b
UNIVERSITÄT
BERN

In Conflict

An Empirical Study of Merge Conflict Resolutions in Open-Source Projects

Bachelor's Thesis

Yael van Dok

Supervised by

Prof. Dr. Timo Kehrer

M. Sc. Alexander Boll

Software Engineering Group

Institute of Computer Science

Faculty of Science

University of Bern

September 10, 2023

Abstract

Modern software development often entails the usage of a version control system such as Git to facilitate collaborative work. Whenever work from multiple parties needs to be integrated into a shared code base, so-called merge conflicts are bound to occur. Resolving such merge conflicts must be done manually by the developers, and doing so can be a straining endeavour. In this study, we investigate the feasibility of a generative approach to merge conflict resolutions that might ease the strain on the developers. By generating all possible resolutions to the merge conflicts and presenting the best-performing to the developers, their workload could be lessened. To investigate the feasibility, we analysed 7'866 open-source projects of various programming languages on GitHub. We determined whether the developers resolved the merge conflicts by using so-called derivable resolutions, i.e., resolutions that can be derived from the preceding commit history, and which in turn could be generated automatically using our approach. We performed this analysis on multiple levels of granularity. We analysed 140'705 conflicting merges, which in turn contained 644'203 conflicting files, which in turn contained 1'451'231 conflicting chunks. We found that (i) in about 75 % of all cases, developers picked derivable conflicting chunk resolutions, and in about 66 % of all cases, they picked derivable conflicting file resolutions, and in about 35 % of all cases, they picked derivable conflicting merge resolutions. Furthermore, we found that (ii) in about 80 % of all cases, conflicting merges were found to have between one and five conflicting chunks. Their derivability rates lie between 49 % and 20 %. Further, (iii) conflicting merges with one conflicting chunk constituted about 45 % of all analysed conflicting merges, and their derivability rate lies at about 49 %. These results indicate that the generative approach we envision is feasible, as it could automate a substantial amount of the merging scenarios we encountered at a low computational cost.

Contents

1 Introduction

Modern software development is most often a collaborative effort. A team of developers works on a project with a shared code base, changing, adding and removing work simultaneously. It is crucial to know what work was introduced when and by whom, such that overseeing the development process and managing different versions of the project is possible. Such an intertwined workflow introduces the need for an organisational tool that keeps track of the shared resources' state, dimension and history.

To this end, so-called version control systems (VCS) are used. [7] Multiple such tools exist, all with their own special feature sets, advantages and disadvantages. At their core, they all track the contents within a specified project and document their changes over time. [5, p. 4-5]

In this work, we focus on Git, one of the most popular VCS.[1] Git is an open-source command-line tool that is used by both large software companies like Microsoft[2] and Google[3] as well as independent developers. It is suited for many different types of projects, as it is very versatile and places few limitations on the specifics of the projects it can manage.

Git can be conceptualised as a set of tools that act on a specified file system, namely the project folder. It provides features to add, alter and remove files and folders from within

[1] https://git-scm.com/
[2] https://github.com/microsoft/
[3] https://github.com/google/

that file system, and it keeps track of all these changes within a designated history. Furthermore, it offers a branching system, making concurrent work on the same files possible. As it is a distributed VCS, meaning that each developer hosts their own copy of the project on their local machine, making them independent of a central server, it lends itself well to remote and offline work. Supplementary features such as the tagging system make version management more convenient.

For all the streamlining Git brings to collaborative development, there are still instances where the developers' workflow is interrupted and they must step in to resolve issues. Namely, whenever multiple parties want to integrate their work to a shared code base (in a process called *merging*), problems may arise. While Git provides its own merging techniques that are able to automatically integrate a large number of changes from multiple sources into a shared code base, in some cases, these algorithms fail, and the developers are tasked with resolving the conflicts themselves. Such conflicts (called *merge conflicts*) must be resolved manually, as Git's merging algorithms do not account for syntactical correctness or semantic meaning of the files they merge, and therefore do not guarantee an error-free result. Ensuring these qualities in the subsequent merge conflict resolution is an important task the developers must undertake themselves.

Resolving merge conflicts can be a straining endeavour. It pulls the developers out of their normal workflow and forces them to take a step back to assess unfamiliar code and find solutions to (sometimes) complex problems. [6] This disrupts the development process, and therefore, it is in the developers' interest to keep such instances of merge conflicts at a minimum.

As of now, there is no other solution than resolving merge conflicts manually. In this work, we investigate the feasibility of a new approach to resolving merge conflicts: By generating all possible resolutions to all conflicts within a merge, ranking them in terms of their syntactical and semantic correctness, and providing the best performing to the developers to choose from, the developers' workload could be reduced. Furthermore,

the risk of introducing errors that occur easily when creating a manual resolution could be lowered. Generating such merge conflict resolutions would be a complex task, as all conflicts within all files within the merge must be taken into account, but the benefits of automating their resolutions would warrant the effort.

This thesis investigates the feasibility of such a generative approach to merge conflict resolutions. To this end, we analysed a selection of open-source projects hosted on GitHub[4], and determined how the developers resolved the merge conflicts that arose during these projects' development. We analysed whether the developers resolved the conflicts in a so-called *derivable* way, meaning that they stuck to resolutions that can be generated (or derived) automatically from the parent merges and their files. We performed this analysis on multiple levels of granularity (on a conflicting merge, conflicting file and conflicting chunk level) to gain detailed insights into how exactly developers resolve their conflicts. This insight allows us to draw conclusions about the feasibility of a merge conflict resolution generator.

1.1 Research Questions

Our goal in this work is to investigate the feasibility of a merge conflict resolution generator. To this end, we introduce the following research questions:

- RQ1: How often do developers pick derivable merge conflict resolutions, on a conflicting chunk, conflicting file and conflicting merge level?
- RQ2: What are the specific properties of such derivable resolutions?

These questions relate to the feasibility in the following ways: If we find that in many (or most) cases, developers pick derivable resolutions, then there is high chance for suc-

[4] `https://github.com/`

cess of such a merge conflict resolution generator, as it would automate the actions these developers performed to resolve the merge conflicts (RQ1). If the derivable resolutions they picked furthermore have specific properties like low numbers of merge conflicts within the files of the merges, then this would be an indicator for the feasibility, as their resolutions could be generated at low computational cost (RQ2).

1.2 Main Contributions

Our work contributes to the research in the following ways:

- We provide a tool (*GitAnalyzerPlus*) that allows for the analysis of merge conflict resolutions within projects managed by Git on multiple levels of granularity.
- We provide quantitative results of such an analysis on a selection of 7'866 projects hosted on GitHub.
- With these two results, we can partially decide the feasibility of a generative approach to resolving merge conflicts.

1.3 Structure of this Work

We go into detail about what exactly derivable merge conflict resolutions are and what our generative approach to them looks like in Chapter 2. We go into related work to such generative approaches in Chapter 3. We explain how we obtained the results of our analysis in Chapter 4, and the results of this analysis are shown in Chapter 5. Possible threats to the validity of our results are found in Chapter 6, and a discussion of our results is found in Chapter and 7. We conclude our study and discuss possible future work in Chapter 8.

2 Background and Motivation

This Chapter provides the necessary theoretical background on Git and its merge procedure. It further introduces the terminology we will use in the subsequent Chapters, and at last, it goes into the details of the generative approach to merge conflict resolutions whose feasibility we study in this work.

2.1 Git

Launched in 2005 and still under active open-source development, Git has gathered huge popularity over the years, to the result that it is the most widely used VCS in open-source software development spaces nowadays. Being a command-line tool, it is available for all major operating systems and runs on a multitude of different hardware. Many IDEs and code editors integrate Git's functionalities into their interfaces, and hosting sites such as GitLab[5] and GitHub[6] are specifically designed to host Git projects and make use of its features.

Git offers many features and tools that facilitate version management and collaborative work. For the sake of brevity, we focus only on the components that are relevant to this thesis here, namely the branching and commit system. All of the information provided

[5] https://about.gitlab.com/
[6] https://github.com/

here is taken from *Pro Git* [3]. A revised and updated version of said book can be found on Git's official website.[7]

2.1.1 Commits

One of the central tasks of any VCS is the tracking of a project's evolution over time, i.e., the tracking of its files and their changing versions. [7] Git achieves this by using so-called *commits*. A commit is essentially a snapshot of the project at a specific point in time. It includes information about the current project state, namely its folder structure and the files and their contents within. Commits are issued by developers, and upon their creation, they are inserted into the project's history, the so-called *commit history*. Whenever a developer sets up a new project with Git, an initial commit is issued, and this commit marks the root of the project's commit history. Subsequent commits are appended to this root commit.

A new commit can only be issued if there is change between the project's current state and the state specified in the most recent commit. Such changes include additions or deletions of and alterations to files. Whenever such a change has occurred, and the developer has created a commit, the commit is inserted into the history, marking a new state of the project. This state then becomes the current one, and a new commit can only be issued if it differs from this most recent state.

It is important to note that the developer can restore any project state included in the commit history by *checking out* the commit in question. This action will revert all project resources back to their state specified in the commit, making it possible to retrace every step of the project's history.

[7] `https://git-scm.com/book/en/v2`

2.1.1.1 Example

To illustrate the concepts so far, we give the following example: Developer Alice wants to set up a new project with Git. She creates her first files (main.py and cfg.py) and issues a first commit A (which includes said files), thereby setting up the main branch (see Section 2.1.2). She then makes some changes to main.py. Its state now differs from the one specified in commit A. She includes these changes in a new commit B which is appended to commit A.

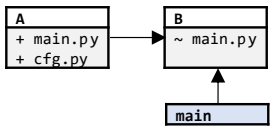

Figure 2.1: Commit history of Alice's project. The alphabetically named boxes represent the issued commits with their included changes to the specified files. + indicates a file being added, ~ indicates a file being altered, - indicates a file being deleted. The arrows in between the commits illustrate the progression of the project's history. The blue boxes represent branches. The arrows extending from them to certain commits represent their current location.

2.1.2 Branches

To structure the commit history and allow for concurrent development on the same files, commits are placed within so-called *branches*. Branches mark specific chains of development. Upon creation of a project, a default branch is set up. This branch is typically called the main (in older versions master) branch. If the developer wants to work on a specific issue and does not want to include their changes in the main branch

(as the convention in many software development teams is that this branch is reserved for major versions of the system only), they create a new branch and switch to it. This new branch marks a deviation from the current branch (the main branch) in the project's history. Starting from the commit they created the branch from, a divergent path of commit history is opened up, and any commits they make will be placed within that branch.

Commits on different branches do not interfere with one another, since they are kept in separate parts of the history. They are not aware of each other's existence, and they only interact whenever the developer issues a merge, a concept that we will go into later. It is important to note that branching is not restricted to the main branch. Any branch in the history can be branched from, allowing for multiple levels of sub-branches.

2.1.2.1 Example

We use our previous example to illustrate the concept of branching:

Alice's project so far includes the default branch main which contains two commits A and B. She now wants to work on some special features that she doesn't want to include in the main branch (yet). Therefore, at commit B, the latest commit of branch main, she creates a new branch dev and switches to it. The subsequent commits C (alterations to main.py, insertion of db.py) and D (alterations to db.py) are placed within this branch, and the main branch is not aware of their existence and any changes they include. After she has completed her work on dev, she switches back to main, and issues commits E (insertion of alg.py) and F (alterations to main.py, deletion of cfg.py). These commits are not known to dev either, as its history has deviated from main at commit B. Finally, she switches back to branch dev, and creates another branch called exp. On this branch, she issues a commit G, which includes alterations done to main.py. Like the dev branch, branch exp is not aware of any commits done to main after their deviation point at

commit C, and it will also not be aware of any new commits added in branch dev after commit D.

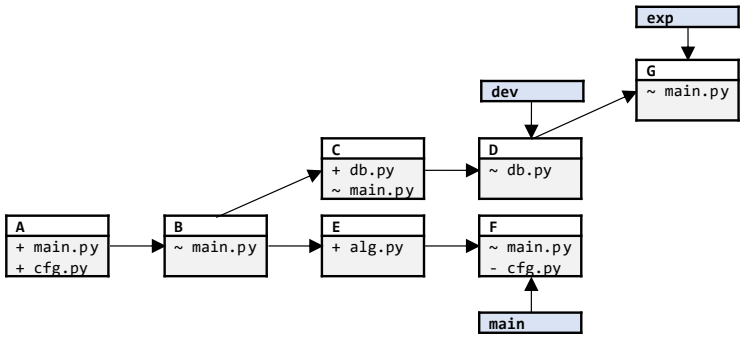

Figure 2.2: Updated commit history of Alice's project. The alphabetically named boxes represent the issued commits with their included changes to the specified files. + indicates a file being added, ~ indicates a file being altered, - indicates a file being deleted. The arrows in between the commits illustrate the progression of the project's history. The blue boxes represent branches. The arrows extending from them to certain commits represent their current location.

As can be seen in Figure 2.2, the commit history of a Git project is a rooted connected directed acyclic graph.[8] The initial commit A represents the root of this graph. Each commit is a node and it represents all changes of the project from the commit's parent(s), i.e., its direct predecessor nodes.

To build a complete snapshot of the project when checking out a certain commit, Git replays the cumulative changes of the complete commit path from the root to the specified commit. As an example, to build the snapshot of commit D, Git will replay the changes from the root commit A, commit B commit C and lastly, commit D.

[8] https://en.wikipedia.org/wiki/Directed_acyclic_graph

2.1.3 Merging

While commits on separate branches do not affect one another, at some point, developers may want to merge one branch into the other to combine their changes. To do so, they issue a *merge*. A merge is a command that instructs Git to include all the changes made in one branch in another one. Git offers several merging algorithms to achieve this. In this work, we only consider the default merge strategy, namely the three-way merge, using the default merge algorithm ORT.[9]

The three-way merge strategy takes the latest commit of the branch to be merged, the latest commit of the branch to be merged into, and the latest commit that is present in both branches (the base commit), i.e., the one from where the two branches deviated. Git processes all the commits on both branches starting from the deviation point (i.e., the base commit, the last common ancestor in the commit history), and, where possible, integrates all the changes together, resulting in a new commit that contains all the changes from both branches. The resulting files are then included in this new commit, the so-called *merge commit*. In the commit history, such a merge is represented as the union of the two branches.

2.1.3.1 Example

To illustrate such a merge, we adapt our example from earlier:

Alice's project has three branches main, dev and exp. She has issued work on all branches, but now, she wants the work from dev to be included in main. To this end, she switches to branch main (on commit F) and starts a merge of branch dev into branch main. Git considers the latest commit on dev (commit D), the latest commit on main (commit F), and the latest commit both branches share (commit B). It gathers up all

[9] `https://git-scm.com/docs/git-merge`

the changes on dev from commit B to D, as well as all the changes on main from commit B and to F, and it tries to replay them all on the commit B. If successful, this will lead to a new commit H which includes all the changes from main and dev, applied to their common ancestor commit B. The branch exp is not affected by this merge, as it deviates from dev on commit D, and its changes will not be included in the merge.

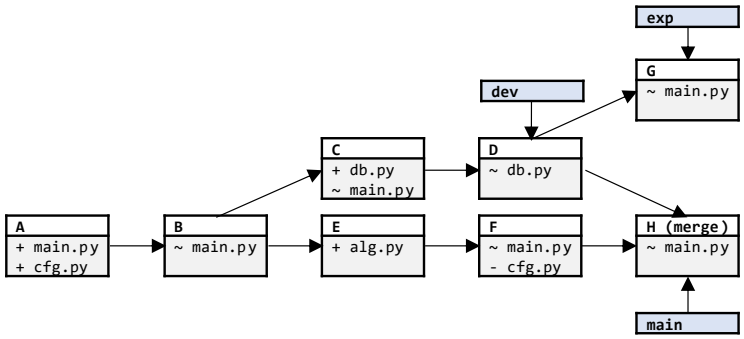

Figure 2.3: Three-way merge of branch dev into branch main. As before, the alphabetically named boxes represent the issued commits with their included changes to the specified files. + indicates a file being added, ~ indicates a file being altered, - indicates a file being deleted. The arrows in between the commits illustrate the progression of the project's history. The blue boxes represent branches. The arrows extending from them to certain commits represent their current location.

2.1.4 Merge Conflicts

Git merges files on a line-basis, meaning that Git processes each file and tries to combine all changed lines within all commits into one final file. It is during this process where problems may arise, namely when both branches contain commits that change the same files on the same lines. In this case, Git does not know which change (i.e., which code from which branch) to include in the file. Git's merging algorithms make no considerations to syntax or semantics, meaning that there is no heuristic for Git to determine which changes to include in the file. Whenever such a merge conflict occurs, Git signals to the developer that they must step in to resolve the issue. They do so by examining the files

in question, creating working versions of the code, and committing the result. As soon as they have committed, the merge is considered resolved.

2.1.4.1 Example

To illustrate what such merge conflicts look like, we, again, use our previous example:

Alice has issued the merging of branch dev into branch main. Some of the changes on branch dev (namely the insertion and alteration of db.py in commit C and commit D) will be integrated without issues, since these files exist only on dev and therefore do not interfere with any files on main. Analogously, the insertion of alg.py on main in commit E integrates without issues, as does the deletion of cfg.py in commit F, as no alterations to this file were issued on dev. However, as both branches issue changes to main.py (in commit C and F) on the same lines, a merge conflict ensues, and Alice is tasked with resolving it.

```
1  def main():
2  <<<<<<< main (HEAD)
3    print('Hello!')
4  =======
5    print('Greetings!')
6  >>>>>>> dev
7
8  if __name__ == '__main__':
9    main()
```

Figure 2.4: The main.py file during the merge. Non-conflicting code is overlayed green, whereas conflicting code is overlayed red. Git's merging markers are overlayed grey.

Git gathers all conflicting code within so-called *merging markers* to visually highlight the conflict. In our example, Git adds merging markers in lines 2, 4 and 6. Between merging marker 1 (line 2) and merging marker 2 (line 4) lies the code that was issued in branch main, and between merging marker 2 (line 4) and merging marker 3 (line 6) lies the code that was issued in branch dev.

In the subsequent work, we use the term *conflicting chunk* (CC) for the sequence of lines from merging marker 1 to merging marker 3 (lines 2 up to and including 6 in

our example). One conflicting chunk represents one merge conflict. We call the file the conflicting chunk belongs to (main.py in our example) the *conflicting file* (CF), and analogously, we call the merge the conflicting file belongs to (commit H in our example) the *conflicting merge* (CM).

It is important to note that in this example, there is only one conflicting chunk within the conflicting file, as only one segment of the code conflicts. However, in case of multiple conflicting code segments separated by non-conflicting code, the conflicting file will have multiple conflicting chunks.

2.2 Merge Conflict Resolutions

To resolve such conflicting chunks, the developers have several options. We call these options *conflicting chunk resolutions* (CCR). They can be grouped into three main categories.

First, they may pick the code from the first branch, disregarding the the code from the second branch, or vice versa. We denote such a resolution a *canonical* conflicting chunk resolution. Second, they may opt to include both code segments in the resulting file, or none at all (meaning that they delete all conflicting lines). In this case, we call the resolution a *semi-canonical* conflicting chunk resolution. The third option they have is to disregard both options and include their own custom code. We call such a conflicting chunk resolution *non-canonical*.

2.2.1 Example

In our example, there is only one conflicting chunk, which Alice can resolve in the following ways:

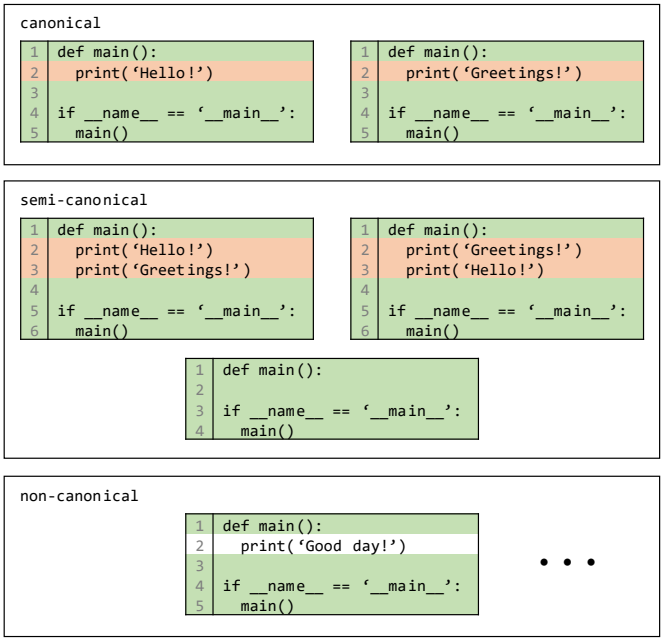

Figure 2.5: Possible conflicting chunk resolutions of the conflicting chunk within main.py, grouped within the three categories.

Whatever conflicting chunk resolution (CCR) Alice chooses to resolve the conflict, as soon as she has, she commits the resulting file. This file represents a *conflicting file resolution* (CFR). As there was only one conflicting file within the conflicting merge (which she has now resolved), the merge is complete, and the resulting commit is a *conflicting merge resolution* (CMR). This marks the merging process as complete, and she can carry on with her work.

In this example, there is only one conflicting file within the conflicting merge, with only one conflicting chunk in its code. In reality, there may be multiple conflicting files with multiple conflicting chunks each, which makes resolving the conflicts more complex. The conflicting files and conflicting chunks may be interlinked or depend on each other syntactically or semantically, necessitating more considerations when resolving them, such that in the end, syntactical and semantic integrity of all files is maintained.

2.3 A Generative Approach to Merge Conflict Resolutions

As stated in Chapter 1, we want to investigate the feasibility of a tool that automatically generates all possible merge conflict resolutions to a conflicting merge. We laid out in Section 2.1.4 what conflicting merges, conflicting files and conflicting chunks are and how they come about. Now, we will use the terminology and concepts introduced in Section 2.2 (conflicting chunk resolutions and their three categories, conflicting file resolutions and conflicting merge resolutions) to illustrate the generative approach whose feasibility we want to investigate.

2.3.1 Derivability

A tool that automatically generates merge conflict resolutions can only generate resolutions that can be derived from existing resources, i.e., from the project's commit history. Therefore, we introduce the following concepts.

2.3.1.1 Derivability of Conflicting Chunk Resolutions

The canonical and semi-canonical conflicting chunk resolutions introduced in Section 2.2 are pre-determined and can be derived from the existing commit history, as they only contain code from the three file versions used in the three-way-merge (i.e., the files from both parent commits and the file from their common ancestor commit). We therefore call these conflicting chunk resolutions *derivable*. As is illustrated in Figure 2.5, there are two canonical conflicting chunk resolutions, and three semi-canonical ones. Non-canonical conflicting chunk resolutions are custom-made by the developer at the time of the merge, and therefore, they cannot be derived from the preceding commit history. We call this category of conflicting chunk resolutions *non-derivable*. In theory, there are infinitely many such non-canonical conflicting chunk resolutions, as the developer has infinitely many choices to resolve the conflict such that syntactical and semantic correctness of the code is maintained.

2.3.1.2 Derivability of Conflicting File Resolutions

We expand this concept of derivability to conflicting file resolutions by stating the following: A conflicting file resolution is *derivable* if all the conflicting chunks within were resolved in a derivable manner, i.e., if they were resolved canonically or semi-canonically. Further, it must hold that none of the non-conflicting lines that surround these conflicting chunks were changed. This is necessary, because even if all conflicting chunks within the conflicting file are resolved in a derivable manner - if the code surrounding them has changed, then the file itself cannot be derived automatically from its parent files, making its automatic generation impossible.

In the subsequent work, we will refer to such non-conflicting code segments within conflicting files as *contexts*. Therefore, to reiterate, a conflicting file resolution is derivable if all its conflicting chunks were resolved in a derivable way and none of its contexts were changed. If this does not hold, the conflicting file resolution is *non-derivable*.

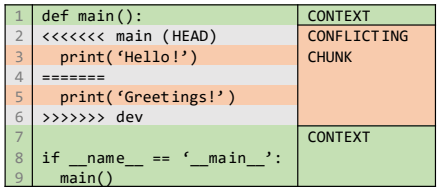

Figure 2.6: Illustration of the concept of contexts, applied to the main.py file from our example. Line 1 marks the first context, lines 2-6 mark the conflicting chunk, and lines 7-9 mark the second and last context.

2.3.1.3 Derivability of Conflicting Merge Resolutions

As we laid out in Section 2.2, a conflicting merge can have multiple conflicting files. We therefore expand the concept of derivable conflicting file resolutions to conflicting merge resolutions by stating that a conflicting merge resolution is *derivable* if all its conflicting

file resolutions are derivable. If at least one of them is non-derivable, the conflicting merge resolution itself is *non-derivable*.

2.3.2 Complexity

Using this concept of derivability on multiple levels, we can now illustrate what the generative approach whose feasibility we investigate looks like. Namely, we envision a merge conflict resolution generator that generates derivable conflicting merge resolutions. As detailed in the previous Section 2.3.1, derivable conflicting merge resolutions consist of derivable conflicting file resolutions. Any such derivable conflicting file resolution in turn consists of derivable conflicting chunk resolutions.

These considerations have the following implications for the size of the set of derivable conflicting merge resolutions:

Notation 2.1
Let m be a conflicting merge with conflicting files f_i for $i \in \{1, ..., n\}$ where $n \in \mathbb{N}_{\geq 1}$. Every conflicting file f_i has conflicting chunks c_{ij} for $j \in \{1, ..., m_i\}$ where $m_i \in \mathbb{N}_{\geq 1}$.

Let $R_{CC}(c_{ij})$ be the set of resolutions for conflicting chunk c_{ij}. As there are five derivable resolutions (two canonical ones and three semi-canonical ones) for every conflicting chunk, it holds that:

$$|R_{CC}(c_{ij})| = 5$$

Let $R_{CF}(f_i)$ be the set of resolutions for conflicting file f_i. As every derivable conflicting file resolution is a combination of such derivable conflicting chunk resolutions, it holds that:

$$|R_{CF}(f_i)| = \prod_{j=1}^{m_i} |R_{CC}(c_{ij})| = \prod_{j=1}^{m_i} 5 = 5^{m_i}$$

Let $R_{CM}(m)$ be the set of resolutions for conflicting merge m. As every derivable conflicting merge resolution is again a combination of such derivable conflicting file resolutions, it holds that:

$$|R_{CM}(m)| = \prod_{i=1}^{n} |R_{CF}(f_i)| = \prod_{i=1}^{n} 5^{m_i} = 5^{\sum_{i=1}^{n} m_i} = 5^K \text{ with } K = \sum_{i=1}^{n} m_i$$

2.3.3 Generational Cost

As shown in Notation 2.1, the cardinality of the set of derivable conflicting merge resolutions grows exponentially with the number of conflicting chunks. This can pose a serious issue for finding the best solution in this set, since simply generating all derivable conflicting merge resolutions would be very costly as the number of conflicting chunks rises. A tool that generates all possible derivable conflicting merge resolutions might therefore only be viable if the amount of conflicting chunks is low. This consideration motivates our second Research Question (see Section 1.1). If our study finds that most conflicting merges only contain a low amount of conflicting chunks, then this bodes well for the feasibility of such a generator.

2.3.3.1 Dependency of Merge Conflicts

Simply generating the whole set of derivable conflicting merge resolutions may not be viable for conflicting merges with many conflicting chunks. Therefore, it is important to consider strategies that limit the resolution set of the generation. One such strategy is grouping conflicting chunks that depend on each other syntactically or semantically together and generating their resolutions independently of other non-dependent chunks. This reduces the complexity of the generation by breaking up the generation of all conflicting merge resolutions into sub-generations of conflicting chunk and conflicting file resolutions that can be conducted independently of each other. Further, by inspecting the nature of the dependencies between these conflicting chunks, one can exclude conflicting file resolutions and conflicting merge resolutions that will lead to invalid code from the generation beforehand. This will further limit the size of the resolution set.

We illustrate these strategies by giving three examples:

Example 2.1
Suppose that there is a conflicting merge m from branch A into branch B. m contains one conflicting file f which contains two conflicting chunks c_1 and c_2. Suppose that branch A places within these chunks some code segments that depend on each other. Namely, in c_1, some method $func$ is defined, and in c_2, said function $func$ is invoked. In this case, a resolution for c_2 cannot be proposed independently of the resolution of c_1.

In cases like Example 2.1, all conflicting chunks depend on each other. When generating the resolutions for this conflicting merge, these dependencies must be taken into account, and the obvious brute-force strategy to account for them is to simply generate all derivable conflicting merge resolutions. As explained in Notation 2.1, this computation grows exponentially and is only viable for small conflicting chunk counts. In this Example, many of these generated conflicting merge resolutions will be invalid, namely when $func$ is invoked in the resolution of c_2, but the resolution of c_1 does not contain the definition of $func$.

In such cases, one could investigate the nature of the dependencies beforehand to determine what combinations of conflicting chunk resolutions will lead to invalid results. One can then exclude them from the resolution set of the generation. In our Example 2.1, one could exclude conflicting merge resolutions that invoke method $func$ in the resolution of c_2 while not including the method definition in the resolution of c_1. This could dramatically reduce the resolution set.

Example 2.2
Suppose that there is a conflicting merge m from branch A into branch B. m contains one conflicting file f which contains two conflicting chunks c_1 and c_2. Suppose that branch A places within these chunks some code segments that do not depend on each other. Namely, in c_1, it sets the background colour of some element within a graphical application, and in c_2, it alters the name of another element within said

application. In this case, resolutions for c_2 can be proposed independently of the resolution of c_1, as c_2 does not reference any variables used in c_1 and vice versa.

In cases like Example 2.2, as the conflicting chunks do not depend on each other, one can independently generate their resolutions. Afterwards, one can test which resolution for one conflicting chunks is the best performing by fixing the resolution for the other conflicting chunk, and analysing the different resolutions. Once such an optimal resolution is found, one can then fix this resolution and repeat the process for the other conflicting chunk.

In our Example 2.2, one could first generate the conflicting chunk resolutions for c_1 and c_2, and find a combination that compiles. Then, one could fix a resolution for c_1 and compile and test all different resolutions for c_2 independently. Next one can fix the best resolution for c_2 and try out all resolutions for c_1. This would result in linear growth, once a compiling combination of conflicting chunk resolutions is found.

Example 2.3
Suppose that there is a conflicting merge m from branch A into branch B. m contains two conflicting files f_1 and f_2, which in turn contain conflicts c_1 and c_2. Suppose that f_1 and f_2 belong to two entirely separate programs found within the same project. In this case, the resolutions for c_1 and c_2, and subsequently f_1 and f_2 do not depend on each other at all.

In cases like Example 2.3, as the conflicting chunks and conflicting files do not depend on each other at all, it is possible to generate the derivable conflicting file resolutions for both conflicting files independently of each other, and determine for each resolution set which one is the best performing.

In our Example 2.3, one could therefore generate the conflicting file resolutions for both f_1 and f_2, and test for each resolution set which one is the best performing. As both conflicting files only contain one conflicting chunk, this would result in linear growth of the generation.

2.3.3.2 Implications for our Study

As we do not perform any syntactical or semantic analysis of the merge conflicts in our study, we cannot say in how many cases of conflicting merges their conflicting chunks depend on each other, and further, of what nature these dependencies are. Therefore, we cannot determine in how many cases the strategies outlined above could be applied.

However, we can investigate their feasibility in terms of complexity. If our study finds that most conflicting merges only contain a low number of conflicting chunks, this bodes well for the feasibility, because even in the worst case scenario where all conflicting chunks depend on each other, and the whole set of derivable conflicting merge resolutions would need to be generated, this generation could be performed at low computational cost. These considerations are reflected in our second Research Question (see Section 1.1).

3 Related Work

Before we discuss the methodology of our study, we go into other work that has been done in the realm of merge conflict resolution. As our study is a direct continuation of another empirical study conducted by Severin Buchser [2], we lay our main focus on how our approach differs and goes beyond said study's approach, mitigating several of its limitations in the process. Afterwards, we will briefly discuss other work.

3.1 Buchser's Study

In his study, Buchser analysed how developers resolve their merge conflicts on a conflicting chunk level, conflicting file and conflicting merge level. He analysed 8000 open-source projects hosted on GitHub. The goal of his study was identical to ours - investigating the feasibility of a merge conflict resolution generator.

He analysed the conflicting chunks found within the conflicting merges and their resolution rates - namely if they were resolved canonically or non-canonically. Using this conflicting chunk resolution analysis, he determined whether the conflicting file resolutions and further the conflicting merge resolutions were canonical, i.e., whether they only contained canonical conflicting chunk resolutions.

He found that while 80.63 % of all conflicting chunks were resolved canonically, only about 46.29 % of conflicting file resolutions and 34.11 % of conflicting merge resolutions were made up using only canonical conflicting chunk resolutions.

3.2 Differences of our Approach

Our work builds directly on top of Buchser's, but it differs on both a conceptual and technical level. We will discuss these differences here. The details of our methodology can be found in Chapter 4, and we provide a comparison of our study results with Buchser's in Section 5.3.

3.2.1 Expanded Analysis of Conflicting Chunk Resolutions

In his study, Buchser determined whether conflicting chunks were resolved canonically or non-canonically. Our analysis goes further and determines whether conflicting chunks were resolved canonically, semi-canonically or non-canonically. This allows for a more detailed insight into the conflicting chunk resolutions, and therefore, into the merging behaviour of the developers.

Furthermore, our analysis takes the relative position of each conflicting chunk within the conflicting file into account when analysing its resolutions. Whereas Buchser's approach marks a conflicting chunk as being resolved canonically if its code is found anywhere in the conflicting file resolution, our approach only does so if it is found at its correct position within the code. This eliminates the possibility of false positives when marking canonical conflicting chunk resolutions.

3.2.2 Consideration of Contexts and Derivability

As we detailed in Section 2.3.1, in order for a conflicting file resolution to be derivable, and therefore possible to be automatically generated, all conflicting chunks found within must be resolved canonically or semi-canonically, and all contexts must remain unchanged.

Buchser did not consider contexts within his study. He only analysed the conflicting chunk resolutions and did not determine whether contexts were changed. As our approach also takes them into account, it paints a more complete picture of how developers

resolve their merge conflicts and whether they stick to derivable resolutions or not. This, in turn, allows us to investigate the feasibility of a merge conflict resolution generator more accurately.

3.2.3 No Limitations of Conflicting Chunk Counts

In his study, Buchser only analysed conflicting merges with at most 12 conflicting chunks. This limit was set due to the approach and implementation of his analysis tool. Our analysis approach is different, and it eliminates this limitation, allowing for unconstrained study sets. Furthermore, the implementation of our analysis tool guarantees better scalability in general, as it uses multi-threading and result aggregation within databases (see Chapter 4).

3.3 Other Related Work

Apart from the study we discussed above, not much work has gone into the analysis of merge conflict resolutions. Ghiotto et al. [4] conducted an empirical study of merge conflicts found within the commit histories of more than 2'731 open source Java projects. First, they analysed five Java projects and their about 1'000 conflicting merges manually, to gather insight into the types of merge conflicts most often encountered. Then, they ran an automated analysis of 2'731 Java projects. Within these projects, they analysed 25'328 conflicting merges, which included 175'805 conflicting chunks. Their main findings were that 87 % (automated analysis) and 83 % (manual analysis) of all conflicting chunks contained code that could be wholly derived from the parent files. Further, about 60 % of conflicting merges contained more than one conflicting chunk, and in 14 % to 46 % of these cases, the conflicting chunks were dependent on each other.

4 Methodology

This Chapter details our experimental setup and lays out the analysis algorithm we use to gather our results.

4.1 Experimental Subjects

To get comparable results to Buchser's study, we chose to use the same projects that he analysed in his work. He specified six programming languages (Python, Java, C++, Go, JavaScript and TypeScript), and for each of said languages, he selected 500 of the most highly rated projects on GitHub that are marked as belonging to that language. In addition, he made two selections of 2500 projects with randomly selected names. The first selection includes projects with high ratings, the other includes projects with low ratings. In both of these cases, the projects could use any programming language. This leads to a total of 8000 projects.

In the meantime of our study and his, 1.675 % of the these projects were deleted or did otherwise become unavailable on GitHub. We opted not to replace them with other projects, and so we analysed the remaining 7'866 projects.

4.2 Analysis

We forked Buchser's tool *GitAnalyzer* and adapted it to allow for our more elaborate approach (see Section 3.2). We called it *GitAnalyzerPlus* to reflect its origin. *GitAnalyzerPlus* is a command-line tool written in Java which allows for the cloning and merge

conflict analysis of Git projects. It uses the GitHub REST API[10] to acquire the repositories and it uses JGit[11], an implementation of Git written in Java, to run the merge conflict resolution analysis detailed below. The source code is accessible for faculty members on the GitLab server of the University of Bern.[12]

4.2.1 Workflow

GitAnalyzerPlus takes in a list of GitHub projects, clones their repository to the user's local machine and runs its merge conflict resolution analysis on each project. The merge conflict resolution analysis runs multi-threaded on multiple levels (on a project level, on a conflicting merge level, and on a conflicting file level), and its results are stored in a local database.

4.2.1.1 On Project Level

For every project, all commits in the history that are marked as merges are scheduled for analysis. We limit our analysis to merges with two parents, as Buchser found in his work that this type of merge makes up the vast majority of all merges. [2, p. 27]

4.2.1.2 On Conflicting Merge Level

For every merge, its parent commits are pulled up and re-merged using Git's default merging strategy (three-way merge with the ORT merging algorithm). If the resulting merge contains conflicts, meaning that at least one of the files within the merge could not be merged together from its parent files and their common ancestor without conflicts,

[10] `https://docs.github.com/en/rest?apiVersion=2022-11-28`
[11] `https://eclipse.dev/jgit/`
[12] `https://gitlab.inf.unibe.ch/SEG/theses/git-analyzer-plus`
For non-faculty members: If you would like to access the code, contact us via email at yael.vandok@students.unibe.ch.

then the merge is marked as a conflicting merge, and the files that contain said conflicts are marked as conflicting files.

4.2.1.3 On Conflicting File Level

For every conflicting file, its merge conflicts and the file variants found in the commits involved in the three-way merge (i.e., the file variants found in both parent commits of the conflicting merge and their common ancestor) are formatted together. This results in a file that is structurally equivalent to the one Git uses during the merging process (see Figure 2.4). This file includes the conflicting chunks the developer had to resolve during the merge. We dub this file the *unmerged resolution file* (URF).

The file variant found within the conflicting merge itself is the one the developer committed to resolve these conflicting chunks. We dub this file variant the *actual resolution file* (ARF).

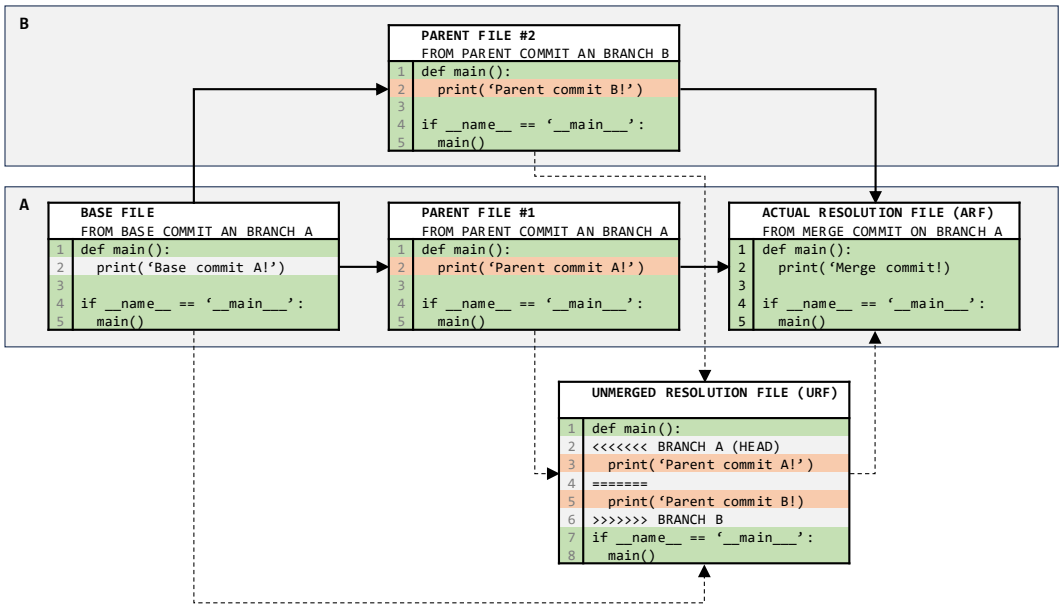

Figure 4.1: Different file variants involved in the recreation of a conflicting merge of branch B into branch A. The base file variant and both parent file variants are formatted together along with their merge conflicts, resulting in the unmerged resolution file (URF) which contains the conflicting chunks. The file variant found in the conflicting merge, i.e. the actual resolution file (ARF), contains the resolutions to these conflicting chunks. Note that all files except the unmerged resolution file are included in the project's commit history in their respective commits.

As our goal is to determine how the developers did resolve the merge conflicts, we compare the unmerged resolution files and the actual resolution files to determine how they differ. This gives us insight into whether the conflicting chunks were resolved in a derivable manner and whether the contexts have changed, which in turn allows us to determine whether the conflicting file resolution is derivable.

4.3 File Comparison

The comparison between an unmerged resolution file (subsequently called URF) and its associated actual resolution file (subsequently called ARF) happens in multiple steps. To illustrate them, we will use a running example with the two files specified below.

	UNMERGED RESOLUTION FILE (URF)	SECTION	INDEX	MAPPED
1	def main()			
2	# say hello			
3	<<<<<<< BRANCH A (HEAD)			
4	print('Hello!')			
5	=======			
6	print('Greetings!')			
7	>>>>>>> BRANCH B			
8				
9	# give compliment			
10	<<<<<<< BRANCH A (HEAD)			
11	print('Your shoes are nice!')			
12	=======			
13	print('Your shirt is nice!')			
14	>>>>>>> BRANCH B			
15				
16	# give comment on the weather			
17	<<<<<<< BRANCH A (HEAD)			
18	print('It's warm outside!')			
19	=======			
20	print('It's cold outside!')			
21	>>>>>>> BRANCH B			
22				
23	# say goodbye			
24	<<<<<<< BRANCH A (HEAD)			
25	print('See you!')			
26	=======			
27	print('Goodbye!')			
28	>>>>>>> BRANCH B			
29	if __name__ == '__main__':			
30	main()			

	ACTUAL RESOLUTION FILE (ARF)	SECTION	INDEX
1	# this is the main function		
2	def main()		
3	# say hello		
4	print('Greetings!')		
5			
6	# give compliment		
7	# about the outfit		
8	print('Your shoes are nice!')		
9	print('Your shirt is nice!')		
10			
11	# give comment on the season		
12	print('It's a nice summer day!')		
13			
14	# say goodbye		
15	print('Farewell!')		
16	if __name__ == '__main__':		
17	# start the main function		
18	main()		

Figure 4.2: The URF and ARF we use to illustrate the analysis process. The URF file includes the conflicting chunks to be resolved, and the ARF contains the resolved conflicting chunks, i.e. the conflicting chunk resolutions.

4.3.1 Creation of the Section List of the URF

First, the URF is divided into sections. All conflicting chunks are located and the code lines in between the merging marker 1 and merging marker 2 (i.e., the code from the first branch) are denoted as sections of type CCP1. Analogously, the code lines in between merging marker 2 and merging marker 3 (i.e., the code from the second branch) are denoted as sections of type CCP2. All non-conflicting code segments surrounding these sections (i.e., the contexts) are marked as sections of type CTX. This results in a list of

sections that represents the structure of the unmerged resolution file. Each section is indexed with a section index, denoting its place within the file.

4.3.1.1 Empty Contexts and Empty Conflicting Chunk Parts

If the URF starts with a conflicting chunk, meaning that there is no first context, an empty section of type CTX is inserted at the start of the section list. Analogously, if the URF ends with a conflicting chunk, meaning that there is no last context, an empty section of type CTX is inserted at the end of the section list. Contexts in between conflicting chunks can not be empty, because if they were, their surrounding conflicting chunks would be aggregated into one conflicting chunk.

Similarly, if any of the code segments found in the conflicting chunks are empty, i.e., there are no lines between merging marker 1 and merging marker 2 or between merging marker 2 and merging marker 3), then an empty section of either type CCP1 or CCP2 is inserted at the corresponding place in the section list.

This guarantees that the section list has a standardised structure of alternating sections of type CTX and bundles of sections of type CCP1 and CCP2, starting and ending with a section of type CTX. This is necessary for the following steps of our algorithm to work.

	UNMERGED RESOLUTION FILE (URF)	SECTION	INDEX	MAPPED
1	def main()	CTX	1	
2	# say hello			
3	<<<<<<< BRANCH A (HEAD)			
4	print('Hello!')	CCP1	2	
5	=======			
6	print('Greetings!')	CCP2	3	
7	>>>>>>> BRANCH B			
8		CTX	4	
9	# give compliment			
10	<<<<<<< BRANCH A (HEAD)			
11	print('Your shoes are nice!')	CCP1	5	
12	=======			
13	print('Your shirt is nice!')	CCP2	6	
14	>>>>>>> BRANCH B			
15		CTX	7	
16	# give comment on the weather			
17	<<<<<<< BRANCH A (HEAD)			
18	print('It's warm outside!')	CCP1	8	
19	=======			
20	print('It's cold outside!')	CCP2	9	
21	>>>>>>> BRANCH B			
22		CTX	10	
23	# say goodbye			
24	<<<<<<< BRANCH A (HEAD)			
25	print('See you!')	CCP1	11	
26	=======			
27	print('Goodbye!')	CCP2	12	
28	>>>>>>> BRANCH B			
29	if __name__ == '__main__':	CTX	13	
30	main()			

	ACTUAL RESOLUTION FILE (ARF)	SECTION	INDEX
1	# this is the main function		
2	def main()		
3	# say hello		
4	print('Greetings!')		
5			
6	# give compliment		
7	# about the outfit		
8	print('Your shoes are nice!')		
9	print('Your shirt is nice!')		
10			
11	# give comment on the season		
12	print('It's a nice summer day!')		
13			
14	# say goodbye		
15	print('Farewell!')		
16	if __name__ == '__main__':		
17	# start the main function		
18	main()		

Figure 4.3: Sections of the URF. As can be seen in the SECTION column, every conflicting chunk (sections CCP1 and CCP2) is enclosed by contexts (sections CTX). Their section indices are shown in the INDEX column.

4.3.2 Section and Line Mapping

Next, a section and line mapping between the URF and ARF is performed. To this end, the ARF is traversed, and where possible, the sections of the URF are mapped to lines of the ARF. The mapping is based on Java's in-built String comparison, and the algorithm proceeds as detailed in Algorithm 4.1. This results in information about which (if any) of URF's sections were found in ARF, and which (if any) of ARF's lines could therefore be mapped.

```
ARF.sections = []
for section in URF.sections:
  if section is empty:
    mark section as mapped
  else:
    if section.lines exist within ARF:
      mark section as mapped
      add section to ARF.sections
      mark the corresponding lines in the ARF as belonging to section
```

Algorithm 4.1: Section and line mapping between URF and ARF

	UNMERGED RESOLUTION FILE (URF)	SECTION	INDEX	MAPPED
1	def main()	CTX	1	TRUE
2	# say hello			
3	<<<<<<< BRANCH A (HEAD)			
4	print('Hello!')	CCP1	2	
5	=======			
6	print('Greetings!')	CCP2	3	TRUE
7	>>>>>>> BRANCH B			
8		CTX	4	TRUE
9	# give compliment			
10	<<<<<<< BRANCH A (HEAD)			
11	print('Your shoes are nice!')	CCP1	5	TRUE
12	=======			
13	print('Your shirt is nice!')	CCP2	6	TRUE
14	>>>>>>> BRANCH B			
15		CTX	7	
16	# give comment on the weather			
17	<<<<<<< BRANCH A (HEAD)			
18	print('It's warm outside!')	CCP1	8	
19	=======			
20	print('It's cold outside!')	CCP2	9	
21	>>>>>>> BRANCH B			
22		CTX	10	TRUE
23	# say goodbye			
24	<<<<<<< BRANCH A (HEAD)			
25	print('See you!')	CCP1	11	
26	=======			
27	print('Goodbye!')	CCP2	12	
28	>>>>>>> BRANCH B			
29	if __name__ == '__main__':	CTX	13	
30	main()			

	ACTUAL RESOLUTION FILE (ARF)	SECTION	INDEX
1	# this is the main function		
2	def main()	CTX	1
3	# say hello		
4	print('Greetings!')	CCP2	3
5		CTX	4
6	# give compliment		
7	# about the outfit		
8	print('Your shoes are nice!')	CCP1	5
9	print('Your shirt is nice!')	CCP2	6
10			
11	# give comment on the season		
12	print('It's a nice summer day!')		
13		CTX	10
14	# say goodbye		
15	print('Farewell!')		
16	if __name__ == '__main__':		
17	# start the main function		
18	main()		

Figure 4.4: Result of the section and line mapping between the URF and the ARF. As shown in the SECTION and INDEX column of the ARF, six sections were found in total, namely Section(CTX, 1), Section(CCP2, 3), Section(CTX, 4), Section(CPP1, 5), Section(CPP2, 6) and Section(CTX, 10). They are shown at their respective place line-wise within that column. Their mapping status shown in the MAPPED column of the URF reflects their successful mapping.

4.3.2.1 Special Case: Empty ARF

If the ARF is empty, meaning that it contains no lines, then the section and line mapping and all further analysis steps are skipped. Rather, all empty sections of the URF are marked as mapped, whereas the non-empty ones are marked as unmapped. The results are then aggregated according to the procedure detailed in Section 4.3.5.

4.3.2.2 Checking the Section List Structure

After the section and line mapping is finished, we determine if the mapped sections were mapped in order by checking if the section indices of the sections in the ARF increase with each consecutive section. If this is not the case, then contexts or conflicting chunks were moved from their original position within the code to another position. Since we cannot say what implications this restructuring has for the conflicting chunk resolutions and contexts, as we do not perform any syntactical or semantic analysis on the code, we terminate the analysis and mark the conflicting file as *skipped*, stating the changed section list structure as the reason for the skip (see Section 4.4).

If however the sections are mapped in order, then we check if there are unmapped lines in the ARF. If no unmapped lines remain in the ARF, then the conflicting file resolution is derivable, as all contexts were mapped (and therefore remained unchanged) and all conflicting chunks were resolved canonically or semi-canonically (as either CPP1, CPP2, both or neither were mapped). In this case, our analysis is complete and we gather the results according to the procedure detailed in Section 4.3.5.

In our example, lines 1, 7, 11-12 and 15-18 remain unmapped (see Fig. 4.4). Whenever such unmapped lines remain, we continue with the analysis as detailed in the next Section.

4.3.3 Handling Unmapped Lines

In the case of unmapped lines within the ARF, either contexts were changed and/or conflicting chunks were resolved non-canonically. To determine what exactly occurs, all consecutive unmapped lines are gathered into sections with type NONE and section index -1. They are inserted within the section list of the ARF at the place that they appear in the code.

	UNMERGED RESOLUTION FILE (URF)	SECTION	INDEX	MAPPED
1	def main()	CTX	1	TRUE
2	# say hello			
3	<<<<<<< BRANCH A (HEAD)			
4	print('Hello!')	CCP1	2	
5	=======			
6	print('Greetings!')	CCP2	3	TRUE
7	>>>>>>> BRANCH B			
8		CTX	4	TRUE
9	# give compliment			
10	<<<<<<< BRANCH A (HEAD)			
11	print('Your shoes are nice!')	CCP1	5	TRUE
12	=======			
13	print('Your shirt is nice!')	CCP2	6	TRUE
14	>>>>>>> BRANCH B			
15		CTX	7	
16	# give comment on the weather			
17	<<<<<<< BRANCH A (HEAD)			
18	print('It's warm outside!')	CCP1	8	
19	=======			
20	print('It's cold outside!')	CCP2	9	
21	>>>>>>> BRANCH B			
22		CTX	10	TRUE
23	# say goodbye			
24	<<<<<<< BRANCH A (HEAD)			
25	print('See you!')	CCP1	11	
26	=======			
27	print('Goodbye!')	CCP2	12	
28	>>>>>>> BRANCH B			
29	if __name__ == '__main__':	CTX	13	
30	main()			

	ACTUAL RESOLUTION FILE (ARF)	SECTION	INDEX
1	# this is the main function	NONE	-1
2	def main()	CTX	1
3	# say hello		
4	print('Greetings!')	CCP2	3
5		CTX	4
6	# give compliment		
7	# about the outfit	NONE	-1
8	print('Your shoes are nice!')	CCP1	5
9	print('Your shirt is nice!')	CCP2	6
10		NONE	-1
11	# give comment on the season		
12	print('It's a nice summer day!')		
13		CTX	10
14	# say goodbye		
15	print('Farewell!')	NONE	-1
16	if __name__ == '__main__':		
17	# start the main function		
18	main()		

Figure 4.5: Result of the aggregation of unmapped lines within the ARF. As can be seen in the SECTION and INDEX columns of the ARF, the unmapped lines (line 1, 7, 11-12 and 15-18) were gathered into multiple Section(NONE, -1).

Next, we determine what implications the placing of these sections of unmapped lines have for the conflicting chunk resolutions and contexts. To this end, we analyse each of these sections of unmapped lines and their (eventual) surrounding mapped sections. There are four possible scenarios where such sections of unmapped lines may lie in the code, and we lay them all out in the following Sections. To simplify our explanations, we

will refer to the section list of the ARF as ARF.sections, and we will refer to the section list of the URF as URF.sections. Additionally, we will use the following notation:

- ARF.sections.length refers to the length of the ARF.sections list. In our example, ARF.sections.length is 10. Analogously, URF.sections.length refers to the length of the URF.sections list. In our example, URF.sections.length is 13.
- ARF.sections[i] refers to the section found in the ARF.sections list at index i. The index i is one-based, and we refer to it as the section list index. In our example, ARF.sections[3] is the section Section(CCP2, 3).
- ARF.sections[i].index refers to the section index of the section found in the ARF.sections list at index i. In our example, ARF.sections[3].index is 3.

4.3.3.1 Unmapped Lines at the Beginning of the File

ARF.sections[1].index = -1: If there are unmapped lines at the very beginning of the file, then the following two scenarios are possible:

1. ARF.sections[2].index = 1: In this case, the next section that could be mapped afterwards is the Section(CTX, 1), i.e., the first context. Therefore, additional lines were added beforehand, and since this marks a change to this first context, we mark it as unmapped.
2. ARF.sections[2].index = k, $1 < k \leq$ URF.sections.length: In this case, no sections up until the one with section index k could be mapped. We therefore mark all of the sections with section index $<$ k as unmapped.

In our example, additional lines are added before Section(CTX, 1) (see Figure 4.5). Therefore, according to the first case scenario above, we mark the it as unmapped (see Figure 4.6).

4.3.3.2 Unmapped Lines at the End of the File

ARF.sections[ARF.sections.length - 1].index = -1: Another possibility is that there are unmapped lines at the end of the file. Again, there are two possible scenarios:

1. ARF.sections[ARF.sections.size - 2].index = URF.sections.length - 1: In this case, the last sections that could be mapped before is Section(CTX, URL.sections.length - 1), i.e., the last context. Therefore, additional lines were added afterwards, and since this marks a change to this last context, we mark it as unmapped.

2. ARF.sections[ARF.sections.size - 2].index = k, $1 \leq k <$ URF.sections.length: In this case, no sections down until the one with section index k could be mapped. We therefore mark all of the sections with section index $>$ k as unmapped.

In our example, Section(CTX, 10) is the last mapped section before the one with unmapped lines (Section(NONE, -1)) (see Figure 4.5). Therefore, according to the second scenario above, we mark all sections with section index $>$ 10, namely the sections Section(CCP1, 11), Section(CCP2, 12) and Section(CTX, 13) as unmapped (see Figure 4.6).

4.3.3.3 Unmapped Lines in between Neighbouring Sections

If there are unmapped lines in between neighbouring sections (i.e., sections whose section indices are consecutive numbers), then additional lines were inserted in between two mapped sections. There are two possible scenarios:

1. One of the two mapped sections is of type CTX, whereas the other is either of type CCP1 or CCP2. In this case, by convention, we mark the section that CTX corresponds to as unmapped.

2. The two sections are of type CCP1 and CCP2. In this case, we mark them both as unmapped.

In our example, there is an unmapped line (line 7) in between the neighbouring sections Section(CTX, 4) and Section(CCP1, 5) (see Figure 4.5). As Section(CTX, 4) denotes a context, according to the first case above, we mark it as unmapped.

4.3.3.4 Unmapped Lines in between Non-Neighbouring Sections

If there are unmapped lines in between non-neighbouring sections (i.e., sections whose sections indices are not consecutive numbers), then all sections in between these non-neighbouring sections could not be mapped, indicating non-canonical conflicting chunk resolutions and/or changed contexts. In this case, we mark all sections with section indices in between the sections indices of these non-neighbouring sections as unmapped.

In our example, there are unmapped lines (lines 10-12) in between the non-neighbouring sections Section(CPP2, 6) and Section(CTX, 10) (see Figure 4.5). We therefore mark all sections in between these two (namely sections Section(CTX, 7), Section(CCP1, 8) and Section(CCP2, 9)) as unmapped (see Figure 4.6).

4.3.4 Handling Unmarked Sections

After all unmapped lines are analysed according to the four scenarios above, sections within the URF that do not have a mark already are marked as unmapped. This step finalises the mapping.

In our example, only Section(CCP1, 2), the second section in URF.sections, is left without a mapping (see Figure 4.5). It is subsequently marked unmapped (see Figure 4.6).

	UNMERGED RESOLUTION FILE (URF)	SECTION	INDEX	MAPPED
1	def main()	CTX	1	FALSE
2	# say hello			
3	<<<<<<< BRANCH A (HEAD)			
4	print('Hello!')	CCP1	2	FALSE
5	=======			
6	print('Greetings!')	CCP2	3	TRUE
7	>>>>>>> BRANCH B			
8		CTX	4	FALSE
9	# give compliment			
10	<<<<<<< BRANCH A (HEAD)			
11	print('Your shoes are nice!')	CCP1	5	TRUE
12	=======			
13	print('Your shirt is nice!')	CCP2	6	TRUE
14	>>>>>>> BRANCH B			
15		CTX	7	FALSE
16	# give comment on the weather			
17	<<<<<<< BRANCH A (HEAD)			
18	print('It's warm outside!')	CCP1	8	FALSE
19	=======			
20	print('It's cold outside!')	CCP2	9	FALSE
21	>>>>>>> BRANCH B			
22		CTX	10	TRUE
23	# say goodbye			
24	<<<<<<< BRANCH A (HEAD)			
25	print('See you!')	CCP1	11	FALSE
26	=======			
27	print('Goodbye!')	CCP2	12	FALSE
28	>>>>>>> BRANCH B			
29	if __name__ == '__main__':	CTX	13	FALSE
30	main()			

	ACTUAL RESOLUTION FILE (ARF)	SECTION	INDEX
1	# this is the main function	NONE	-1
2	def main()	CTX	1
3	# say hello		
4	print('Greetings!')	CCP2	3
5		CTX	4
6	# give compliment		
7	# about the outfit	NONE	-1
8	print('Your shoes are nice!')	CCP1	5
9	print('Your shirt is nice!')	CCP2	6
10		NONE	-1
11	# give comment on the season		
12	print('It's a nice summer day!')		
13		CTX	10
14	# say goodbye		
15	print('Farewell!')	NONE	-1
16	if __name__ == '__main__':		
17	# start the main function		
18	main()		

Figure 4.6: Final result of the mapping. As can bee seen when compared with Figure 4.5, Section(CTX, 1) is now marked as unmapped (according to Section 4.3.3.1). Further, Section(CCP1, 11), Section(CCP2, 12) and Section(CTX, 13) are now marked as unmapped (according to Section 4.3.3.2). Contrary to the intermediate result in Figure 4.5, Section(CTX, 4) is now also marked as unmapped (according to Section 4.3.3.3). Finally, Section(CTX, 7), Section(CCP1, 8) and Section(CCP2, 9) are marked unmapped (according to Section 4.3.3.4). And lastly, Section(CCP1, 2) is also marked as unmapped (according to 4.3.4).

4.3.5 Result

After the analysis is complete, all sections within the URF are marked either mapped or unmapped, and information about unmapped lines are found in the section list of the ARF. The final results are then gathered using the following logic:

For every conflicting chunk, if either its associated CCP1 section or its associated CCP2 section is mapped, then the resolution is *canonical*. If both sections are mapped, or neither of them, with no unmapped lines in between the enclosing contexts (which would indicate a *non-canonical* resolution), then the conflicting chunk resolution is *semi-canonical*. For

every context, if its associated CTX section is mapped, then it has remained *unchanged*. If it is unmapped however, it has *changed*.

	UNMERGED RESOLUTION FILE (URF)	SECTION	INDEX	MAPPED	RESULT
1	def main()	CTX	1	FALSE	CONTEXT (CHANGED)
2	# say hello				
3	<<<<<<< BRANCH A (HEAD)				CONFLICTING CHUNK RESOLUTION (CANONICAL)
4	print('Hello!')	CCP1	2	FALSE	
5	=======				
6	print('Greetings!')	CCP2	3	TRUE	
7	>>>>>>> BRANCH B				
8		CTX	4	FALSE	CONTEXT (CHANGED)
9	# give compliment				
10	<<<<<<< BRANCH A (HEAD)				CONFLICTING CHUNK RESOLUTION (SEMI-CANONICAL)
11	print('Your shoes are nice!')	CCP1	5	TRUE	
12	=======				
13	print('Your shirt is nice!')	CCP2	6	TRUE	
14	>>>>>>> BRANCH B				
15		CTX	7	FALSE	CONTEXT (CHANGED)
16	# give comment on the weather				
17	<<<<<<< BRANCH A (HEAD)				CONFLICTING CHUNK RESOLUTION (NON-CANONICAL)
18	print('It's warm outside!')	CCP1	8	FALSE	
19	=======				
20	print('It's cold outside!')	CCP2	9	FALSE	
21	>>>>>>> BRANCH B				
22		CTX	10	TRUE	CONTEXT (UNCHANGED)
23	# say goodbye				
24	<<<<<<< BRANCH A (HEAD)				CONFLICTING CHUNK RESOLUTION (NON-CANONICAL)
25	print('See you!')	CCP1	11	FALSE	
26	=======				
27	print('Goodbye!')	CCP2	12	FALSE	
28	>>>>>>> BRANCH B				
29	if __name__ == '__main__':	CTX	13	FALSE	CONTEXT (CHANGED)
30	main()				

Figure 4.7: Result aggregation of the file comparison between URF and ARF.

This results in information about how many canonical, semi-canonical and non-canonical conflicting chunk resolutions are contained in the conflicting file resolution, along with the number of unchanged and changed contexts. This in turn makes it possible to determine whether the conflicting file resolution is derivable - namely, if there are no changed contexts and only canonical and semi-canonical conflicting chunk resolutions.

4.4 Output

The results of the file comparison detailed in Section 4.3.5 (the number of canonical, semi-canonical and non-canonical chunk resolutions and the number of changed and unchanged contexts) are inserted into a schema within a local database. Every conflicting file represents one entry. In addition to these numbers, the file name is stored as meta-

data, along with the state of the analysis (whether the file comparison was successful, and if not, why it failed or why it was skipped).

To link the conflicting files back to the conflicting merges they belong to, these conflicting merges are also put into the database within their own schema. Their entries include metadata like the commit hash value, the total number of files included in the merge, and the state of the analysis as well as the reason for failure, should our algorithm not be able to recreate the conflicting merge from the project's commit history. This can happen if commits were deleted or the project's commit history is corrupted.

Analogously, the projects are stored in the database, including metadata like the project name, the amount of contributors and tags, as well as the amount of total commits, merges and octopus merges (i.e., merges with more than two parents). The state of the analysis for the project is also stored, denoting whether its analysis was successful or whether our algorithm failed to access the project's resources. Such failures can happen if the project's commit history is corrupted, and if this occurs, the exact reason for failure is stored in the database as well.

This results in a database structure as follows:

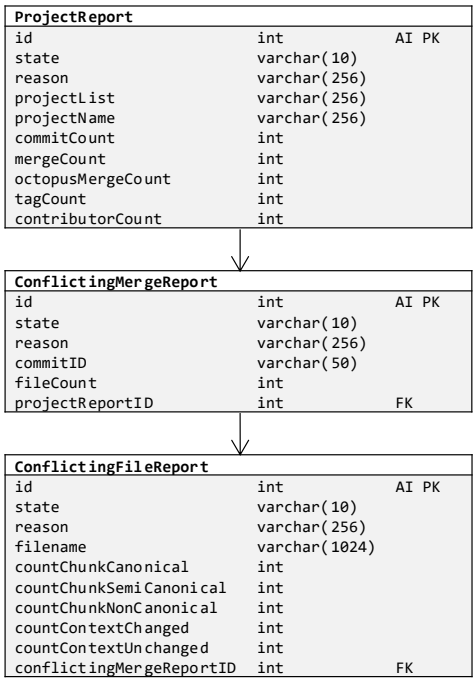

Figure 4.8: Database schema used to gather our results. For each database, the first column denotes the names of the rows found within the database, the second column denotes the types associated with said names, and the third column indicates whether the rows are auto-incrementing (AI), and whether they constitute a primary key (PK) or a foreign key (FK).

5 Results

This Chapter details the results of our analysis. We will explain them and discuss their findings briefly here. A more extended discussion will follow in Chapter 7. *GitAnalzerPlus-DataAnalysis*, the Python application we developed to gather our results from the databases (see Section 4.4), is accessible for faculty members on the GitLab server of the University of Bern.[13]

Note that within all Tables, columns marked with $c_{\text{subscript}}$ denote absolute numbers, whereas columns marked with $r_{\text{subscript}}$ denote relative percentages.

5.1 Overview of the Dimensions of our Study Sets

The following Tables show how many projects, conflicting merges, conflicting files, conflicting chunks and contexts we analysed in total. They give an overview of the dimension of our analysis.

[13] `https://gitlab.inf.unibe.ch/SEG/theses/git-analyzer-plus`
For non-faculty members: If you would like to access the code, contact us via email at yael.vandok@students.unibe.ch.

5.1.1 Projects

List	c_{total}	r_{ok}	r_{skip}	r_{fail}
C++	500	76.60	23.40	0.00
GO	498	79.12	20.88	0.00
Java	500	65.00	35.00	0.00
JavaScript	500	53.00	47.00	0.00
Python	499	67.94	32.06	0.00
TypeScript	498	79.32	20.68	0.00
Random (Low Rating)	2434	3.45	96.55	0.00
Random (High Rating)	2437	8.25	91.75	0.00
All Lists	7866	30.33	69.67	0.00

Table 5.1: Overview of the analysed projects. c_{total} shows the total amount of scheduled projects. r_{ok} shows the percentage of successfully analysed projects. r_{skip} shows the percentage of projects for which the analysis was skipped, and r_{fail} shows the percentage of projects for which the analysis failed.

Table 5.1 gives an overview of the projects we scheduled for analysis. As we detailed in Section 4.1, we picked about 500 projects for six major programming languages (C++, GO, Java, JavaScript, Python and TypeScript), and another approximate 2500 projects with randomly selected names with both low ratings (Random (Low Rating)) and high ratings (Random (High Rating)) on GitHub. The exact numbers are listed in the c_{total} column. These lists were taken from Buchser's study, as we mentioned in Section 4.1.

The r_{fail} column indicates projects for which the analysis failed, due to the project histories being corrupted. This case never occurred.

The r_{ok} column shows how many of these projects were analysed successfully, meaning that we found conflicting merges in their commit histories which we could subsequently analyse. The r_{skip} column indicates the percentage of skipped projects - namely projects with no such conflicting merges in their histories.

What becomes evident immediately is that there were vast differences in between lists in terms of the skipped projects. While for the specified programming languages, the rates of projects with conflicting merges range from about 50 % to about 80 %, for the projects with randomly selected names, the rates are below 10 %, meaning that a vast majority of those projects were found to not have any conflicting merges.

In the case of projects with low ratings, this number is not surprising, as such projects are most likely personal projects with only a few (if not just one) developers as well as few commits and merges. The chance for merge conflicts that come about with collaborative work is therefore lowered. For projects with high ratings, this low amount of projects with conflicting merges is less easily explainable, and a detailed analysis of these projects would be necessary to determine why so few merge conflicts occurred (see Chapter 8).

5.1.2 Conflicting Merges

List	c_{total}	r_{ok}	r_{skip}	r_{fail}
C++	65566	99.92	0.08	0.00
GO	12176	99.92	0.08	0.00
Java	15189	99.95	0.05	0.00
JavaScript	5936	99.90	0.10	0.00
Python	13959	100.00	0.00	0.00
TypeScript	25936	91.64	8.36	0.00
Random (Low Rating)	304	99.67	0.33	0.00
Random (High Rating)	1639	100.00	0.00	0.00
All Lists	140705	98.40	1.60	0.00

Table 5.2: Overview of the analysed conflicting merges. c_{total} shows the total amount of scheduled conflicting merges. r_{ok} shows the percentage of successfully analysed conflicting merges. r_{skip} shows the percentage of conflicting merges for which the analysis was skipped, and r_{fail} shows the percentage of conflicting merges for which the analysis failed.

Table 5.2 shows how many conflicting merges we scheduled for analysis. The total amount for each list is found in column c_{total}. As can be seen, the projects found in the C++ list contributed the majority of conflicting merges to our analysis, while the projects found in the Random (Low Rating) list (i.e., projects with randomly picked names and low ratings) contributed the least amount by far.

The r_{ok} column shows how many of these conflicting merges were analysed successfully, meaning that they did contain conflicting files, which we analysed further and whose results can be found in Table 5.3. The r_{skip} column indicates the percentage of skipped conflicting merges. These conflicting merges did not contain any conflicting files - a scenario which should not be possible, as a conflicting merge by definition must include at least one conflicting file.

We suspect that these cases happened due to a bug in JGit's implementation of the Recursive Merger which we used to perform the three-way merges.[14] The associated class offers a merge functionality, which performs the three-way merge and returns a boolean, indicating whether the merge could be issued without conflicts. If the returned value is false, meaning that conflicts did arise, it is possible to get the paths of the conflicting files by accessing the mergeResults list property of the merger. This list is supposed to hold all conflicting file paths, complete with information about which of their lines are in conflict. However, in the skipped cases, this list was empty, constituting an invalid scenario, and thereby indicating a bug in JGit.

In terms of percentages, these skipped conflicting merges make up only a tiny minority. However, there is one outlier. In projects of the TypeScript list, about 8 % of the conflicting merges were skipped. We do not know how many projects contributed to this

[14] https://download.eclipse.org/jgit/site/6.3.0.202209071007-r/apidocs/org/eclipse/jgit/merge/RecursiveMerger.html

high percentage (whether it was just one of them or many), and this would need further investigation (see Chapter 8).

We opted to skip these conflicting merges. The remaining ones could be analysed successfully, as is evident by the r_{fail} rate, which would indicate failures of our analysis due to issues during recovery of the conflicting merges from the commit histories of the projects.

5.1.3 Conflicting Files

List	c_{total}	r_{ok}	r_{skip}	r_{fail}
C++	291990	98.23	1.52	0.25
GO	52794	99.22	0.61	0.17
Java	96921	99.71	0.25	0.04
JavaScript	18076	98.36	1.57	0.07
Python	40768	98.50	1.33	0.17
TypeScript	136907	98.57	1.25	0.18
Random (Low Rating)	509	87.43	12.57	0.00
Random (High Rating)	6238	82.83	17.15	0.02
All Lists	644203	98.47	1.34	0.18

Table 5.3: Overview of the analysed conflicting files. c_{total} shows the total amount of scheduled conflicting files. r_{ok} shows the percentage of successfully analysed conflicting files. r_{skip} shows the percentage of conflicting files for which the analysis was skipped, and r_{fail} shows the percentage of conflicting files for which the analysis failed.

As shown in Table 5.3, we analysed a total of 644'203 conflicting files contained within the conflicting merges detailed in the Section 5.1.2. Again, the total numbers for each list are shown in the c_{total} column. Most conflicting files were found within projects from the C++ list, which also contributed most conflicting merges.

The r_{ok} column indicates the percentage of successfully analysed conflicting files. The analysis we conducted on these files is the one detailed in Sections 4.2 and 4.3, and their

results can be found in Tables 5.4 and 5.5. A small percentage of conflicting files (shown in column r_{skip}) were skipped, as they were non-line-based files (image files like .png and .jpg files, binaries like .dll files, and folders). As we explained in Section 4.3.2.2, if conflicting chunks or contexts are moved from their original position off to another position within the file during merging, we cannot meaningfully state anything regarding the derivability of the conflicting file resolution as a whole. We therefore would also skip these conflicting files. However, for all conflicting files we analysed, this case never occurred.

Our analysis failed for another small set of the conflicting files (indicated by column r_{fail}). This was always due to an internal exception thrown by JGit. In all of these cases, the exception stated that the file could not be recovered from the project's history. We suspect that this was either due to a bug in JGit, or due to incorrectly set up commit histories or deleted commits. We did not investigate the exact reasons further, and we opted to exclude these files from the further analysis aggregation as they account only for a small percentage of all conflicting files. In future work, the exact reasons for failure could be determined (see Chapter 8).

5.1.4 Conflicting Chunks

List	c_{total}	$r_{\text{canonical}}$	$r_{\text{semi-canonical}}$	$r_{\text{non-canonical}}$
C++	672779	71.46	9.75	18.79
GO	115482	51.29	12.65	36.06
Java	136712	77.96	6.11	15.92
JavaScript	119112	45.24	5.33	49.43
Python	71879	61.80	11.87	26.33
TypeScript	322912	68.17	9.29	22.54
Random (Low Rating)	1145	53.45	21.40	25.15
Random (High Rating)	11210	68.47	12.47	19.06
All Lists	1451231	67.07	9.31	23.62

Table 5.4: Overview of the analysed conflicting chunks. c_{total} shows the total amount of analysed conflicting chunks. $r_{\text{canonical}}$ shows the percentage of canonically resolved conflicting chunks. $r_{\text{semi-canonical}}$ shows the percentage of semi-canonically resolved conflicting chunks, and $r_{\text{non-canonical}}$ shows the percentage of non-canonically resolved conflicting chunks.

The resolution rates for conflicting chunks obtained by our analysis are shown in Table 5.4. Their total amounts are shown in column c_{total}, with the other columns showing the relative percentages of resolution. About two thirds of all conflicting chunks were resolved canonically, with the highest percentages found for projects within the Java list (about 78 %), and the lowest percentage found for projects within the JavaScript list (about 45 %).

5.1.5 Contexts

List	c_{total}	$r_{\text{unchanged}}$	r_{changed}
C++	959603	80.37	19.63
GO	167865	81.87	18.13
Java	233356	91.32	8.68
JavaScript	136892	71.17	28.83
Python	112034	81.51	18.49
TypeScript	457868	77.68	22.32
Random (Low Rating)	1590	53.65	46.35
Random (High Rating)	16377	77.85	22.15
All Lists	2085585	80.54	19.46

Table 5.5: Overview of the analysed contexts. c_{total} shows the total amount of analysed contexts. $r_{\text{unchanged}}$ shows the percentage of contexts that were left unchanged, and r_{changed} shows the percentage of contexts that were altered.

As was explained in Sections 4.2 and 4.3, our conflicting file analysis did not only determine the conflicting chunk resolutions rates, but also whether changes were made to contexts. The results of the latter part of the analysis are shown in Table 5.5, with the c_{total} column indicating the total numbers of found contexts, and the $r_{\text{unchanged}}$ and r_{changed} columns indicating the percentage of unchanged and changed contexts respectively.

Developers in general did not change the contexts. A great majority of contexts found within all lists were left as is, as is indicated by the high percentages in the $r_{\text{unchanged}}$ column.

One outlier to this high percentage of unchanged contexts however is the Random (Low Rating) list (i.e., projects with randomly picked names and low ratings). There, about half of the contexts were changed. As the total amount of contexts found within this list is the lowest by far, we do not think that this has significant implications for our

findings. However, the reason why so many context changes occurred within projects in this list could be investigated further (see Chapter 8).

5.2 Derivability

This Section details how many of the analysed conflicting files and conflicting merges were found to be derivable. As we detailed in Section 2.3.1, a conflicting file resolution is derivable if all its contexts remained unchanged and all its conflicting chunks were resolved canonically or semi-canonically. A conflicting merge resolution in turn is derivable if all the conflicting file resolutions it contains themselves are derivable.

5.2.1 Overview

Tables 5.6 and 5.7 show the absolute numbers of analysed conflicting file and conflicting merge resolutions and their respective derivability rates.

5.2.1.1 Conflicting Files

List	c_{total}	$r_{\text{derivable}}$	$r_{\text{non-derivable}}$
C++	286824	62.18	37.82
GO	52383	69.95	30.05
Java	96644	77.92	22.08
JavaScript	17780	60.42	39.58
Python	40155	57.49	42.51
TypeScript	134956	69.48	30.52
Random (Low Rating)	445	37.98	62.02
Random (High Rating)	5167	70.23	29.77
All Lists	634354	66.47	33.53

Table 5.6: Derivability rates of conflicting files. c_{total} shows the total amount of analysed conflicting files. $r_{\text{derivable}}$ shows the percentage of conflicting files that were resolved in a derivable manner, and $r_{\text{non-derivable}}$ shows the percentage of conflicting files that were resolved in a non-derivable manner.

The total amount of analysed conflicting files are shown in the c_{total} column. About two thirds of all these conflicting files received a derivable resolution, and one third subsequently received a non-derivable resolution, as is indicated by the $r_{\text{derivable}}$ and $r_{\text{non-derivable}}$ columns. The derivability rates are fairly consistent for the lists of programming languages, ranging from about 57 % to 70 %, with the only slight outlier being the Java list with about 78 %. This is to be expected, as this list showed very high rates for unchanged contexts (see Table 5.5) and canonical and semi-canonical conflicting chunk resolutions (see Table 5.4), which are the necessary conditions for a conflicting file resolution to be derivable.

The projects found within Random (High Rating), i.e., projects with randomly picked names and high ratings, have a similar rate. The only outlier are the conflicting files found within projects of the Random (Low Rating) list, as only about 38 % of these

conflicting files were resolved in a derivable manner. Overall, the impact of this list on our findings is small, as it only features few projects which contributed only a small number of conflicting files and conflicting merges.

5.2.1.2 Conflicting Merges

List	c_{total}	$r_{\text{derivable}}$	$r_{\text{non-derivable}}$
C++	65511	34.57	65.43
GO	12166	34.36	65.64
Java	15181	46.85	53.15
JavaScript	5930	35.36	64.64
Python	13959	32.07	67.93
TypeScript	23767	30.66	69.34
Random (Low Rating)	303	28.05	71.95
Random (High Rating)	1639	34.17	65.83
All Lists	138456	34.99	65.01

Table 5.7: Derivability rates of conflicting mserges. c_{total} shows the total amount of analysed conflicting merges. $r_{\text{derivable}}$ shows the percentage of conflicting merges that were resolved in a derivable manner, and $r_{\text{non-derivable}}$ shows the percentage of conflicting merges that were resolved in a non-derivable manner.

Of all the conflicting merges we analysed (shown in column c_{total}), about one third were resolved in a derivable way. As was the case for the conflicting files, the derivability rates (shown in columns $r_{\text{derivable}}$ and $r_{\text{non-derivable}}$) were fairly consistent within the lists of programming languages (with percentages ranging from about 30 % to 35 %). Only the Java list had a higher percentage of about 45 %, which is in line with this list having the highest rate of derivable conflicting file resolutions.

The Random (Low Rating) list shows the lowest derivability rate (about 28 %). As we stated before, since this list contains very few conflicting merges, their impact on our findings is small.

5.2.2 Categorised Derivability Rates

To gather further insight into the properties of derivable and non-derivable resolutions, we categorise them by their amount of conflicting chunks (in the case of conflicting files) and by both their amount of conflicting chunks and their amount of conflicting files (in the case of conflicting merges).

As we explained in Section 2.3.3, how many conflicting chunks are included within the conflicting merges and how they are distributed over the conflicting files are important aspects to consider when investigating the feasibility of a merge conflict resolution generator, as they determine the cost and complexity of the generation. Furthermore, whether the conflicting chunks within a conflicting merge are dependent on each other determines whether certain strategies could be applied to reduce the resolution set of the generation (see Section 2.3.3.1).

As we did not perform any syntactical or semantic analysis of the merge conflicts and their resolutions, we cannot say how often dependencies between conflicting chunks occurred, and of what nature these dependencies were. Subsequently, we do not know in how many cases strategies to lower the computational cost by reducing the resolution set of the generation could be applied. Therefore, in the following Sections, we will assume the worst case scenario - namely, that all conflicting chunks within all conflicting merges depend on each other, and that the reduction of the resolution set is not possible. This means that for every conflicting merge, we assume that the whole set of all derivable conflicting merge resolutions must be generated.

If we find that the conflicting merges and conflicting files have favourable characteristics like low conflicting chunk counts, then this means that even in such a worst case scenario, the generation of the merge conflict resolutions would be feasible at a low computational cost.

5.2.2.1 Conflicting Files

c_{CC}	c_{count}	r_{count}	$r_{\text{derivable}}$	$r_{\text{non-derivable}}$
1	515563	81.27	73.91	26.09
2	61424	9.68	38.79	61.21
3	21267	3.35	36.78	63.22
4	10247	1.62	30.76	69.24
5	5845	0.92	28.47	71.53
6	3825	0.60	23.53	76.47
7	2793	0.44	29.47	70.53
8	1793	0.28	20.97	79.03
9	1384	0.22	23.48	76.52
10	1024	0.16	16.70	83.30
11	831	0.13	16.85	83.15
12	611	0.10	20.95	79.05
13-6087	7747	1.22	16.47	83.53

Table 5.8: Derivability of conflicting files, categorised by their conflicting chunk count. c_{CC} denotes the conflicting chunk count of each category. c_{count} shows the amount of conflicting files within each category. r_{count} shows how much each category makes up percentage-wise out of all conflicting files. $r_{\text{derivable}}$ shows the percentage of conflicting files that were resolved in a derivable way within each category, and $r_{\text{non-derivable}}$ shows the percentage of conflicting files that were resolved in a non-derivable way within each category.

In Table 5.8, the c_{CC} column denotes the amount of conflicting chunks which we used to categorise the conflicting files by. The c_{count} column displays the total amount of conflicting files found within for each conflicting chunk count, and the r_{count} column shows how much these amounts make up percentage-wise out of all analysed conflicting files. The $r_{\text{derivable}}$ and $r_{\text{non-derivable}}$ columns display the percentages of derivable and non-derivable conflicting file resolutions respectively.

The vast majority of all conflicting files we analysed (namely about 81 %) did only contain one conflicting chunk, and about 74 % of these files were resolved in a derivable

way. There is a sharp decline in the amount of conflicting files found for higher conflicting chunk counts, and conflicting files with more than four conflicting chunks make up no more than one 1 % each. This holds true for all the remaining conflicting chunk counts, and for the sake of readability, we aggregated the conflicting files with more than 12 conflicting chunks together in one single row. As is shown in r_{count}, these conflicting files make up only 1.22 % of the overall amount.

With the higher conflicting chunk counts comes a lower derivability rate of the conflicting file resolutions, and by the fifth row, i.e., conflicting files with five conflicting chunks, more than 70 % of the included conflicting files were resolved in a non-derivable manner.

5.2.2.2 Conflicting Merges

c_{CF}	c_{count}	r_{count}	$r_{\text{derivable}}$	$r_{\text{non-derivable}}$
1	61696	44.56	41.66	58.34
2	25043	18.09	34.12	65.88
3	12292	8.88	28.93	71.07
4	7869	5.68	28.54	71.46
5	5317	3.84	26.29	73.71
6	3956	2.86	25.88	74.12
7	3001	2.17	24.96	75.04
8	2344	1.69	24.70	75.30
9	1788	1.29	24.27	75.73
10	1478	1.07	24.56	75.44
11	1304	0.94	25.15	74.85
12	1043	0.75	25.22	74.78
13	881	0.64	24.29	75.71
14	802	0.58	26.06	73.94
15	575	0.42	24.17	75.83
16	566	0.41	21.20	78.80
17	520	0.38	28.85	71.15
18	534	0.39	22.28	77.72
19	412	0.30	25.00	75.00
20	416	0.30	25.24	74.76
21-27615	6833	4.94	23.30	76.70

Table 5.9: Derivability rates of conflicting merges, categorised by their conflicting file count. c_{CF} denotes the conflicting file count of each category. c_{count} shows the amount of conflicting merges within each category. r_{count} shows how much each category makes up percentage-wise out of all conflicting merges. $r_{\text{derivable}}$ shows the percentage of conflicting merges that were resolved in a derivable way within each category, and $r_{\text{non-derivable}}$ shows the percentage of conflicting merges that were resolved in a non-derivable way within each category.

Table 5.9 shows the analysed conflicting merges categorised by their conflicting file count. Analogously to Table 5.8, the c_{CF} column denotes the conflicting file count used to categorise the conflicting merges, whereas the c_{count} column denotes the amount of conflicting merges found for each conflicting file count. The r_{count} column shows how much these conflicting merges makes up percentage-wise out of all analysed conflicting merges, and the $r_{\text{derivable}}$ and $r_{\text{non-derivable}}$ columns show how many of them were resolved in a derivable or non-derivable way respectively.

As can be seen, the amount of occurrences becomes smaller with increasing conflicting file counts. Whereas 45 % of all conflicting merges were found to have one conflicting file, less than 20 % were found to have two, and less than 10 % were found to have three. By the fifth row, i.e. conflicting merges with five conflicting files, the amount of conflicting merges makes up less than 5 % of all conflicting merges. This downwards trend continues, and conflicting merges with more than 20 conflicting files make up less than 1 % each. This holds true for all remaining conflicting file counts, and we therefore represent all of these conflicting merges with more than 20 conflicting files in a single row in the Table. In total, they make up less than 5 % of all conflicting merges.

With the increasing number of conflicting files comes a decreasing derivability rate. Whereas conflicting merges with one conflicting file have a derivability rate of about 40 %, it lowers down to 35 % for conflicting merges with two conflicting files, and by the fifth row, i.e. conflicting merges with five conflicting files, it stabilises at about 25 %, meaning that about a quarter of conflicting merges with more than five conflicting files were found to be derivable.

c_{CC}	$c_{CM,}$	r_{CM}	$r_{CM,\ d.}$	$r_{CM,\ non\text{-}d.}$	c_{CF}	r_{CF}	$r_{CF,\ d.}$	$r_{CF,\ non\text{-}d.}$
1	62981	45.49	48.75	51.25	62981	9.93	48.92	51.08
2	24433	17.65	33.59	66.41	39616	6.25	47.31	52.69
3	11918	8.61	24.80	75.20	26200	4.13	48.20	51.80
4	7568	5.47	23.71	76.29	20296	3.20	49.51	50.49
5	4990	3.60	19.74	80.26	16170	2.55	50.19	49.81
6	3519	2.54	16.11	83.89	12959	2.04	48.15	51.85
7	2636	1.90	15.17	84.83	11115	1.75	50.27	49.73
8	2105	1.52	15.01	84.99	9907	1.56	52.63	47.37
9	1739	1.26	16.79	83.21	8981	1.42	53.16	46.84
10	1360	0.98	13.46	86.54	7692	1.21	51.74	48.26
11	1131	0.82	12.11	87.89	6713	1.06	51.08	48.92
12	986	0.71	15.92	84.08	6302	0.99	54.27	45.73
13	786	0.57	11.45	88.55	5539	0.87	54.13	45.87
14	720	0.52	13.06	86.94	5600	0.88	56.70	43.30
15	596	0.43	12.75	87.25	4695	0.74	53.25	46.75
16	562	0.41	12.63	87.37	4757	0.75	53.44	46.56
17	409	0.30	13.20	86.80	3571	0.56	57.18	42.82
18	493	0.36	12.37	87.63	4663	0.74	59.17	40.83
19	414	0.30	14.25	85.75	4237	0.67	60.87	39.13
20	361	0.26	15.51	84.49	3811	0.60	60.14	39.86
21-27615	7819	5.65	15.17	84.83	368549	58.10	78.09	21.91

Table 5.10: Derivability rates of conflicting merges, categorised by their conflicting chunk count. c_{CC} denotes the conflicting chunk count of each category. c_{CM} shows the amount of conflicting merges within each category. r_{CM} shows how much each category makes up percentage-wise out of all conflicting merges. $r_{CM,\ d.}$ shows the percentage of conflicting merges that were resolved in a derivable way within each category, and $r_{CM,\ non\text{-}d.}$ shows the percentage of conflicting merges that were resolved in a non-derivable way within each category. Further, the amount of conflicting files found within the conflicting merges of each category are listed in c_{CF}. r_{CF} shows how much each category makes up percentage-wise out of all conflicting files. $r_{CF,\ d.}$ shows the percentage of conflicting files that were resolved in a derivable way within each category, and respectively, $r_{CF,\ non\text{-}d.}$ shows the percentage of conflicting files that were resolved in a non-derivable way.

Table 5.10 shows the analysed conflicting merges categorised by their conflicting chunk count. c_{CC} indicates the conflicting chunk count, and c_{CM} indicates how many conflicting merges were found for each of these counts. r_{CM} shows how much each category makes up percentage-wise out of all conflicting merges, and $r_{\mathrm{CM,\,d.}}$ and $r_{\mathrm{CM,\,non\text{-}d.}}$ show how many of the conflicting merges were resolved in a derivable and non-derivable manner respectively. To gather further insight, the conflicting files found within the conflicting merges and their derivability rates are shown in the remaining columns. c_{CF} shows the amount of conflicting files found within each category, and r_{CF} shows how much this amount makes up percentage-wise out of all conflicting files. Analogously to the conflicting merges, $r_{\mathrm{CF,\,d.}}$ and $r_{\mathrm{CF,\,non\text{-}d.}}$ show the percentages of derivable and non-derivable conflicting file resolutions.

About 45 % of all conflicting merges contained only one conflicting chunk within one conflicting file, and in about 49 %, the conflicting merge was resolved in a derivable manner. Conflicting merges with two conflicting chunks make up the second biggest category (with about 18 % of all conflicting merges), and the derivability rate for this category lies at about 34 %). Only about 8 % of all conflicting merges contained three conflicting chunks, and about 25 % of these conflicting merges were resolved in a derivable manner. The number of conflicting merges lowers as the conflicting chunk count, increases and by the tenth row, i.e., conflicting merges with ten conflicting chunks, said amounts makes up less than 1 % for each category. Conflicting merges with conflicting chunk counts from one to five make up about 80 % of all conflicting merges.

Interestingly, even though the conflicting merges with up to 20 conflicting chunks make up about 95 % of all analysed conflicting merges, the conflicting files found these conflicting merges only constitute about 40 % of all analysed conflicting files. This indicates that the conflicting chunks found within conflicting merges with a high amount of conflicting chunks are distributed over a large amount of conflicting files. This is in-line

with the finding in Section 5.2.2.1 that a vast majority of conflicting files only contains one conflicting chunk.

An interesting trend is the increase of the derivability rate for conflicting files as the conflicting chunk count rises. Whereas for conflicting merges with one conflicting chunk, about 49 % of the conflicting files found within were resolved in a derivable way, for conflicting merges with 20 conflicting chunks, about 60 % of all conflicting files found within were resolved in a derivable way. For conflicting merges with more than 20 conflicting chunks, the average derivability rate of conflicting files lies at about 78 %. Curiously, even though this derivability rate of the conflicting files increases, the derivability rate of the conflicting merges themselves decreases. This indicates that conflicting merges with a high amount of conflicting chunks contain a high amount of derivable conflicting file resolutions, and their low derivability rate as a whole is due to a low amount of non-derivable conflicting file resolutions.

5.3 Comparison with Buchser's Results

As we mentioned, our study is a direct continuation of the study conducted by Buchser. However, he limited his analysis to conflicting merges with at most 12 conflicting chunks. To compare our results with his, we categorised our results according to this limitation, and we display these results in Table 5.11 and Figures 5.1 and 5.2.

As we explained in Section 3.2, our approach differs from his in multiple aspects. First, we also analysed the contexts found within conflicting files and determined whether they were changed. Second, we also considered semi-canonical resolutions of conflicting chunks. Third, we eliminated the possibility of falsely mapped conflicting chunks by restricting their mapping to the place within the file were they should appear. (However, as we stated in Section 5.1.3, such a case never occurred.)

We further introduced the concept of derivability, which is more strict than the concept of canonical file and merge resolutions Buchser used in his work. In his study, conflicting

merge resolutions and conflicting file resolutions are called *canonical* if all conflicting chunks found within were resolved canonically respectively. In our study, a conflicting file is derivable if all its conflicting chunks were resolved in a canonical or semi-canonical way and no contexts were changed. Subsequently, a conflicting merge is derivable if all its conflicting files were resolved in a derivable manner.

Because of these differences, we expect our derivability rates to be lower than the rates of canonical conflicting file and merge resolutions that Buchser found in his study. [2, p. 23]

5.3.1 Categorised Resolution Rates

c_{CC}	$c_{O.,\ CM}$	$c_{B.,\ CM}$	$r_{O.,\ CM,\ d.}$	$r_{B.,\ CM,\ c.}$	$c_{O.,\ CF}$	$c_{B.,\ CF}$	$r_{O.,\ CF,\ d.}$	$r_{B.,\ CF,\ c.}$
1	62981	60515	48.75	40.55	184215	175243	55.21	50.31
2	24433	23319	33.59	33.29	26876	25598	28.24	32.30
3	11918	11376	24.80	27.44	8949	8393	24.61	27.98
4	7568	7113	23.71	25.76	3938	3770	20.11	26.05
5	4990	4732	19.74	23.44	1985	1927	14.66	21.64
6	3519	3356	16.11	20.11	1142	1082	14.36	21.16
7	2636	2498	15.17	19.82	662	615	13.75	19.19
8	2105	1990	15.01	19.50	451	432	13.53	20.83
9	1739	1646	16.79	20.72	288	265	14.58	23.40
10	1360	1277	13.46	19.11	186	169	15.05	24.26
11	1131	1057	12.11	15.23	164	151	9.76	17.88
12	986	915	15.92	21.42	76	67	19.74	32.84

Table 5.11: Comparison of our results with Buchser's. c_{CC} denotes the conflicting chunk count we used to categorise the conflicting merges and conflicting files by. Every subsequent pair of columns separated by vertical lines contains our results in the first column (indicated by the *O.* in the subscript), and Buchser's results in the second column (indicated by the *B.* in the subscript). $c_{O.,\ CM}$ and $c_{B.,\ CM}$ show how many conflicting merges we and Buchser analysed respectively. $r_{O.,\ CM,\ d.}$ shows how many of the conflicting merges in $c_{O.,\ CM}$ we found to be derivable. $r_{B.,\ CM,\ c.}$ shows how many of the conflicting merges in $c_{B.,\ CM}$ Buchser found to be canonical. Analogously, $c_{O.,\ CF}$ and $c_{B.,\ CF}$ show how many conflicting files we and Buchser analysed respectively. $r_{O.,\ CF,\ d.}$ shows how many of the conflicting files in $c_{O.,\ CF}$ we found to be derivable. $r_{B.,\ CF,\ c.}$ shows how many of the conflicting files in $c_{B.,\ CF}$ Buchser found to be canonical.

Table 5.11 shows the comparison of our results with Buchser's. Columns $c_{O.,\ CM}$ and $c_{B.,\ CM}$ show how many conflicting merges we and Buchser analysed respectively. As can be seen, our numbers are higher. This is to be expected. Whereas we both analysed the same projects, we conducted our study about six months after Buchser conducted his, and in the meantime, these projects received further development, naturally leading to

more conflicting merges and conflicting files. Since the numbers do not differ greatly, we view this as an indication for the correctness of our results.

$r_{\text{O., CM, d.}}$ shows the derivability rates of the conflicting merges we analysed, and $r_{\text{B., CM, d.}}$ shows the canonical rates Buchser found for the conflicting merges he analysed. As we expected, the derivability rates are lower than the canonical rates in most cases. Curiously, for conflicting merges with one and two conflicting chunks however, the derivability rates we found are higher than the canonical rates, which is counter-intuitive, as our concept of derivability is more strict than Buchser's concept of canonical resolutions (see Section 5.3). We will go into possible explanations for this later in the next Section 5.3.2.

$c_{\text{O., CF}}$ and $c_{\text{B., CF}}$ show how many conflicting files we and Buchser analysed respectively. Again, our numbers are higher, but not greatly so. $r_{\text{O., CF, d.}}$ shows how many of the conflicting files we analysed were derivable, $r_{\text{B., CF, c.}}$ shows how many of the conflicting files Buchser analysed were canonical. The derivability rates are lower in most cases, as we would expect, except for conflicting files with one conflicting chunk. These conflicting files showed higher derivability rates, which is counter-intuitive, as it was for the conflicting merges, due to our more strict concept of derivability. We will go into possible explanations for this in Section 5.3.2 below.

5.3.2 Findings

Our results are in line with the ones Buchser obtained in his study. The numbers of conflicting merges and conflicting files that we analysed are higher than the ones Buchser's analysed, but not greatly so. Further, our derivability rates for both conflicting merges and conflicting files are generally lower, while indicating the same trends that Buchser found in his study (see Figures 5.1 and 5.2). They also do not differ greatly. This strengthens Buchser's results.

The only oddity is that for conflicting merges with one and two conflicting chunks, as well as for conflicting files with one conflicting chunk, our derivability rate was higher than Buchser's canonical rate. We have two possible explanations for this.

One possibility is that the derivability rate is higher simply due to our bigger study sets. We analysed 62'981 conflicting merges with one conflicting chunk, whereas Buchser analysed 60'515. Further, we analysed 24'433 conflicting merges with two conflicting chunks, whereas Buchser analysed 23'319. And lastly, we analysed 184'215 conflicting files with one conflicting chunk, whereas he analysed 175'243. It is possible that these additional study subjects led to an increased derivability rate.

Another possibility is that our consideration of semi-canonical conflicting chunk resolutions lead to this higher derivability rate. As we detailed in Section 2.2, there are three possible semi-canonical conflicting chunk resolutions. Two of them constitute the concatenation of the code segments from both branches. Such semi-canonical resolutions would be considered canonical using Buchser's approach, so they cannot be responsible for the higher derivability rates. However, the third resolution constitutes the removal of all conflicting code. Such a resolution would be considered non-canonical using Buchser's approach. Therefore, if the conflicting chunks within a conflicting merge are resolved using this last semi-canonical resolution (and all of the contexts within the conflicting files remain unchanged), then we consider the conflicting merge resolution and the conflicting file resolutions found within *derivable*, whereas Buchser would consider them *non-canonical*. Such cases could therefore lead to an increased derivability rate over the canonical rate.

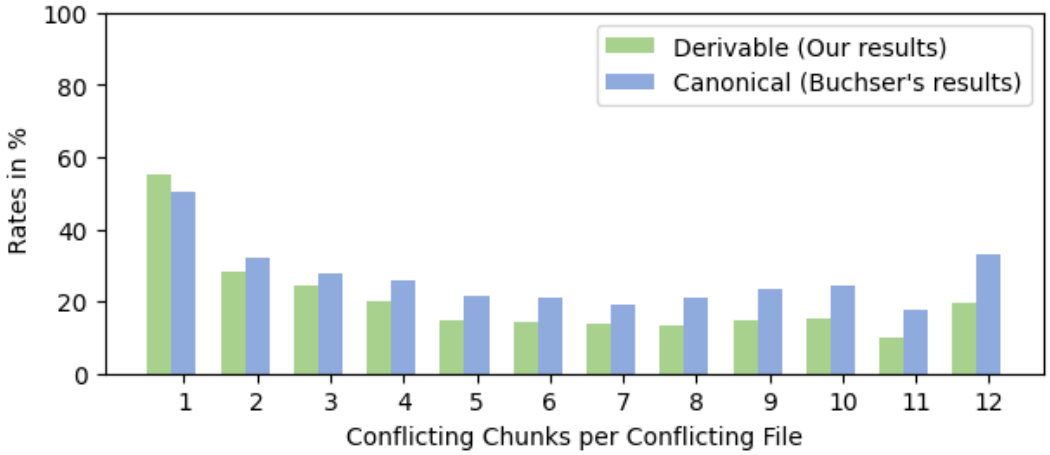

Figure 5.1: Comparison of the derivability rate (our result) and the of the canonical rate (Buchser's result) for conflicting files with at most 12 conflicting chunks. A conflicting resolution file is considered *canonical* if all conflicting chunks found within were resolved canonically. A conflicting file resolution is considered *derivable* if all conflicting chunks found within were resolved canonically or semi-canonically and no contexts were changed.

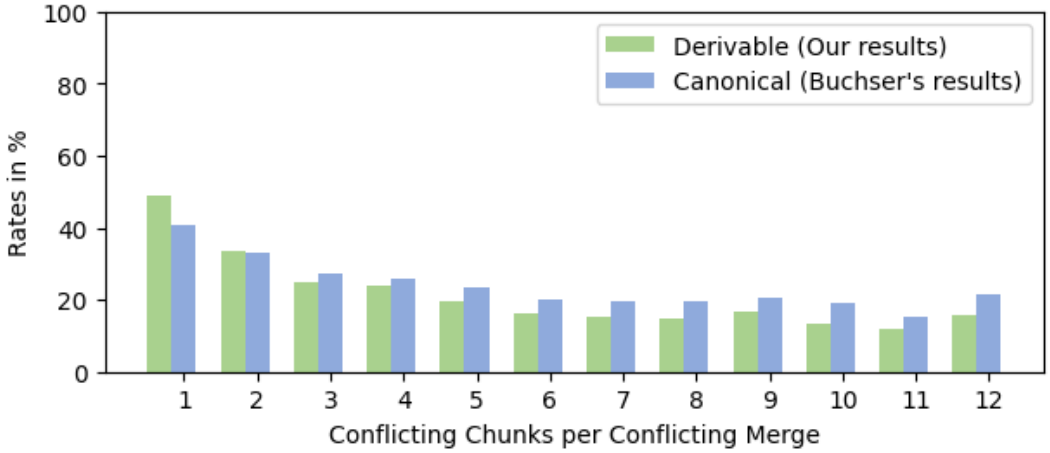

Figure 5.2: Comparison of the derivability rate (our result) and the of the canonical rate (Buchser's result) for conflicting merges with at most 12 conflicting chunks. A conflicting merge is considered *canonical* if all conflicting chunks found within its conflicting files were resolved canonically. A conflicting merge resolution is considered *derivable* if all conflicting file resolutions it contains are derivable.

6 Threats to Validity

Before we discuss the results obtained by our analysis further, we detail the limitations of our approach and discuss the possible threats to validity.

6.1 Internal Validity

Our analysis has limitations that might threaten the validity of our results.

6.1.1 Limitations of our File Comparison

The file comparison we perform (see Section 4.3) can lead to false negatives in three ways, which will in turn lower the derivability rates of the analysed conflicting files.

First, the section and line mapping used to determine how a conflicting chunk conflict was resolved and whether a context was changed operates strictly line-based, using the simple Java String comparison. This means that whenever there is a difference in at least one character between a line within the unmerged resolution file and the actual resolution file, the mapping will not succeed. This leads to lines and sections incorrectly being marked as unmapped in the case of white-space or formatting differences. This is undesirable, as our goal in this study is to determine whether developers resolve merge conflicts in a derivable way in terms of behaviour, and not whether the code they commit has equal formatting to the one found in the parent files. This issue could be mitigated by using an auto-formatter to format both files beforehand, unifying their code structure.

Second, our mapping ignores syntactical and semantic equivalence. Our algorithm only maps lines and sections if they are equal, but not if they are equivalent in either

syntax or semantics, which leads to sections not being mapped even when the generative approach whose feasibility we study would be able to generate an equivalent result. As an example, if a developer resolves a conflicting chunk by picking the code from one branch (i.e., they choose a canonical resolution), but they change the name of one variable found within this code, our algorithms will deem the resolution non-canonical, even though it is semantically equivalent to the canonical resolution. This is undesirable, as we want determine how developers resolve merge conflicts in terms of behaviour, not in terms of (just) picking the right code. This issue could be mitigated by testing for syntactical or semantic equivalence at least in easy cases such as variable renaming. Testing for semantic equivalence might not be viable, as e.g., the Halting Problem is not computable.

Both of these limitations can lead to sections being falsely marked as unmapped, which lowers both the canonical and semi-canonical conflicting chunk resolution rates and the rate of unchanged contexts. This in turn lowers the derivability rates for the conflicting files.

Another limitation of our mapping can lead to false intermediate results. Since we do not perform any syntactical and semantic analysis of the files we analyse, we cannot say what happens within unmapped lines of the actual resolution files. This especially has implications if lines were added in between mapped sections. In the case of additional lines in between mapped contexts and mapped conflicting chunk parts, as an example, we can not determine what the developer added, and whether we should consider the context to be changed or the conflicting chunk resolution to be non-canonical. By convention, in these cases, we decided to mark the context as changed, rather than the conflicting chunk to be solved non-canonically (see Section 4.3.3.4). This decision has no effect on the derivability of the conflicting file (as both changed contexts and non-canonical conflicting chunk resolutions render the conflicting file resolution non-derivable), but it effects the resolution rates of the conflicting chunks and the rates of changed contexts.

In conclusion, these limitations can lead to sections being falsely marked as unmapped. While this is undesirable in terms of accuracy of our analysis, it means that the derivability rates we found in Sections 5.2 would only be higher. All of these limitations could furthermore be addressed in future work (see Chapter 8).

6.1.2 Issues with JGit

There were multiple instances of our implementation returning invalid (intermediate) results, namely during the recreation of conflicting merges and during the analysis of conflicting files (see Tables 5.2 and 5.3). As we detailed in the accompanying text of these Tables, we suspect that this was due to bugs within JGit or incorrectly set up projects and commit histories. Only a low number of conflicting merges and files was affected by this however, and since we decided to just skip them and not account for them in the further result aggregation, we do not think that they have a significant detrimental effect on the validity of our findings.

6.2 External Validity

As we stated in Section 4.1, we analysed the same projects that Buchser analysed in his work. The wide range of programming languages that he chose guarantees that the results do not merely depict merging practices particular to the development culture of a certain language, but rather, it offers a broad view on different types of development practices. As we could not find significantly different derivability rates between these different languages (see Section 5.2), we view this as an indicator that our findings can be generalised to projects of other languages.

However, it is important to consider that while the projects we analysed were classified on GitHub as belonging to a certain programming language, this does not mean that their

repository must only contain source code files of said language. GitHub uses Linguist[15] to determine what languages are used within a project, and it will use the language that occurs most frequently percentage-wise to label the project as to belonging to that language.[16] So, while a project that is labelled as JavaScript will contain files that are written in JavaScript code, it may also contain files written in other languages like HTML and CSS. Therefore, we cannot say how many of the files we analysed were actually written in the language the project was labelled as, and therefore, whether the findings can be generalised for source code files of this language specifically.

Furthermore, projects very often contain non-source code files like project management tool files (e.g., Maven[17] files used in Java development), configuration files and READMEs in their repositories. As our analysis made no distinction between source code files and non-source code files, we cannot state with certainty how developers tend to resolve merge conflicts that occur in source code files as opposed to such auxiliary files.

While this means that our findings cannot be generalised to the actual source code files of a specific language, it does not threaten the validity of our results. Our goal in this study is to determine how developers resolve merge conflicts on a project-wide basis, not on a source code file basis, and as our approach analyses all conflicting files no matter their type, our results paint a complete picture of the merge conflict resolution behaviour of the developers when managing projects as a whole. If one wanted to investigate the developers' merging behaviour in more detail on a specific file basis, one could adapt our tool easily to account for a more detailed analysis of different file types (see Chapter 8).

[15] https://github.com/github-linguist/linguist
[16] https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-repository-languages
[17] https://maven.apache.org/

7 Discussion

In this Chapter, we discuss our results and the implications they have for our research questions.

7.1 RQ1: How often do developers pick derivable merge conflict resolutions, on a conflicting chunk, conflicting file and conflicting merge level?

We found that in about 75 % of all cases, developers picked derivable conflicting chunk resolutions. They furthermore mostly left the contexts unchanged (in about 80 % of all cases). About 66 % of all conflicting files resolutions were found to be derivable, and about 35 % of all conflicting merges were resolved in a derivable manner.

There were vast differences between the lists in terms of found conflicting merges and conflicting files. Most conflicting merges and conflicting files were found within projects from the C++ list (about 45 % in both cases). The list of projects written in TypeScript made up the second biggest amount (about 20 % in both cases). The projects written in Python, Java and GO all contributed about 10 % of conflicting merges and between 10 % and 15 % of conflicting files. Projects written in JavaScript only contributed about 5 % of conflicting merges and about 3 % of conflicting files. By far the smallest amount of conflicting merges and files came from both lists of projects with randomly selected names (Random (Low Rating) and (Random(High Rating)). The highly rated ones made up about 1 % of conflicting merges and files, and the lowly rated ones made up about

0.2 % of conflicting merges, and less than 0.1 % of conflicting files. Due to the very small amounts of conflicting files and conflicting merges found in these last two lists, we do not think that their impact on our finding is substantial, and subsequently, we focus on the lists of specified programming languages.

The derivability rates for the different lists of specified programming languages do not differ much on conflicting file and conflicting merge level. For conflicting files, they range from about 57 % to about 78 %, and for the conflicting merges, they range from about 30 % to about 47 %.

Interestingly, the resolutions rates of the conflicting chunks and the rate of changed contexts were not a clear indicator for the derivability of the conflicting files and conflicting merges. Projects written in Java had very high rates of derivable conflicting chunk resolutions and rates of unchanged contexts, and, as one would expect, they had high derivability rates for both conflicting files and conflicting merges. However, projects written in JavaScript had low rates of derivable conflicting chunk resolutions and lower rates of unchanged contexts, but the derivability rates for conflicting files and conflicting merges were not significantly lower than the ones for lists like C++, which had both a higher derivability rate on a conflicting chunk level as well as a higher rate of unchanged contexts. Furthermore, the derivability rates of the JavaScript projects were higher on a conflicting file level and conflicting merge level than the ones for projects written in Python, whose derivability rate for conflicting chunks and rate for unchanged contexts were higher in both cases.

7.1.1 Findings

We gather from these results that a merge conflict resolution generator is feasible for projects of different programming languages. In a substantial amount of all merge conflict scenarios no matter the language, developers resolved the merge conflicts in a derivable way which could be automated by such a tool.

However, further investigation into the nature of the merge conflicts found in the projects of different languages is necessary. As we already detailed in Chapter 6, we did not solely analyse merge conflicts found in source code files of the specified language, but rather, we analysed all merge conflicts found in all files within the projects' repositories. It would be of interest to see in what files (source code files or non-source code files) the merge conflicts tend to occur, and whether the developers' merging behaviour changes when handling such different file types. Furthermore, it would be of interest to determine whether source code files of different programming languages show different rates of derivability. This would shed further light on what merging scenarios of which file types and languages would be most suited to our generative approach.

7.2 RQ2: What are the specific properties of such derivable resolutions?

About 80 % of all conflicting merges were found to have between one and five conflicting chunks, and their derivability rates lie between 49 % and 20 %. Conflicting merges with one conflicting chunk constituted about 45 % of all analysed conflicting merges, and they were resolved in a derivable way in about 49 % of cases. Conflicting merges with more than five conflicting chunks generally showed lower derivability rates as a whole, averaging at about 15 %.

Interestingly, the derivability rates of the conflicting file resolutions found within these merges increased with higher conflicting chunk counts. This indicates that a majority of conflicting files found in such conflicting merges with high conflicting chunk counts were resolved in a derivable way, and the non-derivability of the conflicting merges as a whole came about due to a low amount of non-derivable conflicting file resolutions.

We found that the vast majority of all conflicting files (about 81 %) contained only one conflicting chunk. In about 74 % of these cases, the conflicting files were resolved in a derivable manner. Conflicting files with more than three conflicting chunks made

up only a small percentage (about 6 %), and they showed lower derivability rates from about 30 % for conflicting files with four conflicting chunks down to 16 % for conflicting files with more than 12 conflicting chunk counts.

Further, the majority of conflicting merges contained only a low number of conflicting files, with about 80 % having at most five conflicting files, and with about 45 % only having one conflicting file. Their derivability rates ranged from 42 % for conflicting merges with one conflicting file to about 26 % for conflicting merges with five conflicting files. Conflicting merges with more than five conflicting files generally showed lower derivability rates, with an average of about 25 %.

7.2.1 Findings

We gather from these results that in the vast majority of cases, a merge conflict resolution generator is feasible, as the computational cost to generate the derivable resolutions is low due to the low number of conflicting chunks within the conflicting merges. Even when assuming the worst case scenario where all conflicting chunks within the conflicting merge depend on each other, and no strategies to limit the resolution set can be applied (see Section 2.3.3.1), the generation of the whole set of derivable conflicting merge resolutions could be carried out at a low computational cost. Furthermore, in a considerable amount of these merge conflict scenarios, the developers resolved the conflicts in a derivable way which could be automated by such a generator. This further strengthens its feasibility.

For conflicting merges with more than five conflicting chunks, our approach might be less feasible, as the generational cost increases exponentially in the worst case, and because these conflicting merges generally showed low derivability rates. However, it is important to consider that conflicting merges with high conflicting chunk counts generally showed high percentages of derivable conflicting file resolutions (see Table 5.10). This, combined with the findings that about 81 % of all conflicting files only contained one conflicting chunk, and that in 74 % of these cases, they were resolved in a deriv-

able way, indicates that in many such cases, the conflicting merges with high conflicting chunk counts contained many conflicting files with only one conflicting chunk, and that these conflicting files were often resolved in a derivable way. If the conflicting chunks within these conflicting files do not depend on each other (which could be likely due to their placement in different conflicting files), then one could apply the strategies we discussed in Section 2.3.3.1 to limit the resolution set of the generation, and our generative approach might become viable even for such conflicting merges with many conflicting chunks. However, to determine if and how often these strategies could be applied, a detailed investigation into the dependencies between the conflicting chunks would need to be conducted (see Chapter 8).

8 Conclusion and Future Work

We found that in many occurrences of merge conflicts, a generative approach to their resolution is feasible. Most often, the number of conflicting chunks within the conflicting merges is low, which would allow for a complete generation of all possible derivable resolutions at a low computational cost. Furthermore, in many of these cases, the developers resolved the merge conflicts in a derivable way which such a tool could automate.

However, more research into the specific properties of merge conflicts and their resolutions by the developers is needed.

To this end, our analysis could be extended. As we detailed in Chapter 6, our approach has certain limitations which could be mitigated to allow for better insight into the merging behaviour of developers. The use of a formatter during the section mapping (see Section 4.3) is one such possible step. By eliminating white-space and formatting differences between the unmerged resolution file and the actual resolution file, the accuracy of the mapping could be increased, allowing for a more accurate determination of the derivability of the merge conflict resolutions, and subsequently, for better insight into the merging behaviour of the developers.

Further, as our analysis was purely line-based, one could adapt our tool to conduct a syntactical and semantic analysis of the merge conflicts and their subsequent resolutions. By taking these aspects into account, one could determine how often so-called semi-structured merge conflicts [1] occur, and whether developers resolve them in a derivable manner or not.

Since we did not analyse the impact of the specific file types on the merge conflict resolutions, one could account for this in a subsequent study by adapting our tool to conduct a more detailed analysis that investigates this impact. This would give further insight into the merging behaviour of developers in different scenarios and whether they behave differently when resolving merge conflicts in source code files as opposed to non-source code files. Furthermore, such an analysis would allow for better insight into which files within the projects tend to conflict most often.

As our study provided purely quantitative results of the derivability of merge conflicts, a qualitative study on what exactly happens in non-derivable merge conflict resolutions would shed light on how developers resolve merge conflicts when they do not just pick the code form one or both branches. It would especially be of interest to see whether these non-derivable merge conflict resolutions occurred due to white-space and formatting differences or simple syntactical changes (like renamed variables) which could be addressed by a sophisticated merge conflict resolution generator.

Moving one step further, as the goal of our work was to determine the feasibility of a merge resolution generator by providing quantitative results, one could investigate other feasibility aspects. To this end, conducting a user study would be of interest to determine if there is general interest in such a tool by developers, and if so, what they envision such a tool to look like and what features they would expect it to entail.

To conclude, much more research into merge conflicts and their resolutions is needed to further investigate the feasibility of a generative approach to merge conflict resolutions. However, our results so far are promising, and we are confident that further research is warranted.

List of Figures

List of Tables

Bibliography

[1] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. “Understanding semi-structured merge conflict characteristics in open-source java projects”. In: *Empirical Software Engineering* 23 (2018), pp. 2051–2085.

[2] Severin Buchser. “An empirical study on the human role in merge conflict resolution”. Bachelor’s Thesis. University of Bern, 2022.

[3] Scott Chacon and Ben Straub. *Pro Git*. Springer Nature, 2014.

[4] Gleiph Ghiotto et al. “On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github”. In: *IEEE Transactions on Software Engineering* 46.8 (2018), pp. 892–915.

[5] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O’Reilly Media, Inc.", 2012.

[6] Nicholas Nelson et al. “The life-cycle of merge conflicts: processes, barriers, and strategies”. In: *Empirical Software Engineering* 24 (2019), pp. 2863–2906.

[7] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. “Version control system: A review”. In: *Procedia Computer Science* 135 (2018), pp. 408–415.