Towards Semi-Automated Merge Conflict Resolution: Is It Easier Than We Expected?

Alexander Boll*
University of Bern
Switzerland

Yael van Dok*
University of Bern
Switzerland

Manuel Ohrndorf*
University of Bern
Switzerland

Alexander Schultheiß
Paderborn University
Germany

Timo Kehrer
University of Bern
Switzerland

ABSTRACT

In version control systems such as Git, concurrent modifications on the same artifacts can cause merge conflicts that may disrupt the development workflow by requiring manual intervention. While research on software merging focused on sophisticated techniques that hardly had any impact in practice, we present an empirical feasibility study on semi-automated conflict resolution using a fixed set of only a few language-agnostic conflict resolution patterns. In a large-scale quantitative analysis, we simulate the performance of our hypothetical conflict resolution strategy by classifying 131,154 merge conflict resolutions of a diverse sample of 10,000 GitHub projects according to these resolution patterns. We shed light on the derivability of merges on multiple levels of granularity: the conflicting merge commit, its conflicting files and their individual conflicting chunks. 87.9% of chunks are derivable individually, while 34.5% of merges are derivable as a whole. Interestingly, however, by inspecting potential factors affecting derivability, we observe that there are stronger correlations considering individual files than considering the entire merge. A short yet preliminary answer to whether semi-automated conflict resolution is easier than we expected is: yes, it might be, particularly if we use the right level of granularity for proposing conflict resolutions. Through our comprehensive analysis, we aspire to bridge the gap between academic innovations on sophisticated merge techniques and real-world merge conflict scenarios, laying the groundwork for more effective and widely accepted automatic merge tools.

CCS CONCEPTS

• **Software and its engineering → Collaboration in software development**; • **General and reference → Empirical studies**.

KEYWORDS

merging, conflict resolution, derivability, merge generator, merge recommendation, git mining, empirical study

EASE 2024, 18–21 June, 2024, Salerno, Italy
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Alexander Boll, Yael van Dok, Manuel Ohrndorf, Alexander Schultheiß, and Timo Kehrer. 2023. Towards Semi-Automated Merge Conflict Resolution: Is It Easier Than We Expected?. In *Proceedings of The 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Version Control Systems (VCS) are an integral tool for software development in companies or open-source communities. Following the paradigm of optimistic versioning, several working copies of the same development artifact, e.g., a source code file, may be edited independently without the need for being locked and unlocked. *Software merging* is the most fundamental operation supporting such optimistic versioning by reconciling concurrent changes to multiple working copies of a shared development artifact [23].

It has been reported that 10-20% of merge attempts encounter a failure, and failure rates can approach even 50% in some cases [8, 18, 34]. Thus, merge conflicts bother developers in their daily work and hamper continuous integration [16]. Virtually everybody who has worked with Git, or any other mainstream VCS, has encountered conflict markers in a *tentative merge* that results from a failed merge attempt, as illustrated in Fig. 1 (file `main.py (tentative)` in the middle). In general, a tentative merge can contain multiple so-called *conflicting chunks* [16], a single one is depicted in the tentative merge in Fig. 1. All lines between `<<<<<<< main (HEAD)` and `|||||||`, in this case line 3, represent the changes in the repository branch we merge into (*ours*). Changes in the branch we are trying to merge (*theirs* in line 5) and the base version of both branches of the conflicting file (*base* in line 7) are similarly delimited by markers. The resolution of the conflict is left to the developers' discretion by editing the respective file.

Research has tackled the challenges of software merging for decades, with the goal of minimizing manual effort. The proposals can roughly be classified into techniques that aim at (i) increasing the accuracy of *conflict detection* by lifting the unstructured, line-based merging of today's mainstream VCSs to a (semi-)structured, syntactic [21–23, 33] or even semantic level [23, 29], (ii) *preventing merge conflicts* by early conflict detection [8, 17, 18] and raising conflict-awareness [14], and (iii) pursuing a *semi-automated conflict resolution* by relying on language-specific conflict resolution

*These authors contributed equally to the paper.

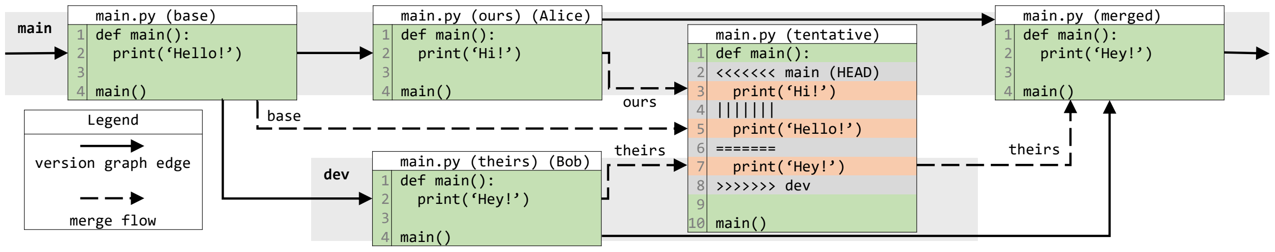


Figure 1: An example of collaborative development on two branches: *main* and *dev*. A Python file is edited in both branches, leading to a merge conflict. Non-conflicting code is shown in green, conflicting code in red, Git's merge markers in gray.

patterns [7, 10]. However, it has long been recognized that sophisticated merge techniques proposed in the literature had almost no impact [15], and the situation has shown little improvement thus far [16]. For the time being, we accept the status quo that merge facilities provided by mainstream VCSs are undisputed in practice.

A few empirical studies have been conducted with the aim of better understanding the nature of merge conflicts and how developers deal with them in practice [16, 24, 25, 28, 32, 34]. In particular, Yuzuki et al. [32] report that, for conflicting chunks caused by concurrent changes on a single line of code, developers adopted one of the alternatives (i.e., *ours* or *theirs*) in 99% of the cases without any modification. Ghiotto et al. [16] found that many conflicting chunks encompass all lines of code present in the merged result. They suggest that rearranging the code lines of chunks may be helpful for conflict resolution, where the merge tool takes the role of a recommender and proposes such rearrangements.

To evaluate the feasibility of such a recommender, we conservatively focus on those conflict resolutions for which we hypothesize that their generation could be feasible in practice. Intuitively, we consider the resolution of a conflicting chunk to be *derivable* if it can be directly constructed from the conflicting parent versions or their base version without any modification (more details in Sec. 3.1.1). To investigate this intuition in detail, we define a fixed set of language-agnostic conflict resolution patterns where one of the patterns can resolve a single chunk. This yields a classification scheme that classifies conflicting chunks into derivable and non-derivable ones. We aggregate all observations made for individual chunks and their surrounding non-conflicting lines of code (i.e., context) to the level of conflicting files and merge commits. File or merge commits are derivable if all their conflicting chunks are derivable and their context remains unchanged (stable).

Based on our precise yet simple notion of derivable conflict resolution, we seek to answer our overall research question through a large-scale simulation and quantitative analysis of merge commits extracted from the development histories of open-source projects hosted on GitHub. We followed established sampling criteria for gathering a large set of 10,000 heterogeneous yet representative projects. We replayed the merge commits of these projects' development histories. Using the merged parents and their common ancestor, we called Git's three-way-merge facility to check whether the original merge attempt led to a conflict, while the merge commit itself served as ground truth of how a merge conflict has been resolved. Based on this information, we investigate the derivability of chunk resolutions and merge commits, and how the derivability is affected by a number of variables, including general project characteristics, the programming language of source code artifacts, and previously used resolution strategies within the repository.

In a nutshell, the derivability rates are promising but substantially vary between the different levels of granularity wrt. individual conflicting chunks, files and commits. On the most fine-grained level, we found 87.9% of conflicting chunks to be derivable, with minor differences among development artifacts of different kinds and programming languages. On the most coarse-grained level, we found 34.5% of merge commits to be derivable, with unstable context and increasing numbers of chunks per conflicting merge being the major factors with a negative impact on the derivability. While these numbers largely confirmed our hypotheses, we got some interesting new insights on the granularity level of conflicting files. Namely, conflicting files receive a rather homogeneous treatment wrt. our conflict resolution patterns, meaning that developers tend to choose one resolution type per file, even for files with many conflicting chunks. Further, by inspecting influencing factors wrt. derivability, we found the strongest correlations on the level of files. This is an indicator that predicting an automated resolution might be feasible for files. Thus, to give a short and preliminary answer to *"is semi-automated conflict resolution easier than we expected?"*: Yes, it might indeed be easier, particularly if we use the right level of granularity for proposing conflict resolutions.

Our insights pave the way for further research. Through our comprehensive analysis, we aspire to bridge the gap between academic innovations on sophisticated merge techniques and real-world merge conflict scenarios, laying the groundwork for more effective and widely accepted automatic merge tools. We summarize our contributions as follows:

- A simple and language-agnostic concept of *merge commit derivability* which we assume to be feasible in practice;
- an open-source tool, *MeGA*, for efficiently analyzing thousands of merge commits wrt. derivability and collecting additional (meta-) data for further analysis;
- a *huge dataset* comprising the merge commits of a representative and heterogeneous collection of 10,000 projects hosted by GitHub, amounting to 131,154 conflicting merge commits, comprising 837,518 conflicting files;
- a *large-scale quantitative analysis* of the gathered merge commits wrt. derivability, the findings of which can be used to inform further research on constructing an automatic merge recommender;
- A full *replication package* [1] to reproduce and validate our results.

2 BACKGROUND ON MERGE CONFLICTS

While several conceptual organizations for the version space of software artifacts have been proposed in the literature [11], mainstream VCSs such as Git are based on organizing versions in a directed acyclic graph, which is commonly known as version graph. A version graph is composed of parallel branches that, in turn, consist of a sequence of revisions (aka. commits).

Fig. 1 shows an exemplary version graph where Alice and Bob collaborate on a simple Python greeting program. Initially, Alice implements a `Hello!` *program in a* main.py *file and commits it to branch* `main` *(top-left corner). Bob creates a new branch* `dev` *and changes the greeting to* `Hey!`*. Concurrently, Alice also changes the greeting on branch* `main` *to* `Hi!`*. Thus, two alternatives of the program coexist.*

Regardless of whether branches are named or implicit (e.g., a local copy of a remote branch), branches may eventually be merged to join the distributed work effort. Conceptually, a merge operation combines two alternative versions v_1 and v_2 of a development artifact into a merged artifact v_m [11]. The preferred way of software merging in the context of version control is three-way merging (cf. Fig. 1), which consults the common base version v_b (aka. ancestor or base) to make certain merge decisions [23].

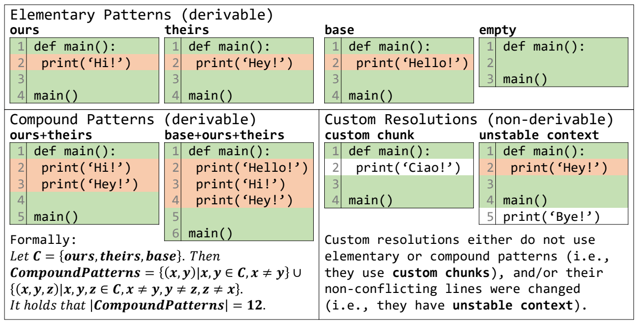
Merging in Git follows the principle of *state-based merging*: It determines symmetric differences along three ways ($v_b \Delta v_1$, $v_b \Delta v_2$, and $v_1 \Delta v_2$) to identify those parts of the development artifacts which are (a) the same in all three versions, (b) the same in two versions, or (c) different in all three versions. The symmetric differences are determined in a line-based manner, treating all development artifacts uniformly as text. This way, the common and distinct parts of a file are represented as line sequences. The merge result v_m includes all parts of type (a) and all parts of type (b) which differ from v_b [20]. All other cases indicate a conflict, meaning a part is different in all of v_1, v_2, v_b.

Alice and Bob agree to unify their work, and decide to merge `dev` *into* `main`*. To their displeasure, a merge conflict occurs.*

In case of a conflict, Git creates a *tentative merge* that requires resolution by a developer. A tentative merge consists of one or more *tentative files* that represent concurrently edited files whose individual parts overlap with respect to the common ancestor (i.e., the base version). A tentative file consists of one or more *chunks*. Each chunk includes the concurrently edited parts of v_1 and v_2 as well as the part of v_b at the location where they overlap. Special *conflict markers* [19, 31] are used to indicate the origin (v_1, v_2, or v_b) of these parts.

In a three-way merge, Git distinguishes three versions in a chunk with respect to the branch on which the merge operation is executed. The *ours* version contains the individual part of the branch on which the merge operation is executed (aka. current branch). The *base* version contains the state of the common ancestor in the version graph (i.e., before changes in branches were made). Lastly, the *theirs* version contains the individual part of the branch being merged into the current branch.

As shown in Fig. 1, Alice and Bob face a conflict in main.py *with a conflicting chunk in Lines 2-8. The chunk has been written directly into* main.py *and requires resolution before they can complete the merge. The chunk contains the conflicting lines* `print('Hi!')` *(ours),* `print('Hello!')` *(base), and* `print('Hey!')` *(theirs). Conflict markers, highlighted gray, identify these components. All other files (not shown) and code highlighted green were merged without conflict.*


Elementary Patterns (derivable)
ours
theirs
base
empty
Compound Patterns (derivable)
ours+theirs
base+ours+theirs
Custom Resolutions (non-derivable)
custom chunk
unstable context

ours:
```
def main():
 print('Hi!')

main()
```

theirs:
```
def main():
 print('Hey!')

main()
```

base:
```
def main():
 print('Hello!')

main()
```

empty:
```
def main():

main()
```

ours+theirs:
```
def main():
 print('Hi!')
 print('Hey!')

main()
```

base+ours+theirs:
```
def main():
 print('Hello!')
 print('Hi!')
 print('Hey!')

main()
```

custom chunk:
```
def main():
 print('Ciao!')

main()
```

unstable context:
```
def main():
 print('Hey!')

main()
print('Bye!')
```

Formally:
Let $\boldsymbol{C} = \{\boldsymbol{ours}, \boldsymbol{theirs}, \boldsymbol{base}\}$*. Then* $\boldsymbol{CompoundPatterns} = \{(\boldsymbol{x},\boldsymbol{y}) | \boldsymbol{x},\boldsymbol{y} \in \boldsymbol{C}, \boldsymbol{x} \neq \boldsymbol{y}\} \cup \{(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) | \boldsymbol{x},\boldsymbol{y},\boldsymbol{z} \in \boldsymbol{C}, \boldsymbol{x} \neq \boldsymbol{y}, \boldsymbol{y} \neq \boldsymbol{z}, \boldsymbol{z} \neq \boldsymbol{x}\}$.
It holds that $|\boldsymbol{CompoundPatterns}| = \boldsymbol{12}$.

Custom resolutions either do not use elementary or compound patterns (i.e., they use **custom chunks**), and/or their non-conflicting lines were changed (i.e., they have **unstable context**).

Figure 2: Derivable resolution patterns and non-derivable custom resolutions of main.py of Fig. 1.

Git can automatically resolve concurrent changes in different locations of a file, but those in the same location require manual resolving, which can be a tedious and error-prone process. Manual resolution is supported by modern IDEs and VCS tools that often present a side-by-side view of the individual versions (i.e., *ours*, *base*, and *theirs*). While a developer can simply select one of the conflicting versions (e.g., *ours*), they may also introduce custom changes to the conflicting file in order to resolve the merge conflict.

To resolve the conflict in Fig. 1, Alice and Bob manually edit main.py*, remove all conflict markers and the undesired ours and base version. They then mark the file as resolved and commit the merge.*

3 RESEARCH METHODOLOGY

Striving for our overall goal of exploring the feasibility of a simple semi-automated merge conflict resolution, we first present our envisioned resolution strategy in Sec. 3.1. Thereupon, in Sec. 3.2, we derive three research questions guiding our empirical study that we answer through a large-scale quantitative analysis of merge commits extracted from the development histories of Git repositories hosted on GitHub, as presented in Sec. 3.3.

3.1 Envisioned Conflict Resolution Strategy

We inductively define our hypothetical conflict resolution strategy by first introducing a fixed set of simple patterns for deriving conflict resolutions on the granularity level of individual chunks, which we then aggregate to define the derivability of merge commits, accounting for the chunks' context.

3.1.1 Derivability of Chunk Resolutions. We consider the resolution of a chunk to be derivable if it can be directly constructed from the conflicting parent versions or their base version. More precisely, we define a classification scheme based on the following patterns:

Elementary patterns (derivable). Elementary patterns describe resolutions where a developer chose *exactly one* or *none* of the alternative parts of a chunk. Thus, we define the four derivable elementary patterns *ours*, *theirs*, *base*, and *empty*. We illustrate these patterns in the upper part of Fig. 2, which constitute derivable resolutions of the conflict from our motivating example in Fig. 1. Here, the patterns correspond to selecting either `print('Hi!')`, `print('Hey!')`, `print('Hello!')`, or removing conflicting text.

Compound patterns (derivable). Compound patterns comprise all possible combinations of elementary patterns except the *empty* pattern, which is a neutral element wrt. combinations. Thus, a conflict resolution matches a compound pattern if a developer has combined the unmodified content of the *ours*, *theirs* or *base* version. While any ordering of these elementary patterns is permitted, we do not consider a reshuffling of lines between the patterns due to the combinatorial explosion of possible options. We consider their derivation infeasible for a practical, lightweight approach to semi-automated conflict resolution. Instead, we have a fixed set of 12 compound patterns; the formal definition of this set as well as two example elements are shown in the lower left part of Fig. 2.

Custom Resolutions (non-derivable). Custom resolutions are chunk resolutions that cannot be derived using our patterns above. We classify custom resolutions as non-derivable as it is infeasible to derive them without additional input from the developer. The lower right part of Fig. 2 shows a custom resolution on the left, where the developer replaced the conflicting line with `print('Ciao!')`.

3.1.2 Derivability of Merge Commits. While the derivability of all conflicting chunks of a tentative merge result yields a necessary precondition for deriving merge commits, we also require that non-conflicting lines, in the sequel referred to as *context*, remain unchanged (stable). This is necessary because, even if all chunk resolutions are derivable, the actual resolution of a conflict may take place in the chunks' surrounding context. The lower right part of Fig. 2 shows such an example; on the right, line 5 was appended, rendering the context unstable. To sum up, a merge commit is derivable if all its chunks are derivable *and* their context stable. Otherwise, the merge commit is non-derivable.

3.2 Research Questions

We define three research questions that guide our study.

RQ1: *How many conflicting chunk resolutions are derivable, and how are they distributed over our resolution patterns?* Based on Ghiotto et al.'s findings [16], our hypothesis is that developers choose a derivable chunk conflict resolution in most cases. With RQ1, we seek to measure how many chunk resolutions found in the histories of open-source projects are derivable, and how the resolutions are distributed over our resolution patterns (cf. Sec. 3.1). If our hypothesis is confirmed, we can continue with a deeper inspection of the derivability of entire merge commits.

RQ2: *Are merge commits derivable, and does this vary with an increasing number of conflicting chunks?* With RQ2, we seek to measure how many merge commits comprise *only* derivable chunk resolutions and how often a chunk's context is modified during a merge. Our hypothesis is that the derivability rate of merge commits comprising only a single conflicting chunk is lower than the derivability rate of individual chunks due to the instability of context, the extent of which is yet an open question. Furthermore, we hypothesize that the derivability rate drops with an increasing number of chunks, but it is yet unclear whether this effect can be observed in practice.

Table 1: Overview of our collection of GitHub projects.

	projects	commits	merges	conflicted merge %	sampled merges
C#	1,000	2,690,761	406,269	54.6	16,611
C	1,000	8,884,850	714,432	43.6	15,015
C++	1,000	7,308,952	1,126,367	45.6	22,011
Go	1,000	3,211,249	627,271	88.2	13,204
JavaScript	1,000	291,441	35,586	99.3	2,685
Java	1,000	992,236	126,451	65.0	6,023
PHP	1,000	3,044,459	656,015	46.4	20,163
Python	1,000	471,755	66,488	81.5	3,481
Rust	1,000	2,633,980	442,468	60.0	10,609
TypeScript	1,000	3,256,112	440,869	58.1	21,352
total	10,000	32,785,795	4,642,216	64.2	131,154

RQ3: *Which quantitative factors influence derivability, and do they depend on the granularity level of conflicting merges?* With RQ3, we seek to explore which factors might influence the derivability of merge results on different levels of granularity. Hereby, we hope to gain insights for constructing heuristics that could later guide automated merge conflict resolution. Aiming to be language-agnostic, we limit our analysis to generic factors related to a merge (e.g., chunk count, resolution patterns, derivability) and prior repository history (e.g., length of branches before merges, prior resolution patterns, number of commits).

3.3 Study Design and Implementation

We seek to answer our research questions through a large-scale quantitative analysis of merge commits extracted from the histories of Git repositories on GitHub. In addition to providing rich metadata for each project, GitHub provides transparency and access to the version histories of Git repositories, which currently is one of the most popular and widespread VCSs. For the sake of our empirical study, we implemented *MeGA* (MergeGitAnalyzer), a command-line tool written in Java. It uses GitHub's REST API [2] to acquire repositories, and JGit [3], an implementation of Git written in Java, to run the actual analysis. *MeGA* is available as an open-source project and can efficiently analyze thousands of Git histories by implementing a multi-level parallel processing for repositories, merge commits and their files. A detailed configuration setup used for our study is part of our replication package.

3.3.1 GitHub Project Collection. To gather a representative, large-scale and heterogeneous collection of projects from GitHub, we followed established sampling techniques based on popularity metrics. Specifically, we first selected the ten most popular languages based on the number of stars received in Q3 2023 according to GitHut [4]: Python, JavaScript, Go, C++, Java, TypeScript, C, C#, PHP and Rust. Then, we defined search queries to retrieve repositories for each programming language according to GitHub's project type classification [5], sorting the repositories in descending order by their number of stars, and collecting the first 1,000 entries. Using this method, we collected a total of 10,000 projects as study subjects, an overview of which is given in Tab. 1. We sampled (at most) the last 100 conflicting merges per project to reduce the bias of projects with extensive histories.

3.3.2 Merge Commit Simulation and Analysis.

Core Analysis Workflow. Since we cannot directly analyze conflict resolutions from a project's history, we simulate (i.e., replay) merges from the history and analyze the chosen resolutions.

Given a list of GitHub projects, *MeGA* searches the entire history of each project for three-way merge commits to simulate. Merge commits with more than two parents (aka. octopus merges) are skipped, as they only constituted about 0.32% of merges found in the projects. For each merge commit, *MeGA* determines the involved parent and base commits and simulates a merge using Git's default merging strategy [6].

If the simulated merge contains conflicts, *MeGA* collects all files with conflicts (*tentative files*) and their corresponding *committed* version. *MeGA* then traverses the tentative file and tries to match the chunk parts (*ours*, *base*, *theirs*) of each chunk and the context surrounding the chunks within the *committed* file using a line-based text equality matching. If a chunk part or context part is found, the corresponding lines in the *committed* file are marked as *mapped* to the respective part in the *tentative* file. Once such a matching has been performed for all chunk and context parts, *MeGA* determines whether derivable patterns were used to resolve the conflicts and whether the context is stable. Compound patterns, that are permutations of each other, are grouped together by *MeGA*, resulting in 4 elementary and 12 compound patterns, see Sec. 3.1.

If all lines of a committed file are mapped, then the context is stable, and all conflicting chunks have been resolved using a derivable pattern. This is the case for our example in Fig. 1, where the lines of the committed file `main.py (merged)` would be mapped to the tentative file `main.py (tentative)` as follows: $\langle 1\rangle \mapsto \langle 1\rangle$, $\langle 2\rangle \mapsto \langle 7\rangle$, $\langle 3,4\rangle \mapsto \langle 9,10\rangle$. However, non-mapped lines in a committed file indicate that either context modifications occurred or non-derivable conflict resolutions were used. Namely, if non-mapped lines are found between two adjacent context parts or two chunk parts belonging to the same chunk, they indicate a non-derivable conflict resolution. Alternatively, if non-mapped lines are found between a chunk part and its adjacent context, they indicate an unstable context. If non-mapped lines are found between non-adjacent chunk or context parts, then all chunks in between receive a non-derivable resolution, and all context parts in-between are unstable.

Dealing with Non-Linear Alignments. Developers may change the order of chunk resolutions when resolving conflicts, meaning that they move a chunk resolution after or before the resolution of the preceding or following chunks. We found that only 2.22% of files in our sample were reordered.

Such reorderings are challenging for a recommender due to possible combinatorial explosions. Especially for large files with many chunks, restructurings can result in ambiguous matches between the chunks of the tentative merge and the commit. As such ambiguities may tamper our resolution pattern classification, we conservatively treat restructured files as non-derivable (in RQ2) and make no statement about the derivability of their contained chunks. For the classification of individual chunks in RQ1, we exclude all chunks of restructured files, as we cannot confidently classify them.

Whitespace Normalization. To minimize the effect of whitespace differences in our mapping and subsequently on the derivability rates of chunks and merges, *MeGA* normalizes whitespace in *tentative* and *committed* files prior to analysis. This is a lightweight, language-agnostic formatting that removes all empty lines within the files, removes all leading and trailing whitespace from non-empty lines and reduces all other whitespace to a single space character `' '`. In a preliminary analysis, we measured the effect of whitespace normalization on our analysis results and found slightly more derivable patterns (i.e., in the mean, 0.376% *ours*, 0.0564% *theirs*, and 0.687% all other derivable patterns) and 1.12% fewer *non-derivable* resolutions. This suggests that developers occasionally apply custom formatting while resolving merge conflicts, particularly for the less common (see Sec. 4.1) resolution patterns.

3.3.3 Statistical Data Analysis. *MeGA* stores the information about chunk resolutions and eventual context modifications, giving insight into how often each derivable pattern was used (RQ1), and, by aggregation, whether the merge commit is derivable (RQ2). Lastly, *MeGA* collects various other data about the merge, serving as the basis for a statistical correlation analysis (RQ3). All data is then analyzed and visually represented by Jupyter Notebooks, which are part of our replication package.

4 RESULTS

In this section, we summarize our results structured by research questions, while all details can be found in the replication package.

4.1 RQ1: Derivability of Chunk Resolutions

In Fig. 3, we show how chunks are distributed over our resolution patterns (i.e., how often they match one of our (non-)derivable resolution patterns of Fig. 2), separated by GitHub project types. We find that, overall, 87.9% of chunks are resolved by using one of our derivable patterns, with the elementary patterns *ours* and *theirs* being used most often. Altogether, the *ours* pattern was used in 62.4% of cases, more than twice as often as any other pattern. The *theirs* pattern was used more often than a non-derivable resolution in the majority of project types. We accumulated the frequencies of all remaining patterns in the *other* category, which even summed up remained the least used category. Nevertheless, we found each of our derivable patterns, even the most complex ones. Ordered by pattern frequency they are: *ours+base* (0.661%), *base* (0.974%), *ours+base+theirs* (1.22%), *theirs+base* (1.29%), *ours+theirs* (1.34%), *empty* (2.45%), *theirs* (17.6%), *ours* (62.4%), and *non-derivable* (12.1%).

Fig. 3 also shows a visible variance between the project types, e.g., while only 6.61% of C++ project chunks are non-derivably resolved, 20.3% are non-derivably resolved in Rust projects. Similarly, the distribution within the derivable patterns varies between the project type, but the inequalities $\mathit{ours} > \mathit{theirs} > \mathit{other}$ hold for every project type but C.

While Fig. 3 gives a breakdown following GitHub's classification of projects, other file types may be part of these merges (e.g., a Java project's merge could also include .json files, .xml files, etc.). Thus, we additionally analyzed whether source code files feature

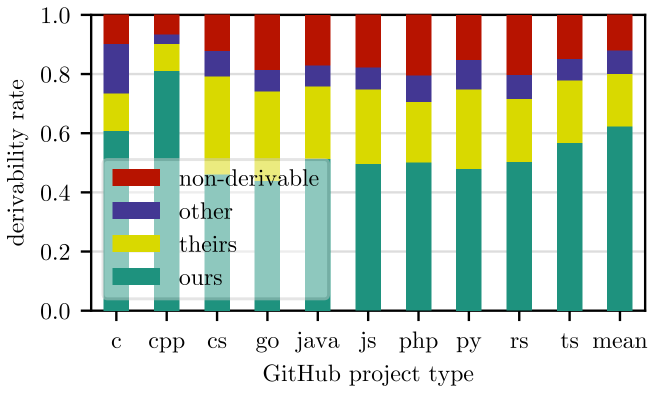


Figure 3: Chunk resolution patterns for different languages.

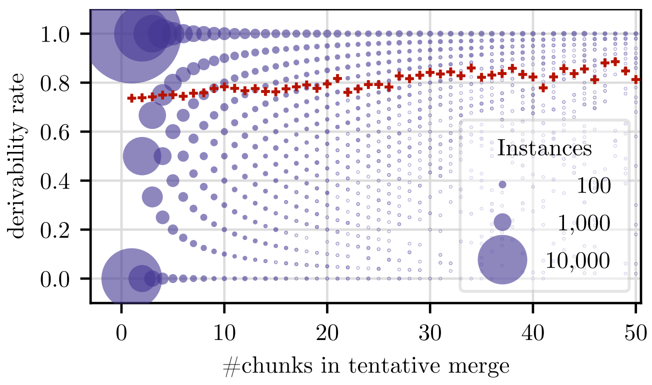


Figure 4: Derivability rates of chunks by tentative merge size. Bubbles represent instances of our sample; red crosses represent the mean chunk derivability rates.

different resolution patterns by considering only files matching a project's type (e.g., only .java files of a Java project). We found that the overall derivability rate of chunks does not change for source code files, and only the distribution within patterns *ours*, *theirs*, and *other* varied slightly in comparison to Fig. 3.

Fig. 4 gives an overview of chunk derivability rates with respect to the number of chunks of their tentative merge. A bubble at (x, y) represents tentative merges comprising x chunks, having a derivability rate of $y = \frac{i}{x}$ with i of the x chunks being derivable. The bubble size represents the number of instances of such merges in our sample. A red cross at (x, y) represents that chunks of a merge with x chunks have a mean derivability rate y. We see that the vast majority of merges has few chunks (left-most bubbles are bigger) and that merges of all sizes usually are resolved with a high derivability rate (upper bubbles are bigger and more abundant). The mean derivability rate of chunks increases slightly from around 0.75 to 0.85, but the right-most means show a high variance, as there are fewer tentative merges with a high number of chunks.

RQ1: How many conflicting chunk resolutions are derivable, and how are they distributed over our resolution patterns?
We found 87.9% of chunks to be derivable. The pattern *ours* was used most often to resolve chunks in every language and more than 60% overall. Chunks in the source code and other files of other types are mostly resolved in the same way. Merges with a low number of chunks dominate. The derivability rate of individual chunks increases slightly with a higher number of chunks in a tentative merge.

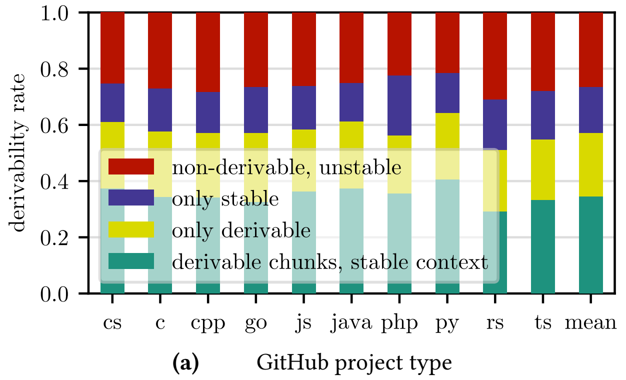
4.2 RQ2: Derivability of Merge Commits

We list our overall findings of RQ2 in Fig. 5a. All languages (in terms of GitHub project type) show a merge commit derivability of 30 to 40%; about 20% of the merges have a modified context while all chunks are derivable, and 16.5% of merges have at least one non-derivable chunk with a stable context. This leaves 26.4% of the chunks with both non-derivable chunk(s) and a modified context. For all languages, there are more merges where a modified context prevented derivability than merges where non-derivable chunk(s) prevented derivability. In contrast to our results for RQ1, the variability between the different languages is much reduced.

In Fig. 5b, we give a breakdown of the derivability rate by the number of chunks. The derivability drops with an increase of chunks, from around 45% derivability for a single chunk, to around 20% for merges with high chunk counts. With higher chunk counts, there are fewer instances of merges where only the context remains stable. The fact that the derivability levels off at around 20% (even for high chunk counts up to 200) can be explained by two factors: (i) the slight increase of the derivability rate for higher chunk counts (cf. Fig. 4), and (ii) the chunks of a tentative merge are *not resolved independently* – otherwise it would decrease at an exponential rate with the number of chunks. For a chunk count, e.g., 40, the mean derivability rate is 0.823 (cf. Fig. 4), which would result in a theoretical rate of $0.823^{40} = 0.000416$ for independent resolutions – we measured a derivability of 0.206 instead (cf. Fig. 5b).

The derivability of merge commits, especially of bigger size, seems discouraging at first glance. However, if tentative merges are usually resolved with a few different resolution patterns, at least the combinatorial explosion would be limited for a recommender. We just learned that chunks are not resolved independently; thus, we investigate the 'homogeneity' in merges and files. We define 'homogeneity' as the number of different resolution patterns used in a merge (cf. Fig. 6a) or a file (cf. Fig. 6b). If a merge uses only the pattern *ours* to resolve all chunks, it is counted in the bottom bar. If it also contains a *theirs* chunk, it is counted in the penultimate bar, etc. As tentative merges increase in chunk number, only around 30% of them are uniformly resolved with exactly one pattern. In contrast, individual files with many chunks are resolved much more homogeneously, even for very large files.

RQ2: Are merge commits derivable, and does this vary with an increasing number of conflicting chunks?
30 – 40% of the merge commits are derivable for all languages. Considering only chunks for merge commit derivability (omitting context), ca. 60% of merge commits are derivable. Merge commit derivability is more consistent language-wise than chunk derivability of RQ1. The derivability rate of merge commits falls from ca. 45% to ca. 20% as the chunk count increases. The homogeneity of resolutions is much more uniform on file than on merge level.



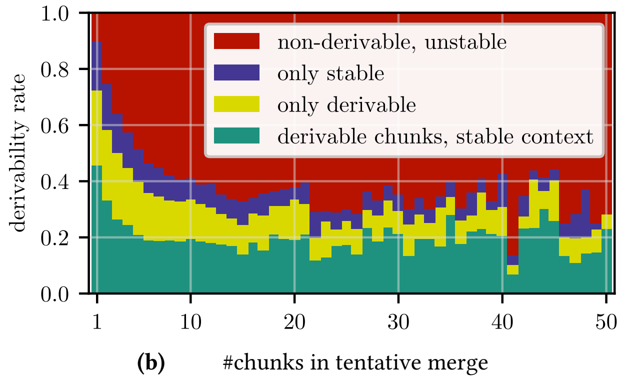
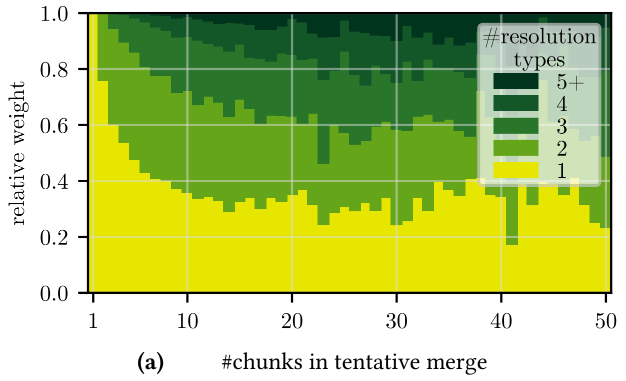


Figure 5: Distribution of derivability of merge commits (Fig. 5a), broken down by the number of chunks (Fig. 5b).



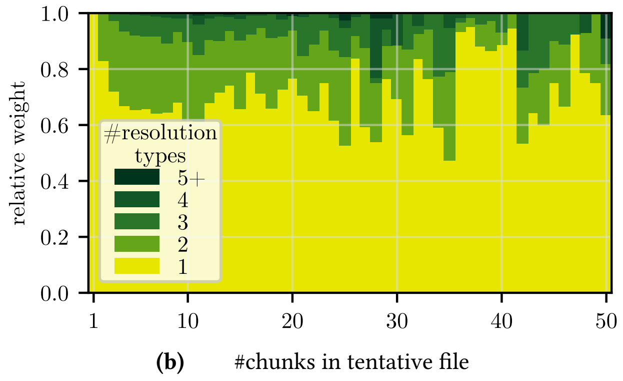


Figure 6: Homogeneity of resolutions in merges (Fig. 6a) and files (Fig. 6b) of different chunk counts.

4.3 RQ3: Factors correlating with Derivability

To learn more about the factors that influence whether a merge commit is derivable, whether chunks are derivable, and whether context is stable, we construct a correlation matrix, combining multiple variables, in Fig. 7. The upper part of the matrix pertains to variables of a merge and how it is resolved, the lower part to variables of a file and how it is resolved. Only the pairs of column 4 and rows 4 deviate by name (shown in red). Variables of the matrix marked with a '%'-symbol represent the ratio of chunks that satisfy the variable's condition. Variables without the '%'-symbol are binary. We compute the rank correlations ρ after Spearman [12], as none of our variables are normally distributed. In the matrix, correlations with $|\rho| \geq 0.3$ are shown with a number. The (very few) insignificant correlations are presented uncolored, same as completely uncorrelated variables. All other correlations are highly significant, with $p < 0.001$, even for weak correlations. Note that in contrast to many correlation analyses, here, some variables are related by definition, e.g., *ours%* + *theirs%* + *base%* + ... = *chunks derivable%*.

While the derivability of a merge commit (upper part of Fig. 7) is only influenced by whether all chunks are derivable and its context is stable, the latter shows a higher correlation, echoing our findings in Fig. 5b. Both *ours* and *theirs* are neutral in regard to a stable context, but are weakly positively correlated to chunk and overall derivability. A higher presence of *ours* chunks indicates fewer *theirs* chunks used in a resolution, and vice versa. The more exotic resolution patterns, i.e., *base%* and below, show only very weak correlations, but a higher presence of them comes with a more unstable context and less derivable merge commit.

In addition to the factors we show here, we also measured all pairwise correlations of the variables of the matrices and the following: number of chunks and conflicting files in the tentative merge; the percentage of conflicting source code files, the length of conflicting files; whether named or implicit branches were merged; how many commits and authors (up to the base version) the branches had before merging; prior used resolution patterns and when the merge commits took place. However, none of these factors showed a correlation stronger than 0.3 to any of the factors in this matrix.

As we have seen in RQ2, the resolution homogeneity in files can be much higher than in merges. Therefore, we also investigate file correlations. In the lower part of Fig. 7, we observe an almost identical replication of the upper part, but with stronger correlations – all colored cells are again highly significant with $p < 0.001$. On the file level, some variables not depicted in Fig. 7 also showed correlations, e.g., how often the *ours* pattern was used in files of the same file type prior to the current merge correlated to 'file derivable' with 0.35, to *ours* with 0.46, and to *theirs* with −0.35.

RQ3: Which quantitative factors influence derivability, and do they depend on the granularity level of conflicting

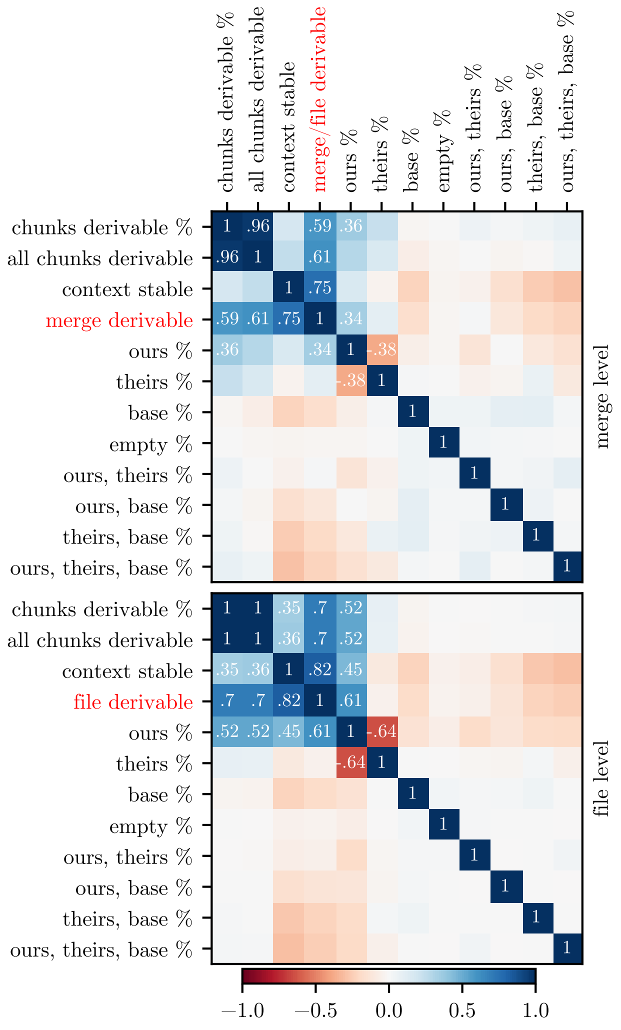


Figure 7: Correlation matrices of factors that potentially influence the derivability of merges and files.

merges?
Few of the analyzed factors show strong correlations. Correlations are similar, though more pronounced on file granularity opposed to merge granularity. Using *ours* is inversely correlated to using *theirs*. Using more exotic resolution strategies slightly correlates with modifying context and thus merges becoming non-derivable.

5 DISCUSSION

We discuss our results in terms of observations and implications wrt. our overall research goal and potential threats to validity.

5.1 Observations and Implications

RQ1. Building on Ghiotto et al.'s [16] findings, our hypothesis that developers favor derivable chunk resolutions was strongly supported. Overall, 87.9% of chunks are derivable, with the pattern *ours* being predominant, exceeding 50% across most analyzed languages. Most individual chunks being resolved using derivable patterns emphasizes the potential for semi-automated resolutions for individual chunks.

RQ2. As analyzed in Figs. 5a and 5b, the derivability of merge commits drops significantly compared to the derivability of individual chunks. At first glance, these results do not support the idea of a (simple) semi-automated merging tool. However, we must consider that a single non-derivable conflicting chunk may categorize the merge as non-derivable. Moreover, our approach to classify derivable resolutions is, on purpose, highly conservative in that we do not allow restructurings in the merge commit. Considering that the probability of a non-derivable conflicting file increases with the size of a merge, these results are also not too surprising.

With this in mind, it is noteworthy that non-derivable changes tend to be more prevalent in the *context* rather than within *chunks*. Consequently, a merge not being derivable solely due to an unstable context can still be considered a positive outcome. While leaving the context for developers, the resolutions of individual chunks could potentially be generated.

The most obvious candidates for semi-automated resolution are merges with few chunks. Considering Fig. 4 (see large bubbles in the upper left corner) and Fig. 5b (see left-most bars), such merges have the highest derivability, occur most often, and are also the least complex ones considering a possible combinatorial explosion.

As indicated by Fig. 4 (see red crosses), developers choose derivable chunks more often when tentative merges grow large. This relation suggests that, in large merges, developers fall back to a simple resolution pattern more often. However, the derivability rate still does not approach 1.0, even for extremely large merges with hundreds of chunks, resulting in about 10% of chunks that are not resolvable without additional information.

A similar effect can be observed for the homogeneity of conflict resolutions and the chunk count. In Fig. 6a (see right-half), the homogeneity slightly increases for (most) larger merges. Furthermore, it is noteworthy that this effect becomes even more pronounced when considering the homogeneity per file, as depicted in Fig. 6b (see right-half). This difference in Figs. 6a and 6b indicates that developers tend to choose one resolution type per file, even though in a whole merge, the resolution types may differ. Thus, a tool that supports developers in resolving merge conflicts could propagate a resolution strategy from a single chunk to the whole file, while propagating the resolution for the entire merge may require additional (probably structural or even semantic) information.

Language-wise, we found only marginal differences with respect to Figs. 3 and 5a. Except in Fig. 3, the *ours* pattern in C and C++ projects is more predominant. However, when filtering for specific source code files, the difference vanishes. These results can be interpreted such that there are at least certain basic resolution strategies that are language-agnostic.

RQ3. Here, we examined factors influencing the derivability of merge or file resolutions. Few factors showed strong correlations, with file-level granularity (cf. Fig. 7) exhibiting more pronounced correlations than merge-level granularity. This is an indicator that it is easier to give predictions on a file level than on the merge level, echoing our findings of homogeneity. The inverse correlation between the *ours* and *theirs* pattern (see middle part of the matrices), and the correlation of exotic resolution strategies with modified context, provide insights into developer preferences and potential heuristics for automated conflict resolution. However, most factors we analyzed for RQ3 did not show any linear correlation, e.g., we found the commit time or the usage of implicit branches had no impact on the conflict resolution.

In general, there are countless potentially influencing factors; in our analysis, we picked those that seemed most likely from our intuition and experience. Nevertheless, the sample of factors we analyzed led us to conclude that merge conflicts only weakly follow project-specific factors and are rather resolved individually. In other words, automatically resolving merge conflicts is still a highly challenging problem. However, our in-depth analysis provides some interesting starting points for building the next generation of automatic merging tools.

5.2 Threats to Validity

5.2.1 Internal Validity. To conduct our analysis, we implemented *MeGA* using the well-maintained JGit library. Our tool incorporates various algorithms and techniques designed for efficiently analyzing thousands of Git histories. Arising problems are logged [1] for the analyzed files and merges. Only a small fraction of samples encountered technical problems: We excluded 1.18% of all merge commits, as they only contained non-line-based binary files, e.g., images. Moreover, for 1.59% of merge commits, we could not determine the specified base commit due to missing or pruned repository information. We further excluded 1.83% of the merge commits, which JGit flagged false-positively as conflicting, while no conflicting file was present. In projects exceeding 100 merges (see Sec. 3.3), we resampled to obtain 100 unproblematic merges each. Moreover, 1.62% of the commits in our sample are duplicated due to forked projects on GitHub.

Our analysis replays merges using Git's default merge strategy. In general, we did not know the exact settings and tools used by the developers to resolve the merge conflict. Thus, some developers may have perceived merges differently. However, assuming a (hypothetical) recommender system, it is reasonable to assume a default merging strategy to estimate the possibility of automation.

The mapping of the tentative merge to the merge commit is performed using a textual line-based mapping. Using structural mappings, more mappings of chunks or their context may be found. In principle, this could have improved our results concerning the derivability of merges. To mitigate this problem while still being language-agnostic, we introduced whitespace normalization, as described in Sec. 3.3.2. For whitespace-sensitive languages such as Python, this does not have a noticeable impact in our results.

We solely analyzed conflicting files in a merge to assess context stability, not accounting for potential modifications developers performed in non-conflicting files during the merge.

Finally, we rely on GitHub's classification [5] to determine the language of a project, which may be ambiguous, especially for larger projects that are often polyglot. However, our conclusions do not rely on the language classification, and therefore, such ambiguities do not impact their validity.

5.2.2 External Validity. The generalizability of our findings depends on several factors concerning our sampling strategy: First, we analyzed only Git repositories as it is the state-of-the-art VCS – other mainstream VCSs may handle merge conflicts differently. However, all mainstream VCSs use similar three-way merging techniques, from which we expect similar results. Second, we only analyzed ten languages, and our results may not be generalizable to conceptually divergent niche languages. Nevertheless, we covered the most popular languages of diverse programming paradigms. Third, we selected the 1000 most popular GitHub projects per language. A project's popularity might influence how merging is handled, though. We limited sampling to 100 merges per project to prevent bias from projects with extensive histories. Although relevant influencing factors for RQ3 might have been missed, our approach revealed promising leads for future research.

6 RELATED WORK

Ghiotto et al. [16] conducted the most closely related study to ours. They analyzed individual chunks and their resolution in detail in open-source Java projects. They found that 87% of the chunk resolutions added no new code which inspired our work. However, we take a much more conservative view than Ghiotto et al. by focusing on *derivable resolutions*, hypothesizing that they could be practically derived. Specifically, our definition of derivable chunks avoids a combinatorial explosion of possible rearrangements of conflicting code lines, and we consider context stability as another necessary condition for the derivability. Moreover, as opposed to their language-specific approach, we deliberately keep our definitions and analyses language-agnostic. The generic functionality of mainstream VCSs has been one of the most important factors for their success [15]. Consequently, we did ourselves to Java projects but gathered a larger set of heterogeneous GitHub projects.

Other empirical studies on the nature of merge conflicts and how developers deal with them are less related to our work. Yuzuki et al. [32] found that our derivability patterns, *ours* and *theirs*, constitute up to 99% of conflicts within a method body. While this suggests an even higher fraction of derivable chunks, the findings are based on only ten open-source Java Projects. The studies conducted by Zimmermann [34] and Nguyen et al. [25] are comparable to ours from a methodological point of view in that they are mining open-source repositories for merge commits. Yet, they are guided by different research questions that focus on integration and conflict rate evolution in a project. Shen et al. [28] studied semantic conflicts, which are out of the scope of our study as they are not detected by Git, but in later stages of compiling and testing. Though limited to a manual qualitative analysis, they found that such conflicts usually need a non-derivable resolution. Interestingly, such resolutions often still seem to be similar to one of our patterns *ours* or *theirs*. Qualitative results on how developers organize conflict management have been obtained through semi-structured interviews conducted by Nelson 2019 et al. [24]. Interestingly, participants reported situations where too many conflicts involved in a merge attempt were simply resolved by discarding one of the branches. We can empirically confirm this finding as, from a certain number of conflicts in a merge, their derivability tends to increase again.

Most empirical studies, however, revolve around the performance evaluation of sophisticated techniques involved in software merging. Traditionally, there is a line of research that aims at providing empirical evidence for (semi-)structured merging leading to an increased accuracy in conflict detection compared to unstructured merging [9, 27, 28], which is largely orthogonal to our research goal of improving conflict resolution. Our work rather aligns with resolution techniques utilizing program synthesis [26] and deep-learning [13, 30]. While these approaches are still in their infancy, they might become feasible in the future. In fact, we favor simplicity in terms of language-agnostic conflict resolution patterns that are easy to understand for developers, but their application in terms of a recommender system may well be combined with machine learning to predict their applicability. In this regard, our comprehensive dataset may serve as a basis for supervised learning.

7 CONCLUSION

We conducted a large-scale empirical study on how merge conflicts are resolved in practice. In our study, we analyzed merge conflicts from 10,000 GitHub projects that cover the most popular programming languages. We found that the vast majority of chunk resolutions are derivable (87.9%). In almost all derivable cases, developers chose the simple elementary patterns *ours* and *theirs* (62.4% and 17.6% respectively). Further, we found that around 35% of merge commits are derivable, but the derivability drops to around 20% with increasing chunk count due to unstable context and non-derivable chunks. Lastly, we investigated which factors influence derivability. Here, we identified few factors with high correlation (e.g., the *ours* pattern) and some factors with low correlation (e.g., compound patterns such as *ours+theirs*, or *ours+base*).

Based on our results, we believe that semi-automated conflict resolution is feasible in general and that, for some use cases, it may be easier than expected. So far, however, our conflict resolution strategy is just hypothetical, and the next consequent step towards turning it into reality is to develop a recommender that predicts the most likely resolution pattern for a given conflict, or resorts to a manual resolution if no derivable resolution seems appropriate. While we leave this step for future work, both our findings and the gathered dataset serve as a promising starting point.

ACKNOWLEDGMENTS

This work has been supported by the Swiss National Science Foundation (SNSF) under grant 219719.

REFERENCES

[1] 18-01-2024. https://zenodo.org/records/10511517.
[2] 18-01-2024. https://docs.github.com/en/rest?apiVersion=2022-11-28.
[3] 18-01-2024. https://eclipse.dev/jgit.
[4] 18-01-2024. https://madnight.github.io/githut/#/stars/2023/3.
[5] 18-01-2024. https://github.com/github-linguist/linguist.
[6] 18-01-2024. https://git-scm.com/docs/git-merge.
[7] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. 2010. Colex: A Web-based Collaborative Conflict Lexicon. In *IWMCP*. ACM, 42–49.
[8] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *TSE* 39, 10 (2013), 1358–1375.
[9] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. 2019. The Impact of Structure on Software Merging: Semistructured Versus Structured Merge. In *ASE*. IEEE, 1002–1013.
[10] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. 2008. Managing Model Conflicts in Distributed Development. In *MODELS (LNCS, Vol. 5301)*. Springer, 311–325.
[11] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *CSUR* 30, 2 (1998), 232–282.
[12] Joost CF De Winter, Samuel D Gosling, and Jeff Potter. 2016. Comparing the Pearson and Spearman Correlation Coefficients Across Distributions and Sample Sizes: A Tutorial Using Simulations and Empirical Data. *Psychological methods* 21, 3 (2016), 273.
[13] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu K. Lahiri. 2023. DeepMerge: Learning to Merge Programs. *TSE* 49, 4 (2023), 1599–1614.
[14] H.-Christian Estler, Martin Nordio, Carlo A. Furia, and Bertrand Meyer. 2014. Awareness and Merge Conflicts in Distributed Software Development. In *ICGE*. IEEE, 26–35.
[15] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. 2005. Impact of Software Engineering Research on the Practice of Software Configuration Management. *TOSEM* 14, 4 (2005), 383–430.
[16] Gleiph Ghiotto, Leonardo Murta, Márcio de Oliveira Barros, and André van der Hoek. 2020. On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub. *TSE* 46, 8 (2020), 892–915.
[17] Mário Luís Guimarães and António Rito Silva. 2012. Improving Early Detection of Software Merge Conflicts. In *ICSE*. IEEE, 342–352.
[18] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive Conflict Minimization through Optimized Task Scheduling. In *ICSE*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE, 732–741.
[19] Petra Kaufmann, Horst Kargl, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel. 2010. Representation and Visualization of Merge Conflicts with UML Profiles. In *ME*. 53–62.
[20] Sanjeev Khanna, Keshav Kunal, and Benjamin C Pierce. 2007. A Formal Investigation of Diff3. In *FFTTCS*. Springer, 485–496.
[21] Simon Larsén, Jean-Rémy Falleri, Benoit Baudry, and Martin Monperrus. 2023. Spork: Structured Merge for Java With Formatting Preservation. *TSE* 49, 1 (2023), 64–83.
[22] Olaf Leßenich, Sven Apel, Christian Kästner, Georg Seibt, and Janet Siegmund. 2017. Renaming and Shifted Code in Structured Merging: Looking Ahead for Precision and Performance. In *ASE*. IEEE, 543–553.
[23] Tom Mens. 2002. A State-of-the-Art Survey on Software Merging. *TSE* 28, 5 (2002), 449–462.
[24] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. 2019. The life-cycle of merge conflicts: processes, barriers, and strategies. 24, 5 (2019), 2863–2906.
[25] Hoai Le Nguyen and Claudia-Lavinia Ignat. 2018. An Analysis of Merge Conflicts and Resolutions in Git-Based Open Source Projects. *CSCW* 27, 3-6 (2018), 741–765.
[26] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu K. Lahiri, and Mike Kaufman. 2021. Can Program Synthesis be Used to Learn Merge Conflict Resolutions? An Empirical Analysis. In *ICSE*. IEEE, 785–796.
[27] Georg Seibt, Florian Heck, Guilherme Cavalcanti, Paulo Borba, and Sven Apel. 2022. Leveraging Structure in Software Merge: An Empirical Study. *TSE* 48, 11 (2022), 4590–4610.
[28] Bowen Shen, Cihan Xiao, Na Meng, and Fei He. 2021. Automatic Detection and Resolution of Software Merge Conflicts: Are We There Yet? *CoRR* abs/2102.11307 (2021). arXiv:2102.11307
[29] Marcelo Sousa, Isil Dillig, and Shuvendu K Lahiri. 2018. Verified Three-Way Program Merge. *PACMPL* 2, OOPSLA (2018), 1–29.
[30] Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri. 2022. Program Merge Conflict Resolution via Neural Transformers. In *ESEC/FSE*. ACM, 822–833.
[31] Konrad Wieland, Philip Langer, Martina Seidl, Manuel Wimmer, and Gerti Kappel. 2013. Turning Conflicts into Collaboration. *CSCW* 22, 2-3 (2013), 181–240.
[32] Ryohei Yuzuki, Hideaki Hata, and Kenichi Matsumoto. 2015. How We Resolve Conflict: An Empirical Study of Method-Level Conflict Resolution. In *SWAN*. IEEE, 21–24.
[33] Fengmin Zhu, Fei He, and Qianshan Yu. 2019. Enhancing Precision of Structured Merge by Proper Tree Matching. In *ICSE*. IEEE, 286–287.
[34] Thomas Zimmermann. 2007. Mining Workspace Updates in CVS. In *MSR*. IEEE, 11.