


How code composition strategies affect merge conflict resolution?

Heleno de S. Campos Junior  [Universidade Federal Fluminense | helenocampos@id.uff.br]
Gleiph Ghiotto L. de Menezes [Universidade Federal de Juiz de Fora | gleiph@ice.uff.br]
Márcio de Oliveira Barros [Universidade Federal do Estado do Rio de Janeiro | marcio.barros@uniriotec.br]
André van der Hoek [University of California, Irvine | andre@ics.uci.edu]
Leonardo Gresta Paulino Murta [Universidade Federal Fluminense | leomurta@ic.uff.br]

Abstract

Software developers often need to combine their contributions. This operation is called merge. When the contributions happen at the same physical region in the source code, the merge is marked as conflicting and must be manually resolved by the developers. Existing studies explore why conflicts happen, their characteristics, and how they are resolved. This paper investigates a subset of merge conflicts, which may be resolved using a combination of existing lines. We analyze 10,177 conflict chunks of popular projects that were resolved by combining existing lines, aiming at characterizing and finding patterns frequently addressed by developers to resolve them. We found that these conflicting chunks and their resolutions are usually small (they have a median of 6 LOC and 3 LOC, respectively). Moreover, 98.6% of the analyzed resolutions preserve the order of the lines in the conflicting chunks. We also found that 72.7% of the chunk resolutions do not interleave lines from different contributions more than once. Finally, developers prefer to resolve conflicts containing only *Import* statements using lines from the local version of the conflict. When used as heuristics for automatic merge resolution, these findings could reduce the search space by 94.7%, paving the road for future search-based software engineering tools for this problem.

Keywords: *Version control systems, software merge, conflict resolution, search-based software engineering*

1 Introduction

Nowadays, collaboration is common when developing software products. Developers usually employ branches (Appleton et al., 1998) to facilitate this cooperation, with each branch representing a distinct line of development. Sometimes, these branches live for a long time. Other times, they are used for quick changes. Nonetheless, when the time comes, these branches may need to be combined to integrate the different contributions. This integration is called a merge (Mens, 2002). A merge operation can be applied over two or more branches. The most common type of merge occurs over two branches. Thus, we focus on merges of two branches without distinguishing the branches' lifespan. The last versions of each branch are called v_1 and v_2 , where v_1 is the branch the developer is working on and v_2 is the branch merged into v_1 .

When merging branches, conflicts may arise due to modifications to the base code that happen in parallel in the same physical region. In a single merge operation, multiple files or even multiple parts of a file may be marked by the merge algorithm as conflicting. Each conflicting part is called a conflicting chunk and is composed of conflicting lines from v_1 and v_2 . Conflicts must be manually resolved by the developers, who often need to interrupt their work to reason, understand, and possibly interact with others, to resolve the conflict (Costa et al., 2014; Nelson et al., 2019). Indeed, previous work (Kasi and Sarma, 2013; Ghiotto et al., 2020) report that up to 20% of all merges result in textual conflicts.

Resolving merge conflicts is a time-consuming and error-prone task. Thus, any automation of such a task is welcome.

Previous work (Ghiotto et al., 2020) shows that 87% of the merge conflict resolutions employ existing code, pulled from one or both of the versions being merged, without actually modifying any of these lines of code. This may happen by cherry-picking some lines or using all of them. Developers may use lines of code from just one version; they may also pick lines from both, but doing so without interleaving them (essentially, concatenating whatever lines pulled from one version with the lines pulled from the other version); or they may mix lines of code from both versions, interleaving them. Hence, a potential strategy for resolving conflicts is using Search-based Software Engineering (Harman et al., 2012) to explore the combinations of existing lines of code to generate potential candidates, which could be validated using compilation and tests. Given the number of potential rearrangements of the source code, studying whether the search space for solutions can be reduced is beneficial for automated approaches. In addition, finding situations where resolution patterns are more likely to be used can also be beneficial, because candidates using such patterns could then be explored first, and thus save time – be it developer time looking over suggestions, or automated compilation/testing time if a tool is built that takes a suggested set of potential resolutions and tries them in order. This challenging problem was not explored in detail yet in the literature and motivated our paper (we refer the reader to (Ghiotto et al., 2020; Accioly et al., 2018) to obtain a more extensive background on merge conflicts).

In this paper, we analyze a subset of the conflicts analyzed by Ghiotto et al. (2020). Specifically, we analyze 10,177 conflicting chunks from 1,076 open-source Java projects that

were resolved by the developers using a combination of the conflicting lines. Thinking from an automatic tool perspective, finding the correct combination of lines to resolve a conflict is a combinatorial problem, whose size tends to be exponential to the size of the problem. Thus, our main goal is to study whether there are patterns in these resolutions that might make the problem easier. To achieve this, in a previous paper published at SBES 2022 (Campos Junior et al., 2022), we: (i) characterized these conflicts and resolutions, (ii) investigated whether the developers preserve the partial order of the lines from the conflicting chunks in the resolution, (iii) investigated patterns regarding how the developers combine the conflict lines to compose the resolution. In this paper, we extend the work to (iv) investigate how the language constructs present in the conflict influence the choice of resolution, and (v) characterize the language constructs present in the resolutions of the conflicts. Understanding these two new aspects may be useful for devising new automated conflict resolution heuristics based on the language construct patterns involved in the conflict, alleviating the burden of manually resolving such conflicts. For example, we may find that the developer often chooses a specific resolution pattern when a specific language construct is present in a conflict. We may also find that specific language constructs are more often chosen to compose the resolution. Having heuristics based on the findings of these papers can provide developers with suggestions that, when presented at the top of a list of potential resolutions, are more likely to fulfill the desired resolution – thereby making it easier for the developer.

We found that merge conflicts resolved by combining the conflicting lines of code are usually small (median of one chunk and one file per merge, 6 LOC per chunk, and 3 LOC per resolution). In addition, most resolutions (98.6%) do not violate the partial order of each version involved in the conflict. We also found that chunks that contain only *Import* statements are often resolved using lines from only one of the versions, and there is a preference for using lines of v_1 when resolving these conflicts.

Finally, we found that 72.7% of the resolutions do not interleave lines from both versions of the chunk more than once. By leveraging our findings, we demonstrate that the search space for resolutions could be reduced by 94.7% for median-sized conflicts. Thus, search-based techniques could benefit from these results to automatically find the best combination of lines for resolving a conflict.

This paper is an extended version of a conference paper published at SBES 2022 (Campos Junior et al., 2022). The main extensions consist of a study on how the language constructs present in the conflict influence the choice of the resolution and the characterization of the language constructs present in the resolutions of the conflicts.

The rest of this paper is organized as follows. In Section 2, we discuss the materials and methods used throughout the study. Section 3 presents the results of our analysis. In Section 4, the results are discussed. The threats to the validity of our results are discussed in Section 5. Section 6 describes related work. Finally, the concluding remarks are presented in Section 7.

2 Material and Methods

This section describes the research questions of this study, the dataset, and the method used to collect data, and the methods used to analyze the data and answer the research questions.

2.1 Research questions

The five research questions we focus on in this paper are:

- RQ1. What is the size of the conflicts and their resolutions?
- RQ2. Do developers preserve the conflicting chunk’s partial order in the resolutions?
- RQ3. Are there any patterns in the resolutions?
- RQ4. Which language constructs appear in the chunks and lead to specific resolution patterns?
- RQ5. Which language constructs are used in the resolutions, and how do they influence the resolution pattern choice?

The first research question aims at characterizing the conflicting chunks and their resolutions. We organized the analysis along three perspectives: the conflicting merges as a whole, each conflicting file in isolation, and each conflicting chunk independently. Addressing this research question provides insight into how conflicts are distributed across multiple chunks and files, as well as the size of these chunks in terms of LOC. The larger the number of files and chunks, the more difficult it becomes to manually merge them and to develop automated tools. Therefore, it is important to first understand the issue of size.

Regarding the second research question, preserving the chunk’s partial order means that the resolution lines respect the order of the lines from both versions involved in the conflicting chunk. Consider the conflicting chunk extracted from the project *unitycoders/uc_pircbotx*¹ and displayed in Listing 1. The content of v_1 is displayed between the <<<<<< and the ===== markers. Analogously, the content of v_2 is displayed between the ===== and the >>>>>> markers. In this example, the developer resolved the conflict by using lines 7, 10, and 11 from v_1 , and lines 13 and 14 from v_2 . However, the order in which v_1 lines were put in the resolution was changed (lines 10 and 11 before line 7)², as displayed in Listing 2. This is a clear example of where the partial order of the chunk is violated.

Considering that valid programs must adhere to the programming language grammar, it makes sense to hypothesize that violating the partial order of the chunk is not common. Thus, in RQ 2 we investigate whether this is true.

In the third research question, we investigate if there are any patterns regarding how developers combine lines in the conflicting chunk resolutions. For example, what is the source of each line used in the resolution? Is it more frequent to use lines from v_1 or v_2 ? In which situations are lines from each version of the chunk used? Finding such patterns might be useful for developing automated approaches to support the developers.

¹https://github.com/unitycoders/uc_pircbotx

²https://github.com/unitycoders/uc_pircbotx/commit/7a55eddfedd4b39ff8bdaa8ca83773ac04adde89

Listing 1: Chunk from project cucumber/cucumber-jvm.

```

1  }
2
3  @Override
4  <<<<<<<
5  public void sendSuccess() {
6
7  }
8
9  @Override
10 public String getArgument(int id) {
11     return getArgument(id, null);
12     =====
13     public void respondSuccess() {
14
15     >>>>>>>
16     }
17 }

```

Listing 2: Conflict resolution adopted by the developer.

```

1  }
2
3  @Override
4  public String getArgument(int id) {
5      return getArgument(id, null);
6  }
7
8  public void respondSuccess() {
9
10 }
11 }

```

The fourth research question investigates the language constructs present in the chunks that lead to the resolution patterns found in the third research question. We do not analyze how these language constructs are used, i.e., their structure, since this can lead to progressively expensive resolution techniques, as exemplified by structured merge techniques (Seibt et al., 2021). Nonetheless, we characterize each conflicting chunk according to the outermost language constructs it conveys in lexicographic order, following the approach used by Ghiotto et al. (2020). For instance, we classify a conflicting chunk where v_1 calls a method, and v_2 has an if statement as “if statement, method invocation”. This question aims to identify patterns in the chunks that frequently lead to specific resolution strategies. These findings could be valuable for automated conflict resolution tools. More details on the analysis of language constructs are discussed in Section 2.5.

Finally, we perform another frequent patterns analysis in the fifth research question. While in RQ4 we look at the language constructs in the chunk, in this case, we analyze the language constructs present in the resolution adopted by the developers. The goal is to identify situations that lead to specific resolution patterns, which can be useful for devising new automated resolution heuristics. We follow a similar approach to the fourth question by classifying the resolutions according to the language constructs they convey.

2.2 Data collection

Ghiotto et al. (2020) collected and labeled the resolution strategy used in 175,805 conflicting chunks from 25,328 merges that occurred in 2,731 open-source Java projects. They found that 50% of the chunks were resolved by adopting the whole

Table 1. Statistics about the projects and the dataset.

Characteristic	Mean (std. deviation)	Median
Number of commits	3,125 ± 10,710	932
Number of merges	555 ± 5,033	70
Number of failed merges	38 ± 125	12
Number of developers	20 ± 31	9
Dataset chunks	9 ± 26	2
Dataset merges	5 ± 12	2

of v_1 , 25% by adopting the whole of v_2 , 3% by concatenating one version after the other (v_1v_2 or v_2v_1), 9% by combining conflicting lines, and 13% by adding new code. In this paper, we are interested in the chunks that were resolved with combination, which excludes those that are resolved with v_1 , v_2 , v_1v_2 , v_2v_1 , and new code. Starting from the 15,571 chunks resolved with combination from Ghiotto et al. (2020)’s dataset, some chunks needed to be discarded. A total of 1,998 chunks were discarded because we needed to query GitHub to get additional data about them, but they were not available anymore. Moreover, 264 chunks were discarded due to inconsistent conflict markers, e.g., cases of files with many chunks causing the markers to overlap each other. We also discarded 2,583 additional conflicting chunks because the merge involved additional changes beyond just the conflicting lines, making it not possible to isolate the resolution from the rest of the file. Finally, we discarded 549 chunks because they belong to projects that are implicit forks of other projects in the dataset, representing duplicated data. Thus, the dataset used in this paper comprises 10,177 conflicting chunks that occurred in 5,346 merges from 1,076 open-source GitHub Java projects. Table 1 lists some statistics about the projects in the dataset and the dataset itself. For example, the projects have, on average, 3,125 commits and 20 developers who participated in the project over time.

For each conflicting chunk, we collected its size, as represented by the number of lines in v_1 and v_2 ; the resolution size, represented by its number of lines; and the contents of both the conflicting chunks and the merge resolution. We also collected the language constructs used in the conflicting chunk and in the merge resolution.

To locate the conflict resolution content, Ghiotto et al. (2020) perform a diff between the conflicting file and the resolution file. Using the line indexes of the prefix and suffix lines along with the information provided by the diff about added and removed lines, they calculate the new line indexes where the resolution is placed. Therefore, even if the content of the prefixes and suffixes are not unique within a file, the identification of the resolution region remains accurate because the method relies on line indexes rather than the content itself. This ensures a robust and precise location of the resolution content. For instance, if a line is in the tenth line and three lines are removed before it, the resulting line is the seventh.

In RQ1, we investigate characteristics such as the number of chunks per conflicting merge and conflicting file. Even though our selection focused on the chunks that are resolved

with combination, our analysis also looks at the overall complexity of that merge. Thus, some merges may contain not only chunks resolved with combination, but may also include chunks resolved with other strategies. To answer RQ1, where appropriate, these additional chunks were included to show the full complexity of the merge.

2.3 Partial order analysis

Verifying whether the resolution of a conflicting chunk preserves the chunk's partial order is not a trivial task to perform manually, especially for bigger conflicts. Thus, we developed Algorithm 1 to perform this task automatically, when given a list with v_1 's lines, a list with v_2 's lines, and a list with the resolution lines (m).

The algorithm uses dynamic programming to find whether the resolution (m) can be obtained by combining the content of v_1 and v_2 in a partially sorted manner. A matrix is constructed, where each cell represents the length of the longest matching subsequence of lines found thus far. The algorithm explores all possible partial sorting scenarios by iteratively considering potential extensions of the current subsequence based on the matches from v_1 and v_2 . If the length of any subsequence equals the length of the resolution (m), the algorithm concludes that the pattern can be found in a partially sorted manner and returns True. Otherwise, it returns False.

Algorithm 1 Partial Order checking

```

1: function PartialSorting( $v_1, v_2, m$ )
2:   Append None to the end of sequences  $v_1$  and  $v_2$ 
3:   Create a 2D matrix matrix with dimensions
     ( $\text{len}(v_1), \text{len}(v_2)$ )
4:   for  $i = 0$  to  $\text{len}(v_1) - 1$  do
5:     for  $j = 0$  to  $\text{len}(v_2) - 1$  do
6:        $up \leftarrow left \leftarrow 0$ 
7:       if  $i > 0$  then
8:          $up \leftarrow \text{matrix}[i - 1][j]$ 
9:         if  $m[up] == v_1[i - 1]$  then
10:           $up \leftarrow up + 1$ 
11:        end if
12:      end if
13:      if  $j > 0$  then
14:         $left \leftarrow \text{matrix}[i][j - 1]$ 
15:        if  $m[left] == v_2[j - 1]$  then
16:           $left \leftarrow left + 1$ 
17:        end if
18:      end if
19:       $\text{matrix}[i][j] \leftarrow \max(up, left)$ 
20:      if  $\text{matrix}[i][j] == \text{len}(m)$  then
21:        return True
22:      end if
23:    end for
24:  end for
25:  return False
26: end function

```

After executing the partial order algorithm on all chunks in the dataset, we randomly sampled 30 chunks (21%) that *violate* the partial order for performing manual analysis to further understand the characteristics of these chunks. Our

intuition is that even though for some chunks the partial order was violated, the order of the lines does not matter when considering the programming language syntax. For example, if every line of a chunk resolution is an *import* statement without mixing wildcard imports with fully-named imports, then the order does not matter since these types of statements are independent of the others. On the other hand, for other conflicts, like the one displayed in Listings 1 and 2, using a different line ordering would break the syntax of the source code.

2.4 Resolution pattern analysis

To answer RQ3, we analyze the composition of the conflicting chunks' resolution regarding the source of each line (v_1 or v_2). For each chunk resolution, we count how many lines originate from v_1 and v_2 . If a line appears in both v_1 and v_2 , then the line counts as 0.5 for each side. Thus, we normalize the v_1 and v_2 line count, and their sum will always be the same as the number of lines in the resolution. We use this count to calculate the normalized proportion of v_1 and v_2 in the resolution.

We also use the source of each line to derive patterns in the resolution. For example, consider $v_1 = (A, B, C)$ and $v_2 = (D, E)$. The resolution $R_1 = (B, D)$ has the pattern $v_1 v_2$, because its first element comes from v_1 and the second from v_2 . To simplify the amount of possible patterns, we group consecutive lines from the same source into a single element. Following the previous example, the resolution $R_2 = (A, C, D, E)$ would also be represented by the pattern $v_1 v_2$, instead of $v_1 v_1 v_2 v_2$. This strategy of grouping consecutive elements was also used by Brindescu et al. (2020b) to identify interleaved commit patterns in different branches. Using this strategy, we classify the chunks' resolutions into pattern groups according to their composition: using only lines from v_1 or v_2 , using lines from v_1 followed by lines from v_2 , or vice versa, and using more intricate patterns that interleave lines from v_1 and v_2 more than once.

2.5 Language constructs analysis

To answer RQ4 and RQ5, we classified the conflicting chunks and resolution content using the outermost language constructs present in each one. The outermost language constructs are those that reside at the higher level of the code snippet's Abstract Syntactic Tree (AST). This approach is based on the methodology adopted by Ghiotto et al. (2020), which focuses on identifying the governing concerns in the code fragment by using these outermost constructs. Working at this level creates a balance in identifying common patterns with not having too many such "patterns" that, in truth, represent single instances; that is, as multiple small changes over less frequent fine-grained constructs are represented by their governing coarse-grained constructs, these fewer coarse-grained constructs abstract the specific details of the changes to focus on its construct's general goal, allowing better grouping and comparison with other changes.

For example, given a piece of source code, such as the one displayed in Listing 2, we can see that it contains the

outermost language constructs Annotation and Method declaration. Thus, when we consider this source code according to its language constructs, we classify it using such language constructs in lexicographical order and using commas as separators: Annotation, method declaration. Throughout this paper, we use this definition to classify chunk and resolution types.

The conflicting chunks' language constructs are available in Ghiotto et al. (2020)'s dataset, allowing us to answer RQ4 using that data. However, the language constructs present in each conflicting chunk' resolution required for answering RQ5 are unavailable. Thus, considering that such extraction requires parsing the source code, and we are interested only in the resolution fragment that corresponds to the conflicting chunk, we opted for manual extraction. Since the manual analysis of the resolution for the entire dataset was not feasible, we selected a sample of the initial 10,177 chunks.

To determine a sample size that is representative of the dataset, we used Cochran's sample size formula (Israel, 1992) using a maximum variability of 0.5, a confidence level of 99%, and an error level of 10%. The calculated sample size is 166 conflicting chunks. Thus, we randomly selected 166 conflicting chunks from the dataset to compose the sample to answer RQ5. Two authors were responsible for the data extraction. A random subsample of 16 (representing 10% of the sample) chunks was selected to be analyzed by both authors independently. They reached a consensus of 100% of their extracted data. Thus, one of the authors manually extracted the language constructs for the remaining 150 conflicting chunks. As a sanity check of the sample's representativeness, we calculated the standard error (Altman and Bland, 2005) for the distributions of each collected metric in the dataset (chunk and resolution sizes and derived proportions). Then, we checked whether these metrics' mean in the sample is within the range of the dataset mean, considering the standard error. We found that all metrics are in the range except one (percentage of intersection between v_1 and v_2 content). Nonetheless, it is very close (difference of 0.001392).

In this study, we extract association rules from our dataset to answer RQ4 and RQ5. Association rules extraction is a data mining technique that allows the discovery of frequent patterns in a dataset of the type $X \rightarrow Y$, where X is called the *antecedent*, and Y is called the *consequent*. This pattern means that when X happens, Y also happens. Three metrics help to understand this concept. The first one is *support*, which indicates the frequency of the rule in the dataset. Second, *confidence* measures the percentage of all transactions satisfying X that also satisfy Y . Finally, the *lift* of a rule can be calculated by $\text{confidence}(X \rightarrow Y) / \text{support}(Y)$. A *lift* value greater than one means that there is an increased chance of Y happening when X happens. For example, a *lift* of 2.3 means a 130% increase in the chances of Y happening when X happens. A *lift* value smaller than one means that the occurrence of X decreases the chance of Y happening. In our context, the *antecedent* contains the language constructs present in the conflicting chunks for RQ4 and in the resolution for RQ5. The *consequent* contains the resolution patterns from RQ3.

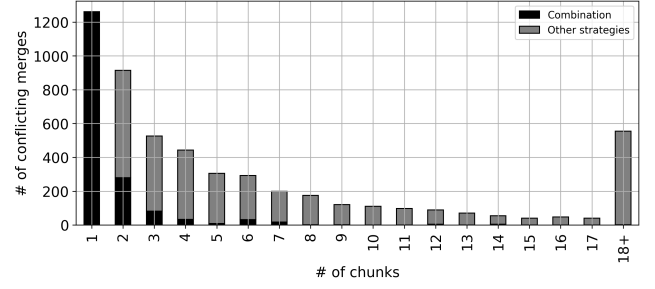


Figure 1. Number of conflict chunks per conflict merge.

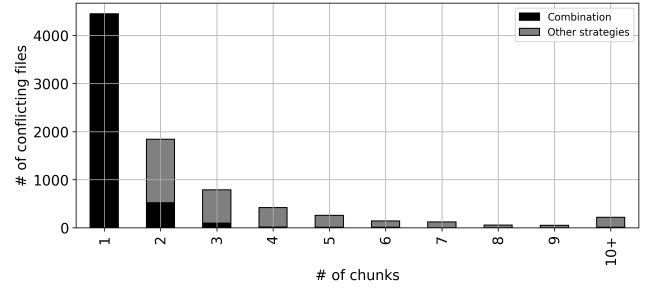


Figure 2. Number of conflicting chunks per conflicting file.

3 Results

This section presents and discusses the results obtained for each research question.

3.1 RQ1. What is the size of the conflicts and their resolutions?

As discussed in Section 2.2, in some merges only the combination strategy was used to resolve the conflicts, but many have 1 or 2 chunks resolved with combination and other chunks resolved with different strategies. Thus, Figure 1 includes data of 52,083 chunks from the 5,346 merges of our dataset, of which 10,177 chunks are resolved using combination. Chunks that were resolved with combination are represented with a darker color, whereas chunks that were resolved with another resolution strategy are represented in a lighter color.

Most conflicting merges with at least one conflicting chunk resolved with combination have a small number of conflicting chunks. In fact, 75% of all merges have 8 or fewer chunks. The median number of chunks per merge is 3. From a total of 5,346 conflicting merges, 1,726 (32.3%) were resolved entirely with combination. The proportion of chunks that were resolved with combination decreases as the number of chunks increases. If we consider only the 1,726 conflicting merges where all chunks were resolved with combination, 73% have one chunk, and 89.2% of them have up to two chunks. The median of chunks per merge, in this case, is 1.

Another perspective that may overwhelm the developer when confronting conflicting merges is the number of chunks per conflicting file. Considering that the context of a file may be more restricted than the context of an entire merge, it seems reasonable that chunks within the same file are dealt with together. Figure 2 shows the distribution of the number of conflicting chunks per conflicting file.

We found that the analyzed merges have a median of a sin-

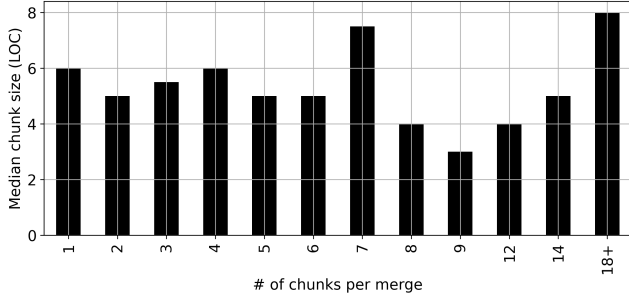


Figure 3. Median chunk size for conflicting merges in relation to the number of chunks.

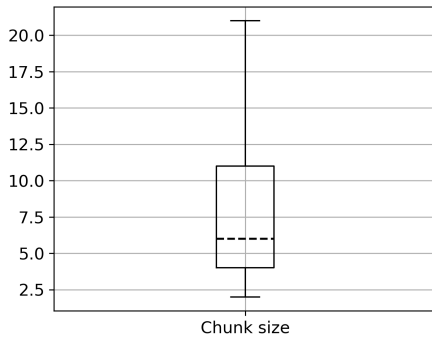


Figure 4. Distribution of the conflicting chunks' size (LOC). Outliers excluded.

gle file with conflicts (min 1, max 129). Moreover, 92.8% of the files have 5 or fewer conflicting chunks. From a total of 8,347 conflicting files, 5,124 (61.4%) were resolved entirely with combination. Similar to Figure 1, the proportion of chunks resolved with combination decreases as the number of chunks per file increases. In fact, if we consider only the 5,124 files resolved entirely with combination, 97% of them have one or two chunks and the median of chunks is one.

Considering the 5,346 merge conflicts with at least one conflicting chunk resolved with combination, we found that 42% have only one file and one chunk.

In addition to the number of chunks per merge and per file, Figure 3 shows the median chunk size, in terms of LOC, for merges with a varying number of chunks. For example, merges with one conflicting chunk have a median of 6 LOC. In comparison, merges with two chunks have a median of 5 LOC. No particular trend was observed in this analysis.

We also analyzed the size of conflicting chunks individually. Figure 4 shows a boxplot representing the distribution of the size of conflicting chunks. The median size is 6 LOC, with a minimum of 2 and a maximum of 2,545. However, 75% of the chunks have up to 11 LOC.

Finally, Figure 5 shows the distribution for the size of both v_1 and v_2 sides of the conflicting chunks individually and the size of the resolution adopted by the developers. The median size of v_1 is 2 LOC and of v_2 is 3 LOC. The median size of the resolution is also 3 LOC. Compared to the median size of the chunks (6 LOC), in the median case, half of the conflicting chunk lines are discarded in the resolution.

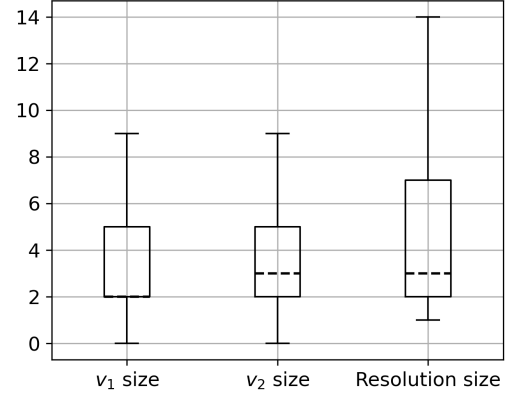


Figure 5. Boxplots for the distribution of the size (LOC) of v_1 , v_2 , and the chunk resolution, respectively. Outliers excluded.

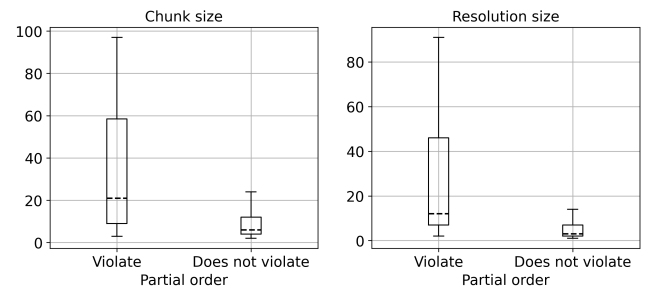


Figure 6. Chunk and resolution size for chunks that violate and do not violate the partial order. Outliers excluded.

Finding 1: In total, 42% of the merge conflicts with at least one conflicting chunk resolved by combining existing lines have only one conflicting chunk and one conflicting file. The conflicting chunks have a median of 6 LOC, and the resolutions have a median of 3 LOC. Compared to the general case, which includes more resolution strategies, the merge conflicts in our dataset have more but smaller conflicting chunks. The implication of this finding is related to the feasibility of resolving this type of conflict automatically. Toward this objective, a small number of small chunks is good, considering that the combination problem is exponential to the number of lines in v_1 and v_2 .

3.2 RQ2. Do developers preserve the conflicting chunk's partial order in the resolutions?

From the 10,177 analyzed chunks, we found that only 142 (1.40%) violate the chunk's partial order in their resolution. Chunks that violate the partial order have a median of 21 LOC and their resolutions have a median of 12 LOC. In comparison, chunks that do not violate the partial order have a median of 6 LOC and their resolutions have a median of 3 LOC. Thus, chunks where the resolution violates the partial order are usually bigger than chunks that do not violate the partial order. Figure 6 shows the size distribution of chunks and their resolutions for both groups of chunks.

To investigate the relationship between the chunk size and the phenomena of violating the partial order, Figure 7 shows the proportion of chunks that violate and do not violate the partial order as the chunk size grows. For example, 60.9% of the chunks having more than 500 LOC violate the partial

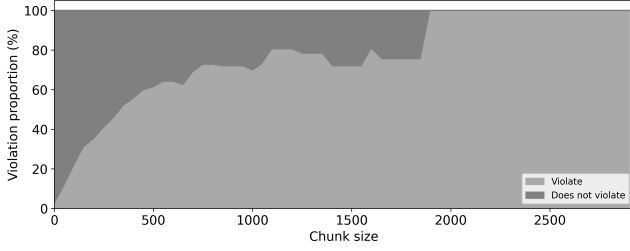


Figure 7. Proportion of chunks that violate and do not violate the partial order as the chunk size grows.

order.

Finding 1 suggests that the chunks are usually small. The previous paragraphs suggest that most of the small chunk resolutions do not violate the partial order of the chunk. Thus, it seems reasonable to assume that an automatic algorithm to resolve conflicts could ignore cases that violate the partial order and reduce the search space of candidate resolutions for small chunks. This strategy could go even further, since we also observed that the bigger the chunk, the greater the chance of violating the partial order. Thus, the automatic algorithm could use this information to tune itself by ignoring the partial order depending on the chunk size.

As discussed in Section 2.3, we manually analyzed a sample of 30 (21%) chunks where the resolution violates the partial order. The goal was to understand if the order of the resolution lines would matter considering the programming language syntax. We found that in 19 of the 30 analyzed chunks (63.3%) the order of the lines is relevant due to syntax restrictions and forcing partial ordering would hinder finding a correct resolution. The median size of the chunk, in this case, is 19 LOC. On the other hand, for the remaining 11 chunks, although their resolutions violate the partial order of the chunk, their order does not matter within the resolution scope, i.e., considering just the resolution lines and not the entire file. Moreover, 9 of them include only *import* statements and 2 include variable declarations that are not used within the resolution scope, thus any shuffle of the existing lines would yield a valid resolution. The median size of the chunk for this case is 6 LOC.

Finding 2: Only 1.4% of the analyzed chunks violate the partial order. The chance of violating the partial order increases as the chunk's size increases. We also found that despite violating the partial order in the resolution, the order of the lines does not matter in the resolution scope in 36.7% of a manually analyzed sample of chunks that violate the partial order. Thus, an automatic resolution algorithm that enforces partial ordering to restrict the search space would have failed in only 90 (0.88%) of the 10,177 analyzed chunks, which have resolutions that violate partial ordering (1.4%) and are not tolerant to ordering changes (63.3%).

3.3 RQ3. Are there any patterns in the resolutions?

Figure 8 shows a histogram with the normalized percentage (explained in Section 2.4) of resolution lines coming from v_1 and v_2 . The y-axis shows different ranges of the normalized v_1 (left) and v_2 (right) percentages considered and the x-axis

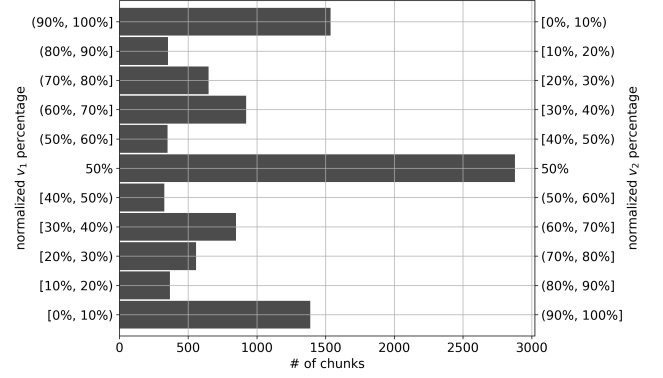


Figure 8. Histogram showing the number of chunks for the normalized percentages of lines in the resolution that come from v_1 (left y-axis) and v_2 (right y-axis).

shows the number of chunks for each of these ranges. For example, the first bar (top) of the histogram shows that a little more than 1,500 chunks have resolutions with from 90% to 100% of their lines coming from v_1 and from 0% to 10% of lines coming from v_2 . The middle bar of the histogram shows that almost 3,000 chunks have resolutions with 50% of their lines coming from v_1 and 50% from v_2 .

Note that since the number of lines in the resolution is discrete, some percentages are only possible for larger resolutions. For example, a resolution with 10% of its lines coming from v_1 or v_2 must have at least 10 lines in total, which would represent one line coming from v_1 or v_2 . In comparison, resolutions with the median number of lines (3 LOC) can have one, two, or three lines coming from either v_1 or v_2 . In each of these cases, they would have 0%/100%, 33.3%/66.7%, 66.7%/33.3%, 100%/0% of v_1/v_2 , respectively. Thus, median resolutions could be placed in four different histogram bars, depending on the distribution of their composition.

We observed that the highest concentration of chunks in Figure 8 occurs in the middle of the histogram, which represents chunks with 50% of the resolution lines coming from v_1 and 50% coming from v_2 . Only chunks with an even number of lines in the resolution may have this composition. On average, conflicting chunk resolutions have 51% of their lines coming from v_1 and 49% from v_2 .

The number of chunks in the extremes of the histogram is also noteworthy. This means that a high percentage (28.7%) of chunks have resolutions with a low percentage of its lines coming from either v_1 or v_2 . Moreover, we observed that 25.8% of all chunks have either only lines from v_1 (100% v_1 and 0% v_2) or only lines from v_2 (0% v_1 and 100% v_2) in their resolution. Thus, the high concentration of chunks in the two extremes is mostly explained by this case.

From the observations above, we classify the resolutions into four groups. The groups v_1 *only* and v_2 *only* include chunks where the resolution contains only lines from v_1 or v_2 , respectively. The groups v_1v_2 and v_2v_1 include chunks where the developer uses consecutive lines from v_1 or v_2 , and then uses lines from the other version, without going back to the previous version. That is, the lines from different versions involved in the chunk are not interleaved more than once.

We found that 1,367 (13.4%) of the chunks use the v_1 *only* pattern and 1,260 (12.4%) use the v_2 *only*. This means that 25.8% of the chunks use a subset of lines from one version of the conflict. For the cases where the lines come from both ver-

sions involved in the conflict, we found that 2,677 (26.3%) of the chunks use the v_1v_2 pattern. In comparison, 2,093 (20.6%) use the v_2v_1 pattern. Thus, 46.9% of the analyzed conflicting chunks do not interleave lines from different versions involved in the chunk more than once. These four patterns cover 7,397 (72.7%) of all chunks in our dataset. The remaining 27.3% use intricate patterns such as multiple alternations between lines from v_1 and v_2 .

The average size of v_2 for chunks resolved with the v_1 **only** pattern is 1 (min 0, max 41). In comparison, the average size of v_1 for chunks resolved with the v_2 **only** pattern is 0.77 (min 0, max 34). This indicates that the developer often uses only lines from one version involved in the conflict when the other version has few LOC.

Figure 9 shows the proportion of resolution patterns used in the chunks for different chunk size deltas ($v_2 - v_1$). A negative delta means that v_1 is bigger than v_2 . Figure 9 also shows the relative frequency of each chunk size delta in the dataset at the top of the Figure. Note that, since some chunk size delta values are not frequent in the dataset, values not in the figure's center are represented by an interval of chunk size deltas. For example, the bar at the chunk size delta value of -260 encompasses chunk size deltas that are > -310 and ≤ -260 .

Figure 9 shows that most chunks are concentrated with low chunk size delta values. It also shows that none of the resolution patterns dominate the most frequent delta values. However, when $v_1 > v_2$ the v_2 **only** strategy is used in only 0.4% of the chunks. This can also be observed when $v_2 > v_1$, with the v_1 **only** strategy being used in only 1.4% of the chunks. This reinforces that the developer often uses only lines from one version involved in the conflict when the other version is small in size. To complement this analysis, we also calculated the correlation between the v_2 percentage in the resolution and the chunk size delta. Since the data distribution is not normal, we used Spearman's Rank Correlation Coefficient (Hinkle et al., 2003). We found a moderate positive ($\rho = 0.674$) correlation between the two variables. This means that as the chunk size delta increases (i.e. $v_2 > v_1$), the v_2 percentage in the resolution also increases. Analogously, the v_1 percentage decreases ($\rho = -0.666$).

To finish this analysis, we can also observe in Figure 9 that when the chunk size delta is 0, that is, v_1 and v_2 have the same size, the developer rarely uses only lines from one of the versions. When this happens, in 41.5% of the chunks the v_1v_2 pattern is used. The v_2v_1 pattern is used in 35.2% of the chunks, and 20.3% of the chunks use other patterns. This observation may be useful for devising automatic approaches that use the chunk's characteristics, such as v_1 and v_2 size, to prioritize the type of resolutions to search for.

Table 2. Top-10 language constructs according to the number of chunks where they are used.

Language construct	Number of chunks
Method invocation	5,005
Variable	2,765
Import	2,735
Comment	2,158
Attribute	2,029
If statement	1,813
Method signature	1,570
Method declaration	917
Annotation	846
Return statement	543

Finding 3: On average, half of the resolution contents come from v_1 and the other half from v_2 . We identified four resolution patterns that describe 72.7% of the cases: using only lines from v_1 (13.4%) or v_2 (12.4%), and using lines from either v_1 or v_2 followed by lines from the other version involved in the conflict (26.3% for v_1v_2 and 20.6% for v_2v_1). Finally, we also found that when both versions involved in the conflict have the same size, the developer mostly uses v_1v_2 (41.5%) or v_2v_1 (35.2%). We found a moderate correlation ($\rho = 0.674$) suggesting that the bigger a version involved in the conflict is, the bigger the chance of the developer choosing lines from that version to compose the resolution. These results may be useful for devising heuristics to resolve conflicts. For example, the heuristic can: (i) prioritize lines from the bigger version involved in the conflict, (ii) prioritize the v_1v_2 and v_2v_1 patterns when both versions involved in the conflict have the same size, and (iii) exclude resolutions that interleave lines from both versions involved in the conflict more than once and still find the correct resolution for 72.7% of them.

3.4 RQ4. Which language constructs of the chunks lead to specific resolution patterns?

This research question investigates the relationship between the language constructs within the conflicting chunks and the resolution patterns. Table 2 shows the top 10 most frequent language constructs according to the number of chunks where they are used.

According to Table 2, *Method invocation* is the most frequent language construct in the chunks of our dataset, being used almost two times more than the second and third most frequent (*Variable* and *Import*).

We also analyzed the combination of language constructs present in the conflicting chunks, which we call chunk type. Table 3 displays the top-5 chunk types in the dataset and their frequency. We found that 25.5% of all chunks in our dataset are composed only of *Import* statements. They have a median size of 2 LOC for each chunk and the resolution. The remaining chunks in the dataset have a diverse range of language

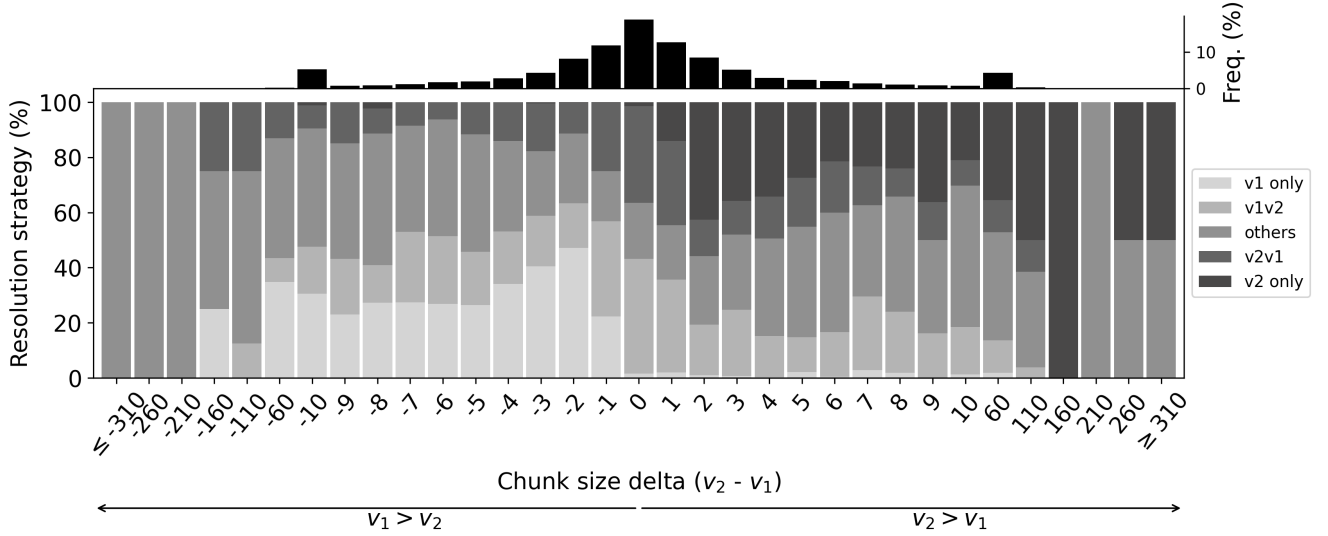


Figure 9. Frequency of each chunk size delta (top) and proportion of resolution patterns used for each chunk size delta ($v_2 - v_1$).

Table 3. Top-5 types of conflicting chunks according to the language constructs they present.

Chunk language constructs	Frequency
Import	25.5%
Method invocation, variable	6.6%
Method invocation	5.3%
Attribute, method invocation	3.7%
Attribute	3.1%

constructs. For instance, 6.6% are classified as "method invocation, variable"; 5.3% as "method invocation"; 3.7% as "attribute, method invocation"; and 3.1% as "attribute".

Import only chunks are the most common for all four main resolution patterns discussed in the previous research questions. Such pattern happens in 45.3% of the chunks resolved with the v_1 *only* pattern, 47.7% for the v_2 *only* pattern, 20.9% for v_1v_2 , and 14.4% for the v_2v_1 pattern. This is not surprising, given that *Import*-only chunks are the most common overall in the dataset. Similar distributions occur for the remaining top-5 chunk types of the overall dataset, except for the "method invocation, method signature" type of chunk, which appears as the fifth most frequent (3.9%) for chunks resolved with v_2v_1 .

We calculated the lift of each chunk type (according to its language constructs) that leads to specific resolution patterns. Table 4 shows the five biggest lift values found. The biggest lift obtained is for cases where the chunk has the language constructs *Annotation* and *Method signature*. In this case, the chance of using the resolution pattern v_2v_1 is increased by 251% (lift = 3.51). This pattern occurs 75 times in the dataset. We looked at some examples of chunks for this rule and found that usually, v_1 has a *Method signature*, and v_2 has an *Annotation* and a *Method signature*. In these cases, the resolution usually has the *Annotation* and one of the *Method signatures*. The average v_1 size is 1.04 LOC and for v_2 is 2 LOC. The average resolution size is 2.08 LOC. It should be noted that this rule happens predominantly (48 of 75 times) in a single project (*freenet/fred*). The remaining occur-

Table 4. Association rules between chunk types (language construct) and resolution patterns.

Association rule	Sup.	Lift
Annotation, method signature $\rightarrow v_2v_1$	75	3.51
Method invocation, method signature $\rightarrow v_2v_1$	82	3.30
Import $\rightarrow v_2$ <i>only</i>	602	1.87
Import $\rightarrow v_1$ <i>only</i>	617	1.77
Method invocation, variable $\rightarrow v_1v_2$	281	1.58

rences were observed in 13 other projects. Listing 3 shows an example of this case from project the *freenet/fred* and Listing 4 shows the adopted resolution.

Listing 3: Chunk from project *freenet/fred* showing one case of the association rule Annotation, Method signature $\rightarrow v_2v_1$

```

1      super.unregister(container,
2          ↪ context);
3      }
4      <<<<<<<<
5      public synchronized boolean isCancelled(
6          ↪ ObjectContainer container) {
7      =====
8      @Override
9      public synchronized boolean isCancelled()
10         ↪ {
11      >>>>>>>>
12         return cancelled;
13     }

```

Listing 4: Resolution of the chunk displayed in Listing 3.

```

1      super.unregister(container,
2          ↪ context);
3      }
4      @Override
5      public synchronized boolean isCancelled(
6          ↪ ObjectContainer container) {
7         return cancelled;
8     }

```

The second biggest lift was obtained for chunks that contain *Method invocation* and *Method signature* constructs. When these types of chunks occur, there is a 230% increased chance of the resolution adopted by the developer following the v_2v_1 pattern. This occurs 82 times in the dataset. However, the project (xetorthio/jedis) is responsible for 62 of the 82 occurrences. The other occurrences are spread across 16 other projects from the dataset. The average chunk size when this rule happens is 4.22 LOC and the average resolution size is 2.16 LOC. Listing 5 shows an example of this case and Listing 6 shows its resolution.

Listing 5: Chunk from project xetorthio/jedis showing one case of the association rule Method invocation, Method signature $\rightarrow v_2v_1$.

```

1      }
2
3      @Override
4      <<<<<<<
5      public void assertOnRightPage() {
6          assertThat(queryField, is(notNullValue
7              ↪ ());
8          assertThat(searchButton, is(
9              ↪ notNullValue()));
10
11      =====
12      public void assertLocation() {
13      >>>>>>>
14
15      }
16
17      public SearchResultsPage searchFor(
18          ↪ String keyword) {

```

Listing 6: Resolution of the chunk displayed in Listing 5.

```

1      }
2
3      @Override
4      public void assertLocation() {
5          assertThat(queryField, is(
6              ↪ notNullValue()));
7          assertThat(searchButton, is(
8              ↪ notNullValue()));
9
10     }
11
12     public SearchResultsPage searchFor(
13         ↪ String keyword) {

```

The next most frequent patterns happen when the conflicting chunk has only *Import* statements. In these cases, the chances of using only v_1 or v_2 lines in the resolution are increased by 87% and 77%, respectively. On the other hand, the chances of using the patterns v_1v_2 and v_2v_1 decrease by 18% and 43%, respectively. This result is useful for automated conflict resolution tool builders. They can, for example, detect whether a conflict includes only import statements and prioritize their search for a solution using only lines from one of the versions involved in the conflict chunk.

We conducted an analysis to understand the patterns observed in the resolution of chunks containing only Imports. In these cases, most resolutions use the v_1 *only*, v_2 *only*, and v_1v_2 patterns, each accounting for approximately 20% of the instances. The remaining resolutions primarily use the v_2v_1 pattern (11%) and other more complex mixed patterns. Although the v_1v_2 pattern is among the most commonly used for resolving this type of conflict, its likelihood of being employed decreases by 18% (lift 0.82) when dealing with import-only chunks.

Table 5. Association rules between chunk types (language construct) and resolution patterns with lift smaller than one.

Association rule	Sup.	Lift
Import $\rightarrow v_2v_1$	303	0.57
Import $\rightarrow v_1v_2$	560	0.82
Attribute $\rightarrow v_2v_1$	56	0.84
Method invocation $\rightarrow v_1$ <i>only</i>	65	0.89
If statement, Method invocation $\rightarrow v_1v_2$	63	0.94

By further analyzing these chunks, we found a distinct pattern. When the resolution pattern is v_1 *only*, the conflicting lines were removed in v_2 in 70% of the cases. In comparison, when the resolution pattern is v_2 *only*, the conflicting lines were removed in v_1 79% of the time. This suggests a clear trend: if a chunk contains only imports and v_1 has removed the conflicting lines, it is resolved with v_2 *only*, and vice versa. However, for the v_1v_2 and v_2v_1 resolution patterns, such a pattern does not hold. In these cases, the sizes of v_1 and v_2 are similar, indicating that the resolution is more likely to involve a combination of both sides' imports. Therefore, the observed decrease in the usage of v_1v_2 and v_2v_1 patterns for import-only chunks may be attributed to the tendency of these chunks to have one side (either v_1 or v_2) removing the conflicting lines. This specific pattern within the data set suggests that the simpler resolutions (v_1 *only* and v_2 *only*) are more appropriate and preferred in these scenarios.

Other cases of negative lifts (smaller than 1) are displayed in Table 5. One case of a negative lift occurs with the *Attribute* construct. When the resolution contains only *Attribute* constructs, the chance of using the v_2v_1 pattern is decreased by 16%. This rule occurs 56 times in the dataset, spread across 44 different projects. The average chunk size for this case is 5.48 LOC, and the average resolution size is 3.27 LOC.

Previous studies indicate that unstructured merge approaches are problematic when dealing with conflicts that involve only imports because, in most cases, imports are not dependent on order. However, one should not completely ignore ordering in these cases, as this may lead to the false notion that there are no conflicts when there actually are. For example, the jFSTMerge tool (Cavalcanti et al., 2017) addresses this specific case. We take a different approach by analyzing situations where the conflict consists solely of imports and observing developers' resolution patterns in these cases. Therefore, while Cavalcanti et al. (2017) propose a way for the jFSTMerge tool not to report conflicts in situations with imports, we provide evidence of how these cases are resolved by developers, hoping that this information can be used in conflict resolution heuristics based on the organization of conflicting lines.

Finding 4: Method invocation is the most frequent language construct in chunks overall, although *Import* is the most common chunk type. We found some situations that increase the chance of a developer choosing specific resolution patterns, but they occur mostly in a few specific projects. The most prominent finding is for chunks that contain only *Imports*. The chance of resolving these chunks with the v_1v_2 or v_2v_1 pattern decreases by 18% and 43%, respectively. In addition, for these chunks, when v_1 is empty, it is usually resolved with v_2 *only* and vice versa. Automated conflict resolution tools can use this finding for prioritizing the resolution search using only lines from one version involved in the conflict.

3.5 RQ5. Which language constructs are used in the resolutions, and how do they influence the resolution pattern choice?

As previously discussed, we manually analyzed a sample of 166 conflicting chunk resolutions to answer this research question. The distribution of resolution patterns in these chunks is displayed in Table 6. This sample's most common resolution pattern is *Others* (30%). *Others* groups all resolution patterns that intertwine lines from both versions involved in the conflict more than once in the resolution. The second most common resolution pattern is v_2v_1 , occurring in 26% of the sample, followed by v_1v_2 in 23% of the resolutions. The distribution of resolution patterns in the sample is similar to the distribution in the dataset, reinforcing its representativeness.

Table 6. Distribution of resolution patterns in the chunks analyzed for RQ5.

Resolution pattern	Occurrences	(%)
v_1 <i>only</i>	20	12%
v_2 <i>only</i>	15	9%
v_1v_2	43	26%
v_2v_1	39	23%
Others	49	30%

Table 7 shows the top-5 types of resolution according to the language construct they present. We found that 13.2% of the resolutions in the sample are composed only of *Import* statements. They have a median size of 2 LOC. The second most frequent type of resolution is “method invocation, variable”, happening in 4.2% of the resolutions in the sample. “comment, method declaration”, “comment, method invocation, variable”, and “comment, import” occur in 3% of the resolutions.

Analyzing each resolution pattern, we found that, for pattern v_1 *only*, the most common resolution contains only *Import* statements, occurring in 25% of the resolutions. For the v_2 *only* resolution pattern, “attribute, comment” (13%) is the most common combination of language constructs. For the resolution pattern v_1v_2 , the most frequent combination of language constructs is *Import* (20%), followed by “method

Table 7. Top-5 resolution types according to the language constructs they present.

Chunk language constructs	Frequency
Import	13.2%
Method invocation, variable	4.2%
Comment, method declaration	3.0%
Comment, method declaration, variable	3.0%
Comment, import	3.0%

Table 8. Association rules between resolutions' language constructs and resolution patterns.

Association rule	Sup.	Lift
Annotation, method invocation, method signature, variable $\rightarrow v_2v_1$	3	3.19
Comment, Import $\rightarrow v_2v_1$	3	2.55
Comment, Method invocation, Variable $\rightarrow v_1v_2$	3	2.32
Comment, Method declaration $\rightarrow v_2v_1$	3	2.13
Import $\rightarrow v_1$ <i>only</i>	5	1.89

invocation, variable” (6.9%), “comment, method invocation, variable” (6.9%), “annotation, method declaration” (4.6%), and “method declaration” (4.6%). Finally, for the v_2v_1 resolution pattern, “comment, method declaration”, “comment, import”, and “annotation, method invocation, method signature, variable” occur in 7.7% of the resolutions each.

According to Table 8, the biggest lift obtained is for the “annotation, method invocation, method signature, variable $\rightarrow v_2v_1$ ” rule. This rule occurred 3 times in the sample and has a lift of 3.19, meaning that if the resolution contains these language constructs, there is a 219% increased chance of the resolution being of the v_2v_1 type. Furthermore, the second-biggest lift is for the “comment, import $\rightarrow v_2v_1$ ” rule, with a lift of 2.55, also occurred 3 times in the sample.

We noted a high variation in the language constructs used in the resolution. For this reason, the frequency of the same language constructs being used together and following the same resolution pattern is low. Nonetheless, we observed something interesting for chunks resolved with *Import*. It seems that there is a preference of the developers for using v_1 lines first when resolving the conflicts. We hypothesize that this happens due to the presentation order of the conflict by Git. Usually, the first version that is presented to the user is v_1 . Since the order of *Import* statements does not matter syntactically, developers simply adopt the lines from v_1 first and then lines from v_2 , if applicable. For example, for chunks resolved with *Import*, 23% use the v_1 *only* resolution pattern, 41% use v_1v_2 , 23% use intertwined lines from v_1 and v_2 , 4% use v_2 *only*, and 9% use v_2v_1 . Moreover, analyzing the lift of these cases, we noted that when the chunk is resolved with *Import*, there is an increased chance of 89% (lift = 1.89) of using the v_1 *only* resolution pattern and 58% (lift = 1.58) of using v_1v_2 . On the other hand, the chance is decreased by 50% (lift = 0.50) for using v_2 *only*, 61% (lift = 0.39) for using v_2v_1 , and 23% (lift = 0.77) for using intertwined lines.

We analyzed cases of chunks that were resolved with *Import* and use more complex resolution patterns, such as intertwined lines, to understand the developers reasoning for such a decision. Observe the conflicting chunk from the project rapidftr/RapidFTR-Android displayed in Listing 7. Despite the developer adopting intertwined lines from v_1 and v_2 in the resolution (shown in Listing 8), there is a rationale behind the chosen order. Apparently, imports from the same package were grouped together to ease visualization, probably due to automated ordering used by some IDE. By manually inspecting the remaining cases, we observed this ordering in 60% of the cases. There is no apparent reason for the ordering chosen by the developers for the remaining 40% cases.

Listing 7: Chunk from project rapidftr/RapidFTR-Android showing one case of the association rule *Import* \rightarrow *Others* .

```

1  import com.google.inject.Inject;
2  import com.rapidftr.R;
3  <<<<<<<
4  import com.rapidftr.model.BaseModel;
5  import com.rapidftr.model.User;
6  import com.rapidftr.repository.Repository;
7  import com.rapidftr.service.FormService;
8  import com.rapidftr.service.SyncService;
9  =====
10 import com.rapidftr.RapidFtrApplication;
11 import com.rapidftr.model.Child;
12 import com.rapidftr.model.User;
13 import com.rapidftr.repository.ChildRepository
14     ↪ ;
15 import com.rapidftr.service.ChildService;
16 import com.rapidftr.service.DeviceService;
17 import com.rapidftr.service.FormService;
18 import com.rapidftr.utils.DeviceAdmin;
19 >>>>>>>
20 import org.apache.http.HttpException;
21 import org.json.JSONException;

```

Listing 8: Resolution of the chunk displayed in Listing 7.

```

1  import com.google.inject.Inject;
2  import com.rapidftr.R;
3  import com.rapidftr.model.BaseModel;
4  import com.rapidftr.model.Child;
5  import com.rapidftr.model.User;
6  import com.rapidftr.repository.Repository;
7  import com.rapidftr.service.DeviceService;
8  import com.rapidftr.service.FormService;
9  import com.rapidftr.service.SyncService;
10 import org.apache.http.HttpException;
11 import org.json.JSONException;

```

Although we found some subtle trends when analyzing the data to answer RQ5, we did not find any strong indication of causality. Nonetheless, these observed trends could be valuable for generating new configurations in the context of Search-Based Software Engineering (SBSE). Rather than relying solely on random edits to generate the next configuration, incorporating these patterns, even if they are not highly frequent, could potentially increase the likelihood of reaching a solution more efficiently. By exploring and leveraging these patterns, SBSE approaches could strategically guide the search process, enhancing the overall effectiveness of the resolution strategy.

Finding 5: The lift analysis for RQ5 did not result in interesting rules due to the diverse range of language constructs present in the resolutions. Nonetheless, by further investigating chunks resolved with *Import*, we found that developers have a strong preference for lines coming from v_1 , possibly because of the presentation order of the conflicting chunk by Git. Moreover, when developers intertwine lines from v_1 and v_2 for chunks resolved with *Import*, it is possibly due to automated ordering provided by the IDE.

4 Discussion

Number of conflicting chunks: Ghiotto et al. (2020) found that 40% of the conflicting merges have one chunk and 90% have 10 or fewer chunks. In comparison, in our analysis, which includes merges that have at least one chunk resolved with combination, 23.6% of the merges have one chunk and 81.3% have 10 or fewer chunks. This suggests that merge conflicts that include chunks resolved with combination usually have more chunks than the average case.

Size of conflicting chunks: When considering the size of conflicting chunks, Ghiotto et al. (2020) found that 94% of the chunks have up to 50 LOC in each version, 68% have up to 10 LOC, and 50% up to 5 LOC. In contrast, we found that 98.6% of our analyzed chunks have up to 50 LOC, 88.8% up to 10 LOC, and 76.5% at most have 5 LOC. Regarding the size of each version involved in the conflict, they found that the median v_1 size is 2 and the median v_2 size is 2.5. We found that the chunks analyzed in our study have the same median size for v_1 and a slightly bigger median size for v_2 (3 LOC). Based on these observations, the results indicate that chunks resolved with combination have a slightly bigger v_2 . However, it seems that the chunks in this study are much smaller than in the general case.

Contrary to conflicts resolved entirely with v_1 , v_2 , v_1v_2 , or v_2v_1 , conflicts that are resolved with combination may be harder to resolve using automated approaches. This is due to the exponential nature of the problem of combining existing lines. To illustrate this, we expand on the reasoning behind the number of possible combinations for resolving a conflict.

To start, consider that each line in the conflict may or may not be used in the resolution. In addition, each line may appear in a position in the resolution that is different from its original position in the chunk, i.e., a permutation of the original lines. The formula $\frac{n!}{(n-i)!}$ represents the number of possible permutations of size i for a tuple with n elements. For example, given the tuple $S = (A, B, C)$ with $n = 3$ and considering that we are interested in finding the number of permutations with size $i = 2$. By applying the formula $\frac{3!}{(3-2)!}$, we find that there are 6 different ways of arranging S in tuples of size 2. Namely: $S_1 = (A, B)$, $S_2 = (A, C)$, $S_3 = (B, A)$, $S_4 = (B, C)$, $S_5 = (C, A)$, $S_6 = (C, B)$. Given that we know how to get the number of permutations of a given size for a tuple, we are now interested in candidate resolutions (i.e., tuples) of all sizes from 1 to $v_1 + v_2$, which we call n . Thus, we add the result of the permutations' formula for each size, resulting in Equation 1.

It is important to highlight that all the following equations

must be subtracted by 4 units, which corresponds to the resolutions that use the whole v_1 , v_2 , v_1v_2 , and v_2v_1 since they are not the focus of this paper. The exception is when v_1 or v_2 have zero lines. In this case, only one unit is subtracted from the equations. We omit this subtraction in the equations to make them less cluttered.

$$N(n) = \sum_{i=1}^n \frac{n!}{(n-i)!} \quad (1)$$

To illustrate Equation 1, consider the median conflicting chunk case, with 6 LOC in total. In this case, the total number of different combinations is 1,952. This number grows very quickly as the conflict gets bigger. For example, a conflict with 7 LOC has 13,695 different combinations of its lines. A conflict with 20 LOC on each version, which is much smaller than the biggest conflict in our dataset, has around 2.22×10^{29} different options.

In a permutation, we may have different arrangements of the elements, as shown above. On the other hand, in a combination, the order of the elements is disregarded. We found in RQ2 that only 1.4% of the chunks violate the partial order in the resolution. Since this represents a small number of chunks, we may assume that all resolutions must respect the partial order of the chunk, i.e., the resolutions are combinations of the original lines.

The formula for calculating the total number of combinations of size i for a tuple with n elements is $\frac{n!}{i!(n-i)!}$. This can also be written as $\binom{n}{i}$, which can be read as n choose i . Following the example of the tuple $S = (A, B, C)$, if we are interested in finding the number of combinations of size $i = 2$, we have $\frac{3!}{2!(3-2)!} = 3$. Thus, there are 3 different combinations of size 2 for S : $S_1 = (A, B)$, $S_2 = (A, C)$, $S_3 = (B, C)$. Knowing how to get the number of combinations of a given size, we may use the product rule to calculate ways of choosing lines from v_1 , v_2 , or both. Thus, in Equation 2, the first term $\binom{L_1}{i}$ represents the number of ways of picking i lines from v_1 , where L_1 represents the total number of lines in v_1 . Analogously, the second term $\binom{L_2}{j}$, represents the number of ways of picking j lines from v_2 , where L_2 represents the total number of lines in v_2 . The third term $\binom{i+j}{i}$ represents the number of ways of picking i lines from both v_1 and v_2 . Finally, the goal of the last term $\min(1, i+j)$ is to remove empty resolutions from the sum (when i and j are equal to 0). Using the same principle from Equation 1, we add the result of each iteration, representing different resolution sizes.

$$N(L_1, L_2) = \sum_{i=0}^{L_1} \sum_{j=0}^{L_2} \binom{L_1}{i} \binom{L_2}{j} \binom{i+j}{i} \min(1, i+j) \quad (2)$$

Following the example of the median chunk size, suppose that the conflicting chunk has $L_1 = 2$ and $L_2 = 4$. In this case, applying Equation 2, only 211 combinations respect the partial order of both v_1 and v_2 . This number represents a search space that is 10.8% of the search space using Equation 1. As another example, applying Equation 2 for the case with 7 LOC ($L_1 = 3$ and $L_2 = 4$), yields 659 options (4.8% of the search space using Equation 1) and for the case with 20

LOC on each version, around 1.62×10^{18} options ($< 0.001\%$ of the search space using Equation 1). Table 9 shows the percentage of the search space represented by Equation 2 in relation to Equation 1 for different chunk sizes.

Analyzing Table 9, we can observe that the search space reduction using Equation 2 grows quickly as the chunk size grows. In fact, for chunk sizes $(L_1 + L_2)$ bigger than 14 LOC, the search space represented by Equation 2 in relation to Equation 1 is so small that it cannot be represented using only three decimal points. These cases are highlighted in bold.

We also found in our study that 72.7% of the chunks resolved with combination follow an additional restriction regarding the resolution content. They use only lines from v_1 or v_2 , or they use lines from either v_1 or v_2 followed by lines from the other version. That is, the lines from v_1 and v_2 are not interleaved more than once. Thus, in Equation 3, we remove the third term from Equation 2. The only two options for picking lines from v_1 and v_2 is to have some lines of v_1 then some lines of v_2 , or the opposite. Thus, the third term in Equation 3 represents this option: $(\min(1, i) + \min(1, j))$. We cannot simply add 2 units to the equation, since there are special cases where i and j are equal to 0.

$$N(L_1, L_2) = \sum_{i=0}^{L_1} \sum_{j=0}^{L_2} \binom{L_1}{i} \binom{L_2}{j} (\min(1, i) + \min(1, j)) \quad (3)$$

Using the example of the median case for a chunk with $L_1 = 2$ and $L_2 = 4$ in Equation 3 results in only 104 feasible combinations (49.3% of the search space using Equation 2). For the case with $L_1 = 3$ and $L_2 = 4$, Equation 3 results in 228 feasible combinations (34.6% of the search space using Equation 2). Finally, for the bigger chunk with 20 LOC on each version, Equation 3 results in around 2.2×10^{12} ($< 0.001\%$ of the search space using Equation 2) combinations. Table 10 shows the search space represented by Equation 3 in relation to Equation 2 as the chunk size grows.

We observed that Table 10 shows no relative search space smaller than 0.001%. On the other hand, Table 9 shows that chunks bigger than 14 LOC have a relative search space smaller than 0.001%. Thus, the search space reduction provided by Equation 3 over Equation 2 is smaller than that provided by Equation 2 over Equation 1.

By leveraging the findings of our study to investigate the number of possible combinations for a conflicting chunk, we show it is possible to reduce the search space for a given order of magnitude and thus allow the combinations of small chunks, which we have shown are very frequent, to be addressed in a reasonable time. There is a trade-off between the number of cases that can be covered by the added restrictions and the magnitude of the reduction of feasible combinations. We show that by restricting the resolutions to comply with the partial order of the chunks, there is coverage of 98.6% of the analyzed chunks. In this situation, following the median chunk size, the search space is 10.8% of the search space without restrictions. In other words, the search space is reduced by 89.2%. Using both the partial order restriction and the restriction to allow only resolutions that do not interleave lines from both versions involved in the conflict more

Table 9. Search space size for Equation 2 relative to Equation 1 for different v_1 and v_2 sizes.

		v_2 size (L_2)										
		0	1	2	3	4	5	6	7	8	9	10
v_1 size (L_1)	0	N/A	100.000%	66.667%	42.857%	22.222%	9.259%	3.171%	0.920%	0.232%	0.052%	0.010%
	1	100.000%	100.000%	63.636%	38.333%	18.380%	7.121%	2.300%	0.638%	0.155%	0.034%	0.007%
	2	66.667%	63.636%	46.667%	25.234%	10.809%	3.819%	1.149%	0.301%	0.070%	0.015%	0.003%
	3	42.857%	38.333%	25.234%	12.295%	4.812%	1.579%	0.447%	0.111%	0.025%	0.005%	<0.001%
	4	22.222%	18.380%	10.809%	4.812%	1.748%	0.539%	0.145%	0.034%	0.007%	0.001%	<0.001%
	5	9.259%	7.121%	3.819%	1.579%	0.539%	0.157%	0.040%	0.009%	0.002%	<0.001%	<0.001%
	6	3.171%	2.300%	1.149%	0.447%	0.145%	0.040%	0.010%	0.002%	<0.001%	<0.001%	<0.001%
	7	0.920%	0.638%	0.301%	0.111%	0.034%	0.009%	0.002%	<0.001%	<0.001%	<0.001%	<0.001%
	8	0.232%	0.155%	0.070%	0.025%	0.007%	0.002%	<0.001%	<0.001%	<0.001%	<0.001%	<0.001%
	9	0.052%	0.034%	0.015%	0.005%	0.001%	<0.001%	<0.001%	<0.001%	<0.001%	<0.001%	<0.001%
	10	0.010%	0.007%	0.003%	<0.001%	<0.001%	<0.001%	<0.001%	<0.001%	<0.001%	<0.001%	<0.001%

Table 10. Search space size for Equation 3 relative to Equation 2 for different v_1 and v_2 sizes.

		v_2 size (L_2)										
		0	1	2	3	4	5	6	7	8	9	10
v_1 size (L_1)	0	N/A	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
	1	100.000%	100.000%	85.714%	78.261%	71.186%	64.748%	59.048%	54.077%	49.771%	46.043%	42.803%
	2	100.000%	85.714%	71.429%	59.259%	49.289%	41.300%	34.948%	29.889%	25.829%	22.537%	19.839%
	3	100.000%	78.261%	59.259%	45.000%	34.598%	27.036%	21.492%	17.366%	14.244%	11.840%	9.959%
	4	100.000%	71.186%	49.289%	34.598%	24.843%	18.281%	13.770%	10.594%	8.304%	6.617%	5.350%
	5	100.000%	64.748%	41.300%	27.036%	18.281%	12.758%	9.162%	6.748%	5.080%	3.897%	3.040%
	6	100.000%	59.048%	34.948%	21.492%	13.770%	9.162%	6.302%	4.461%	3.236%	2.399%	1.811%
	7	100.000%	54.077%	29.889%	17.366%	10.594%	6.748%	4.461%	3.044%	2.135%	1.532%	1.122%
	8	100.000%	49.771%	25.829%	14.244%	8.304%	5.080%	3.236%	2.135%	1.450%	1.010%	0.719%
	9	100.000%	46.043%	22.537%	11.840%	6.617%	3.897%	2.399%	1.532%	1.010%	0.684%	0.474%
	10	100.000%	42.803%	19.839%	9.959%	5.350%	3.040%	1.811%	1.122%	0.719%	0.474%	0.320%

than once, there is still coverage of 72.7% of the chunks in our dataset. In practical terms, following the median case example, we reduce the search space of the problem by 94.7% (from 1,952 to 104 options) at the cost of reducing the number of chunks where such a heuristic would succeed.

In addition to the reduction in the search space for resolutions, future resolution techniques could benefit from the characteristics we found regarding both the conflicting chunks and the resolution lines. We found, for example, that when the conflicting chunk is of the type “Annotation, method signature”, there is an increased chance of 251% of resolving it using the $v_1 v_2$ pattern. Nonetheless, we observed that this happened mostly in one project of the dataset. Other interesting cases also were observed, such as when the chunk type is “Method invocation, method signature”, there is an increased 229% chance of using the $v_2 v_1$ resolution pattern. This also occurred mostly in one of the dataset’s projects. These findings suggest that when building automated approaches, it may be worth considering the context of the project where it will be used. We also observed that new automated conflict resolution approaches might be improved by prioritizing the resolution search using only lines from one version involved in the conflict when the chunk contains only *Import* statements. All of these findings are encouraging for the application of search-based techniques to the problem of conflict resolution.

5 Threats to validity

The data of the projects and conflicts we used in our study were collected initially by Ghiotto et al. (2020). Thus, we are subject to the same threats to internal validity associated with their data collection. They mitigated most of these risks by selecting random conflicting chunks, analyzing the conflicting chunk text and resolution, extracting the relevant in-

formation, and comparing it to their automated collection to ensure the automated classification and data extraction was working as intended. In addition, two researchers independently checked samples and discussed cases where they did not agree to reach a consensus. Besides such actions to mitigate risks, we found a new data issue: the authors filtered out explicit repository forks from their data using the metadata from GitHub API that flags a repository as a fork. However, we observed that this filter does not accurately remove all projects that are forks, as it does not detect implicit forks. We analyzed and compared the commit history of every repository in our dataset and filtered out those with any commit in common. Regarding the construct validity, the partial order checking algorithm was thoroughly tested with synthetic and real examples to ensure it worked as intended. In addition, we selected random samples of the chunks after performing the study to verify whether the algorithm classified them correctly.

Finally, regarding the external validity, our results may not be generalized to projects that are not open-source or developed in a programming language other than Java. We expect to mitigate this generalization threat in future works by analyzing more repositories in different programming languages.

6 Related Work

In this Section, we discuss research related to ours. We focus mainly on studies that classify conflict resolutions and that analyze conflicts complexity.

Yuzuki et al. (2015) focused on studying how developers resolved conflicts. They analyzed 779 conflicts at the method level from 10 open-source Java projects. They found that 48% of the conflict methods were caused by the deletion of the involved method in one of the versions. 44% were caused

by concurrent edits, and 8% were caused by method renaming. Regarding the resolution, they found that 99% of the conflicts were resolved by adopting the contents of one of the versions. Different from our study, they classified the conflict resolutions based on the semantics of the changes at the method level. In our case, we focused on the text lines from the chunks.

Nguyen and Ignat (2017) analyzed adjacent-line conflicts of four large open-source projects. They classified the conflict resolutions in three different ways. One was applying changes from both versions. Another option was to apply the changes from one of the versions. Finally, the last option was not applying any change at all. They found different proportions where the resolutions include modifications from both versions, varying from 24.4% to 85%. Different from our study, they focused on adjacent-line conflicts reported by Git. In our study, we do not differentiate between conflict types; instead, we focus on a specific type of resolution strategy.

Ghiotto et al. (2020) performed a large-scale study with 2,731 open-source Java projects, aiming to investigate conflicts characteristics and how developers resolved them. Conflicting chunks' resolutions were classified as choosing one of the conflicting versions, concatenating one version after the other, combining the lines involved in both versions, writing new code, or having no resolution at all. They found that most of the chunks were resolved by adopting one of the versions in conflict. Furthermore, they also found that 87% of the chunks contained only lines that already existed in the conflict, and 90% of the chunks had less than 50 LOC in each version. They identified different patterns regarding how the developers resolved conflicts but did not delve into them. Our paper is a follow-up to this study. Here, we specifically focus on conflicts that were resolved using the combination strategy.

Brindescu et al. (2020a) performed an empirical study using 6,979 conflicts from 143 open-source Java projects. Their goal was to analyze how merge conflicts impact the quality of the software. According to them, a conflict could be resolved by selecting one of the involved versions, interleaving the code present in both versions or adapting the conflicting code. They found a different distribution of resolution patterns when compared to previous work (Yuzuki et al., 2015; Ghiotto et al., 2020). The most common pattern was to adapt the code, occurring in 60.8% of the conflicts. The second most common was to interleave the code (26.4%) and finally, to select one of the versions, occurring in 12.8% of the conflicts. They also found that code involved in conflicts was more likely to be involved in future bugs. Similar to Ghiotto et al. (2020), the authors identified some resolution strategies but did not focus on any particular one. In our paper, we focus specifically on the strategy they classify as "interleaving code".

Pan et al. (2021) performed an empirical study on the Microsoft Edge project, which is a fork of the Chromium project. They first investigated the characteristics of conflicts in the project. Then, based on the characteristics, they proposed an approach that can learn from examples of conflict resolutions of the project and generate resolutions for new conflicts. They found that most conflicts in the project had only 1 or 2 lines. In addition, 98% of the conflicts had less than 50 LOC.

The proposed approach for resolving conflicts was able to find the resolution in 11.4% of all conflicts of C++ files with an accuracy of 93.2%. Different from our study, where we study conflicts from a range of open-source projects, they focused on one specific project, which is a fork of a very popular project.

Some studies (Nelson et al., 2019; Brohi, 2019; Vale et al., 2021; Brindescu et al., 2020b) focused on the difficulty of resolving conflicts from the developer's perspective. Nelson et al. (2019) performed interviews and conducted surveys with developers to investigate how they perceived the difficulty of resolving conflicts. According to the authors, the developers perceived that the conflict complexity had more impact on the resolution difficulty than the conflict size. Brohi (2019) and Vale et al. (2021) analyzed the correlation between different project metrics and characteristics of conflicts. They concluded that resolving conflicts did not involve only looking at the conflict code, but also at the greater picture of the merge. This conclusion was based on the finding that merge metrics were more correlated to the time to resolve a conflict than the merge conflict metrics. Finally, Brindescu et al. (2020b) investigated the feasibility of predicting the difficulty of conflict resolution. According to them, the most important attribute when predicting conflict resolution difficulty was the cyclomatic complexity, which reinforced the findings of the previous studies that analyzed the developer perspective. Different from these studies, which focus on the developer perspective, we focus on how the conflicts can be resolved by automated approaches. To this end, we discuss conflicts' complexity based on the number of possible combinations of conflicting lines.

Some studies attempt to resolve merge conflicts automatically. For example, Zhu and He (2018) introduced AutoMerge, a method for automatically generating candidate resolutions by combining and rearranging program elements involved in detected conflicts. Despite its capability to yield resolutions with an average accuracy of 95.1%, this approach relies solely on user validation without providing validation mechanisms for the generated resolutions. Conversely, Fraternali et al. (2020) proposed Almost-Rerere, leveraging historical resolutions to tackle current conflicts through clustering and genetic algorithms. Although effective in resolving a substantial portion of conflicts, its reliance on textual similarity metrics might not fully capture complex conflict patterns.

Incorporating Automated Program Repair (APR) into merge conflict resolution, Xing and Maruyama (2019) utilized APR systems to synthesize resolutions based on conflicting class elements and merged classes. While promising, the approach lacks extensive evaluation against real-world conflicts. Similarly, Pan et al. (2021) aimed to resolve conflicts in projects like Microsoft Edge by identifying patterns and synthesizing resolutions using a Domain-Specific Language (DSL). Despite achieving a resolution for 11.4% of conflicts, challenges remain in addressing a vast majority of conflicts and ensuring generalizability across diverse project contexts.

Recent endeavors have explored novel approaches to merge conflict resolution. Dinella et al. (2022) introduced DeepMerge, which utilizes machine learning models to pre-

dict resolutions based on encoded conflict representations. While exhibiting promise, DeepMerge's performance remains limited, particularly in resolving complex, non-trivial merges. GMerge by Zhang et al. (2022) explored the use of large neural language models in resolving semantic and textual conflicts, presenting GMerge as a viable solution with considerable accuracy in semantic conflict resolution. However, challenges persist in effectively addressing small conflicts and ensuring robustness across various conflict types and sizes. Furthermore, Svyatkovskiy et al. (2022) introduced MergeBERT, leveraging transformer-encoder models to classify resolution strategies for token-level conflicts. MergeBERT has demonstrated significant improvements in classification accuracy compared to previous approaches, achieving an accuracy of 63.2% for Java projects and outperforming previous merge tools. Lastly, Dong et al. (2023) introduced MergeGEN, which utilizes a generative encoder-decoder model for processing conflict representations and generating resolutions in an auto-regressive manner instead of using classification. This approach sets new benchmarks in merge conflict resolution, achieving an accuracy of 67.7% for Java projects and also state-of-the-art accuracy for CSharp and JavaScript projects.

In this paper, we demonstrate the feasibility of combining conflicting lines and show common patterns that can be used as heuristics to help the process of generating the correct conflict resolution candidate. We imagine that a new approach could use these findings to generate candidates, which are then validated through testing and compilation. The advantage of such an approach over machine learning-based approaches appears to lie in the validation process. While machine learning-based approaches rely on historical data, i.e., what they have learned from similar past situations, to resolve conflicts, search-based approaches would employ some form of validation, such as compilation and testing, to validate the generated candidates and ensure they fulfill the role of resolving the conflict while maintaining developers' intentions. Thus, we believe that in terms of solution correctness, search-based approaches may yield superior results compared to machine learning-based approaches.

Our work complements previous studies by investigating the conflict characteristics and the resolution patterns of conflicts that were resolved by combining existing lines. We also shed light on the complexity of potentially automating this type of conflict resolution.

7 Conclusion

This paper investigates how developers resolve conflicting chunks by combining the conflicting lines. To achieve this, we analyze 10,177 conflicting chunks from 1,076 open-source Java projects.

We found that the subset of conflicts we analyzed has a median of 6 LOC and their resolutions have a median of 3 LOC. We also found that 98.6% of the conflicts do not violate the partial order of the chunk and that in 72.7% of them, the developers do not interleave lines from different versions involved in the conflict more than once. Finally, we found that developers prefer to resolve conflicts containing only *Import*

statements using lines from the local version of the conflict.

We leverage our findings to argue that reducing the search space of feasible conflict chunk resolutions is possible. For instance, the search-space reduction for the median chunk case is 94.7%. This reduction is even bigger for harder cases, where the chunk has more lines. This encourages the adoption of search-based techniques for automating conflict chunk resolution.

In future work, we intend to investigate if our findings hold for different programming languages. We also plan to investigate the remaining resolution patterns that were not analyzed in this paper. Furthermore, we plan to analyze the code characteristics of the resolutions for each pattern. Finally, we also plan to leverage our findings to develop an approach that employs search-based techniques to generate candidate resolutions for conflicting chunks. We hope that such an approach can help to alleviate the pain of developers when dealing with merge conflicts.

Artifacts availability

The artifacts used in this paper are publicly available at <https://doi.org/10.6084/m9.figshare.24012579>.

Acknowledgements

The authors would like to thank CNPq (grant 311955/2020-7) and FAPERJ (grants E-26/010.101250/2018, E26/010.002285/2019, E26/201.038/2021, and E26/200.591/2021) for the financial support.

References

- Accioly, P., Borba, P., and Cavalcanti, G. (2018). Understanding semi-structured merge conflict characteristics in open-source Java projects. *Empirical Software Engineering*, 23(4):2051–2085.
- Altman, D. and Bland, J. (2005). Statistics notes: Standard deviations and standard errors. *British Medical Journal*, 331(7521):903–903.
- Appleton, B., Berczuk, S., Cabrera, R., and Orenstein, R. (1998). Streamed lines: Branching patterns for parallel software development. In *Proceedings of PloP*, volume 98, page 14.
- Brindescu, C., Ahmed, I., Jensen, C., and Sarma, A. (2020a). An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering*, 25(1):562–590.
- Brindescu, C., Ahmed, I., Leano, R., and Sarma, A. (2020b). Planning for untangling: Predicting the difficulty of merge conflicts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 801–811. IEEE.
- Brohi, M. Z. (2019). Software Practitioners Perspective on Merge Conflicts and Resolution. Master's thesis, University of Passau.

- Campos Junior, H. d. S., de Menezes, G. G. L., Barros, M. d. O., van der Hoek, A., and Murta, L. G. P. (2022). Towards merge conflict resolution by combining existing lines of code. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, pages 425–434.
- Cavalcanti, G., Borba, P., and Accioly, P. (2017). Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27.
- Costa, C., Figueiredo, J. J., Ghiotto, G., and Murta, L. (2014). Characterizing the problem of developers' assignment for merging branches. *International Journal of Software Engineering and Knowledge Engineering*, 24(10):1489–1508.
- Dinella, E., Mytkowicz, T., Svyatkovskiy, A., Bird, C., Naik, M., and Lahiri, S. (2022). Deepmerge: Learning to merge programs. *IEEE Transactions on Software Engineering*, 49(4):1599–1614.
- Dong, J., Zhu, Q., Sun, Z., Lou, Y., and Hao, D. (2023). Merge conflict resolution: Classification or generation? In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1652–1663. IEEE.
- Fraternali, P., Herrera Gonzalez, S. L., and Tariq, M. M. (2020). Almost rerere: An approach for automating conflict resolution from similar resolved conflicts. In *Web Engineering: 20th International Conference, ICWE 2020, Helsinki, Finland, June 9–12, 2020, Proceedings 20*, pages 228–243. Springer.
- Ghiotto, G., Murta, L., Barros, M., and van der Hoek, A. (2020). On the nature of merge conflicts: A study of 2,731 open source java projects hosted by github. *IEEE Transactions on Software Engineering*, 46(8):892–915.
- Harman, M., Mansouri, S. A., and Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):1–61.
- Hinkle, D. E., Wiersma, W., and Jurs, S. G. (2003). *Applied statistics for the behavioral sciences*, volume 663. Houghton Mifflin College Division.
- Israel, G. D. (1992). Determining sample size. Technical report, University of Florida.
- Kasi, B. K. and Sarma, A. (2013). Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 732–741. IEEE.
- Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, 28(5):449–462.
- Nelson, N., Brindescu, C., McKee, S., Sarma, A., and Dig, D. (2019). The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Software Engineering*, 24(5):2863–2906.
- Nguyen, H. L. and Ignat, C.-L. (2017). Parallelism and conflicting changes in Git version control systems. In *IWCES'17 - The Fifteenth International Workshop on Collaborative Editing Systems*, pages 1–1, Portland, Oregon, United States. HAL-Inria.
- Pan, R., Le, V., Nagappan, N., Gulwani, S., Lahiri, S., and Kaufman, M. (2021). Can program synthesis be used to learn merge conflict resolutions? an empirical analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 785–796.
- Seibt, G., Heck, F., Cavalcanti, G., Borba, P., and Apel, S. (2021). Leveraging structure in software merge: An empirical study. *IEEE Transactions on Software Engineering*, 48(11):4590–4610.
- Svyatkovskiy, A., Fakhoury, S., Ghorbani, N., Mytkowicz, T., Dinella, E., Bird, C., Jang, J., Sundaresan, N., and Lahiri, S. K. (2022). Program merge conflict resolution via neural transformers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 822–833.
- Vale, G., Hunsen, C., Figueiredo, E., and Apel, S. (2021). Challenges of resolving merge conflicts: A mining and survey study. *IEEE Transactions on Software Engineering*.
- Xing, X. and Maruyama, K. (2019). Automatic software merging using automated program repair. In *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, pages 11–16. IEEE.
- Yuzuki, R., Hata, H., and Matsumoto, K. (2015). How we resolve conflict: an empirical study of method-level conflict resolution. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, pages 21–24, Montreal, QC, Canada. IEEE.
- Zhang, J., Mytkowicz, T., Kaufman, M., Piskac, R., and Lahiri, S. K. (2022). Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 77–88.
- Zhu, F. and He, F. (2018). Conflict resolution for structured merge via version space algebra. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25.