b
UNIVERSITÄT
BERN

An empirical study on the human role in merge conflict resolution

Bachelor's Thesis

Severin Buchser

from

University of Bern

Faculty of Science, University of Bern

29.12.2022

Prof. Dr. Timo Kehrer

Alexander Boll

Software Engineering Group

Institute of Computer Science

University of Bern, Switzerland

Abstract

Modern software engineering often employs a version control system (VCS) to collaborate on a project, but collaboration inevitably leads to merge conflicts, hindering the automation of merging. Many attempts to predict, prevent and characterize conflicts have been made, but more research on the topic of merge conflict resolution needs to be conducted. Therefore, we test the feasibility of an approach that aims to improve the resolution of existing merge conflicts to add to this research field. To investigate the feasibility of the approach, we studied how often humans intervene in the resolution of merge conflicts for 8500 open-source projects from GitHub, containing a total of 119 794 conflicting merges, 217 712 conflicting files, and 297 126 conflicting chunks. We found that (i) 34.11 % of conflicting merges were resolved by picking one of the conflicting chunks in each conflict. Further, (ii) 46.29 % of conflicting files were resolved using the above method, and (iii) 80.63 % of conflicting chunks, i.e., single conflicts, were resolved using the abovementioned method. Additionally, we found that (iv) language or popularity of the projects does not influence the resolution process, but (v) the number of conflicts in a conflicting merge, respectively, in a conflicting file does play a role. We also studied two possible indicators, commit count and contributor count, on their predictive power on conflicting merges, and the results show that (vi) they are insignificant. We concluded that the approach to improve the resolution of conflicting merges is infeasible because the rate at which the conflicting merges were resolved by picking an existing chunk is too low in combination with performance problems concerning a growing number of conflicts.

Contents

1 Introduction

Concurrent collaboration is a necessary aspect of software engineering. A modern and widely spread solution to foster concurrent collaboration is a VCS. As described by Zolkifli et al. [29], a VCS has many advantages, for example, keeping track of older versions of projects and enabling parallel collaboration, which significantly increases productivity as well as efficiency. A VCS automates many procedures, like reverting to an old version of a project or, as mentioned, the collaboration between developers, allowing concurrent changes to be integrated into the project. The VCS used in this thesis is called Git[1]. Git is used by Google[2], Microsoft[3] and other well-known software companies.

The process of integrating concurrent changes is called merging, and is mostly automated, using many different merge techniques which focus on conflict prevention, see Section 3.3. As opposed to syntactical or semantical merge techniques, Git uses a text-based approach which allows projects of all types to involve Git. However, the downside of a text-based approach is the production of additional false-positive conflicts [23]. The reason for this behavior is that no text-based technique can prevent or resolve all conflicts because there is no heuristic to let an algorithm decide on the correct resolution. For such a heuristic to exist, the developer's intent (semantical correctness) or, at the very least, compilability (syntactical correctness) would need to be known. Git does not support either of those and thus cannot choose the correct resolution, prompting the developer to intervene. This problem does not only apply to the merge technique of Git. Today, the perfect merge technique does not exist with the current state of technology, and contributors must evaluate and resolve conflicts manually at some point, no matter the merge technique.

Nelson et al. [22] described the manual conflict resolution process in five steps. Initially, in the development process (step one, development), developers face a structured workflow until they encounter a merge conflict. Resolving conflicts is an interruption of the workflow at which a developer has to take a step back and recognize the conflict (step two, awareness). Then, the information about the conflict and its origin has to be gathered (step three, planning) to determine how to create or choose the correct resolution (step four, resolution). This process becomes difficult quickly with a bigger conflict count [6]. The last step Nelson mentioned is the evaluation of the resolution (step five, evaluation). The contributors

[1] `https://git-scm.com/` accessed on Nov 13, 2022
[2] `https://www.google.ch/` accessed Dec 2, 2022
[3] `https://www.microsoft.com/` accessed Dec 2, 2022

need to check if the resolution is syntactically and semantically correct. The time and effort those tasks consume should be prevented as much as possible, resulting in more comfort when merging and increasing productivity and efficiency [10].

In this work, we want to investigate the feasibility of an automatic merge conflict resolution approach. This approach tries to suggest the best-suited resolution, easing the strain on developers. The idea of this automatic resolver is to generate all possible canonical resolutions of a conflicting merge, see Section 2.5. Each resolution could then be evaluated based on additional criteria like compilation success and test success rate. Take the language Java[4] for example. Java's compiler can detect syntactical errors in a program. Java also has many different testing frameworks whose tests can detect a few semantical errors. With build automation tools like Maven[5], an interface is provided to run the aforementioned tasks (compilation and testing). By tapping into the interface and extracting information about compilation and testing, the state of a program can be evaluated: Programs that do not compile are unusable but could be ranked by the number of errors encountered. The ones that do compile can be ranked by their test scores. The higher the score, the higher it ranks in usability. Based on the rankings of the resolutions with the highest scores, a suggestion on what resolution is best suited could then be provided to the developer.

Since only the feasibility of this approach is discussed in this thesis, it is essential to find out how humans resolve merge conflicts. There are many studies examining the structure of conflicts and how they could be prevented ([12], [7], [16]), but only a few regarding the frequency of human intervention in the merge conflict resolution process [12]. To study human behavior, we collected data from different open-source projects from GitHub[6], categorized by criteria like language or popularity. We developed two tools for the data collection process: The first tool[7] is used for project collection and uses GitHub[6] and its API, the GitHub API[8]. The second, more elaborate tool[9] analyzes the merge conflicts of those projects. When evaluating the results of the analysis, the focus lies on answering the following two research questions:

RQ1: How often do humans intervene in merge conflict resolution for a VCS such as Git?
RQ2: What factors influence the number of merge conflicts and their resolutions in a Git project?

First, we cover background knowledge, necessary to understand our further work in Chapter 2, Theory. Next, we refer to related work, covering additional useful information for this thesis in Chapter 3, Related Work. Our methodology is then explained in Chapter 4, Methodology: What data is used and how the data is extracted and evaluated. This extracted data and the evaluation of it is then presented in Chapter 5, Results and discussed in Chapter 7, Discussion. A conclusion of our work is then given in Chapter 8, Conclusion and Future Work.

[4] `https://www.java.com/` accessed Dec 2, 2022
[5] `https://maven.apache.org/` accessed Dec 2, 2022
[6] `https://github.com/` accessed Dec 2, 2022
[7] `https://gitlab.inf.unibe.ch/sb18n092/projectlistgenerator` accessed Dec 18, 2022
[8] `https://docs.github.com/de/rest?apiVersion=2022-11-28/` accessed Dec 2, 2022
[9] `https://gitlab.inf.unibe.ch/SEG/theses/mergeconflictresolution`, accessed Dec 13, 2022

2 Theory

In this chapter, we will give theoretical background to the general purpose and usage of a VCS and cover its merge process, i.e. the inner workings of the merge process and the formation and resolution methods of merge conflicts.

2.1 Version Control System

In many software projects, a VCS is used. There are many VCS like Git[1], SVN (Apache Subversion)[2], and Mercurial[3]. To give theoretical background, we use terminology and examples from Git, as it is the VCS used in this work. Git is an open-source VCS with many features, as mentioned in Chapter 1, and is perfectly suited for this thesis. For additional reading material on Git, see the book by Chacon and Straub called "Pro Git" [9]. Git uses an updated version of this book in their documentation[4].

The usage of a VCS becomes more important in collaborative, complex, or big projects, as it documents all changes the project undergoes. A change can be categorized into deletions, additions, or modifications which are applicable to a whole file or a few lines within a file. Additionally, changes can be grouped into a so-called *commit* to further improve the overview of the project's history.

Another essential feature of a VCS is the collaboration of different contributors, which is made easy with a tree-like structure consisting of commits as vertices with multiple *branches*. The VCS creates multiple branches with parallel project variants. Using branching, each contributor is able to create new commits and append them to one branch in particular. Those changes won't translate into the other branches - especially since the main version of the project is unaffected by the changes. The tree-like structure implies that one way or another all branches originate from one root. Because of this fact, every pair of branches have at least one *common ancestor*, which itself is a commit of the project.

[1] `https://git-scm.com/` accessed on Nov 13, 2022
[2] `https://subversion.apache.org/` accessed Dec 14, 2022
[3] `https://www.mercurial-scm.org/` accessed Dec 14, 2022
[4] `https://git-scm.com/book/en/v2` accessed Dec 14, 2022

A *version* of a project is therefore the replay of changes since the initialization of the project up to a certain commit within a certain branch. The recovery of such a version is made possible with the above information provided by the VCS. We will sometimes refer to a version by the identification/name of the commit at which the replay of changes stopped.

2.2 Merging

A *merge* attaches the changes of one or more branches to another branch, such that the branch which is merged into will also contain the changes of the other branches. In this thesis, we limit ourselves to merges with two branches since the so-called octopus merges with more than two branches are hardly ever used, see Section 6.1. Let us call the branch which is merged into the *merging branch*. The most recent common ancestor in a merge is called the *merge-base*. Only the changes after the merge-base commit will be translated into the merging branch because all the changes before the merge-base are already adopted in both branches.

The merge is performed by a *merge algorithm*, of which many exist, using different approaches and techniques. But since we are using the default merge algorithm provided by Git, we only focus on Git's default merge algorithm. Further examples of algorithms can be found in Section 3.3.2. Furthermore, the default merge algorithm of Git only allows three-way-merging and is text-based, meaning that the merge is performed solely on textual analysis, not considering other factors like syntax.

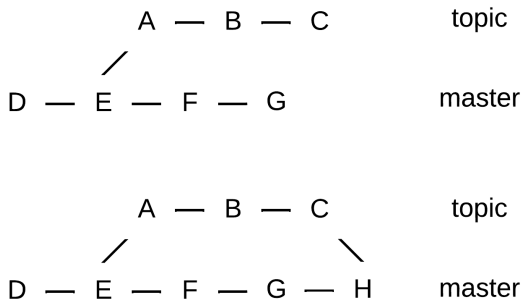

Figure 2.1: Merge example from the official Git merge description with two branches, "topic" and "master". Branch "topic" at commit *C* is merged into "master" at commit *G* producing the merged commit *H*.

To explain how this algorithm works, we use the short example directly from the official git merge description[5]. Look at the example tree-like structure shown in Figure 2.1. It has two branches called "master" and "topic". The "master" branch splits into two branches at commit *E*, which is the merge-base in this example. Assume that at commit *E*, there are two files called "file-1.txt" and "file-2.txt", both containing two lines of content. Both branches receive changes until branch "topic" gets to commit *C* and branch "master" gets to commit *G*. In both branches, both files receive their respective changes. See Figure 2.2 for the contents of the files in the different commits.

[5] `https://git-scm.com/docs/git-merge/` accessed on Nov 2, 2022

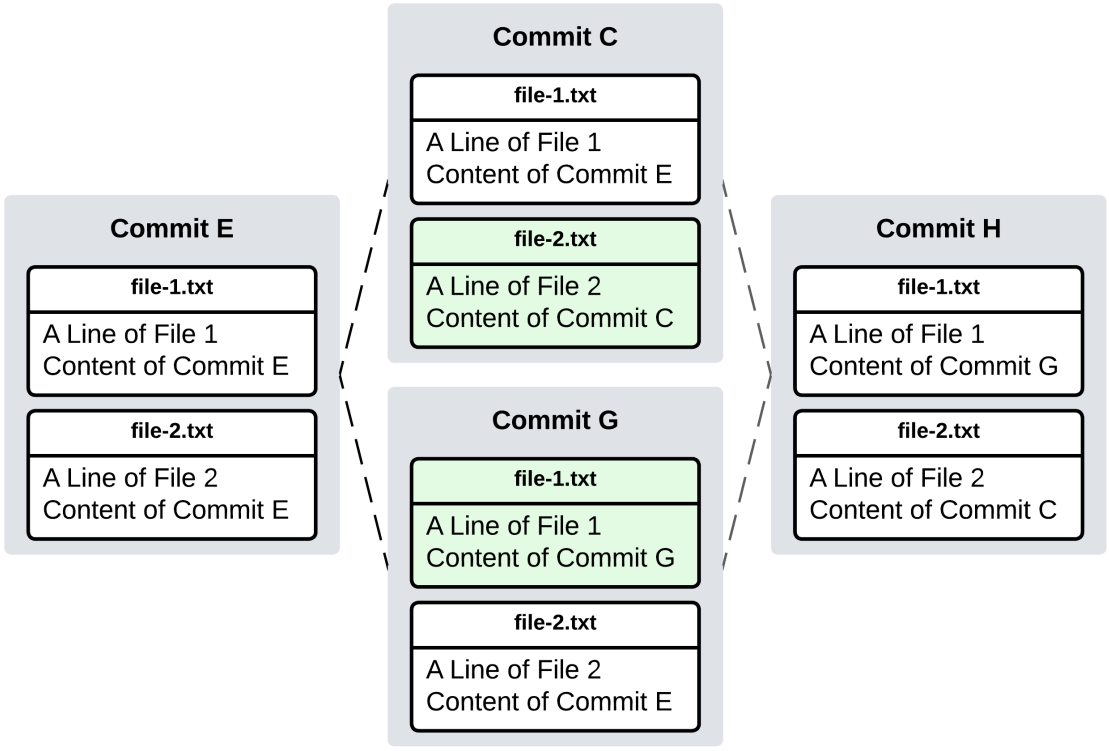

Figure 2.2: Example file contents of the merge shown in Figure 2.1. Files with a green backgrounds contain changes. The, for this merge, irrelevant commits, *A*, *B*, *D*, and *F* are not included in this figure.

At this point, the ”topic” branch is merged into the ”master” branch. Notice that in branch ”topic” only ”file-2.txt” received a change, and in branch ”master” only ”file-1.txt” received a change. The merge combines all changes, and the result will be the commit *H* with files shown in Figure 2.2 as well. One can see that now the change made in branch ”topic” in ”file-2.txt” and the change made in branch ”master” in ”file-1.txt” are both adopted at commit *H*, see Figure 2.2.

2.2.1 Merge Output

In Git, a merge operates on each file individually without any knowledge of the other files. For this reason, we will focus only on an individual file, in other words, the pair of files to be merged. The merge splits the two files into sections of text named *merge chunks* and stores them in an ordered sequence. Usually, a merge chunk consists of a few lines of text. How exactly the chunks are produced is decided by the so-called *merge strategy*. There are many strategies, for example, ORT[6] (Ostensibly Recursive’s Twin) of Git. The concrete implementation of this merge strategy is out of scope for this thesis. Here, we only concentrate on the output produced by the merge.

[6] `https://git-scm.com/docs/merge-strategies#Documentation/merge-strategies.txt-ort/` accessed Nov 22, 2022

After the chunk sequences for each file are created, the merge concatenates the chunks of both sequences in such a way that the resulting sequence of chunks contains the changes of both files as well as the unchanged content. The sequence preserves the order in which the merge will concatenate the text from the chunks to form the merged file.

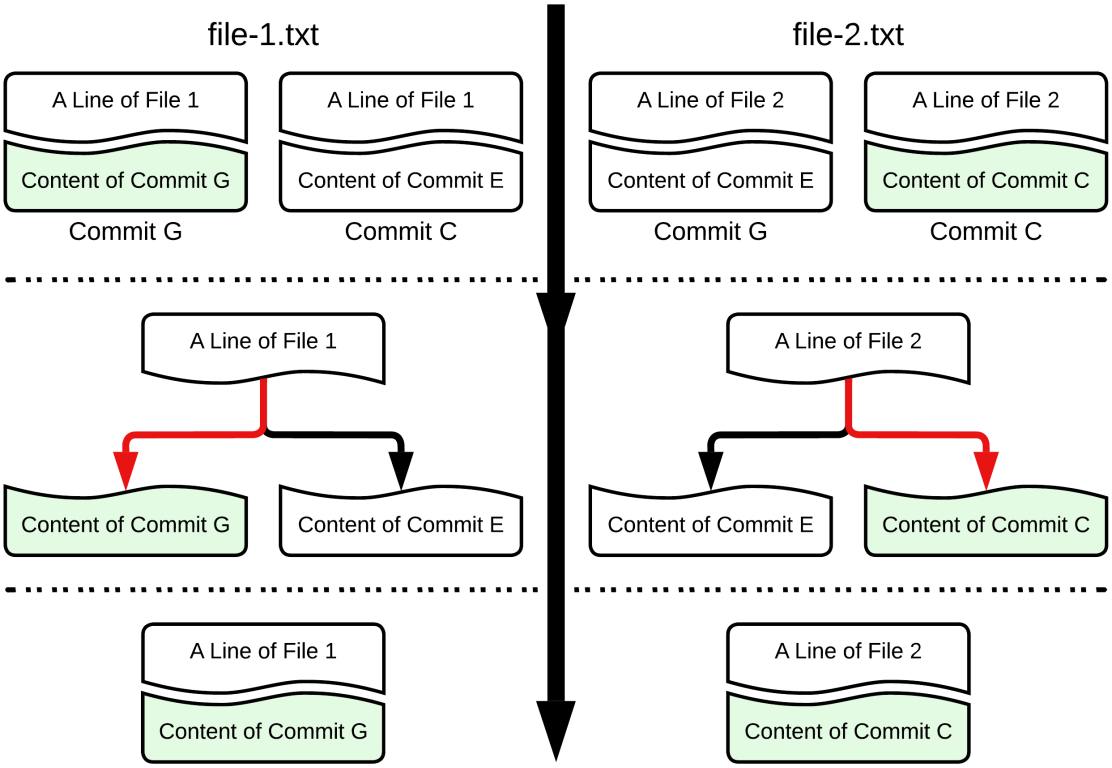

Figure 2.3: Chunk sequences produced by the merge-algorithm containing one conflict. Chunks with a green background contain changes (in this case modifications). The top four sequences are the unmerged sequences of the files and in the middle are the combined sequences. On the bottom are the merged sequences.

In the case of the above example, the merge will first split each file into chunks like shown in Figure 2.3. Then the algorithm detects that the first chunk in every file does not contain a change, so it must be the same chunk in both commits. Thus, the chunks are combined to form one single chunk, representing both. Since the subsequent two chunks in both sequences do not contain the same content, the merge-algorithm will pick the one with a change. One of the chunks must have a change, else they would be the same chunk and could be combined. In the end, this produces a new chunk sequence containing the unchanged content as well as the changed content. The case for addition and deletion works similarly. The chunks are indexed so that the merge detects chunks that are either added or deleted.

2.3 Merge Conflicts

A merge does not always succeed. The problems that can occur whilst merging are called *merge conflicts* or simply just *conflicts*. Merge conflicts arise when one or more contributors make different changes in the same location in two separate branches. This produces two merge chunks for the same location containing different changes. Those two chunks are called conflicting chunks (CCs), and they represent exactly one merge conflict. The first CC is the merge chunk of the merging branch and the second CC is the merge chunk from the other branch. It cannot be automatically inferred which chunk should be used in the merged file because the contributors' intent is unknown to a machine. In this context, the intent of the contributor refers to semantical correctness. Other aspects, like the compilability of the program (syntactical correctness), are also not considered by the merge algorithm of Git. Therefore the developer will be prompted to intervene.

Let us introduce some more notation: A file containing one or more merge conflicts is denoted as a conflicting file (CF). A merge containing at least one CF is called a conflicting merge (CM), and therefore a CM also has at least one merge conflict. See Figure 2.4.

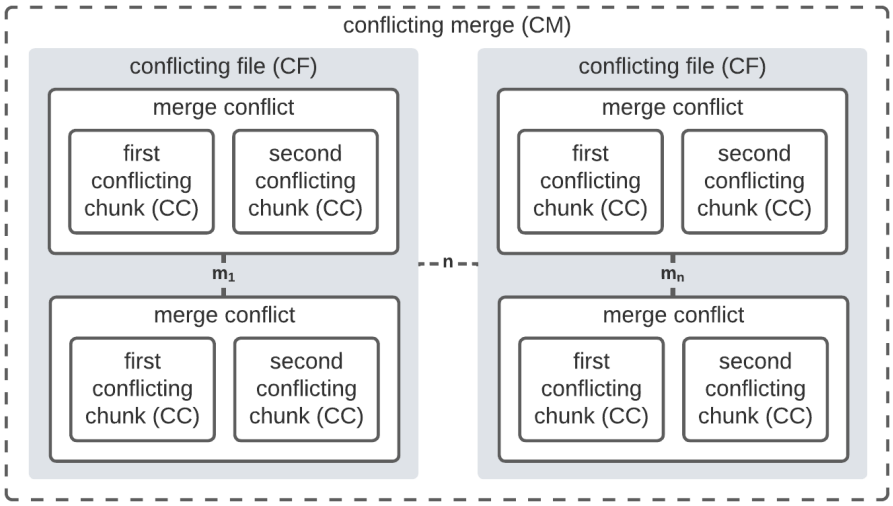

Figure 2.4: Composition visualization of a CM, CF and CCs. There are n CFs within the CM and m_i merge conflicts within the i-th CF.

Again, to better explain how merge conflicts arise and how they are treated, an example is best: Assume the same example as in Section 2.2, with the difference that in commit *C*, "file-1.txt" now also contains a change, see Figure 2.5. Notice that "file-1.txt" has different changes in the same location in both branches. This time, a merge conflict will arise when merging. Even though the merge has conflicts, it can still be partially merged: All non-conflicting merge chunks and files will be merged normally, and for the CCs, both will be included in the final merged chunk sequence. The chunk sequences of the merge are shown in Figure 2.6.

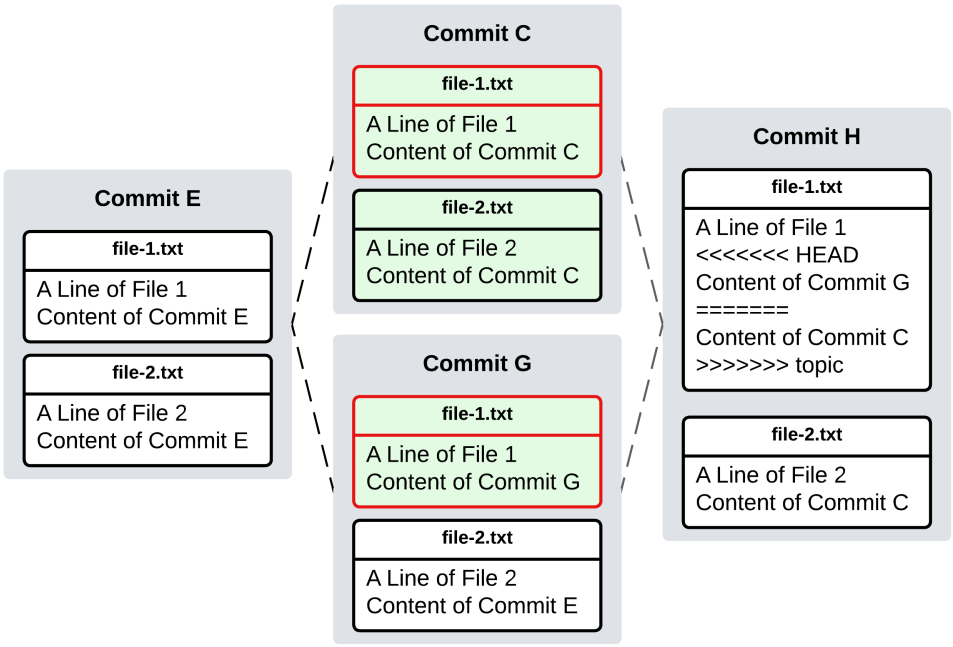

Figure 2.5: Example file contents of the merge shown in Figure 2.1. Files with a green background contain changes. The two files with red borders have conflicting changes. The, for this merge, irrelevant commits, *A*, *B*, *D*, and *F* are not included in this figure.

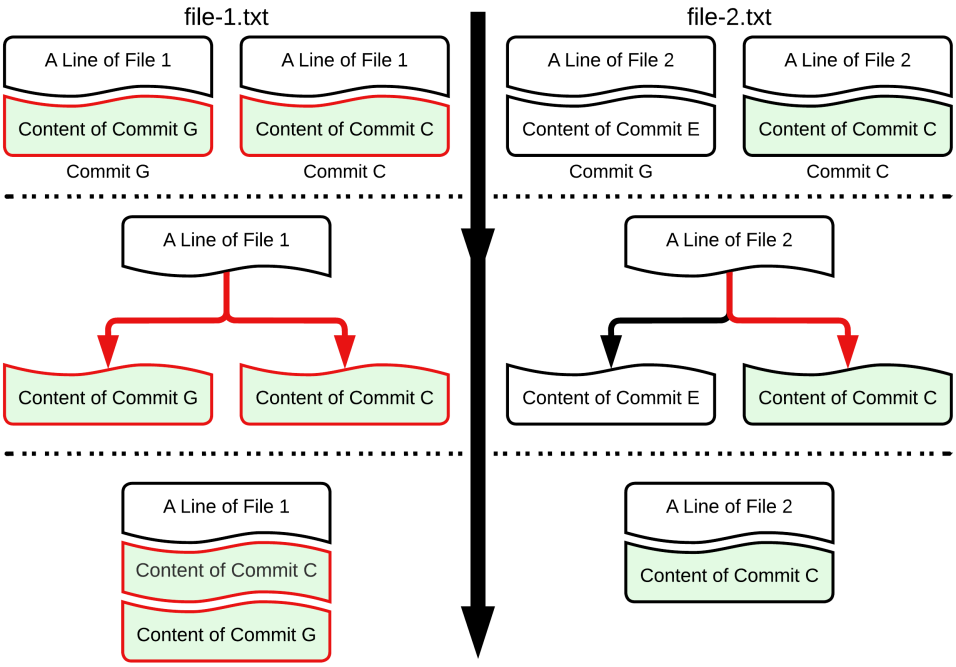

Figure 2.6: Chunk sequences produced by the merge-algorithm containing one conflict. Chunks with a green background contain changes and chunks with a red border mark conflicting chunks. Here ”file-1.txt” contains conflicting merge chunks. The top four sequences are the unmerged sequences of the files and in the middle are the combined sequences. On the bottom are the (partially) merged sequences.

For the final step of producing the merged file, when confronted with a CF, the merge-algorithm produces a partially merged file where both CCs are included, as shown in Figure 2.5, commit *H* ”file-1.txt”. The lines of a partially merged file not part of any merge chunk are called the conflict markers. Listing 1 shows the conflict markers: The beginning marker, ”<<<<<<<”, together with the branch which is merged into, ”HEAD”, is on line 1. Line three contains the separator, ”=======” and line 5 contains the ending marker, ”>>>>>>>”, with the branch which is to be merged into ”HEAD”, ”other-branch-name”.

```
1 <<<<<<< HEAD
2 Content of first CC
3 =======
4 Content of second CC
5 >>>>>>> other-branch-name
```

Listing 1: Example of conflict markers. Lines 1, 3, and 5 contain the conflict markers.

2.4 Resolving Merge Conflicts

If conflicts appear, they must first be resolved by one of the contributors, or else the merge will not go ahead and abort, reverting to before the merge. Note that it is possible to create commits with conflict markers still present since the conflicts can be marked as resolved without actually resolving them. However, this would result in unusable code, which is not the intended way.

To resolve a conflict, a developer can choose one of two ways: The first method is manually picking one of the two CCs. Carrying on the example from the previous section, the contributor can pick the CC from commit *G*, which is the first CC, or the contents in commit *C*, which is the second one. Another way to resolve the conflict is by writing a new resolution and disregarding all other resolutions. This approach also includes committing files with conflict markers still present. Note that a developer could also introduce new changes outside of the conflicting chunks. This issue will be discussed in Section 6.1.

Let us introduce some further notation: The first method is denoted as the *canonical resolution* and the second one as the *manual resolution*. Additionally, a resolution for a complete merge is denoted as a conflicting merge resolution (CMR), and it consists only of the corresponding conflicting file resolutions (CFRs), which themselves consist of conflicting chunk resolutions (CCRs). For each of the three granularities, all conflicts within the domain (merge, file, or chunk) have to be resolved:

- CMR all conflicts of the CM are resolved
- CFR all conflicts of the CF are resolved
- CCR the conflict belonging to the CCs is resolved

To simplify the resolution process, most integrated development environments (IDEs) provide a graphical interface where the conflicts can be resolved, see Figure 2.7.

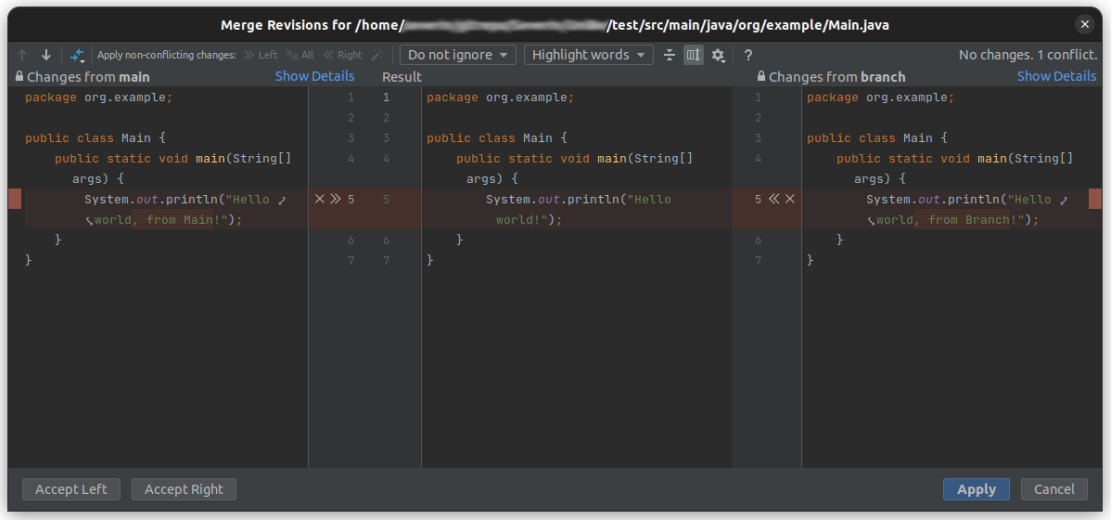

Figure 2.7: A screenshot of the merge conflict resolution dialogue of the IntelliJ IDEA. On the left is the ”HEAD” branch, in the middle a live preview of the merged file, and on the right is the other branch. One can edit the files directly (manual reolution) or can accept the ”left” or ”right” version with the buttons in the bottom left corner (canonical resolution).

2.5 Canonical Conflicting Merge Resolutions

One way of programmatically resolving merge conflicts is by using the canonical resolution approach. The produced CMR is denoted as canonical conflicting merge resolution (CCMR). Generating one CCMR implies the generation of one canonical conflicting file resolution (CCFR) for each CF within the CM. This means that there are $\prod_{i=1}^{n} k_i$ possible CCMRs, where k_i is the number of resolutions for the i-th CF and n is the number of CFs.

Assume that for conflicting file j there are m_j conflicting chunks, then there are $k_j = 2^{m_j}$ CCFRs for file j. This number comes from the fact that for each conflict there are two possible CCs one can choose from. On a side note, in theory there are infinitely many CCMRs one can generate, since the two chunks can be concatenated in infinitely many ways. To demonstrate, denote the first CC as CC1 and the second one as CC2. The following concatenations can be created:

- No CC (empty)
- CC1 or CC2
- (CC1 or CC2) and (CC1 or CC2)
- (CC1 or CC2) and (CC1 or CC2) and (CC1 or CC2)
- (CC1| CC2)* (generalization as a regular expression)

We will only include canonical resolutions which either choose one or the other CC, i.e. CC1 or CC2.

Let $K = \sum_{p=1}^{n} m_p$ be the total number of conflicts in a merge, then by substitution, the number of possible resolutions is given by:

$$\prod_{i=1}^{n} k_i = \prod_{i=1}^{n} 2^{m_i} = 2^{\sum_{i=1}^{n} m_i} = 2^K \tag{2.1}$$

In the example from Section 2.3 there are two CCFRs for ”file-1.txt”, and therefore there are two CCMRs. The CCFRs are shown in Figure 2.8.

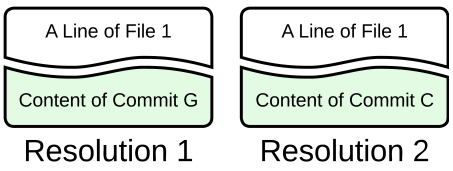

Figure 2.8: Merge resolutions of the merge conflict of branch ”topic” and branch ”master”.

3
Related Work

This chapter includes other empirical studies to understand where our results lie within the existing research. Further, we cover additional information about merges and merge conflict resolution, which aids the discussion of the results in Chapter 7.

3.1 Empirical Studies based on Repository Analysis of Git

As mentioned in Chapter 1, little research about the human role in resolving merge conflicts exists. Ghiotto et al. [12] conducted a detailed study on 2731 open-source Java projects with 25 328 CMs and 175 805 CCs. The study aimed to examine the structure of the merge conflicts and to gather insight into how developers resolve them. Their focus lay on analyzing how single conflicting chunks are resolved and how they depend on each other, if at all. They found that (i) 87 % of CCs were resolved with already existing code from both of the CCs, (ii) 60 % of all conflicting merges had more than two CCs, and (iii) 29 % of conflicting chunks depend on other chunks. Their analysis is based on extracting the line numbers of the CCs and retrieving those lines from the actual conflicting file resolution (ACFR). They then performed a line difference using an existing algorithm by Git and searched for lines of both CCs from the conflict. A CC is marked as correct if there is no new code, meaning that the lines from the ACFR are entirely made out of lines from the CCs. This approach should result in a lower correct rate than ours, which is the case, as seen in Table 5.3 for the Java list. Their result showed that more than 75 %, or in their words, ”a full three quarters” (Ghiotto et al. [12], p. 905), of CCs were resolved by picking one of the chunks, which is the same metric used in this thesis, the correct rate for CCs for the Java list.

Leßenich et al. [17] posed seven hypotheses based on a survey on merge conflicts. The first hypothesis suggests that active branches, determined by the number of commits, are more likely to result in conflicts than inactive branches. However, they analyzed more than 21 thousand merge scenarios and concluded that the number of commits does not strongly correlate with the number of merge conflicts and therefore rejected their first hypothesis.

Another study on the influence of different variables on the number of merge conflicts was conducted by Menezes et al. [20]. They studied 182 273 merges from 80 projects written in eight different programming languages and found that factors like the number of committers, the number of commits, and the number of changed files of a branch have the biggest influence on the occurrence of merge conflicts. However, it is important to note that these factors only concern one of the two branches involved in a merge conflict and are, therefore, not directly comparable to our work.

3.2 Merge Conflict Resolution

Merge conflicts cannot always be avoided, so gathering information on what they affect and how is important. Consider the following questions related to the merge conflict resolution process (Nelson et al. [22], p. 2864):

1. ”How do developers approach and manage merge conflicts?”
2. ”How do developers perceive the difficulty of a merge conflict resolution?”

These questions are interesting because many studies explain the effects merge conflicts bring to the software engineering process, but the developer was rarely in the spotlight. Their research found that developers follow a sequence of five phases when resolving conflicts, as mentioned in Chapter 1. They discovered that 56.18 % of the participants in their survey deferred at least once, meaning they did not deal with the conflict immediately. The main reasons for this behavior were found to be the complexity of the conflicting code and the number of CCs. Unsurprisingly, the most prominent factor in how developers perceive difficulty of CMs is the complexity of the conflicting code. Furthermore, the expertise a developer possesses in the field where the CC lies is the second most prominent difficulty factor. They also found that the merge tools used to gather information about the conflicts and resolve them are perceived as less effective when complexity increases. Brindescu et al. [6] determined that not only complexity but also information gathering dramatically affects the difficulty of the resolution of CMs, meaning that the information-gathering process is dependent on the complexity, and it also affects the perception of the difficulty of the CMs.

3.3 Prevention of Merge Conflicts

Merge conflicts cannot always be prevented, but there are many attempts and approaches to how one could reduce the number of conflicts.

3.3.1 Early Prevention

A way to prevent conflicts is to reduce the possibility of a conflict being able to form. Kasi et al. [16] called this approach ”proactive conflict minimization”. They proposed and tested a tool that restricts dependent tasks or tasks that share common files from being concurrently edited. In their experiment, they successfully avoided the majority of conflicts.

3.3.2 Merge Techniques

Still, with the reduction of conflicts in their early stage, there will always be conflicts. A promising fact, supported by the studies listed below, is the existence of false-positive conflicts. Those are conflicts that only form because the merge-algorithm could not correctly merge the branches and found conflicts where none were necessary. The solution to this problem is to improve the merge techniques.

There are four kinds of merge techniques [21]: Textual, syntactic, semantic, or structural merge techniques. In general, a merge technique tries to ”prevent” merge conflicts by minimizing their amount, but some variations of the techniques also try to resolve existing conflicts.

The technique used by Git is the unstructured technique. Unstructured merges base comparison solely on text, losing important syntactic information. In an environment using one particular programming language and, therefore, syntax, unstructured merges may not be the best technique. The better approach would be using a structured merge technique ([3], [14], [24], [25]), which considers the syntax of a given language. By many accounts, such as Seibt et al. [23], structured merges show fewer merge conflicts in a syntax-based environment than unstructured merges. The downside of the structured technique is a performance deficit. Another method, the semi-structured approach ([1], [8], [2]), tries to balance the performance of the unstructured approach and the improved precision of the structured approach. Therefore this approach is less precise than structured merges, supported by Seibt et al. [23].

The semantic-based approach ([4], [5], [13], [28], [15]), tries to address conflicts that are preventable but are not of syntactical nature. For example, it can detect and prevent conflicting renames of functions or variables.

The last approach is the operation-based, or structural merge technique ([18], [19]), which uses the fact that a VCS records every change in a project. A conflict can be resolved by replaying recordings of all project operations in the form of transformations. Edwards [11] proposes different resolution strategies concerning operation-based merge techniques. The strategy of interest is the so-called *explosion strategy*. It calculates all possible paths of operations that lead to various resolutions. This approach has similarities to the CCMR generation described in this thesis, see Section 4.2.1.

As one can see, many algorithms are trying to minimize merge conflicts as much as possible with varying success, and they should be considered in the approach proposed in Chapter 1.

4 Methodology

Our data collection and analysis approach is presented in the sections below. First, the gathering of the projects and then the extraction of the relevant data about the projects is described.

4.1 Data

As discussed in Section 2.4, a conflict can be resolved by either the canonical or the manual resolution. This thesis analyzes human behavior in the case of merge conflicts, especially which of the two available methods is chosen, see Section 2.4. For each level of granularity of conflict resolution, the number of times a contributor chose one of the conflicting chunks is of interest. For the study, we used open-source projects from GitHub. GitHub provides a network-based solution for the Git-based VCS software-engineering process.

Electing projects was based on three criteria: Random projects with high popularity, random projects with low popularity, and projects with high popularity featuring a specific programming language. The popularity of a project is defined as the number of stars awarded to a project by GitHub users. Starring a project is the same as adding a bookmark to the project to remember it.

4.2 Data Collection

The project collection process uses the GitHub API, which allows users to make search requests for projects based on different criteria. For example, searches for language and name are explicitly searched for, while one can obtain popular or unpopular projects by sorting the returned projects by the number of stars in descending or ascending order.

First, we describe the random project selection. A random project can be obtained by searching for a project by name, where the name is chosen randomly. The random name is generated by selecting a random word from the Unix Words List[1]. The number of projects returned by each search request is limited to 10 projects to further randomize the selection and prevent words strongly associated with software engineering from affecting the selection bias. The projects obtained with this search are expected to be

[1] `https://en.wikipedia.org/wiki/Words_(Unix)` / accessed on Nov 14, 2022

small in file size and commit count, which makes it possible to increase the number of projects to analyze. For each criterion, random projects with high and low popularity, 2500 projects were selected with the above method. How the popularity criterion was accounted for is explained above.

The search for the language-based criteria is simple. The language can be specified in the search request, and the projects can be sorted by stars in descending order to ensure high popularity. Since the popularity of these projects is high, the number of projects is reduced to 500 projects per language because popular projects are expected to be bigger in file size and commit count. Six languages were selected because they are the most used on GitHub based on the website GitHut[2]. Below are the chosen languages, listed in descending order of usage:

1. Python[3]
2. Java[4]
3. C++[5]
4. Go[6]
5. JavaScript[7]
6. TypeScript[8]

The program used for the creation of the project lists is written in Python and is hosted on the GitLab server of the University of Bern[9]. In addition, we developed a second program in Java, which was used to clone the projects, find the conflicts, and analyze them. This program is also hosted on the GitLab server of the University of Bern but in a different repository[10].

The program first clones the project to obtain a local copy. All the merges of the projects up to the point of cloning were then located and re-merged to find the conflicts. For conflicting merges with conflicting chunk count smaller than or equal to 12, all CCMRs were generated. The reason we chose a limit of 12 CCs is explained in Section 6.1. Then each merge resolution was compared to the merge resolution which was actually used in the existing project, denoted as actual conflicting merge resolution (ACMR). Moreover, all canonical file resolutions were compared to the actual used file resolutions, denoted as ACFRs, and the same process was applied to the chunk resolutions. For all the above comparisons, the number of conflicting merges, files, or chunks and the number of times the canonical resolution is used in the actual resolution or equal to the actual resolution is documented.

[2] `https://madnight.github.io/githut/` accessed on Nov 14, 2022
[3] `https://www.python.org/` accessed on Nov 14, 2022
[4] `https://www.java.com/` accessed on Nov 14, 2022
[5] `https://isocpp.org/` accessed on Nov 14, 2022
[6] `https://go.dev/` accessed on Nov 14, 2022
[7] `https://www.javascript.com/` accessed on Nov 14, 2022
[8] `https://www.typescriptlang.org/` accessed on Nov 14, 2022
[9] `https://gitlab.inf.unibe.ch/sb18n092/projectlistgenerator` accessed Dec 13, 2022
[10] `https://gitlab.inf.unibe.ch/SEG/theses/mergeconflictresolution`, accessed Dec 13, 2022

4.2.1 Generating Canonical Conflicting Merge Resolutions

The JGit[11] API is a Git implementation in the Java programming language. JGit makes it possible to filter out all the merges from all the commits of a repository and then re-merge the merge's parent commits. The re-merge produces a sequence of merge chunks as described in Section 2.2 and Section 2.3. One can imagine this sequence as a tree-like structure where each merge chunk is a vertex, and a CC splits the tree. After each such intersection, the edges of both CCs lead back to the next merge chunk or the next chunks if they are conflicting. Let us call this tree the *file-tree*. For each CM, a tree can be generated as well, the *merge-tree*, consisting of file-trees as vertices, see Figure 4.1. One CCMR, therefore, represents one path in the merge-tree. This path can then be separated into multiple sub-paths, each corresponding to one CCFR. The ACMR can be obtained by reading the Git repository and converting the resolution into a comparable object.

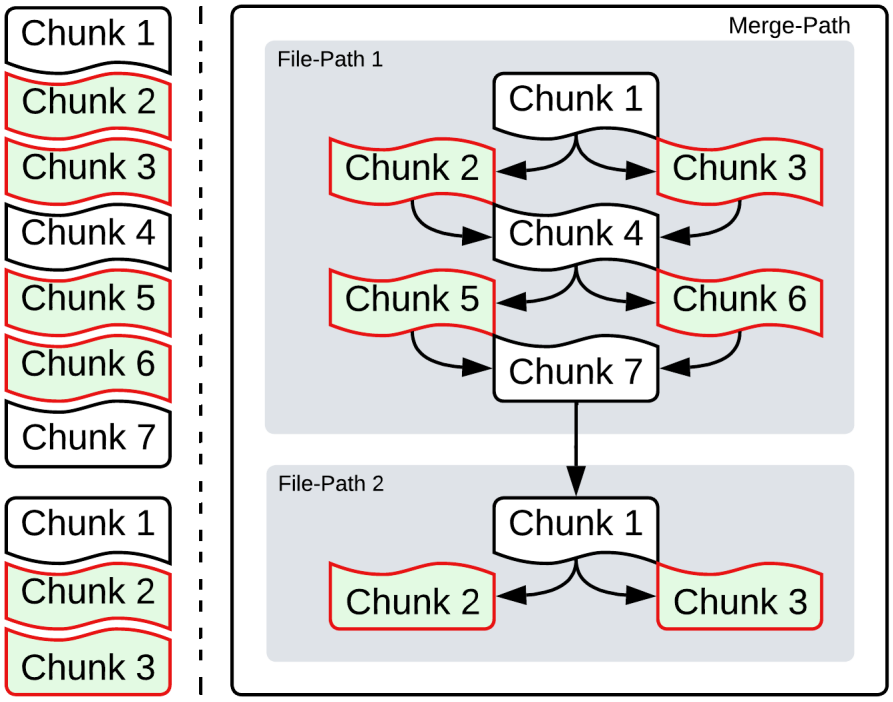

Figure 4.1: Example of a merge-tree. On the left are the merge chunk sequences of two files, and on the right is the merge-tree corresponding to the sequences. A green background represents a merge chunk with changes, and a red border marks conflicting merge chunks.

With Gits' diff-algorithm, the CCMR and the ACMR (ACFR for file granularity) can be compared. If there are no differences, the compared resolutions are the same. Comparing two resolutions works for merge and file granularity. However, the comparison is different for the chunks themselves. To still compare at chunk granularity, the CCs are searched for in the ACFR.

[11] `https://www.eclipse.org/jgit/` accessed on 14 Nov, 2022

If a CCMR or CCFR has no differences to the ACMR, or rather the ACFR or a CC is found in the ACFR, the developer used the canonical resolution method. We then mark the CM, CF, or the conflict belonging to the CCs as *correct*. The number of times a CM, CF, or conflict is marked as correct is denoted as the *correct count*, c_{correct}. The percentage of the correct count, out of the total number, c_{total}, of analyzed CMs, CFs or conflicts, is denoted as the *correct rate*, which is used to answer RQ1:

$$r = \frac{c_{\text{correct}}}{c_{\text{total}}} \tag{4.1}$$

For a CMR to be correct, all its CFRs must be correct. But for a CFR to be correct, all its conflicts must be marked as correct. This relationship manifests itself in the following equation:

$$r_{\text{CM}} \leq r_{\text{CF}} \leq r_{\text{CC}} \tag{4.2}$$

For consistency over the thesis, the r_{CC} is used instead of r_{conflict}. The latter notation would technically be the correct one, but conflicts and conflicting chunks are mostly interchangeable terms since the CCs belong to the conflict, and it is inferrable from context if the conflict or the actual chunks are referred to.

4.2.2 Metadata

Collecting projects' metadata to answer RQ2 is also important. For each project, we collected the number of commits and contributors. Since the relationships between commit count and conflict count and between contributor count and conflict count are possibly very complex, we would like to see how the behavior is in reality. We also collected the total number of merges, first and foremost for completeness of the data and secondly for a qualitative comparison to the number of CMs.

5 Results

In this chapter, the results of the thesis are presented. The results related to the correct rate are shown first, and the results regarding the metadata are second. The sections below give results for both research questions because there are essential measurements for both questions in Section 5.1.

5.1 Correct Rate

The results for the canonical resolutions with the correct count, total count, and correct rate are listed in this section. The results are structured identically for all three levels, merge, file, and chunks. The Table 5.1 contains the results of the CMs, Table 5.2 the ones of the CFs, and Table 5.3 the CCs. The first two columns of the last row labeled ”All Lists” represent the sum of the above values. The correct rate was calculated the same way in every row, including the last row.

Table 5.1: Results for the merge granularity for all lists. Columns $c_{\text{correct, CM}}$ and $c_{\text{total, CM}}$ in the last row are the sum of the above values.

List	$c_{\text{correct, CM}}$	$c_{\text{total, CM}}$	r_{CM} [%]
Python	3737	12 567	29.74
Java	5529	12 359	44.74
C++	19 581	58 020	33.75
Go	3495	10 419	33.54
JavaScript	1615	5071	31.85
Typescript	6281	19 749	31.80
Random Name Desc.	534	1353	39.47
Random Name Asc.	90	256	35.16
All Lists	40 862	119 794	34.11

Table 5.2: Results for the file granularity for all lists. Columns $c_{\text{correct, CF}}$ and $c_{\text{total, CF}}$ in the last row are the sum of the above values.

List	$c_{\text{correct, CF}}$	$c_{\text{total, CF}}$	r_{CF} [%]
Python	9278	21 432	43.29
Java	12 610	22 267	56.63
C++	49 867	109 204	45.66
Go	8029	17 642	45.51
JavaScript	4179	8924	46.83
Typescript	15 642	35 789	43.71
Random Name Desc.	1022	2100	48.67
Random Name Asc.	151	354	42.66
All Lists	100 778	217 712	46.29

Table 5.3: Results for the merge granularity for all lists. Columns $c_{\text{correct, CC}}$ and $c_{\text{total, CC}}$ in the last row are the sum of the above values.

List	$c_{\text{correct, CC}}$	$c_{\text{total, CC}}$	r_{CC} [%]
Python	23 412	29 316	79.86
Java	25 462	29 965	84.97
C++	119 910	148 035	81.00
Go	19 279	24 182	79.72
JavaScript	9725	12 075	80.54
Typescript	38 655	49 893	77.48
Random Name Desc.	2656	3119	85.16
Random Name Asc.	459	541	84.84
All Lists	239 558	297 126	80.63

First, look at the overall result for each level of granularity. The increase of the correct rate from granularity level to granularity level is expected because of the relation in equation (4.2). The one granularity which stands out is the chunk granularity at a correct rate of 80.63 %. Compared to the increase from $r_{\text{All Lists, CM}}$ to $r_{\text{All Lists, CF}}$, the increase from $r_{\text{All Lists, CF}}$ to $r_{\text{All Lists, CC}}$ is larger by $22.16\,\%$. This difference in increase manifests itself in a relatively small percentage of conflicts (CCs), namely 19.37 %, causing 65.89 % of all CMs not being resolvable by a CCMR.

Java produced the highest correct rates, with 44.74 % for the merge granularity and 56.63 % for the file granularity. Both of those values are around 10 % higher than the total correct rate. For the chunk granularity, Java's correct rate stands out less than for the other granularities, but it is still one of the highest values. Similar results are found for the random name with descending popularity list. The difference is

that its correct rate is not quite as high as the one for the Java list. For the merge granularity, the ”Random Name Desc.” list’s correct rate stands at 39.47 %, and for the file granularity, it stands at 48.67 %. Other than Java and ”Random Name Desc.”, the lists are within a 4.50 % interval around the correct rate of all lists, which holds for all granularities.

Now consider the correct counts over every project, disregarding the list separation. In Figure 5.1 and Figure 5.2, there are two plots for each granularity, visualizing the conflicting merges, files, and chunks count against the conflicting merges, files, and chunks correct count. The first (left) plot has linear scaling to show the true scaling of the data, and the second (right) plot shows all the data but with double-logarithmic scaling. The data includes all projects from every list.

The regression equations in Figure 5.1 and Figure 5.2 confirm the above results from Table 5.1, Table 5.2 and Table 5.3. The interesting piece of information represented in the plots, which is not visible otherwise, is that for the merge and file granularity, the data is much more scattered than for the chunk granularity. Some projects with a high CM count or CF count have a relatively low correct count, but those outlier projects are no longer outliers for the chunk granularity.

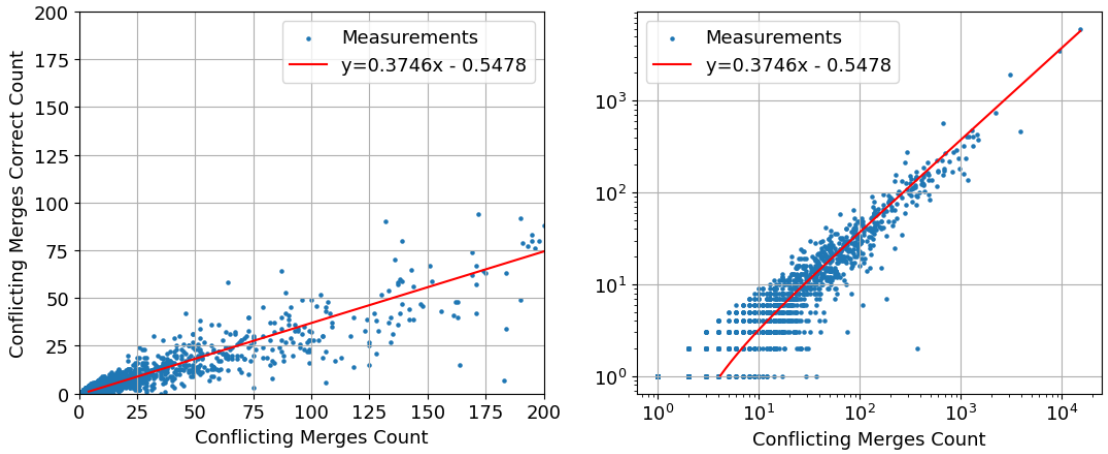

Figure 5.1: Correct count of the merge granularity in relation to the total count over all the projects for each granularity. Both plots contain the same data but over different scales for each granularity. The left plot is constricted to data around the source, whilst the right plot has a double logarithmic scale over all the data.

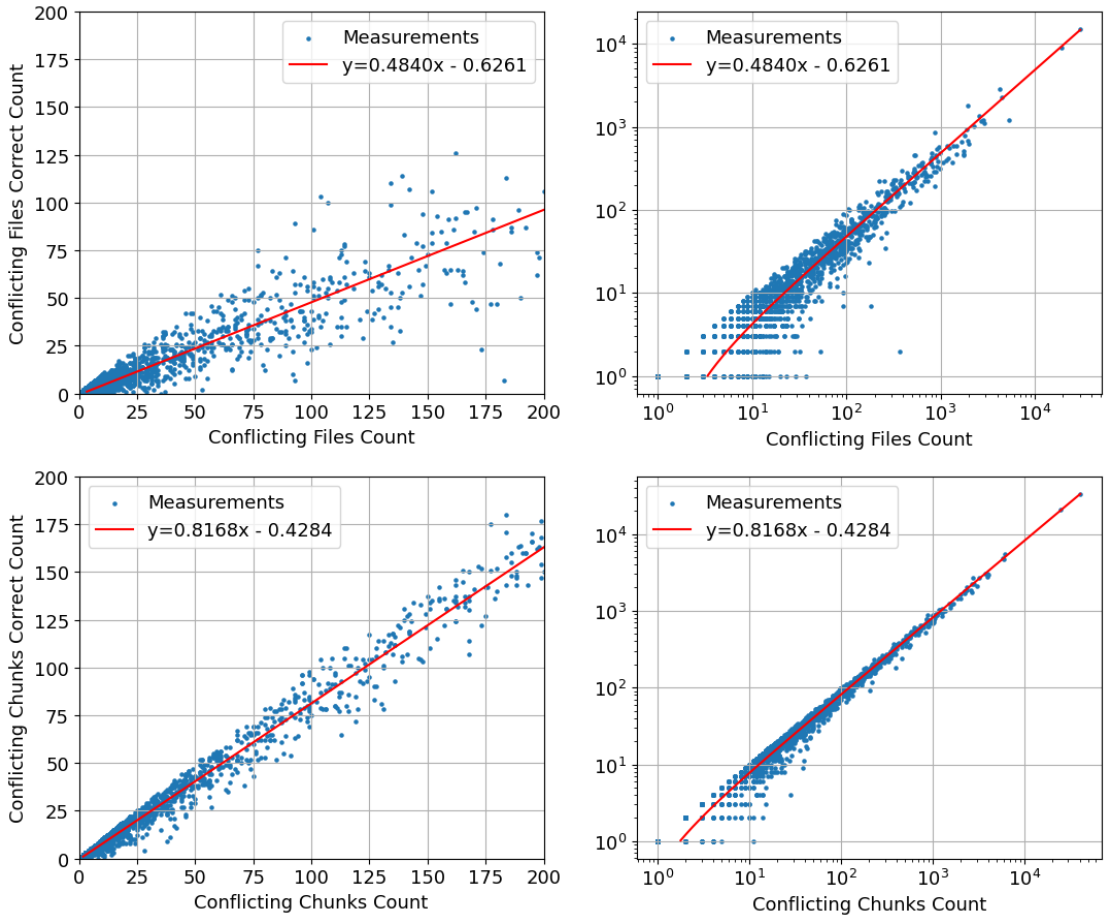

Figure 5.2: Correct count in relation to the total count over all the projects for each granularity. Both plots contain the same data but over different scales for each granularity. The left plot is constricted to data around the source, whilst the right plot has a double logarithmic scale over all the data. From top to bottom: File and chunk granularity.

Lastly, we give the correct rates of CMs and CFs for different conflict counts. Each CM, resp. CF was categorized by its conflict count. For each of the conflict counts $i = 1, \ldots, 12$, the total amount, $c_{\text{total, gran., } i}$, of each granularity (gran.) belonging to the conflict count gets calculated as well as the amount of correct count for each granularity, $c_{\text{correct, gran., } i}$, of the conflict count. With these values, the correct rate $r_{\text{gran., } i}$ for each conflict count i could be calculated. The results are listed in Table 5.4 and visualized in Figure 5.3. One can see that for the CMs, the correct rate and the total count decrease with increasing conflict count, up to six CCs. The CFs behave similarly with the difference that the correct rate increases visibly again at seven CCs. In both plots, there is an anomaly at 11 CCs, but the general shapes of the bar graphs point to the previous conclusions.

Table 5.4: Correct and total counts as well as correct rates for all CMs and CFs over all projects categorized by CC count of each CM or CF. The subscript i denotes the CC count. For example: The total count for CFs with $i = 11$ CC count is $c_{\text{total, CF, 11}} = 151$.

Conflicting Chunks Count	$c_{\text{correct, CM, } i}$	$c_{\text{total, CM, } i}$	$r_{\text{CM, } i}$ [%]	$c_{\text{correct, CF, } i}$	$c_{\text{total, CF, } i}$	$r_{\text{CF, } i}$ [%]
1	24 536	60 515	40.55	88 173	175 243	50.31
2	7763	23 319	33.29	8269	25 598	32.30
3	3122	11 376	27.44	2348	8393	27.98
4	1832	7113	25.76	982	3770	26.05
5	1109	4732	23.44	417	1927	21.64
6	675	3356	20.11	229	1082	21.16
7	495	2498	19.82	118	615	19.19
8	388	1990	19.50	90	432	20.83
9	341	1646	20.72	62	265	23.40
10	244	1277	19.11	41	169	24.26
11	161	1057	15.23	27	151	17.88
12	196	915	21.42	22	67	32.84

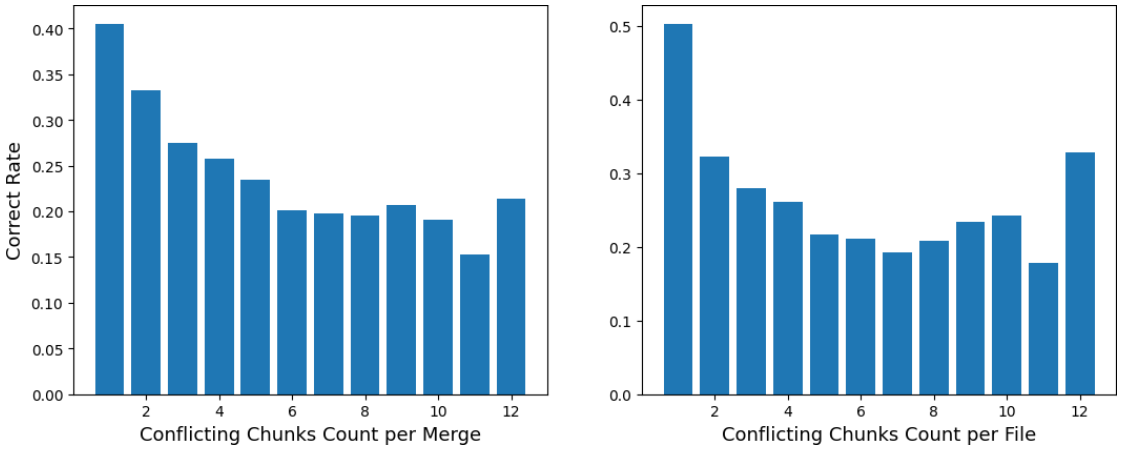

Figure 5.3: Correct rate of CMs and CFs by number of conflicts / CCs.

5.2 Metadata

The metadata was collected for every project in a project list as described in Section 4.2.2. The metadata in Table 5.5 has been summarized for each list by summation over each project in a list. Some data points stand out: C++ has the highest commit and merge count. Also, there is a noticeable difference between JavaScript and TypeScript, even though TypeScript is based on JavaScript. As expected, the random name lists have much fewer contributors, commits, and merges.

Table 5.5: Metadata of the lists. In this table, the last row is the sum of the above values.

List	Contributors	Commits	Merges	Analyzed CMs ($c_{\text{total, CM}}$)
Python	58 553	1 085 693	191 741	13 509
Java	34 977	1 136 621	247 719	13 934
C++	119 127	4 206 996	685 559	64 370
Go	120 753	1 977 436	414 242	11 411
JavaScript	16 584	286 345	55 632	5636
Typescript	128 301	1 920 250	263 727	23 368
Random Name Desc.	6594	136 748	18 435	1617
Random Name Asc.	3977	35 552	2650	302
All Lists	488 866	10 785 641	1 879 705	134 147

We analyzed correlations of various metadata metrics to the CM count. First, the correlation between the CM count and commit count, and then the correlation between CM count and contributor count was calculated. The calculated value is the Pearson correlation coefficient[1]. The correlation between the commit count and the CM count is 0.52, and the correlation between the contributor count and the CM count is 0.20. The cause-and-effect relationships between the CM count and the commit count and CM count and contributors are shown in Figure 5.4. The linear regressions used to approximate the relationships roughly suggest that in 100 commits, there will be approximately two CMs, and for every additional 10 contributors, there will be approximately 1.5 CMs more. Notice that this is a rough approximation. It is solely used to introduce an understanding of how strong the effect of commit count and contributor count on CM count is.

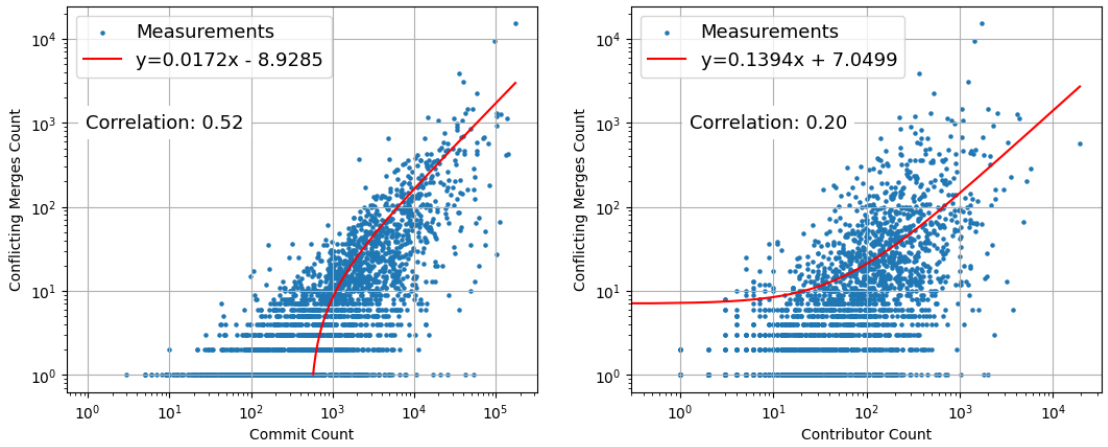

Figure 5.4: Influence of commit count and contributor count on CM count. Both plots have a double logarithmic scale to accommodate all data points. Every project, over every list, is included.

[1] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient accessed Dec 1, 2022

6 Threats to Validity

We address threats to internal, external and construct validity in this chapter. We chose the classification of threats to validity according to Wohlin [26],[27], which is typical for software engineering literature.

6.1 Internal Validity

We did not study projects containing conflicts with very big files to maintain a time- and memory-efficient analysis with a high CM count. Suppose there are conflicting chunks in big files, the duration for comparing merge resolutions increases. Some files, e.g., ”package-lock.json” of some JavaScript projects managed by NPM[1], have lots of conflicting chunks in the same file while having sizes in the range of a few megabytes. These files lead to a significant increase in time for comparisons and heap-memory consumption. Projects which have conflicts with such files were removed and replaced by similar projects without conflicts in very big files. The project was moved from the project list to a project blacklist, and the reduced project list was replenished with a new project by running the same election process but keeping the already elected ones and not accepting the projects on the blacklist. Of all the 8500 projects, there were only 15 such occurrences, and since such problems can occur in every project, no matter the number of commits of a project or what language the project was in, we deem their impact on the selection bias to be small.

There is also the possibility that projects are empty (no commit has been made yet) or contributors emptied projects from GitHub in the time frame from gathering projects to the cloning process, which was, at most, a few hours. This may transpire more often for projects with low popularity, i.e., both lists of random projects, but this occurred only once for projects with high popularity. All those projects were ignored but not replaced because this only affected one project of the Go list, 332 projects of the ”Random Name Desc.” list, and 343 projects of the ”Random Name Asc.” list. This issue was accounted for by enlarging the random project lists.

Git does also not prevent contributors from editing the history, meaning that branches or commits may be altered or deleted. Since altering or deleting a commit is time-consuming and pointless most of the time, since a mistake in the past can mostly be fixed in the present, we can assume that these actions are rare. Furthermore, there is no reason why an altered or deleted commit should be different from any of the

[1] `https://www.npmjs.com/` accessed on Nov 15, 2022

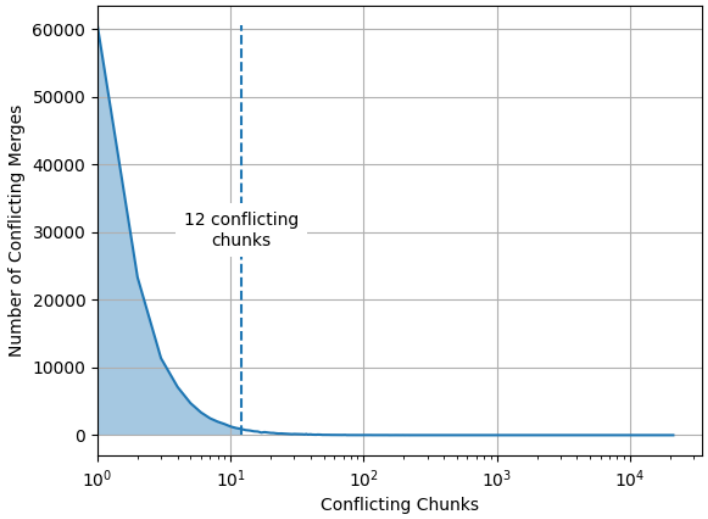

Figure 6.1: Number of merges with certain conflicting chunks count. The plot contains every conflicting merge overall project lists. The line at 12 conflicting chunks marks the cutoff of all the analyzed merges: To the left of the line are the merges which were analyzed. For better readability, this is not a bar graph, but a curve, even though the values are discrete.

other commits, meaning that if one such commit is present in the data, it does not affect the selection bias. For the case when a conflicting merge would have included a deleted branch, one would not be able to check this CM since one of the parent commits would be missing. Our tool would have noticed the missing branch and marked this specific CM as a fail. The analysis, however, showed that this never occurred. The last case, altering a branch, has no impact since the commits necessary for the re-merge still exist.

There are other limitations as well. The number of possible merge resolutions grows exponentially, see equation (2.1). For example a CM with 10 CFs where each CF has 30 CCs, the amount of possible canonical resolutions is given by $\prod_{i=1}^{10} 2^{30} = 2^{300}$. Given this structure of a CM, our tool would need to make approximately 2^{33} comparisons of relatively large data structures, which is infeasible to do in a reasonable amount of time. The way to avoid these merges with many conflicts is to limit the number of possible paths in the merge-tree. In this thesis, the limit was set to $2^{12} = 4096$, which means that conflicting merges with more than 12 CCs are discounted from further analysis. Out of the 130 619 CMs in this thesis, not counting the CMs containing binary files (see Section 6.3), only 10 825 or 8.29 % have more than 12 CCs, see Figure 6.1. For further data about the number of ignored merges, see Table 6.1.

Human intervention in a merge conflict makes comparing merge chunks non-trivial. The problem lies in finding the location of a conflicting chunk in the actual resolution file. Consider the following example: A conflicting file was resolved by picking one of the conflicting chunks, but the contributor makes a new change while merging, a change in the same file that has nothing to do with conflicts. The reason for the change could be a bug discovered while merging. One can quickly determine that the actual resolution

file will never match a canonical file resolution, but there is no way of determining where precisely the conflicting chunks are located in the actual resolution file because the change may have shifted the location of the chunks. This means that the conflicting chunks may be at different locations than the merge results suggest. This poses the problem that there is no definitive way to tell if the chunk in the actual resolution is actually the one we are trying to find. A workaround solution was described in Section 4.2, where the CCs are searched for in the ACFR. This approach causes a different problem: There is the possibility that a copy of a conflicting chunk is present in a conflicting file but in a different location than the actual conflicting chunk. This means that the conflicting chunk could be found even though the conflicting chunk is not even used in the actual resolution file. There are other similar problems. Our approach will increase the correct rate of the CCs, but the change is not quantifiable. Other research like Ghiotto et al. [12] faced the same problem and solved it with a different method, leading to a lower correct rate, differing by about 10 % to our result for Java projects. They also mention that their results may not hold for studies with a larger sample size which is the case for this thesis. Because of this, we assume that the results gathered from our analysis concerning the correct rates of CCs are plausible values.

Our code may be buggy, resulting in wrong measurements. However, we regularly analyzed samples manually as well as wrote unit tests on a prefabricated project from which the conflicts were known to ensure correctness.

6.2 External Validity

As we studied GitHub projects of six programming languages, our results may not apply to projects outside of GitHub or from other programming languages. However, we found that the deviation between the programming languages we studied was small, which indicates that similar results will be found for projects in other languages. The same can be said for random projects with ascending and descending popularity. Moreover, GitHub is the major online repository today. Since GitHub is an open-source project, the projects we studied are diverse in popularity and project size. As the results were very similar, even for these diverse projects, we are confident that projects hosted differently (not on GitHub) would measure similar to ours. All in all, the results of RQ2 give an indication for external validity, see the discussion for RQ2.

6.3 Construct Validity

As mentioned, Git supports different merge algorithms using different merge-strategies. Our analysis focuses on the three-way merge with the default merge strategy, ORT[2], of Git. ORT is based on a recursive strategy and merges two branches. The three-way merge is the default algorithm for Git and is used in most cases. In our analysis, we found 4881 octopus merges or 0.26 % in 1 879 705 merges. Restricting our analysis to one single merge-strategy may cause some commits to be identified as false-positive or

[2] `https://git-scm.com/docs/merge-strategies#Documentation/merge-strategies.txt-ort/` accessed Nov 22, 2022

false-negative merges if developers used a different merge strategy. According to the documentation of the ORT merge strategy[2] of Git, the ORT strategy "reported to result in fewer merge conflicts without causing mismerges by tests done on actual merge commits taken from Linux 2.6 kernel development history". We argue that most developers use the ORT merge strategy of Git, and thus the percentage of non-recursive merges is negligibly small.

Also ignored are merges containing binary files, of which we detected 3528 or 2.63 % out of 134 147 eligible CMs. Although Gits' merge-algorithm can detect conflicts in binary files, a human cannot resolve such conflicts because binary files do not contain human-readable UTF-8-encoded[3] text. Therefore it is pointless to analyze merges containing conflicts in binary files. The total amount of ignored CMs can be found in Table 6.1.

Table 6.1: Data of ignored CMs of the lists. In this table, the last row is the sum of the above values.

List	Total CMs	Analyzed CMs ($c_{\text{total, CM}}$)	Ignored CMs	Ignored CMs (Binary Files)	Ignored CMs (Max. CC count)
Python	13 509	12 567	942	164	778
Java	13 934	12 359	1575	142	1433
C++	64 370	58 020	6350	1348	5002
Go	11 411	10 419	992	132	860
JavaScript	5636	5071	565	73	492
Typescript	23 368	19 749	3619	1442	2177
Random Name Desc.	1617	1353	264	191	73
Random Name Asc.	302	256	46	36	10
All Lists	134 147	119 794	14 353	3528	10 825

Since resolving conflicts is difficult [6], developers will eventually make mistakes when resolving merge conflicts. However, our analysis assumes that the ACMRs are mistake-free. The reason for this assumption is that the ACMRs created by the developers are the only resolutions available, and checking a resolution for correctness is also not possible since total correctness of a program is undecidable because of the undecidability of the halting problem[4].

[3] `https://de.wikipedia.org/wiki/UTF-8/`, accessed Dec 21, 2022
[4] `https://en.wikipedia.org/wiki/Halting_problem` accessed Dec 15, 2022

7 Discussion

7.1 Summary of Key Findings

The results found in Chapter 5 are the following: We found that the correct rate for CMs, with 34.11 %, is much lower than the one for the CCs, with 80.63 %. Only 19.37 % of all conflicts cause 65.89 % of all CMs not being resolvable by a CCMR. Further, Java has the highest CM and CF correct rate and is in the top three for the CC correct rate. The correct rates categorized by CC count are similar for both CMs and CFs. The correct rate for a low CC count is high but decreases again with a higher CC count up to around six or seven CCs and then increases again. For the CMs, the increase is very slight as to the CFs, where the growth is much more noticeable. For the correlation between different measures of project metadata and CM count, it was found that the commit count correlates moderately strongly, and the contributor count correlates weakly.

RQ1: How often do humans intervene in merge conflict resolution for a VCS such as Git?

Based on the correct rate of the CMs, one of three CM is resolvable by a CCMR. Opposed to the correct rate for the CCs, the involvement of humans in CMs is much stronger. About eight out of ten CCs are resolved by the canonical approach. We could only compare the correct rate for CCs for Java projects with prior work. As mentioned in Section 3.1, Ghiotto et al. [12] found that the correct rate for CCs for the Java list is more than 75 %. Our measured value is 84.97 %. It is difficult to say which of the two values more accurately displays the true value because both methods need to compromise to compare the CCs to the ACFR, but since their results match with the expected lower value and ours with the expected higher value, it is reasonable to assume that both values circle an average and are plausible. In turn, this means that values for the other project lists are also plausible regarding our method.

Based on the research of Brindescu et al. [6], the complexity and difficulty of resolving a CMs can quickly increase with the size of the CM, i.e., the lines of code involved in the CM or the number of CCs. This is because the information needed to resolve a CM is not easily obtainable. With an increasing CC count, the information-gathering process will be even more difficult, especially if the different CCs of the

CM are related to each other, meaning that the resolution of one CC affects the resolutions of the other CCs. Obviously, no relations can form for single conflicts, but more such relationships will appear with increasing CCs count. Ghiotto et al. [12] found that 29 % of all CMs have dependencies among their CCs. Our results point in the direction that most of the CCs are simple to resolve since the CC correct rate is so high. This implies that most conflicts are unrelated, meaning that the information needed to resolve one conflict is confined to this one conflict. But if two conflicts are related, resolving them is difficult because the amount of required information grows rapidly. Again Ghiotto et al. [12] concluded that for 87 % of CCs, all the code, or information, to resolve a conflict is found within the two CCs, which aligns with the above statement.

There seems to be a limit of human involvement at around six or seven CCs for both CMs and CFs. This limit could be the result of developers being less willing to gather the necessary information to resolve CMs or CFs with a higher conflict count. Another possible explanation for this effect could be that the contributors get overwhelmed by the amount of CCs, do not know how to resolve them even with the necessary information, and resort back to the canonical approach. It is important to mention that the above reasoning is speculative and should be researched further.

Regardless of the reason, we can conclude that the rate at which humans intervene in CMs depends on the number of conflicts in the corresponding CM. For fewer conflicts, the involvement is weak, becomes stronger up to six or seven CCs, and weakens again with increasing CC count up to the analyzed conflict count of 12. Because of the reasoning above, intuitively, it would make sense that the correct rate would either stay the same or grow slowly for a higher conflict count, but this would need to be researched further. Nevertheless, intervention in single conflicts is less common, whilst for CMs and CFs, it becomes more common, meaning that a small percentage of CCs cause a big part of the CMs and CFs not being resolvable by CCMRs, resp. CCFRs.

RQ2: What factors influence the number of merge conflicts and their resolutions in a Git project?

There is no substantial difference in involvement when considering language or popularity of projects for all three granularities. The only somewhat notable values are the correct rates for Java. Java has the highest correct rate for CMs and CFs, and its correct rate for CCs is in the top three. Since Java has similar statistics, i.e., CMs count, commit count, etc., to other languages researched in this thesis, it can only be assumed that this value came about by statistical chance.

In our research, we considered commit count and contributor count, which could influence the conflict count of projects. We found that the correlation between contributor count and CM count is weak, with a value of 0.2. To gain perspective, Leßenich et al. [17] found as an additional result of their study that there is almost no correlation between contributor count and CM count. It is important to mention that Leßenichs study is much less extensive, analyzing only 163 open-source projects, of which the

maximum contributor count is 497. But still, the correlation is nowhere near expressive enough to conclude meaningful implications on the CM count. The number of contributors of a project is more likely to be an indicator of the project's size rather than the number of CMs. This is shown in Table 5.5, as, in general, a higher contributor count yields a higher commit count.

The correlation between commit count and CM count is stronger, as we found it to be 0.52. Again Leßenich et al. [17] rejected their first hypothesis, suggesting commit count as an indicator with predictive power for the number of conflicts. The correlation between commit count and conflict count was found to be 0.16. This value is not directly comparable to our result, but it indicates that commit count only slightly affects conflict count or, in our case, CM count.

To summarize, the commit count has a moderate influence on the CMs count: About two CMs in 100 commits are found. On the other hand, the contributor count only has a weak correlation to the CMs count.

7.2 Implications

There is much research on merge conflict indication, prevention, recognition, and characteristics, see Chapter 3. However, little has been studied on the topic of the involvement of humans in merge conflict resolution. As of now, we found no research as detailed and extensive as ours. As mentioned in the answer for RQ1, our results for the correct rate of Java projects support the existing research of Ghiotto et al. [12]. As for the correlation between commit count and CM count and contributor count and CM count, our results also support the research conducted by Leßenich et al. [17], see the answer for RQ2.

The results not mentioned in the above paragraph represent new insights into merge conflict resolution and the human role. As discussed, the number of CCs in a CM strongly affects the decisions of a developer resolving a CM, and a relatively small percentage of CCs cause a high percentage of CMs not being resolvable by a CCMR. Consider the resolution approach proposed in Chapter 1. The approach would not be efficient based on our results. Aside from only one out of three CMs being resolvable by a CCMR, there will be other problems associated with the approach, like performance issues, because for every CM the program needs to be compiled and sometimes tested. One could consider new factors, adapt the approach to more closely evaluate the conflict, and base the evaluation of CCMRs suggested to the developer on them. One such factor may be the number of CCs.

7.3 Recommendations

Further research needs to be conducted on the small incorrect percentage of CCs. Reducing this percentage would strongly affect the growth of the correct rate for the CMs. Therefore finding the reasons for failure of the 19.37 % of conflicts is essential. If those reasons are minuscule, i.e., syntactical conflicts, they can be redeemed with little effort. Many approaches have already been proposed, serving the prevention of CMs based on syntax and semantics.

A way to improve the efficiency of the mentioned approach in Chapter 1 would be to treat conflicts in groups based on their relation to one another. Related conflicts could be treated as one, i.e., the resolutions that do not fit together could be filtered out. This would reduce the number of paths, as in Section 4.2.1, and therefore efficiency. The difficulty with this is twofold: First, one would need to detect related conflicts, and second, the incompatible resolutions need to be filtered somehow. With current knowledge, it is difficult to say if the reduction in paths would result in a beneficial, if at all, overall improvement of performance since additional steps would need to be implemented. However, if the path count would grow linearly and not exponentially, it would be worth taking a shot.

Since, in this thesis, the maximum CCs count is set to 12, it would be interesting to see if the trend of the increasing correct rate with increasing CC count still holds for more conflicts. Additionally, the reason for the increasing correct rate for the increasing conflict count could be researched further.

As there was also the argument that related conflicts could play a role in the resolution process, it would also be interesting to see how the related and unrelated conflicts are resolved individually and the differences between the two. Further, it would be interesting to see if there are additional factors like the degree of the relation of the conflicts that affect their resolution, i.e., the developers' decisions while resolving the conflicts.

8
Conclusion and Future Work

To assess the feasibility of the approach proposed in Chapter 1, we analyzed 119 794 CMs of 8500 open source projects from GitHub, each CM containing up to twelve conflicts. All CMs contain 217 712 CFs and 297 126 CCs in total, all included in the analysis process.

We concluded, based on the results of our analysis in Chapter 5, and the discussion of the aforementioned results in Chapter 7, that the approach proposed in Chapter 1 would not be very useful as it is. Because other difficulties like performance cannot be disregarded, the correct rate for CMs of 34.11 % is considered to be too low. By including other merge techniques (Section 3.3.2), which reduce the false-positive rate, or by treating related conflicts as one, reducing the path count of the merge-tree, the performance of the approach could be improved. But even if an algorithm with perfect performance would exist, it will always be limited by bottlenecks like the rate at which tests can be performed. Nevertheless, the correct rate for CCs is very interesting in that it is very high compared to the CMs or CFs. If this rate could be improved even further, it would also drastically impact the correct rate for the CMs. Regardless, improving the correct rate for CMs is vital for future studies.

No matter the somewhat sobering realization that the approach is not feasible, the results of this extensive study bring new, interesting insights into the human role in the merge resolution process.

List of Figures

List of Tables

Glossary

ACFR actual conflicting file resolution. 12, 16, 17, 18, 27, 29

ACMR actual conflicting merge resolution. 16, 17, 18, 28

CC conflicting chunk. 7, 9, 10, 12, 13, 16, 17, 18, 19, 20, 22, 23, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36

CCFR canonical conflicting file resolution. 10, 11, 17, 18, 30

CCMR canonical conflicting merge resolution. 10, 11, 14, 16, 17, 18, 20, 29, 30, 31

CCR conflicting chunk resolution. 9

CF conflicting file. 7, 9, 10, 18, 19, 21, 22, 23, 26, 29, 30, 33, 34, 35, 36

CFR conflicting file resolution. 9, 18

CM conflicting merge. 7, 9, 10, 12, 13, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 28, 29, 30, 31, 33, 34, 35, 36

CMR conflicting merge resolution. 9, 10, 18

IDE integrated development environment. 9

VCS version control system. i, 1, 2, 3, 4, 15, 29

Bibliography

[1] S. Apel, O. Leßenich, and C. Lengauer. Structured merge with auto-tuning: balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, New York, New York, USA, 2012. ACM Press.

[2] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. Semistructured merge. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, New York, New York, USA, 2011. ACM Press.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Autom. Softw. Eng.*, 14(1):3–36, Mar. 2007.

[4] V. Berzins. Software merge. *ACM Trans. Program. Lang. Syst.*, 16(6):1875–1903, Nov. 1994.

[5] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, 4(1):3–35, Jan. 1995.

[6] C. Brindescu, Y. Ramirez, A. Sarma, and C. Jensen. Lifting the curtain on merge conflict resolution: A sensemaking perspective. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2020.

[7] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, New York, New York, USA, 2011. ACM Press.

[8] G. Cavalcanti, P. Borba, and P. Accioly. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.*, 1(OOPSLA):1–27, Oct. 2017.

[9] S. Chacon and B. Straub. *Pro git*. Apress, Elk Grove, CA, 2 edition, Jan. 2014.

[10] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration*. Addison-Wesley Educational, Boston, MA, June 2007.

[11] W. K. Edwards. Flexible conflict detection and management in collaborative applications. In *Proceedings of the 10th annual ACM symposium on User interface software and technology - UIST '97*, New York, New York, USA, 1997. ACM Press.

[12] G. Ghiotto, L. Murta, M. Barros, and A. van der Hoek. On the nature of merge conflicts: A study of 2,731 open source java projects hosted by GitHub. *IEEE trans. softw. eng.*, 46(8):892–915, Aug. 2020.

[13] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, July 1989.

[14] J. J. Hunt and W. F. Tichy. Extensible language-aware merging. In *International Conference on Software Maintenance, 2002. Proceedings*. IEEE Comput. Soc, 2003.

[15] Jackson and Ladd. Semantic diff: a tool for summarizing the effects of modifications. In *Proceedings International Conference on Software Maintenance ICSM-94*. IEEE Comput. Soc. Press, 1994.

[16] B. K. Kasi and A. Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013.

[17] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen. Indicators for merge conflicts in the wild: survey and empirical study. *Autom. Softw. Eng.*, 25(2):279–313, June 2018.

[18] E. Lippe and N. van Oosterom. Operation-based merging. In *Proceedings of the fifth ACM SIGSOFT symposium on Software development environments - SDE 5*, New York, New York, USA, 1992. ACM Press.

[19] E. Lippe and N. van Oosterom. Operation-based merging. *Softw. Eng. Notes*, 17(5):78–87, Nov. 1992.

[20] J. W. Menezes, B. Trindade, J. F. Pimentel, T. Moura, A. Plastino, L. Murta, and C. Costa. What causes merge conflicts? In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, New York, NY, USA, Oct. 2020. ACM.

[21] T. Mens. A state-of-the-art survey on software merging. *IEEE trans. softw. eng.*, 28(5):449–462, May 2002.

[22] N. Nelson, C. Brindescu, S. McKee, A. Sarma, and D. Dig. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empir. Softw. Eng.*, 24(5):2863–2906, Oct. 2019.

[23] G. Seibt, F. Heck, G. Cavalcanti, P. Borba, and S. Apel. Leveraging structure in software merge: An empirical study. *IEEE trans. softw. eng.*, 48(11):4590–4610, Nov. 2022.

[24] H. Shen and C. Sun. A complete textual merging algorithm for software con .guration management systems. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004*. IEEE, 2004.

[25] H. Shen and C. Sun. Syntax-based reconciliation for asynchronous collaborative writing. In *2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE, 2006.

[26] C. Wohlin, M. Höst, and K. Henningsson. Empirical research methods in software engineering. In *Empirical Methods and Studies in Software Engineering*, Lecture notes in computer science, pages 7–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[27] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer, Berlin, Germany, 2012 edition, June 2012.

[28] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Trans. Softw. Eng. Methodol.*, 1(3):310–354, July 1992.

[29] N. N. Zolkifli, A. Ngah, and A. Deraman. Version control system: A review. *Procedia Comput. Sci.*, 135:408–415, 2018.