in different branches automatically [6]. For instance, a conflicting merge occurs when multiple developers change the same region of one file in parallel. In such a situation, the merge cannot automatically combine the contributions, and a developer must be assigned to resolve the conflict manually [5].

Previous studies report that 10% to 20% of the merges end up in a conflict [2, 3, 8]. This shows how frequent merge conflicts are in the life of software developers. Despite the broad array of programming languages available to developers, most studies [1, 6, 7, 9, 13] have investigated merge conflicts on Java projects only, neglecting other programming languages since each programming language has its grammar that should be considered during a tool implementation.

This paper extends Ghiotto et al. [6] approach for extracting information about merges that would demand significant work from researchers considering the processing and implementation effort. The tool automatically recreates all merges for a given software project and extracts the number of merge conflicts, the number of conflicting files and chunks, the most common conflict types, and the resolutions adopted by the developers for all programming languages. Additionally, it is able to extract the language constructs involved in such conflicts for Java, C++, and Python files since they depend on the language's grammar.

The initial evaluation of the tool considered two main goals: reproducibility and generalization. The reproducibility shows the developers' decisions match in 87.75% of [6] analyses while the language constructs match in 67.27% of the cases. On the other hand, the generalization evaluation shows that 100% of the merges analyzed are according to the expected results.

This paper is structured into six sections, besides this introduction. Section 2 introduces relevant concepts to understand the features of the tool. Section 3 presents an overview of the tool. Section 4 presents the main assets of the tool and the software technology used to implement it. Section 5 present initial results. Section 6 presents research and tools that have similar goals. Finally, Section 7 summarizes the paper's contributions and future work.

## 2 SOFTWARE MERGE AND PARSER

VCSs enable developers to manage the projects' history and work collaboratively with other developers. To manage the project's history, developers usually change a set of software artifacts and perform a commit to store a new version in the repository. Additionally, many developers can work on the same project and create

*Both authors contributed equally to this research.

branches to contribute in parallel. Eventually, these branches are also integrated using the merge feature [11].

A merge can be performed automatically or it can report conflicts that should be manually resolved by developers [5]. Using the merge feature implemented in Git [4], a merge is performed automatically when developers change different areas of a software artifact in parallel. On the other hand, when changes are performed over the same region of a software artifact, the VCS reports a conflict that the developer must resolve manually. It is worth mentioning that a conflict may be composed of one or more conflict files that may contain one or more conflict chunks. We also refer to "conflict chunk" as "chunk" in the remainder of the paper.

Listing 1 shows the source code of a chunk extracted from the merge *b14ca5* of the project ANTLR4 [1]. This merge[2] is the result of the integration between the versions *05b0f6* and *b80ad5* and reported eight conflict files, where the file `RuleStartState.java` has the chunk presented in Listing 1. During the merge, the changes are identified and integrated considering the most recent version in common, named merge-base. In this merge, the merge-base is the version *f7d0ca*.

The chunk has three markers: begin, separator, and end. They are represented by the following symbols, respectively: <<<<<<<, =======, and >>>>>>>. The code lines between the begin marker and the separator are called Version 1 (V1). The lines between the separator and the end marker are called Version 2 (V2). The code before the begin and after the end markers is called context, or, more specifically, prefix and suffix, respectively.

**Listing 1: Conflicting chunk of merge b14ca5 from ANTLR4.**

```
1  public final class RuleStartState extends ATNState {
2   public RuleStopState stopState;
3  <<<<<<< HEAD
4   public boolean isPrecedenceRule;
5  =======
6
7   @Override
8   public int getStateType() {
9    return RULE_START;
10  }
11 >>>>>>> b80ad5052d1b693be6e5c0a2be8bf87e15b86f18
12 }
```

As discussed earlier, each conflict must be manually resolved by the developers. For example, the adopted resolution for the chunk displayed in Listing 1 is presented in Listing 2. In this case, the developer's decision is a concatenation of the code of V1 and V2.

The source code of one chunk can be parsed to extract the language constructs in V1 and V2. It is important to identify the language construct involved in the source code of each chunk. Moreover, it is a challenging task since each programming language, such as C++, Python, and Java, has its own set of language constructs defined in the grammar, demanding a new implementation for each programming language. For instance, the conflict depicted in Listing 1 parsed with a Java grammar results in the following language constructs: field declaration (line 4), annotation (line 7), and method declaration (lines 8-10) that comprises return statement and variable (line 9). In this paper, we call language-dependent analyses that depend on the grammar of the programming language. The other analyses are called language-independent analyses.

**Listing 2: Resolution adopted by the developers to resolve the conflict chunk displayed in Listing 1.**

```
1 public final class RuleStartState extends ATNState {
2  public RuleStopState stopState;
3  public boolean isPrecedenceRule;
4
5  @Override
6  public int getStateType() {
7   return RULE_START;
8  }
9 }
```

## 3 APPROACH

The Merge Nature tool aims at supporting researchers to extract the data from merges in the history of projects coded in C++, Java, and Python. Figure 1 shows an overview of this process that encompasses four stages: **pre-processing**, which receives a software repository and extracts information, such as the metadata of versions and the chunks; **chunk analysis**, which extracts the language constructs, the developer's decision, and some metrics, such as the number of conflicting merges and the distribution of developers' decisions; **storage**, which stores the results of the analysis; and **visualization**, which is responsible for showing the information extracted during the merge analyses to a researcher.

The `pre-processing` stage extracts the metadata of merges and the versions involved and verifies if the merge is in conflict. In Figure 1, each merge is represented by a red rectangle, representing a conflicting merge, or a green rectangle, representing a non-conflicting merge. The metadata extracted from merges is focused on four versions: the merge result, the two parents, and the merge-base, which is composed of their identifiers, their authors and committers, and the commit's date and message. Another piece of information is extracted by replaying the merges to identify whether a given merge is a conflicting merge. If the merge is non-conflicting, the analysis is finished. Otherwise, the analysis continues to identify the conflicting files and chunks. Considering the conflict files, the conflicts are generally on the content level. However, there are conflicts in the level of files and directories. It is possible when someone deletes a file and someone else adds lines in the same file, when developers change the directory of files, and so on. The approach also deals with these **conflict file types**.

The `chunk analysis` stage extracts the information from the chunks formerly identified, such as the language constructs, the developers' decisions, and metrics such as the number of lines of code in V1 and V2. Such information is collected for all chunks in the project's history under analysis. The language constructs are extracted using the language's grammar to parse the source code and select the language constructs of the lines inside the chunks.

The developers' decisions are extracted using textual analyses, considering the code of the chunks and their resolution. Considering that the contexts are not changed during the chunk resolution, the textual analyses consist of identifying the sequence of lines between the prefix and the suffix. If the sequence of lines is equal to the code of V1, the developer's decision is **Version 1**. Ghiotto et al. [6] defined other developers' decisions that are used in this

[1] https://github.com/antlr/antlr4

[2] https://merge-nature.netlify.app/501687/antlr4/b14ca56441196d63b8974455c0050bfaee4cb3a4/b14ca56441196d63b8974455c0050bfaee4cb3a4

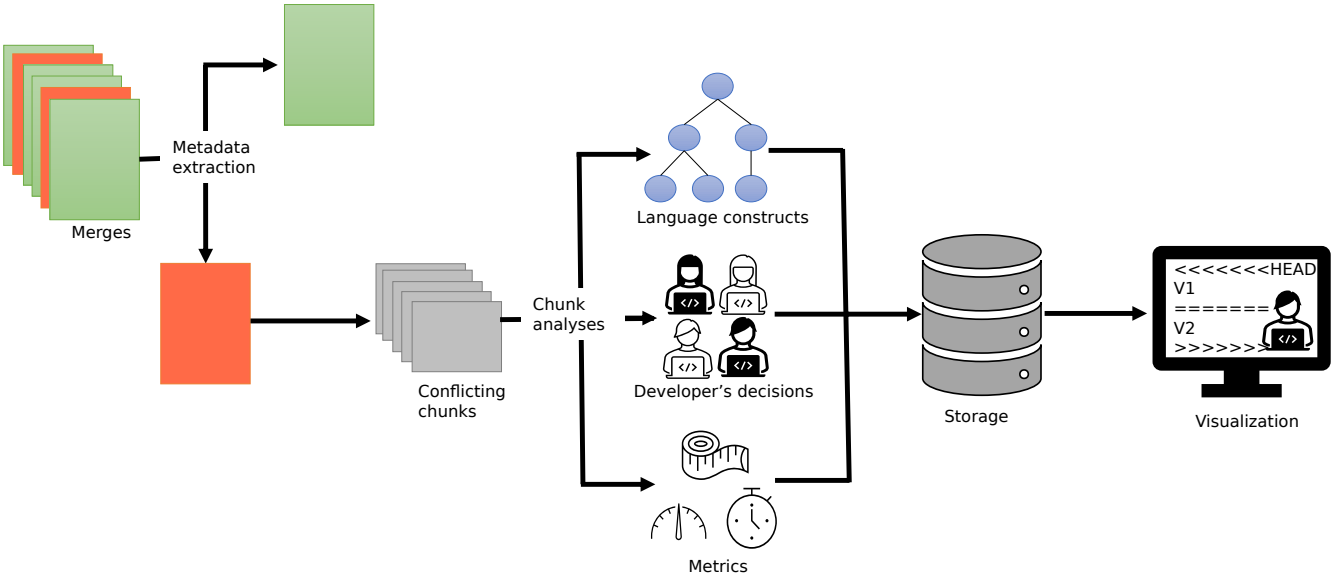


**Figure 1: Merge Nature tool architecture.**

approach as **Version 2**, which is analogous to Version 1; **Concatenation**, which occurs when the sequence of lines is represented by the concatenation of V1 with V2 or V2 with V1; **Combination**, which takes place when the sequence of lines is a subset of V1 and V2 combined in any order; **None**, which happens when the sequence of lines is empty; and **New Code**, which ensues when the developer adds one or more lines that were not included in V1 or V2 in the sequence of lines. For improving the classification, this approach added the following developers' decisions: **Deleted file**, which occurs when the developer deletes the file containing the conflict during the merge resolution; **Postponed**, which takes place when the developer leaves at least one marker of the chunk in the resolution; and **Imprecise**, which occurs when one of the context lines was changed by the developer during the chunk resolution. In that case, it is not possible to determine the resolution precisely.

Finally, metrics, such as the number of lines in a chunk, are extracted from the source code of the chunks. This stage is responsible for most of the analyses performed by this tool, and it may consume most of the processing time depending on the number of merges and the size of the files involved.

The `storage` stage stores the analyses performed in the previous stages to build a collection of analyses, allowing the users to compare them or share the results, which can be stored in a text file or a database. Finally, the `visualization` stage is responsible for presenting the available information and allowing researchers to navigate over the merges, files, or chunks they are interested in.

## 4 IMPLEMENTATION

The Merge Nature tool extracts information about merges by replaying them in the same sequence they appear in the project history. The current implementation[3] supports projects that are versioned in Git and coded in C++, Java, and Python. The decisions of the VCS and the programming languages supported are based on the size of the community that uses Git and the top three object-oriented languages on TIOBE index [4]. Moreover, ANTLR [5] was elected as a parser because it has a set of implemented grammars that enable future extensions.

To analyze a repository, the tool requires two inputs: the path to a Git repository and the number of context lines that should be displayed for each conflicting chunk. A low number of context lines may facilitate the visualization of the chunks, and more context lines may give more insights when someone is trying to understand the conflict. Based on these inputs, the tool identifies and processes the repository's merges, and reports the results on a screen with the data obtained during the steps of pre-processing and chunk analysis. This screen, depicted in Figure 2, is divided into five regions that are explained in this section.

Region 1 shows the analyzed project, including its name, the URL from where it was cloned, and the organization that maintains it. In this case, the project used is `ANTLR4`, which was cloned from GitHub and is maintained by the ANTLR organization. Region 1 has a button entitled *Metrics* that shows the main statistics for the project, such as the number of chunks resolved with each developer decision, the number of files by conflict type, and the number of chunks having each language construct.

Region 2 shows a table with all the merges found by the analyses and information about the conflicts. The table has columns that show the merges' hashes, the number of chunks, and whether this merge is in conflict. For instance, we can observe that merge *b14ca56* is in conflict and has 12 chunks, while merge *6d1d0e0* is non-conflicting.

The tool allows researchers to filter the merges according to their goals. This feature is available by pressing the *Set Filter* button

[3]https://figshare.com/articles/software/MergeNature-main_zip/20644443

[4]https://www.tiobe.com/tiobe-index/

[5]https://www.antlr.org/

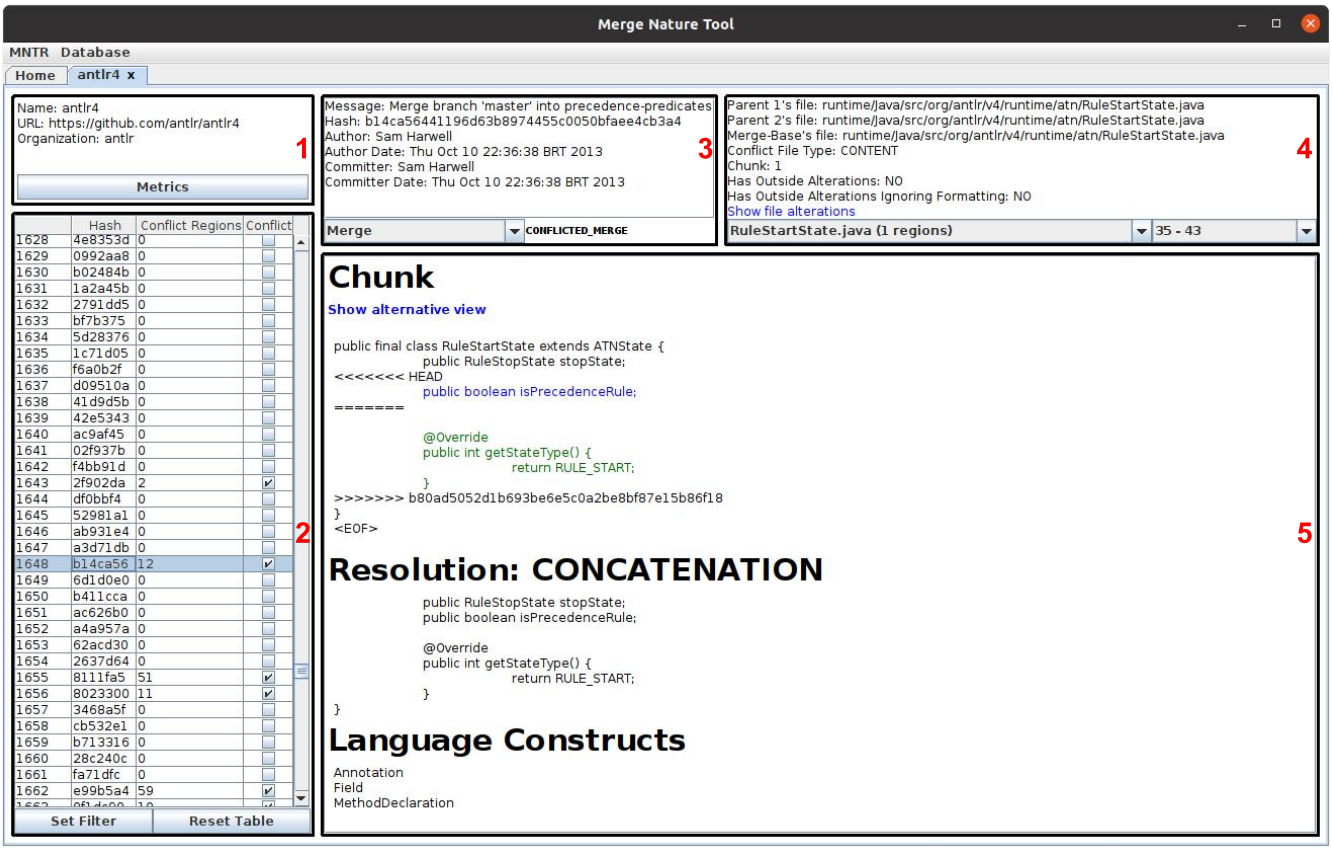


**Figure 2: Main screen of Merge Nature tool.**

in Region 2. For instance, it is possible to order the merges by the number of chunks and filter the chunks resolved using specific developers' decisions. To improve the user experience, they can filter the current content to refine their results, or they can reset the table contents to the initial state by using the *Reset Table* button.

After selecting the set of merges for subsequent analysis, the user can view their characteristics by double-clicking each merge. For instance, by double-clicking the row *b14ca56*, Region 3 shows information about the commits involved in this merge, such as the commits' message, their committers, and their dates. The figure shows the information related to the merge's version, but it is possible to see the information of parents 1 and 2, and merge-base versions by selecting another option in the drop-down menu.

When the selected merge is in conflict, the data for each conflicting file is shown in Region 4. Such data includes the file path in each commit, the conflict file type, and the number of chunks. This region also shows whether the conflict resolution has altered source code outside the chunks, which can be used to evaluate if the merge resolution is restricted to the chunks or if other regions of the code were affected when a developer resolved the merge. Furthermore, the user can click the blue hyperlink *Show file alterations* to open a new screen that shows all changes made by the developer during the conflict resolution process, including changes performed outside of chunks. Lastly, the user can select one chunk file for visualization using the left and the right drop-down menus of Region 4.

Region 5 shows the chunk information selected in the drop-down menu of Region 4. The first information is the chunk content, which presents the code from V1 highlighted in blue and the code from V2 in green. Next, the region shows the developer's decision and the resolution adopted by the developer to address the conflicting chunk. If the user needs to analyze the developer's decision manually – for instance, in Imprecise cases – the *Show file alterations* feature in Region 4 shows the changes performed to resolve the merges in the conflicting file. Region 5 also shows the language constructs present in V1 and V2. It is worth mentioning that when there are nested language constructs, we collect only the most external as in Ghiotto et al. [6]. In this case, the chunk represented by Region 5 has the language constructs annotation (line 7), field (line 4), and method declaration (lines 8 to 10).

## 5 EVALUATION

The Merge Nature tool was evaluated considering two main goals: reproducibility and generalization. The reproducibility was evaluated to verify if the developers' decisions and language constructs of chunks match with the results of Ghiotto et al. [6]. On the other hand, the generalization was evaluated manually to identify if the results for another language, Python, are reliable.

To evaluate reproducibility, we selected five Java projects that have 137 merges that were analyzed by Ghiotto et al. [6] and 498 Java chunks. The tool was able to reproduce the [6] classification of developers' decisions in 87.75% of the chunks. The main difference is explained mainly by the introduction of *imprecise* which represents 9.39% of the chunks. Regarding the language constructs, 67.27% of the chunks had identical language constructs. However, the grammar used in the current version of the tool treats the language constructs in different ways. For instance, Java *Annotation* is treated as one of the modifiers of *Method declaration* and *Attributes*, while it was an external language construct in the previous evaluation. Despite the differences, this evaluation shows that the current tool can reproduce the results of Ghiotto et al. [6] and some divergences should be addressed in future work.

The generalization evaluation was performed on 100 merges extracted from five Python projects. One author manually inspected the chunks of the merges and compared the expected results with the result reported by the tool. Regarding developers' decisions, 100% of the chunks matched the results expected by the author. Considering the language constructs, the tool found three chunks with a parser error caused by syntax problems. All the other cases were a match between what the author manually identified and what the tool identified, which indicated the power of the tool for Python. It also indicates that the tool can replace manual classification, which can be error-prone.

## 6 RELATED WORK

This section presents approaches that deal with merge analyses. Owhadi-Kareshk and Nadi [12] proposed a tool named MERGANSER to support researchers in collecting data regarding merge from Git repositories. The tool can replay merge scenarios to identify merge conflicts and collect data, such as the conflicts files and chunks. MERGANSER was used to collect data from 267,657 merge scenarios from 744 GitHub repositories, comprising seven different programming languages. However, the tool is only able to extract language-independent information, such as those provided by Git.

Brindescu et al. [2] discuss a combination of three tools to identify merge conflicts, extract relevant metrics, such as the presence of bug fixes and new features in merges, and track statements across different versions of a source code. The authors collected data from 143 open-source projects comprising 36,122 merges. However, it is not clear how the data is presented to the user and the collected data is language-independent.

Menezes et al. [10] proposed a tool named MacTool (Merge Attribute Collector). It identifies whether a repository has specific types of files for eight different programming languages: JavaScript, Python, Java, PHP, C#, C++, C, and Ruby. The tool also collects information, such as the number of merges and conflicted merges of a repository. However, while the tool can identify information of files coded in a specific programming language, it does not consider the language representation or grammar.

Ghiotto et al. [6] developed a tool that collects merge conflict metrics and classifies the conflict resolution adopted by the developers. They used the tool to collect and analyze merge conflicts data of 2,371 open-source Java projects. As discussed previously, the current tool is an extension of the former one, given that their tool can only analyze Java files, which does not allow researchers to extract language-dependent information from conflict merges in other programming languages, and extract a subset of our developer decisions, which can be imprecise in some cases. For instance, when the developer postpones a conflict resolution and Ghiotto et al. [6] classifies as new code resolution.

## 7 CONCLUSION

This paper presents a tool to support studies of software merge conflicts dealing with information from three programming languages. Besides supporting the extraction of merge information, the tool helps its users to visualize and analyze merge conflicts. The tool was evaluated considering the power of reproducibility of the results of Ghiotto et al. [6] and the generalization in a set of Python projects. The initial results are encouraging and open the way for new analyses. As future work, we propose to increase the coverage of programming languages through an extensible architecture that would allow researchers to analyze language-dependent information. Other future work would be to use the results to research and improve studies on the conflict merge analysis.

## ACKNOWLEDGEMENTS

We would like to thank BIC/UFJF, VIC/UFJF, CAPES, and CNPq for their financial support.

## REFERENCES

[1] 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 *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 58–67.

[2] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering* 25, 1 (2020), 562–590.

[3] 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*. 168–178.

[4] Scott Chacon and Ben Straub. 2014. *Pro git: Everything you need to know about Git* (second ed.). Apress. https://git-scm.com/book/en/v2

[5] Catarina Costa, Jair Figueirêdo, João Felipe Pimentel, Anita Sarma, and Leonardo Murta. 2019. Recommending participants for collaborative merge sessions. *IEEE Transactions on Software Engineering* 47, 6 (2019), 1198–1210.

[6] Gleiph Ghiotto, Leonardo Murta, Marcio Barros, and Andre Van Der Hoek. 2018. 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 (2018), 892–915.

[7] Mário Luís Guimarães and António Rito Silva. 2012. Improving early detection of software merge conflicts. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 342–352.

[8] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: proactive conflict minimization through optimized task scheduling. In *Proceedings of the 35th ICSE*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE / ACM, 732–741.

[9] Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. 2019. Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 151–162.

[10] José William Menezes, Bruno Trindade, João Felipe Pimentel, Tayane Moura, Alexandre Plastino, Leonardo Murta, and Catarina Costa. 2020. What causes merge conflicts?. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*. 203–212.

[11] Tom Mens. 2002. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 28, 5 (2002), 449–462.

[12] Moein Owhadi-Kareshk and Sarah Nadi. 2019. Scalable software merging studies with merganser. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 560–564.

[13] Bowen Shen, Cihan Xiao, Na Meng, and Fei He. 2021. Automatic detection and resolution of software merge conflicts: Are we there yet? *arXiv preprint arXiv:2102.11307* (2021).