Towards Merge Conflict Resolution by Combining Existing Lines of Code

Heleno de S. Campos Junior
helenocampos@id.uff.br
Universidade Federal Fluminense
Niterói, Brazil

Gleiph Ghiotto L. de Menezes
gleiph@ice.ufjf.br
Universidade Federal de Juiz de Fora
Juiz de Fora, Brazil

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

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

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

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. In this paper, we investigate a specific 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 that developers frequently use 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 77.4% of the chunk resolutions do not interleave lines from different contributions more than once. These findings altogether, when used as heuristics for automatic merge resolution, could enable the reduction of the search space by 94.7%, paving the road for future search-based software engineering tools for this problem.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; **Collaboration in software development**; **Software version control**.

KEYWORDS

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

SBES 2022, October 5–7, 2022, Virtual Event, Brazil

ACM ISBN 978-1-4503-9735-3/22/10…$15.00
https://doi.org/10.1145/3555228.3555229

ACM Reference Format:
Heleno de S. Campos Junior, Gleiph Ghiotto L. de Menezes, Márcio de Oliveira Barros, André van der Hoek, and Leonardo Gresta Paulino Murta. 2022. Towards Merge Conflict Resolution by Combining Existing Lines of Code. In *XXXVI Brazilian Symposium on Software Engineering (SBES 2022), October 5–7, 2022, Virtual Event, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3555228.3555229

1 INTRODUCTION

Nowadays, software is rarely developed by a single person. People often collaborate to compose a software product. To achieve this, developers usually adopt branches that represent separate lines of development. Sometimes these branches live for a long time. Other times they are allocated to making 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. 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 [6, 11]. Indeed, previous work [5, 10] report that from 16% to 54% of all merges result in conflicts.

Resolving merge conflicts is a time-consuming and error-prone task. Thus, any automation to such a task is welcome. Previous work [7] 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 just cherry-picking some lines or using all of them. Developers may use some 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 of both versions, interleaving them. Hence, a potential strategy for resolving conflicts is using Search-based Software Engineering [8] to explore the combinations of existing code. 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. This challenging problem was not explored in detail yet in the literature and motivated our paper (we refer the reader to [1, 7] to obtain a more extensive background on merge conflicts).

In this paper, we analyze a subset of the conflicts analyzed by Ghiotto et al. [7]. 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, which 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, we: (i) characterize these conflicts and resolutions, (ii) investigate whether the developers preserve the partial order of the lines from the conflicting chunks in the resolution, and (iii) investigate patterns regarding how the developers combine the conflict lines to compose the resolution.

We found that merge conflicts resolved by combining the conflicting lines of code are usually small (median of one chunk and one file, and a median of 6 LOC for the whole chunk and 3 LOC for the resolution). We also found that most of the resolutions (98.6%) do not violate the partial order of the chunk. Finally, we found that 77.4% 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 the common conflict case. Thus, search-based techniques could benefit from these results to automatically find the best combination of lines for resolving a conflict.

The rest of this paper is organized as follows. In Section 2 we discuss the materials and methods used throughout the study. Section 3 shows 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 final 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 three 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?

The first research question aims at characterizing the conflicting chunks and their resolutions. We organized the analysis into three perspectives: the conflicting merges as a whole, each conflicting file in isolation, and each conflicting chunk independently. Addressing this research question gives us a sense of how spread out the conflicts are into multiple chunks and multiple files, as well as how large the chunks are in terms of LOC. The larger the number of files and chunks, the more difficult it is likely to be manually merged and to come up with automatic tools. Thus, it is important to first understand the size issue.

Regarding the second research question, preserving the chunk's partial order means that the resolution lines respect the order of the lines from both sides of the conflicting chunk. Consider the conflicting chunk extracted from the project *unitycoders/uc_pircbotx*[1] and displayed in Listing 1 as an example. 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 where the partial order of the chunk is violated.

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  }
```

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.

Finally, in the third question, we investigate if there are any patterns regarding how developers combine lines to resolve conflicts 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 lines from each version

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

of the chunk are used? Finding such patterns might be useful for developing automated approaches to support the developers.

2.2 Data collection

Ghiotto et al. [7] 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 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. [7]'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.

For each conflicting chunk, we collect its size, 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 chunk and its resolution.

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 an algorithm to perform this task automatically, given the conflicting chunk and its resolution content.

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. The analysis was performed manually because our goal is to further understand the characteristics of these chunks. Our intuition was that even though for some chunks the partial order was violated, for some of them 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, 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 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 come 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 v_1 and v_2 percentage in the resolution. This percentage represents the 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_1v_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_1v_2, instead of $v_1v_1v_2v_2$. This strategy of grouping consecutive elements was also used by Brindescu et al. [3] 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.

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 are resolved with combination seems to decrease 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 is 1.

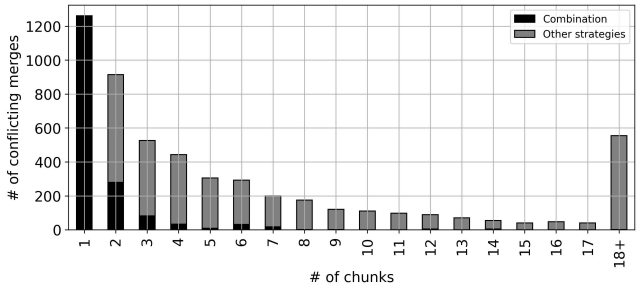

Figure 1: Number of conflict chunks per conflict merge.

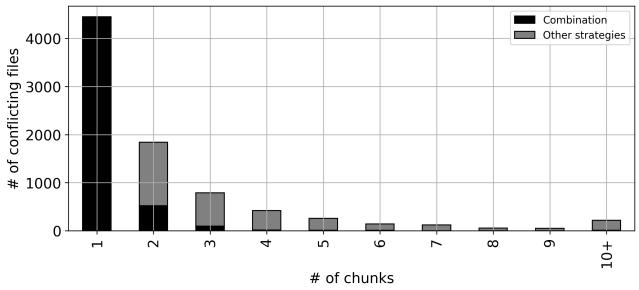

Figure 2: Number of conflicting chunks per conflicting file.

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 single 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.

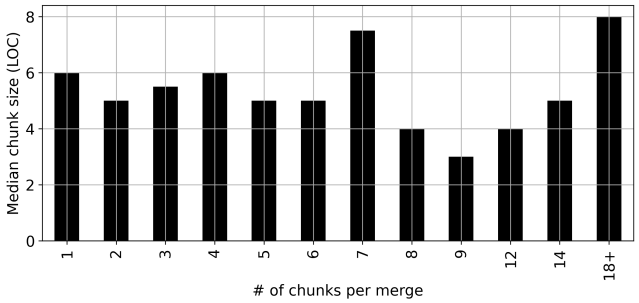

Figure 3: Median chunk size for conflicting merges in relation to the number of chunks.

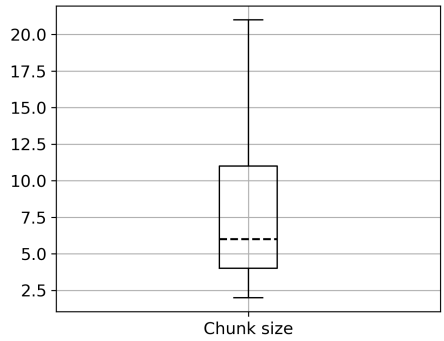

Figure 4: Distribution of the conflicting chunks' size (LOC). Outliers excluded.

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 is discarded in the resolution.

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. In this sense, 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

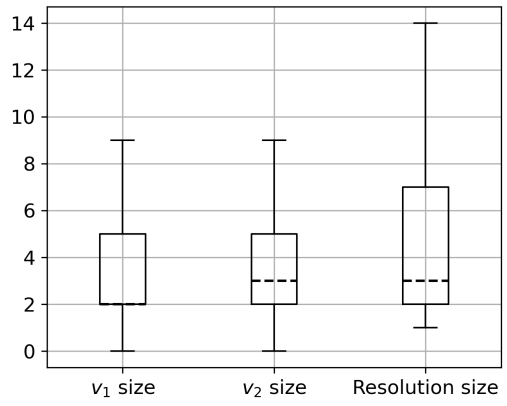

Figure 5: Boxplots for the distribution of the size (LOC) of v_1, v_2, and the chunk resolution, respectively. Outliers excluded.

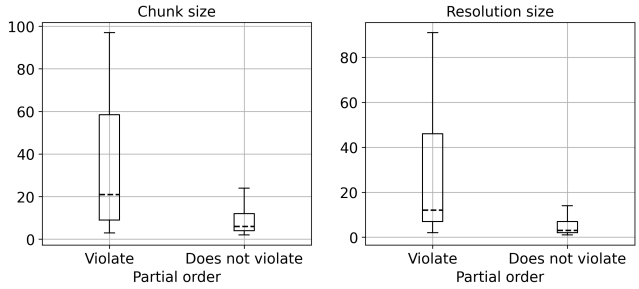

Figure 6: Chunk and resolution size for chunks that violate and do not violate the partial order. Outliers excluded.

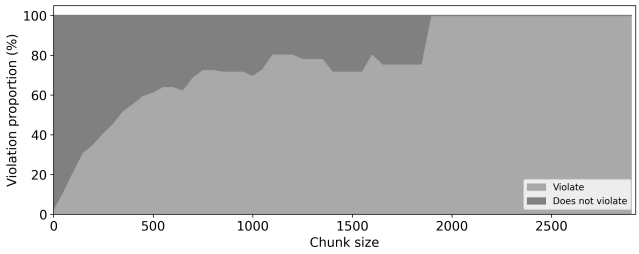

Figure 7: Proportion of chunks that violate and do not violate the partial order as the chunk size grows.

do not violate the partial order have a median of 6 LOC and their resolutions have a median of 3 LOC. Thus, we may say that chunks where the resolution violates the partial order are usually bigger than chunks that do not violate. 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 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 also 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. In other words, if the order is not important for some of these cases, assuming the partial order would not impair finding a correct resolution.

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 avoid 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 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 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% coming 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

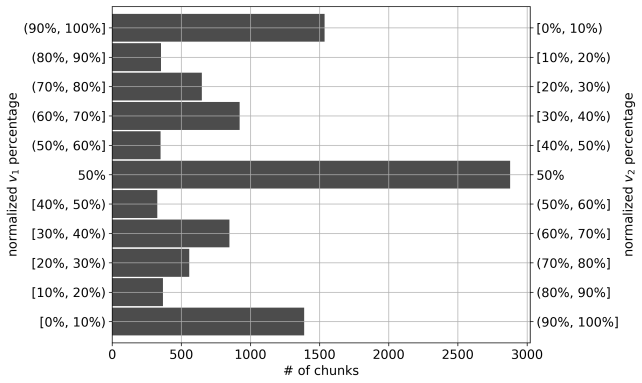

Figure 8: Histogram showing the number of chunks for the normalized percentages of lines in the resolution that come from v_1 (left y-axis) and v_2 (right y-axis).

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 $\boldsymbol{v_1}$ ***only*** and $\boldsymbol{v_2}$ ***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, then use lines from the other chunk side, without going back to the previous side. That is, the lines from different versions of the chunk are not interleaved more than once.

We found that 1,367 (13.4%) of the chunks use the $\boldsymbol{v_1}$ ***only*** pattern and 1,260 (12.4%) use the $\boldsymbol{v_2}$ ***only***. This means that 25.8% of the chunks use a subset of lines from one side of the conflict.

For the cases where the lines come from both sides of 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 sides of the chunk more than once.

The four patterns discussed above cover 7,881 (77.4%) of all chunks in our dataset. The remaining 22.6% 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 $\boldsymbol{v_1}$ ***only*** pattern is 1 (min 0, max 41). In comparison, the average size of v_1 for chunks resolved with the $\boldsymbol{v_2}$ ***only*** pattern is 0.77 (min 0, max 34). This indicates that the developer often uses only lines from one side of the chunk when the other side is small in size.

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 that are not in the center of the Figure 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 of the 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 $\boldsymbol{v_2}$ ***only*** strategy is used in only 0.4% of the chunks. This can also be observed when $v_2 > v_1$, with the $\boldsymbol{v_1}$ ***only*** strategy being used in only 1.4% of the chunks. This reinforces that the developer often uses only lines from one side of the chunk when the other side 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 distribution of the data is not normal, we used Spearman's Rank Correlation Coefficient[9]. We found that there is 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 sides. 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.

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 77.4% 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 chunk side (26.3% for v_1v_2 and 20.6% for v_2v_1). Finally, we also found that when both sides of the conflict have the same size, the developer mostly uses v_1v_2 (41.5%) or v_2v_1 (35.2%). On the other hand, we found a moderate correlation ($\rho = 0.674$) suggesting that the bigger one side of the chunk is, the bigger the chance of the developer choosing lines from that side 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 side of the chunk, (ii) prioritize the v_1v_2 and v_2v_1 patterns when both sides of the chunk have the same size, and (iii) exclude resolutions that interleave lines from both sides of the chunk more than once and still find the correct resolution for 77.4% of them.

4 DISCUSSION

Number of conflicting chunks: Ghiotto et al. [7] found that 40% of the conflicting merges have one chunk and 90% have 10 or fewer

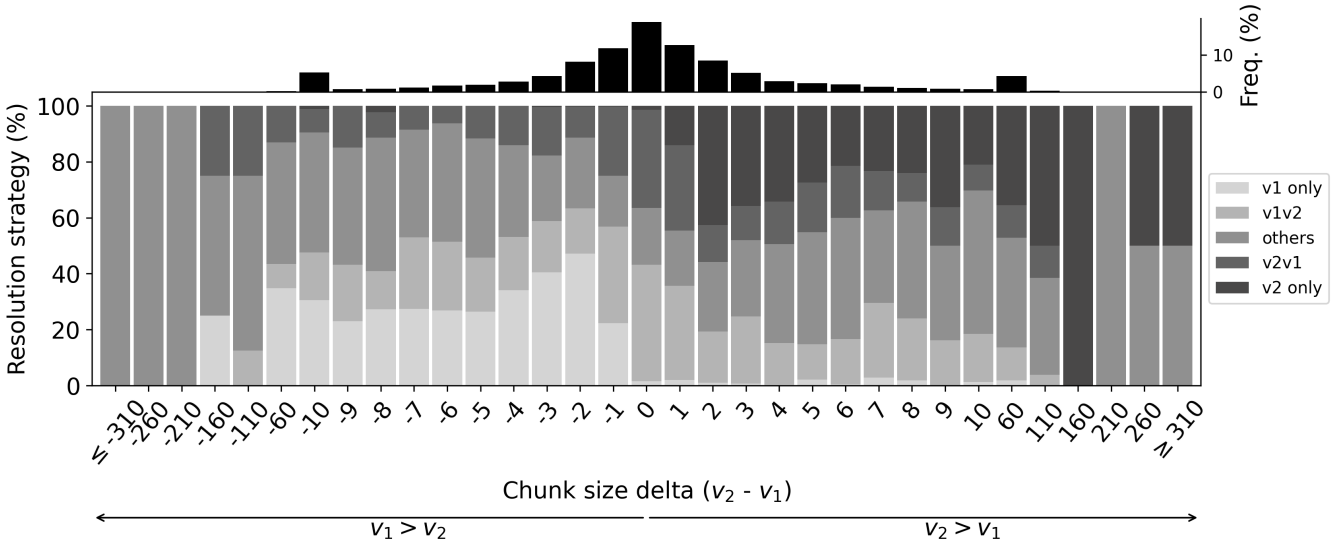

Figure 9: Frequency of each chunk size delta (top) and proportion of resolution patterns used for each chunk size delta (v_2 - v_1).

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. [7] 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% have 5 or less LOC. Regarding the size of each side of the chunk, 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 that are 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 total 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 $n = 3$, consider 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 now are interested in candidate resolutions (i.e. tuples) of all sizes from 1 to $v_1 + v_2$ size, which we call n. Thus, we add the result of the permutations formula for each of the sizes, 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.

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

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 side, 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.

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

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

Analyzing Table 1, we can observe that the search space reduction using Equation 2 grows very quickly as the chunk size gets bigger. 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 places. These cases are highlighted in bold.

We also found in our study that 72.9% of the chunks that are 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 side. 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.

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

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 side, Equation 3 results in around 2.2×10^{12} ($< 0.001\%$ of the search space using Equation 2) combinations. Table 2 shows the search space represented by Equation 3 in relation to Equation 2 as the chunk size grows.

We observed that Table 2 shows no relative search space smaller than 0.001%. On the other hand, Table 1 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 sides of the chunk more than once, there is still coverage of 72.6% 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. These numbers 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. [7]. 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 information, 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 fork projects, 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.

Table 1: 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 2: 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%

6 RELATED WORK

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

Yuzuki et al. [15] focus on studying how developers resolve conflicts. They analyzed a total of 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% are caused by concurrent edits and 8% are 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 classify the conflict resolutions based on the semantics of the changes at the method level. In our case, we focus on the text lines from the chunks.

Nguyen and Ignat [12] analyzed adjacent-line conflicts of four large open-source projects. They classified these conflict resolutions in three different ways. One is applying changes from both versions. Another option is to apply the changes from one of the versions. Finally, the last option is to not apply 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 focus 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. [7] perform a large-scale study with 2,731 open-source Java projects aiming to investigate conflicts characteristics and how they are resolved by developers. Conflicting chunks resolutions are 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 are 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 have less than 50 LOC in each version. They identified different patterns regarding how the developers resolve conflicts but did not delve into any of 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. [2] perform an empirical study using 6,979 conflicts from 143 open-source Java projects. Their goal is to analyze how merge conflicts impact the quality of the software. According to them, a conflict may 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 the previous works [7, 15]. The most common pattern is to adapt the code, occurring in 60.8% of the conflicts. The second most common is 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 is more likely to be involved in future bugs. Similar to Ghiotto et al. [7], 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. [13] perform 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 propose 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 have only 1 or 2 lines. In addition, 98% of the conflicts have 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 focus on one specific project, which is a fork of a very popular project.

Some studies [3, 4, 11, 14] focus on the difficulty of resolving conflicts from the perspective of the developer. Nelson et al. [11] perform interviews and conduct surveys with developers to investigate how they perceive the difficulty of resolving conflicts. According to the authors, the developers perceive that the conflict complexity has more impact on the resolution difficulty than the conflict size. Brohi [4] and Vale et al. [14] analyzed the correlation between different project metrics and characteristics of conflicts. They concluded that resolving conflicts do 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. [3] investigated the feasibility of predicting the difficulty of conflicts resolution. According to them, the most important attribute when predicting the conflict resolution difficulty is the cyclomatic complexity, which reinforces the findings of the previous studies that analyze 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.

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 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 77.4% of them, the developers do not interleave lines from different sides of the conflict more than once.

We leverage our findings to argue that it is possible to reduce the search space of feasible conflict chunk resolutions. 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 is encouraging for 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.19705600.v1.

ACKNOWLEDGMENTS

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

[1] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. 2018. Understanding semi-structured merge conflict characteristics in open-source Java projects. *Empirical Software Engineering* 23, 4 (Aug. 2018), 2051–2085. https://doi.org/10.1007/s10664-017-9586-1

[2] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. An empirical investigation into merge conflicts and their effect on software quality. *Empirical Software Engineering* 25, 1 (Jan. 2020), 562–590. https://doi.org/10.1007/s10664-019-09735-4

[3] Caius Brindescu, Iftekhar Ahmed, Rafael Leano, and Anita Sarma. 2020. Planning for untangling: Predicting the difficulty of merge conflicts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 801–811.

[4] Muhammad Zohaib Brohi. 2019. *Software Practitioners Perspective on Merge Conflicts and Resolution.* Master's thesis. University of Passau. https://www.infosun.fim.uni-passau.de/se/theses/MuhammadZohaibMA.pdf

[5] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 168–178.

[6] Catarina Costa, José JC Figueiredo, Gleiph Ghiotto, and Leonardo Murta. 2014. Characterizing the Problem of Developers' Assignment for Merging Branches. *International Journal of Software Engineering and Knowledge Engineering* 24, 10 (2014), 1489–1508.

[7] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and André van der Hoek. 2020. On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub. *IEEE Transactions on Software Engineering* 46, 8 (2020), 892–915. https://doi.org/10.1109/TSE.2018.2871083

[8] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 1–61.

[9] Dennis E Hinkle, William Wiersma, and Stephen G Jurs. 2003. *Applied statistics for the behavioral sciences*. Vol. 663. Houghton Mifflin College Division.

[10] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 732–741.

[11] Nicholas Nelson, Caius Brindescu, Shane McKee, Anita Sarma, and Danny Dig. 2019. The life-cycle of merge conflicts: processes, barriers, and strategies. *Empirical Software Engineering* 24, 5 (Oct. 2019), 2863–2906. https://doi.org/10.1007/s10664-018-9674-x

[12] Hoai Le Nguyen and Claudia-Lavinia Ignat. 2017. Parallelism and conflicting changes in Git version control systems. In *IWCES'17 - The Fifteenth International Workshop on Collaborative Editing Systems*. HAL-Inria, Portland, Oregon, United States, 1–1. https://hal.inria.fr/hal-01588482

[13] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike Kaufman. 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)*. 785–796. https://doi.org/10.1109/ICSE43902.2021.00077

[14] Gustavo Vale, Claus Hunsen, Eduardo Figueiredo, and Sven Apel. 2021. Challenges of Resolving Merge Conflicts: A Mining and Survey Study. *IEEE Transactions on Software Engineering* (2021).

[15] Ryohei Yuzuki, Hideaki Hata, and Kenichi Matsumoto. 2015. How we resolve conflict: an empirical study of method-level conflict resolution. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*. IEEE, Montreal, QC, Canada, 21–24. https://doi.org/10.1109/SWAN.2015.7070484