Git Merge Conflict Resolution Leveraging Strategy Classification and LLM

Chaochao Shen[1], Wenhua Yang[1,*], Minxue Pan[2], Yu Zhou[1]
[1]College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China
[2]State Key Laboratory for Novel Software Technology, Nanjing University, China
ccshen@nuaa.edu.cn, ywh@nuaa.edu.cn, mxp@nju.edu.cn, zhouyu@nuaa.edu.cn
*Corresponding author

2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS) | 979-8-3503-1958-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/QRS60937.2023.00031

Abstract—In the realm of collaborative software development, version control systems (VCS) like Git play an indispensable role, enabling concurrent development and facilitating seamless integration of disparate code contributions. Despite these benefits, merge conflicts resulting from simultaneous changes to identical code lines often pose significant challenges to the integration process. Addressing this challenge, our paper introduces a novel two-stage approach, termed as CHATMERGE, for resolving Git merge conflicts. CHATMERGE pioneers a unique strategy that employs machine learning to initially predict resolution strategies, and subsequently leverages a large language model, `ChatGPT`, to create resolutions for conflicts that necessitate complex resolution strategies. A series of comprehensive experiments validate CHATMERGE's efficacy, demonstrating its impressive alignment with historical manual resolutions and its superior performance relative to existing, publicly accessible tools. The paper further explores the influence of various classification algorithms and the prompt construction process for `ChatGPT`, providing further insights into the merge conflict resolution process. Moreover, to foster continued advancements in this area, CHATMERGE, along with its associated training and testing datasets, is made publicly available, offering a valuable resource for both developers and researchers. This work, therefore, provides both an innovative solution to merge conflict resolution and a strong foundation for future explorations in this domain.

Keywords–Merge conflict; resolution generation; ChatGPT

1. INTRODUCTION

In modern software development practices, collaborative efforts have become increasingly important. To support effective collaboration, version control systems (VCS), such as Git, are widely employed. These systems offer features that enable developers to work simultaneously on independent branches and subsequently merge their changes into the mainline [1]. Integration is a critical process in collaborative software development, involving the merging of different code files or snippets. However, merge conflicts, arising from concurrent modifications to the same code lines, pose a significant challenge to collaborative development. Prior studies have demonstrated that merge conflicts are a prevalent issue in software development [2], [3], and merge conflicts account for a considerable percentage of all merges, ranging from 10% to 20%. Furthermore, in several large projects, this percentage can even escalate to 50%. Merge conflicts can have detrimental effects on software projects, leading to delays, decreased productivity, and code quality issues.

Git has emerged as the dominant VCS tool with a large user base worldwide [4]. During the merging process, Git is capable of detecting merge conflicts, which occur when multiple developers make simultaneous changes to the same code lines. However, Git lacks the automated capability to resolve these conflicts and instead provides a mechanism for manual intervention. Specifically, it generates independent `conflict blocks` separated by standard markers, necessitating human intervention to resolve conflicts. The manual resolution of merge conflicts in Git is a time-consuming and error-prone task. It involves meticulous analysis of the conflicting code, comprehension of the developers' intentions behind their changes, and informed decisions on how to appropriately integrate the conflicting modifications. This process is often tedious, demanding considerable effort and potentially leading to workflow delays [5], [2]. Furthermore, there is a risk of introducing additional errors or bugs due to oversight or misunderstanding of the conflicting changes [6], [7], [8]. Therefore, despite Git's ability to detect merge conflicts, it relies on human intervention for resolution. The manual nature of conflict resolution in Git poses a challenge to efficient collaboration and can impede the progress of software development projects.

Merge conflicts pose challenges for developers, demanding extra time and effort and causing a decrease in productivity and rise in frustration. To address this, several approaches have been proposed to automate merge conflict resolution in collaborative software development. Some strive for a balance between structured [9], [10] and unstructured merge, called semi-structured merge [11], [12], [13], [14]. These techniques integrate the advantages of structural code representation, such as abstract syntax trees (ASTs), with conventional unstructured text-line-based merge. Some technologies address merge conflicts unique to certain languages or code features, like JavaScript [15] or code refactoring [16], [17]. Also emerging are data-driven generative merge conflict resolution approaches, utilizing deep learning. These approaches use large pre-trained language models (LLMs) for rapid learning or training neural networks for subsequent resolution synthesis tasks [18], [19], [20]. However, despite progress, these approaches are not yet widely used. They often fall short in consistently achieving satisfactory outcomes or providing useful recommendations, reducing their practicality. Existing learning-based techniques focus their efforts on combining the

979-8-3503-1958-3/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00031

existing contents of the input but lack the capacity to generate novel tokens that are not present in the input. This limitation impedes their ability to resolve conflicts that necessitate the introduction of new content. As a result, developers commonly rely on Git's text-line-based conflict detection algorithm before resorting to manual conflict resolution, instead of choosing automatic merging approaches [21]. Furthermore, some recent methods, especially those using deep neural networks, lack publicly accessible tools for developers [20], presenting an implementation challenge for developers without the necessary training data.

This paper introduces an approach named CHATMERGE for conflict resolution. We observe that certain resolution strategies frequently arise during conflict resolution. Consider, for instance, two code modifications, $\mathcal{A}$ and $\mathcal{B}$, both made relative to a common base $\mathcal{O}$. If both $\mathcal{A}$ and $\mathcal{B}$ alter the same line of code, a conflict ensues. Various strategies can resolve this conflict, such as preserving $\mathcal{A}$, keeping $\mathcal{B}$, or concatenating $\mathcal{A}$ and $\mathcal{B}$, among others (detailed in Section 2). If a conflict's resolution strategy can be predetermined, it signifies a successful resolution of at least half of the conflict. This is because certain strategies, such as adopting the version from $\mathcal{A}$ directly, result in straightforward resolutions. However, generating resolutions for more complex strategies (like combining code from $\mathcal{A}$ and $\mathcal{B}$ or introducing new code) poses a greater challenge. Fortunately, numerous historical merge conflict resolutions are available for reference on open-source community platforms. By identifying similar merge conflicts and utilizing LLMs like `ChatGPT`[1], resolutions can be generated. The primary idea of CHATMERGE stems from this. We first predict merge conflict resolution strategies using historical merge conflict data from GitHub. Based on the predicted strategies, different approaches are employed. If a strategy aligns with an existing version that can resolve the conflict, this version is directly applied. For other strategies that require new or combined code, we rely on LLMs and resolutions from similar merge conflicts to generate solutions.

To develop CHATMERGE, we gathered a dataset comprising 102,455 conflict blocks and their respective manual resolutions, derived from 1,174 GitHub repositories. Drawing from previous studies [3] and our own analysis, we categorized the resolution strategies into seven distinct types. To predict the strategies for merge conflict resolution, we encoded the raw conflicts and corresponding edit sequences, using them as input for a Random Forest classifier. For more complex strategies, we adopted a prompt-based approach, selecting merge conflicts bearing code similarities from GitHub and using them as prompts to engage `ChatGPT` via API for resolution generation. Our experimental findings highlight the remarkable performance of CHATMERGE in contrast to other available merging tools. CHATMERGE successfully resolved a substantial number of merge conflicts (97.7% out of 10,000) and achieved a high precision and accuracy rate (63.5% and 65.0%) by aligning with historical manual resolutions. Additionally, we assessed different classification algorithms and examined the impact of prompt construction on the resolutions generated by `ChatGPT`. Results revealed that the chosen Random Forest classification algorithm performs better and the prompt utilized by CHATMERGE also produces desired resolution outcomes.

In summary, this paper makes the following contributions:

- We propose a two-stage approach for resolving merge conflicts. In this approach, an initial phase determines the appropriate resolution strategies for conflicts. Following this, an LLM is employed to generate resolutions for conflicts requiring complex elimination strategies.
- We conducted extensive experiments to evaluate our proposed approach through comparisons with publicly accessible merge conflict resolution tools using a large dataset, and the results demonstrate the superior performance of our approach in generating resolutions.
- We support the wider research community by publicly releasing our merge conflict resolution tool and associated training and testing datasets[2]. This empowers developers and fosters continued exploration in this field.

Paper organization. The remainder of this paper is organized as follows. Section 2 provides some background knowledge. Section 3 presents the proposed approach in detail. Section 4 gives the evaluation settings and presents the experimental results. Section 5 discusses threats to validity. Finally, related work and conclusion are given in section 6 and section 7, respectively.

2. BACKGROUND

In this section, we provide some background knowledge on merge conflicts and their resolutions in Git, as well as an introduction to LLMs.

2.1. Merge Conflicts in Git

Git generally employs the `diff3` algorithm for unstructured, text-line-based merging via the `git merge` command, also known as a recursive three-way merge. This process treats source code files as lines of text, overlooking structural and semantic information, and simply compares two versions on a line-by-line basis. Specifically, given two modifications $\mathcal{A}$ and $\mathcal{B}$ relative to a common base $\mathcal{O}$, the `diff3` algorithm initiates a two-way diff by aligning $\mathcal{A}$ (or $\mathcal{B}$) with $\mathcal{O}$ at the line level. It then temporarily stores these modification records. These records guide merge decisions, typically preserving changes from either $\mathcal{A}$ or $\mathcal{B}$. However, in instances where both $\mathcal{A}$ and $\mathcal{B}$ alter the same line, an automatic reconciliation is not feasible. Here, the `diff3` algorithm generates a conflict file, requiring manual intervention. The conflict contents, termed a `conflict block` (the portion between "`<<<<<<<`" and "`>>>>>>>`"), necessitate developer review and resolution. The `conflict block` is then replaced with this resolution. The integration process is not deemed complete until all `diff3` conflict files are successfully committed using the `git`

[1]https://chat.openai.com

[2]https://doi.org/10.5281/zenodo.8217581

`commit` command. Each merge conflict can be viewed as the result of two code modifications. In our work, we utilize Git commands to replay merge processes and collect `conflict blocks` from `diff3` conflict files.

2.2. Strategies for Manual Resolution of Merge Conflicts

Currently, merge conflicts identified by Git require manual resolution by developers. A study conducted by Ghiotto et al. [3] revealed that, in 81% of cases, conflicts are resolved exclusively using the content within the conflict block, without any modifications. Subsequent research [19], [20], [22] has validated that developers typically follow certain patterns when resolving conflicts. These patterns can be broadly classified into several types, including adopting a specific version, discarding all versions, merging both modified versions entirely (in any order), interleaving lines of code from both versions, and merging existing code lines with newly introduced ones. Among these resolutions, preserving one modified version is the most prevalent strategy, accounting for approximately 60% of cases. Introducing new code is also common, constituting around 15% to 20% of cases. Building on previous research, we further classify manual resolution strategies into seven distinct categories at the code line level, covering all conceivable scenarios. We believe this classification will be beneficial for our subsequent automatic resolution generation. For the remainder of this paper, we will refer to these strategies as: adopting *version* $\mathcal{O}$ in resolution (O), adopting *version* $\mathcal{A}$ in resolution (A), adopting *version* $\mathcal{B}$ in resolution (B), using concatenation of *version* $\mathcal{A}$ *and version* $\mathcal{B}$ in resolution (AB) or its reverse (BA), *none* (NN, which refers to deleting the conflict block during resolution), and *other* (OT). The OT strategy includes combining different lines of code from $\mathcal{A}$ and $\mathcal{B}$, as well as introducing new code. Based on the complexity of generating resolutions corresponding to these strategies, we categorize all strategies other than OT as simple, while OT is considered a complex strategy.

2.3. Large Pre-trained Language Models

LLMs refer to robust artificial intelligence systems designed to comprehend and generate human-like text outputs that are both logical and coherent. Typically, they utilize the Transformer architecture [23], which consists of an encoder for producing an encoded representation of the input and a decoder for generating output tokens. These LLMs are pre-trained on billions of text documents, enabling them to learn language structures, grammar rules, and contextual meanings, thereby equipping them to handle a wide range of Natural Language Processing (NLP) tasks [24], [25]. Due to extensive pre-training on diverse data, LLMs can be directly applied to specialized tasks without the need for further fine-tuning. This is typically achieved through prompt engineering [26], a method where specific instructions, or prompts, are given to an LLM to guide its responses or enhance its capabilities. It has been demonstrated that directly leveraging LLMs through prompts can yield exceptional performance [24]. Researchers have further shown that LLMs can excel in some code-related tasks, such as code completion [27], program synthesis [28], and program repair [29]. This success can be attributed to the naturalness of code [30]. In our approach, we utilize a popular LLM, `ChatGPT`, to generate merge conflict resolutions. `ChatGPT` has garnered significant interest across various domains, with tasks ranging from answering questions to generating code snippets [31]. We task `ChatGPT` with generation by supplying it with a prompt that is explicit, guiding, and suggestive.

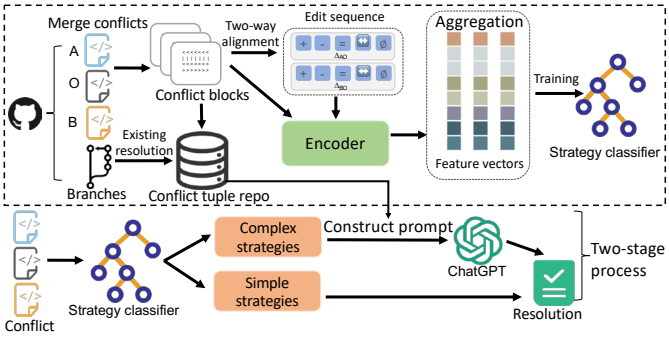

Figure 1. Overview of ChatMerge.

3. Approach

This section provides a detailed explanation of our approach for generating merge conflict resolutions, beginning with an overview of ChatMerge.

3.1. Approach Overview

Figure 1 provides an overview of ChatMerge, an approach devised to produce merge conflict resolutions by predicting resolution strategies and utilizing `ChatGPT` for cases requiring complex strategies. ChatMerge begins by extracting real-world merge conflicts from GitHub commit data and simultaneously mines the corresponding manual resolutions based on branch relationships. The gathered conflict blocks and their associated resolutions compile a conflict tuple repository that offers detailed information on existing merge conflicts, including resolution strategies and the corresponding code changes. To train a classifier, we preprocess the training data, where each conflict arises from two code modifications, $\mathcal{A}$ and $\mathcal{B}$, both relative to a common base $\mathcal{O}$. We execute a two-way alignment between $\mathcal{A}$ (resp. $\mathcal{B}$) and $\mathcal{O}$, generating two edit sequences. After tokenization and token alignment, all conflict and edit sequence information is fed into different encoders (c.f. Section 3.3 for more details) to obtain feature vectors. We then aggregate the obtained features with a label representing the resolution strategy to train the classifier. The training process of the strategy classifier is illustrated within the dashed box at the top of Figure 1. The lower portion shown in the figure outlines the tool's operational process. For a given conflict, ChatMerge utilizes a two-stage process to generate the resolution. Firstly, the trained classifier predicts the resolution strategy. For simple strategies (O, A, B, AB, BA, NN), the resolution involves direct reservations or concatenations of $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{O}$. If the strategy involves combining existing code

Algorithm 1: Manual Resolutions Localization

Input: Conflict file $\mathcal{C}$ and resolved file $\mathcal{M}$
Output: Resolution $\mathcal{R}$ for the conflict

```
1  Procedure LocalizeResRegion(C, M, i)
2      (spos, epos) ← GetConflictStartEnd(C, i)
       prfx ← ⟨BOF⟩ + C[0 : spos]
       sffx ← C[epos : Length(C)] + ⟨EOF⟩
3      if ExistAdjacentConflict(C, i) then
4          prfx ←
             GreedySearch(reverse(prfx), reverse(M))
           sffx ← GreedySearch(sffx, M))
5      end
6      (s, _) ←
         MinimalUniquePrefix(reverse(prfx), reverse(M))
7      (e, _) ← MinimalUniquePrefix(sffx, M)
8      if s ≥ 0 and e ≥ 0 then
9          R = M[Length(M) − s : e]
10         return R
11     else
12         return nil
13     end
14 end
15 Procedure GreedySearch(sffx, M)
       // replace the adjacent conflict
          block with one code snippet
16     (sffx_o, sffx_a, sffx_b) ← SelectOne(sffx)
17     (_, len_o) ← MinimalUniquePrefix(sffx_o, M)
18     (_, len_a) ← MinimalUniquePrefix(sffx_a, M)
19     (_, len_b) ← MinimalUniquePrefix(sffx_b, M)
20     return sffx_x ▷ sffx with the max len
21 end
22 Procedure MinimalUniquePrefix(x, y)
23     Return: the start position of the minimal non-empty
         prefix of x that appears uniquely in y, and the
         length of the prefix, else (-1, -1)
24 end
```

lines or introducing new code (OT), CHATMERGE advances to the second stage. Here, similar conflicts from the tuple repository are retrieved and used to construct a prompt for `ChatGPT`, which generates the corresponding resolution.

3.2. Data Collection and Preprocessing

To train the strategy classifier for merge conflicts, CHATMERGE begins by collecting relevant data from GitHub projects. We select these projects based on a commonly-used list of 2,731 Java projects provided by Ghiotto et al. [3], ensuring the quality of our dataset. To minimize bias and enhance the chances of identifying significant conflicts, we impose the following criteria: each project must have more than 100 commits and 10 stars, involve at least 10 developers, and not be a fork of another project. Upon applying these filters, we curate a dataset from 1,174 original repositories. We proceed by simulating the `git merge` operation on the two parent commits of each commit, aiming to detect any arising conflicts. Whenever a conflict emerges, we extract the conflict blocks delineated by standard `diff3` markers. Through this process, we collect a total of 102,455 real-world merge conflicts. To train the strategy classifier, it is vital to examine the commit data to understand how developers manually resolved these conflicts. Nonetheless, the extraction of merge conflict resolution from historical commit data is not a simple task. In response to this difficulty, Dinella et al. [19] proposed an approach to overcome this challenge, which we adopt in our approach.

The main idea of their approach is to extract prefixes and suffixes for each `conflict block` within the `diff3` conflict file and to locate these elements in an unambiguous manner within the resolved file. By pinpointing both the prefix and suffix within the resolved file, a `conflict block` can be mapped to its resolution. This approach, however, exhibits limitations, particularly when adjacent conflict blocks lack distinct prefixes or suffixes, which often occurs. To address this limitation, we propose certain enhancements. Algorithm 1 demonstrates our improved algorithm. This algorithm accepts as inputs a conflict file $\mathcal{C}$ with markers and the resolved file $\mathcal{M}$, and outputs the conflict's resolution $\mathcal{R}$ extracted from $\mathcal{M}$. The underlying principle of this algorithm is congruent with Dinella et al.'s approach, involving the extraction of the prefix ($prfx$) and suffix ($sffx$) for each `conflict block`, which are then matched in $\mathcal{M}$ utilizing the `MinimalUniquePrefix` procedure. However, our enhancements lie in the introduction of the `GreedySearch` procedure (lines 15-21) to handle scenarios where prefixes or suffixes are absent due to adjacent conflict blocks. Specifically, when confronted with adjacent conflict blocks (lines 3-5), we replace them with $\mathcal{O}$, $\mathcal{A}$, or $\mathcal{B}$, selecting the one with the longest match (using `MinimalUniquePrefix` for comparisons). These enhancements equip the algorithm with the capability to extract resolutions with enhanced accuracy.

After extracting the resolutions for the merge conflicts, we organize each conflict and its respective resolution into what we refer to as a conflict tuple, to be stored for future use. Specifically, each conflict tuple CT is structured as $(\mathcal{O}, \mathcal{A}, \mathcal{B}, \mathcal{R})$, where $\mathcal{A}$ and $\mathcal{B}$ represent the code modifications applied to the same code base $\mathcal{O}$, i.e., $\mathcal{A}$ and $\mathcal{B}$ modify the same line of code. $\mathcal{R}$ signifies the resolution for this merge conflict. Finally, we have extracted a dataset comprising 102,455 real-world merge conflicts along with their corresponding resolutions, and stored them as conflict tuples in a conflict tuple repository.

3.3. Feature Extraction and Representation

Edit Sequence. To train an effective strategy classifier, it is necessary to extract informative features from the raw merge conflict data. For a conflict tuple CT, our classifier aims to predict the most fitting resolution strategy, taking into consideration not just the two differing code modifications $\mathcal{A}$ and $\mathcal{B}$, but also their shared original code base, $\mathcal{O}$. Clearly, the distinctions among $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{O}$ represent crucial features of the merge conflict, which aligns with practical experience and is supported by existing related work [19], [20], [32].

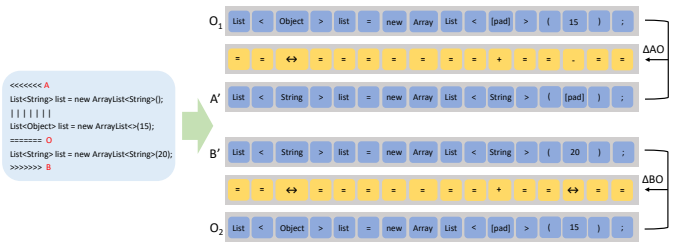

Figure 2. An Example of Edit Sequence.

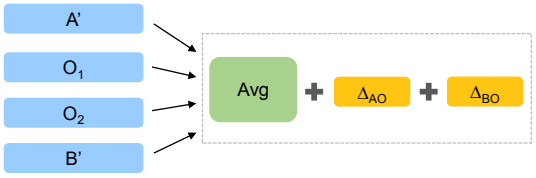

Figure 3. The Aggregation Process of Vectors.

To quantify the differences between $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{O}$, we adopt a concept from prior studies [19] referred to as the *Edit Sequence*, which outlines the modifications transitioning from $\mathcal{O}$ to either $\mathcal{A}$ or $\mathcal{B}$. An edit sequence includes five edit actions [20]: **+** to denote insertions, **-** to signify deletions, **=** for unchanged tokens, $\leftrightarrow$ for replacements, and $\emptyset$ as a padding token. These edit actions, when imposed on a sequence, result in a new sequence [19]. In order to generate the edit sequences between $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{O}$, we align $\mathcal{A}$ (or $\mathcal{B}$) and $\mathcal{O}$ in a two-way manner, thereby creating two edit sequences that encapsulate the modifications. Specifically, we employ a pre-trained tokenizer [33] to transform each code snippet into a token sequence, aligning corresponding pairs by inserting padding symbols to ensure that equivalent tokens align. As a result, we obtain two pairs of aligned sequences ($\mathcal{A}', \mathcal{O}_1$ and $\mathcal{B}', \mathcal{O}_2$), and the two edit sequences (ΔAO and ΔBO) are generated by comparing the aligned tokens. ΔAO and ΔBO represent the edit sequences between $\mathcal{A}'$ and $\mathcal{O}_1$, and between $\mathcal{B}'$ and $\mathcal{O}_2$, respectively. Figure 2 presents an example, where the left side shows the original $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{O}$ code sequences, and the right side displays the aligned sequences $\mathcal{A}'$, $\mathcal{O}_1$, $\mathcal{B}'$, and $\mathcal{O}_2$ after processing. The yellow-highlighted portions represent the edit sequences, namely ΔAO and ΔBO. Besides the edit sequences, the code snippets $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{O}$ themselves also constitute important features. In the following, we will describe how these features are encoded.

Encoding. To encode the four types of aligned code sequences ($\mathcal{A}', \mathcal{O}_1, \mathcal{B}'$, and $\mathcal{O}_2$), we leverage `CodeBert` [33], which acts as the encoder for these sequences. For the two edit sequences, ΔAO and ΔBO, we employ a Bag-of-Words model. `CodeBert`, a Transformer-based neural architecture, has been pre-trained on multiple programming languages, enabling it to learn general-purpose representations that underpin a range of downstream applications such as code search and code documentation generation. To encode the four types of code sequences, we process each of them independently through `CodeBert`, extracting the embedding of the first token (`[CLS]`) in the last hidden state as the vector representation. Due to the limited and unrelated nature of edit types, we employ a Bag-of-Words model to encode the two edit sequences. This model tallies the occurrences of each edit type and expresses them as a vector, yielding an easily implemented, context-independent, and concise representation.

Aggregation. After encoding the code and edit sequences, we need to aggregate these vectors for subsequent training. Figure 3 illustrates our aggregation process for all vectors post-encoding. The comprehensive vector representation of the entire conflict is established by averaging the vector representations of $\mathcal{A}'$, $\mathcal{O}_1$, $\mathcal{B}'$, and $\mathcal{O}_2$. Subsequently, the vector representations of the edit sequences ΔAO and ΔBO are concatenated in order to this interim state from the final dimension, resulting in the final vector representation that is transferred to the subsequent classification model.

3.4. Strategy Classification

Upon encoding and aggregating the features of merge conflicts, alongside the labels indicating the resolution strategies employed, we utilize this data to train our strategy classifier. As introduced in Section 2, we have classified conflict resolution strategies into seven distinct categories. After the extraction of the manual resolution for each merge conflict, an automatic determination of the resolution strategy is made for each merge conflict through straightforward comparison. For example, if the code corresponding to a particular resolution matches the code in version $\mathcal{A}$, discounting any differences in whitespaces and indentation, then the resolution strategy for this scenario is assigned to category A. We assign appropriate resolution strategy labels to all merge conflicts in our dataset. To further ensure the robustness of our model and to avoid overfitting, we also conducted an in-depth analysis of the distribution of these strategies within our dataset to ascertain their alignment with real-world distribution patterns. Figure 4 displays the distribution of resolution strategies within our dataset. When contrasted with established empirical studies [3], [34] that have analyzed a broad spectrum of merge

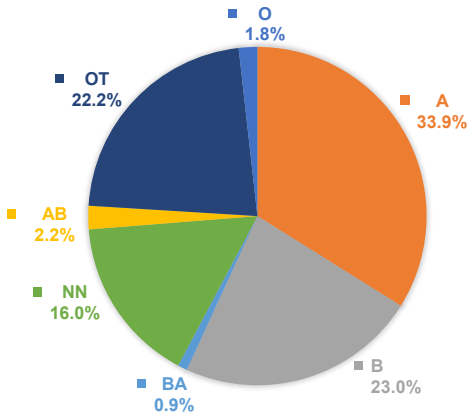

Figure 4. Distribution of Resolution Strategies in Our Dataset.

conflicts, we found a consistent alignment in the distribution of strategies within our dataset.

In our approach, a Random Forest classifier is utilized to determine the resolution strategy for each merge conflict. We allocated 80% of our dataset for training purposes and reserved the remaining 20% for testing. This strategy resulted in the classifier achieving an accuracy of 72.2%, a precision of 78.8%, and a recall of 61.0%. Furthermore, we explored the performance of different classification algorithms for comparative purposes (refer to Section 4.2).

3.5. Resolution Generation

Once the strategy classifier is trained, it can be applied to any given unresolved merge conflict to predict the optimal strategy for its resolution. Depending on the predicted strategy, we generate the corresponding resolution. For simple resolution strategies such as O, A, B, AB, BA, or NN, the generation process is straightforward, involving either the selection of the corresponding version or the concatenation of different versions. For example, if the predicted strategy for a conflict is A, CHATMERGE selects version $\mathcal{A}$ as the resolution. However, if the predicted resolution strategy is more intricate, represented by OT, CHATMERGE utilizes `ChatGPT` to generate the resolution. Here, the crucial aspect is the construction of an appropriate prompt for the model.

In our approach to prompt engineering, the design of the prompt follows these steps. First, we create a real-world scenario that a developer might face during collaborative software development, and present `ChatGPT` with a problem – a merge conflict. We then impose constraints on the output to ensure it includes all the necessary information. Following this, the conflict block that needs resolution is inserted. Given that `ChatGPT` has not previously encountered this type of problem, we provide it with several similar conflict tuples for learning and reference. These similar conflict tuples are selected from our repository of conflict tuples and share the same resolution strategy as the unresolved conflict. Given that the total length of the prompt must not exceed 4,096 tokens, we rank candidate tuples using cosine similarity and select them in descending order (the average number of similar conflict tuples included is 2.3). Figure 5 presents the template of the prompt used in CHATMERGE.

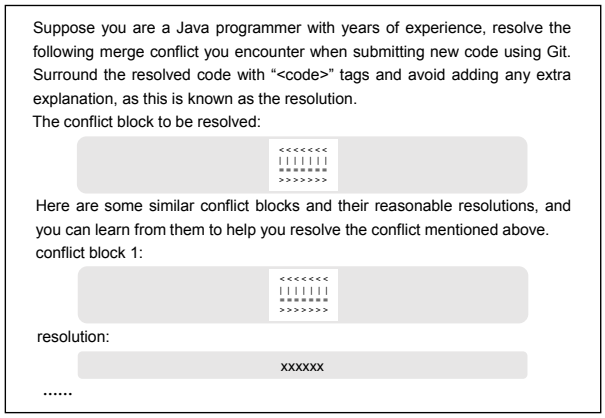

Figure 5. The Template of the Prompt Used in CHATMERGE.

4. EVALUATION

In this section, we present the evaluation of CHATMERGE and explore the following research questions:

- **RQ1: How does the efficacy of CHATMERGE compare with that of publicly available merge conflict resolution tools?** This research question seeks to evaluate the effectiveness of CHATMERGE by comparing its conflict resolution outputs with those of leading alternatives. We assess this by determining the precision and accuracy of how well each tool's generated resolutions match the manually resolved conflicts. Furthermore, to assess the efficiency of CHATMERGE, we measure the average time each tool requires to generate conflict resolutions.
- **RQ2: How does the choice of the classification algorithm affect the performance of CHATMERGE?** Currently, CHATMERGE utilizes the Random Forest algorithm for strategy classification. The intent behind this research question is to examine the potential impact of substituting the Random Forest algorithm with alternative classification algorithms on the effectiveness of the merge conflict resolutions generated by CHATMERGE. By conducting this investigation, our goal is to discern whether a different algorithm could improve CHATMERGE's performance, thereby affirming our rationale for selecting the specific classification algorithm.
- **RQ3: What is the impact of incorporating similar conflict tuples in the prompt on the quality of generated resolutions?** Our approach involves enriching the prompt with similar conflict tuples, thereby guiding `ChatGPT` in its learning process. The prompt thus plays a critical role in capitalizing on the potential of the Language Model. This research question is dedicated to investigating how much this enhancement augments the quality of the generated resolutions. To clarify this, we undertake an ablation study focusing specifically on the implications of embedding similar conflict tuples within the prompt.

Testing dataset and evaluation metrics. The primary goal of our experiments is to assess the quality of resolutions produced by CHATMERGE for merge conflicts. To do this, we need a collection of existing merge conflicts as experimental subjects. Following the method outlined in Section 3.2, we built a testing dataset of merge conflict tuples. This dataset includes specific conflict blocks for each merge conflict, along with the strategies and resolutions manually implemented by developers, serving as our ground truth. Our experimental dataset comprises 10,000 merge conflict tuples, none of which were utilized during CHATMERGE training process. To support future research in this field, we have made this experimental dataset publicly accessible online[2]. Moreover, to assess the quality of the resolutions generated by our tool and other comparative tools, we employ several widely used evaluation metrics derived from existing literature [20]. Along with these,

we also include an execution time metric to evaluate the efficiency of each tool. In the following, we present a succinct overview of the evaluation metrics utilized in our study.

- **Conflicts with Resolutions (conf. w/ res.):** This metric quantifies the percentage of conflicts for which the tool was able to generate a resolution.
- **Precision:** This metric calculates the percentage of generated resolutions that exactly match the manual resolutions of developers (disregarding whitespace and indentation differences).
- **Accuracy:** This metric assesses the percentage of generated resolutions that exactly match the manual resolutions, with respect to the total number of resolutions generated.
- **Time:** This metric provides the average time, in seconds, each tool takes to generate a resolution.

As a concrete example, suppose a tool generates a resolution for 50 out of 100 conflicts and 40 of these generated resolutions exactly match their corresponding manual resolutions. In this scenario, the "conf. w/ res." metric would be 50% (50 out of 100 conflicts have resolutions), while the "Precision" would be 40% (40 correct resolutions out of the total 100 conflicts). The "Accuracy", on the other hand, would be 80% (40 out of 50 generated resolutions are correct).

4.1. RQ1: Approach Efficacy

Baselines. In this RQ, we compare CHATMERGE's performance with existing approaches for the automatic resolution of merge conflicts. This is a topic that has gained significant attention [21], resulting in the development of various methods and tools. Among the publicly available options, we selected JDime [12] and jFSTMerge [14] as baseline approaches, given their ready availability and common usage in related studies [20], [16]. We excluded some other proposed methods from the comparison, such as [20], primarily due to disparities in resolution levels (some operating at the token level, while ours focuses on the line level) and replication difficulties stemming from undisclosed neural network-based training data and code. JDime is a Java-specific merging tool that utilizes a structured-merge approach with auto-tuning. It switches dynamically between unstructured and structured merge depending on the presence of conflicts. When operating in structured merge mode, JDime represents artifacts as context-free syntax trees and employs tree matching and amalgamation algorithms to enhance the precision of difference calculation and merging. We employed the publicly available implementation of JDime [35], running it in semi-structured mode. Contrastingly, jFSTMerge is an enhanced semi-structured merging tool designed to minimize false positives and false negatives during the merge process. It accomplishes this by updating the merged tree based on specific analyses conducted during tree construction. jFSTMerge is universally applicable and supports Java merging as well. We used its publicly available implementation [36] for our comparison.

Lastly, we compared CHATMERGE against `ChatGPT`, known for its superior performance across diverse tasks, including code generation, question answering, and task-oriented dialogue. To ensure a fair comparison, we used the same prompt template as shown in Figure 5, excluding similar conflict blocks. This comparison provides insights into the relative effectiveness of these different approaches for merge conflict resolution.

Results. Table I presents a comparison of the performance between CHATMERGE and other baseline approaches. Overall, CHATMERGE outperforms the other baselines considerably. In terms of conflicts with generated resolutions, CHATMERGE is capable of providing resolution recommendations for the vast majority of conflicts. The scenarios where CHATMERGE fails to generate resolutions primarily stem from difficulties in communication with `ChatGPT`, due to factors such as exceeding context limits, output format violations, or lost connections. While `ChatGPT` shows a reasonable performance in generating resolutions (though slightly lower than CHATMERGE), it too is constrained by these limitations, leading to similar generation failures. Moreover, despite `ChatGPT`'s competent handling of conflicts with resolutions, it falls considerably short of CHATMERGE in precision and accuracy. The available semi-structured baseline approaches are only able to generate resolutions for 34.6% and 18.8% of conflicts, respectively, as they struggle to handle complex contexts and often necessitate manual intervention. In terms of precision and accuracy, metrics reflecting the degree of alignment between generated and manual resolutions, CHATMERGE exhibits a substantial edge. Specifically, CHATMERGE demonstrates a precision of 63.5% and an accuracy of 65.0%, meaning it can generate resolutions that closely align with the ground truth. It is noteworthy that our calculations of precision and accuracy demand exact matches with the ground truth, underscoring the significance of these results. The success of CHATMERGE can largely be credited to the two-stage process it employs. Simple strategies that are directly manageable through preservation or concatenation are filtered out by the strategy classifier, while the more complex strategies are handled by feeding `ChatGPT` a carefully constructed prompt, thereby ensuring the generation of high-quality resolutions.

Table I also includes data on the average time taken by each tool to generate a resolution. From the data, we can observe that JDime takes the longest time, an issue primarily coming from the inherent drawbacks of structured merging. Even with optimizations such as switching between methods and utilizing context-free syntax trees, JDime still necessitates parsing the complete raw file to accomplish the merge process. As an improvement over structured merging, jFSTMerge belongs to the category of semi-structured merging methods and demonstrates significantly better performance, as it can directly tackle conflict blocks, resulting in less processing time. CHATMERGE, falls in the middle in terms of average time spent, generating a resolution in an average of 1.15 seconds. We consider this duration acceptable, especially compared to the time required for manual handling of conflicts. Our analysis indicates that a substantial portion of CHATMERGE's time is consumed by interactions with `ChatGPT`. The time

TABLE I
RESULTS OF CHATMERGE AND BASELINE APPROACHES.

Approach	conf. w/ res. (%)	Precision (%)	Accuracy (%)	Time (s)
JDime	34.6	10.2	29.5	4.14
jFSTMerge	18.8	12.5	66.4	0.65
ChatGPT	95.7	28.5	29.8	2.18
CHATMERGE	97.7	63.5	65.0	1.15

spent on `ChatGPT` is primarily due to network latency, which accounts for its position as the second most time-consuming among all the tools.

4.2. RQ2: Choice of Classification Algorithms

In the training phase of the strategy classifier in CHATMERGE, we utilized the Random Forest algorithm. However, there are multiple alternative classification algorithms available. Therefore, in this experimental setup, we explore six frequently used classification algorithms as potential replacements to Random Forest, namely: Naive Bayes, Support Vector Machine (SVM), Logistic Regression (LR), Decision Tree, XGBoost, and Neural Network (NN). To evaluate the performance of each algorithm on our strategy classification task, we employ four widely recognized metrics in classification tasks: accuracy, precision, recall, and F1-score [37]. Further, we investigate the quality of resolutions generated by our tool under the influence of these various classification algorithms. To accomplish this, we substitute the Random Forest algorithm with each of the alternative algorithms, training individual classifiers and integrating them into CHATMERGE, resulting in distinct variants for comparison. This training and testing of various classifiers are conducted on the identical dataset (as detailed in Section 3.2). The performance evaluation of these different tool variants is carried out using the previously mentioned evaluation dataset. Through this experimental setup, we aim to identify the optimal classification algorithm for our tool, facilitating more accurate and efficient merge conflict resolutions.

Results. Table II provides a comparative analysis of seven different classification algorithms on the merge conflict test set, with accuracy, precision, recall, and F1-score serving as evaluation metrics. These classifiers are tasked with predicting the resolution strategies for merge conflicts. As the table illustrates, the Random Forest algorithm, employed by CHATMERGE, outperforms the others across all metrics, followed by XGBoost and Decision Tree. However, the remaining classifiers fail to show satisfactory performance, thus affirming the rationality of our choice of the Random Forest algorithm for strategy classification. Once the resolution strategy for each merge conflict is determined, the subsequent step involves generating corresponding resolutions. To this end, we further compared the performance of different classification algorithms in the aspect of resolution generation. Table III displays the results of various versions of CHATMERGE, each using a different classification algorithm, in terms of conf. w/ res., precision, and accuracy. In this context, the Random Forest algorithm continues to showcase the best performance. Notably, despite significant variances in the classification performance of these classifiers, they all attain a conf. w/ res. value of over 97%. This high value results from CHATMERGE's two-stage generation process; even if the initial strategy prediction for a merge conflict is inaccurate (for instance, predicting strategy `AB` as `OT`), CHATMERGE will still generate a corresponding resolution. Thus, CHATMERGE generates resolutions for all provided conflicts, which contributes to the high conflicts with resolutions value. However, inaccurate strategy predictions can lead to erroneous resolutions, resulting in lower precision and accuracy for versions of CHATMERGE employing different classification algorithms. In conclusion, the Random Forest algorithm not only excels in classification results but also positively influences the quality of the final generated resolutions.

TABLE II
RESULTS OF DIFFERENT CLASSIFICATION ALGORITHMS

Algorithm	Accuracy	Precision	Recall	F1-Score
Naive Bayes	0.266	0.240	0.282	0.200
SVM	0.429	0.242	0.230	0.209
LR	0.463	0.330	0.257	0.258
NN	0.588	0.551	0.431	0.447
Decision Tree	0.668	0.592	0.590	0.591
Random Forest	**0.722**	**0.788**	**0.610**	**0.668**
XGBoost	0.687	0.725	0.597	0.642

TABLE III
PERFORMANCE OF DIFFERENT VERSIONS OF CHATMERGE.

Version	conf. w/ res. (%)	Precision (%)	Accuracy (%)
Naive Bayes	97.3	28.1	28.9
SVM	97.7	47.8	48.9
LR	97.6	47.7	48.9
NN	97.4	54.9	56.4
Decision Tree	97.6	58.3	59.8
Random Forest	**97.7**	**63.5**	**65.0**
XGBoost	97.5	61.5	63.0

4.3. RQ3: Impact of Similar Conflict Tuples

A well-designed prompt is essential as it guides `ChatGPT` and influences its output. It provides the necessary instructions and context, steering the conversation in a way that allows ChatGPT to generate more precise resolutions for merge conflicts. The quality of the prompt directly influences ChatGPT's ability to produce desired responses. In our approach, we enrich the prompt with similar conflict tuples to enhance `ChatGPT`'s understanding and, thus, its output generation. However, this enrichment process introduces an overhead, and it is crucial to assess its actual contribution to the final result. For this purpose, we conduct an ablation study wherein similar conflict tuples are individually removed from the prompt to gauge their influence. Specifically, we compare the resolutions generated by two versions of CHATMERGE (one that includes

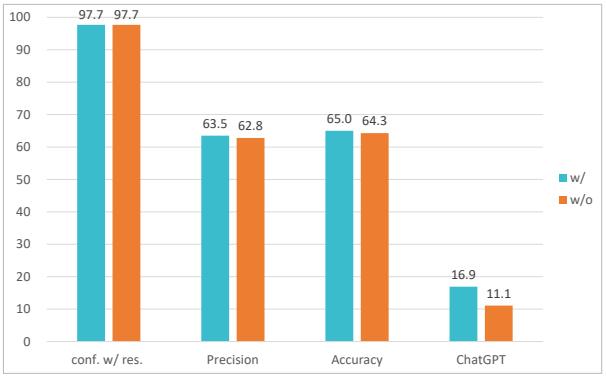

Figure 6. Results: With and Without Similar Conflict Tuples in Prompts.

similar conflict tuples in the prompt and one that does not), focusing on precision and accuracy metrics. This evaluation is conducted using the previously described evaluation dataset. Through this study, we aim to quantify the impact of including similar conflict tuples in the prompt on the quality of the generated resolutions.

Results. Figure 6 compares the performance of two versions of CHATMERGE, one incorporating similar conflict tuples in its prompt while the other does not. As demonstrated in the figure, both versions generate almost the same quantity of conflict resolutions, as there are no significant differences in the initial stages prior to the resolution generation, and the two-stage generation process guarantees the generation of possible resolutions. Despite the marginal divergence in precision and accuracy as depicted in the bar chart, it is noteworthy that the number of conflicts classified under the strategy OT is relatively small, with only 1,412 instances out of the validation dataset in our experiment. The two versions of CHATMERGE primarily differ in the handling of conflicts utilizing strategy OT, while the processing of conflicts using other strategies remains the same. To emphasize the difference, we further examined these 1,412 conflicts and their corresponding resolutions generated by `ChatGPT`. The percentage of resolutions that match manual ones stands at 16.9% and 11.1% for the two versions of CHATMERGE, respectively. This suggests that enriching the prompt with similar conflict tuples positively influences the result. No substantial time differences between the two versions were observed, hence this result is not displayed in the figure. Due to `ChatGPT`'s text length limitation, we did not include many similar conflict tuples in the prompt, averaging around 2.3 tuples per prompt. However, as shown, even with the inclusion of approximately two similar conflict tuples in the prompt, a clear divergence in results is visible. We believe that the performance could further improve if the model supported more similar conflict tuples as input or if we could sequentially input similar conflict tuples.

5. THREATS TO VALIDITY

In this section, we discuss threats to the validity of our work.

Threats to internal validity. This kind of threats refers to factors that might have influenced the results of a study or experiment, excluding the experimental factors themselves. In the context of our research, the primary threat to internal validity may arise from the quality of our dataset. The data for the projects were initially assembled by Ghiotto et al. [3], and we constructed conflict tuples following existing studies [3], [19]. To ensure high-quality conflicts with precise manual resolutions, we applied constraints to each project and improved the algorithm for extracting manual resolutions. This robust procedure laid a solid foundation for our subsequent work. Moreover, the accuracy of the classifier used in our approach to determine the resolution strategy significantly impacts the final result. To dispel any doubt regarding our selection of a specific classification algorithm for strategy classification, we conducted a comparative performance evaluation of multiple classification algorithms on a set of metrics. This comprehensive assessment validated our choice of the Random Forest algorithm. We also performed an ablation study to confirm the usefulness of the prompt structure provided to ChatGPT.

Threats to external validity. This kind of threats relates to the degree to which the findings of a research study can be generalized to other contexts. In relation to our study, one of the threats to external validity lies in the specificity of the program language tested. While our study exclusively evaluated Java, it should be noted that our approach is not language-specific and is not limited by grammar rules. Furthermore, the CodeBert encoder used in our tool has been demonstrated to effectively capture code information in various downstream tasks. Therefore, we expect that our tool's performance would be comparably effective with other programming languages. Meanwhile, we acknowledge that our assessment was only conducted on a select number of projects. To bolster the generalizability of our approach, there is a need for more extensive experimentation on a broader range of projects. An additional potential threat to external validity is the changing nature of software development practices and tools. Over time, the prevalence and nature of merge conflicts, and the strategies developers use to resolve them, could evolve due to factors such as new development methodologies, changes in programming languages, or advancements in IDEs and other software development tools. Consequently, the relevance and effectiveness of our tool may vary with these changes. Thus, continuous evaluation and updates may be necessary to ensure the ongoing efficacy of our tool in diverse and evolving software development contexts.

6. RELATED WORK

In collaborative software development, merge conflicts are a recurring obstacle, a fact that has led to substantial research in this area. Many studies have delved into the exploration of merge conflicts, scrutinizing their prevalence [38], [3], distinct characteristics [39], [21], [22], [40], and strategies for their prevention [41], [42], [43]. These studies underline the widespread presence of merge conflicts and their potentially harmful impact, along with shedding light on the factors that contribute to their occurrence. For instance, Ghiotto et al.

highlighted that conflicts ensue in 10 to 20 percent of all merge attempts [3], whereas Menezes et al. [39] discovered a significant correlation between the emergence of merge conflicts and changes in the files, commits, and committers in the branch under integration. To reduce the number of merge conflicts, one tactic is to predict and prevent their occurrence. A variety of tools, such as CloudStudio [44], CoDesign [45], Palantir [46], Syde [47], and WeCode [48] have been proposed, each designed to alert developers about parallel work and potential conflicts early by monitoring changes in each workspace. Another tactic to tackle merge conflicts is through conflict resolution. This approach is often more direct and appealing, as it allows developers to concentrate on their current tasks without needing to monitor each modification constantly. This leads to improved development efficiency and reduces tension in the development process. The focus of our work is within this latter category of merge conflict resolution.

In the pursuit of resolving merge conflicts, a variety of techniques have been developed. Early merging approaches, such as those by [9], [49], [50], are unstructured and utilize lines of code as the unit of comparison and resolution, making them universally applicable for any textual file. However, these approaches can lead to incomplete or incorrect resolutions due to their inherent limitations. To address these deficiencies, structured merging algorithms have been explored, which leverage the syntactic or semantic information of code snippets [51], [52], [53]. Although these approaches outperform their unstructured counterparts in terms of resolution precision, their language-specific attributes and high computational costs have curtailed their widespread adoption. Recognizing the strengths and weaknesses of both approaches, semi-structured merging strategies were proposed, like FSTMerge by Apel et al. [11], which toggles between different methods. Further advancements were made to strike a balance between precision and performance [12], [54]. In addition, JSFSTMerge adapted an off-the-shelf grammar for JavaScript [15], and jFSTMerge by Cavalcanti et al. included well-designed handlers to reduce false positives and false negatives [14]. Moreover, alternative methods have emerged which address conflict resolution from differing perspectives. Shen et al. [55], for instance, harness the interconnectedness among conflicts to provide interactive and systematic assistance for manual resolution of merge conflicts. IntelliMerge [16] employs a graph-based three-way merging technique that exhibits an enhanced capability for dealing with refactoring changes or other modifications resulting in mismatches.

The integration of machine learning techniques into merge conflict resolution has been a significant recent advancement. For instance, Dinella et al. modeled merge conflict resolution as a supervised sequence-to-sequence learning problem, developing DeepMerge in the process [19]. Similarly, Svyatkovskiy et al. treated merge conflict resolution as a sequence classification task, extracting conflicting regions and context through token-level differencing. They introduced MergeBert, a merge tool based on the bidirectional transformer encoder model [20]. While they achieve promising results, these methods are unable to generate new tokens or facilitate flexible combinations. Our approach provides a unique perspective on conflict resolution, employing a two-stage process. First, we use a strategy classifier to predict the appropriate resolution strategy for merge conflicts. We observed that not all conflicts require complex machine learning techniques and can be resolved by adopting specific versions. For complex conflicts, we employ advanced LLMs in the second stage to assist in resolution generation. MergeBert [20] also utilizes a classification approach to decide conflict resolution patterns but cannot generate new tokens. Moreover, it operates at the token level, addressing one token at a time, and assumes independence among different token-level conflicts, overlooking cases where a single line-level conflict may involve multiple token-level conflicts dependent on each other's content. Our approach integrates the advantages of both classification and generative techniques. It identifies conflicts requiring only simple strategies through classification, enabling their direct resolution. For conflicts demanding complex solutions, our approach restricts the application of machine learning to essential instances, thereby optimizing computations and enabling the generation of new code when necessary. Furthermore, we provide an openly accessible tool to support developers and encourage further research.

7. Conclusion

In modern software development, collaborative efforts have become the predominant approach, allowing multiple developers to work simultaneously in independent branches and integrate their contributions into the mainline. Within this parallel development process, merging is a critical and frequent operation. However, the occurrence of merge conflicts presents challenges when multiple developers make concurrent changes to the same code. To address this issue, we propose a two-stage approach called ChatMerge for generating merge conflict resolutions. This approach first determines appropriate resolution strategies for conflicts and then employs two different methods to generate corresponding resolutions based on the complexity of the strategies: a straightforward approach for simple resolution strategies and an LLM for more complicated ones. Our evaluation of ChatMerge demonstrates its effectiveness, as it can generate resolutions for 97.7% of conflicts with a commendable precision of 63.5% and accuracy of 65.0%, surpassing the performance of existing alternatives. Experimental results also highlight the rationale behind selecting the Random Forest classification algorithm and the usefulness of the enriched prompt adopted in ChatMerge. Furthermore, the language-independent nature of ChatMerge allows easy extension to other programming languages.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Nos. 61972193, 61972197), the Natural Science Foundation of Jiangsu Province (No. BK20201292), and the Collaborative Innovation Center of Novel Software Technology and Industrialization.

REFERENCES

[1] C. Walrad and D. Strom, "The importance of branching models in scm," *Computer*, vol. 35, no. 9, pp. 31–38, 2002.

[2] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *2013 35th Int. Conf. Softw. Eng.*, pp. 732–741, 2013.

[3] G. Ghiotto, L. Murta, M. Barros, and A. van der Hoek, "On the nature of merge conflicts: A study of 2,731 open source java projects hosted by github," *IEEE Trans. Soft. Eng.*, vol. 46, no. 8, pp. 892–915, 2020.

[4] W. Yang, C. Zhang, M. Pan, C. Xu, Y. Zhou, and Z. Huang, "Do developers really know how to use git commands? a large-scale study using stack overflow," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, apr 2022.

[5] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, Association for Computing Machinery, 2012.

[6] H. V. Nguyen, M. H. Nguyen, S. C. Dang, C. Kästner, and T. N. Nguyen, "Detecting semantic merge conflicts with variability-aware execution," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, p. 926–929, Association for Computing Machinery, 2015.

[7] L. D. Silva, P. Borba, W. Mahmood, T. Berger, and J. Moisakis, "Detecting semantic conflicts via automated behavior change detection," in *IEEE Int. Conf. Softw. Maintenance Evol.*, pp. 174–184, 2020.

[8] T. Wuensche, A. Andrzejak, and S. Schwedes, "Detecting higher-order merge conflicts in large software projects," in *IEEE 13th Int. Conf. Softw. Testing Validation Verification*, pp. 353–363, 2020.

[9] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proc. 3rd int. workshop Softw. config. manag.*, pp. 68–79, 1991.

[10] F. Zhu and F. He, "Conflict resolution for structured merge via version space algebra," *Proc. ACM Program. Lang.*, vol. 2, oct 2018.

[11] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: Rethinking merge in revision control systems," in *Proc. 19th ACM SIGSOFT Symp. 13th European Conf. Found. Softw. Eng.*, p. 190–200, Association for Computing Machinery, 2011.

[12] S. Apel, O. Leßenich, and C. Lengauer, "Structured merge with auto-tuning: balancing precision and performance," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.*, pp. 120–129, 2012.

[13] S. Apel, J. Liebig, C. Lengauer, C. Kästner, and W. R. Cook, "Semistructured merge in revision control systems.," in *VaMoS*, pp. 13–19, 2010.

[14] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proc. ACM Program. Lang.*, vol. 1, oct 2017.

[15] A. Trindade Tavares, P. Borba, G. Cavalcanti, and S. Soares, "Semistructured merge in javascript systems," in *34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, pp. 1014–1025, 2019.

[16] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "Intellimerge: A refactoring-aware software merging technique," *Proc. ACM Program. Lang.*, vol. 3, oct 2019.

[17] M. Ellis, S. Nadi, and D. Dig, "Operation-based refactoring-aware merging: An empirical evaluation," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2698–2721, 2023.

[18] J. Zhang, T. Mytkowicz, M. Kaufman, R. Piskac, and S. K. Lahiri, "Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper)," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, p. 77–88, Association for Computing Machinery, 2022.

[19] E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, and S. Lahiri, "Deepmerge: Learning to merge programs," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 1599–1614, 2023.

[20] A. Svyatkovskiy, S. Fakhoury, N. Ghorbani, T. Mytkowicz, E. Dinella, C. Bird, J. Jang, N. Sundaresan, and S. K. Lahiri, "Program merge conflict resolution via neural transformers," in *Proc. 30th ACM Joint European Softw. Eng. Conf. Symp. Found. Softw. Eng.*, p. 822–833, Association for Computing Machinery, 2022.

[21] N. Nelson, C. Brindescu, S. McKee, A. Sarma, and D. Dig, "The life-cycle of merge conflicts: processes, barriers, and strategies," *Empirical Software Engineering*, vol. 24, pp. 2863–2906, 2019.

[22] H. L. Nguyen and C.-L. Ignat, "An analysis of merge conflicts and resolutions in git-based open source projects," *Computer Supported Cooperative Work*, vol. 27, pp. 741–765, 2018.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017.

[24] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[25] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.

[26] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Comput. Surv.*, vol. 55, jan 2023.

[27] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph,

G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[28] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021.

[29] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *Proc. 30th ACM Joint European Softw. Eng. Conf. Symp. Found. Softw. Eng.*, p. 959–971, 2022.

[30] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, p. 122–131, apr 2016.

[31] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," 2023.

[32] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," *arXiv preprint arXiv:1810.13337*, 2018.

[33] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.

[34] S. L. H. Gonzalez and P. Fraternali, "Almost rerere: Learning to resolve conflicts in distributed projects," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2255–2271, 2023.

[35] JDime, "Jdime public implementation," 2023.

[36] jFSTMerge, "jfstmerge public implementation," 2023.

[37] J. D. Novaković, A. Veljović, S. S. Ilić, Ž. Papić, and M. Tomović, "Evaluation of classification models in machine learning," *Theory and Applications of Mathematics & Computer Science*, vol. 7, no. 1, p. 39, 2017.

[38] T. Mens, "A state-of-the-art survey on software merging," *IEEE Trans. Softw. Eng.*, vol. 28, no. 5, pp. 449–462, 2002.

[39] J. W. Menezes, B. Trindade, J. a. F. Pimentel, T. Moura, A. Plastino, L. Murta, and C. Costa, "What causes merge conflicts?," in *Proc. XXXIV Brazilian Symp. Softw. Eng.*, p. 203–212, Association for Computing Machinery, 2020.

[40] M. Mahmoudi, S. Nadi, and N. Tsantalis, "Are refactorings to blame? an empirical study of refactorings in merge conflicts," in *IEEE 26th Int. Conf. Softw. Anal. Evol. and Reeng.*, pp. 151–162, 2019.

[41] M. Owhadi-Kareshk, S. Nadi, and J. Rubin, "Predicting merge conflicts in collaborative software development," in *ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, pp. 1–11, 2019.

[42] P. Accioly, P. Borba, L. Silva, and G. Cavalcanti, "Analyzing conflict predictors in open-source java projects," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, p. 576–586, Association for Computing Machinery, 2018.

[43] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proc. 19th ACM SIGSOFT Symp. 13th European Conf. Found. Softw. Eng.*, p. 168–178, Association for Computing Machinery, 2011.

[44] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer, "Unifying configuration management with merge conflict detection and awareness systems," in *22nd Australian Softw. Eng. Conf.*, pp. 201–210, 2013.

[45] J. y. Bang, D. Popescu, G. Edwards, N. Medvidovic, N. Kulkarni, G. M. Rama, and S. Padmanabhuni, "Codesign: A highly extensible collaborative software modeling framework," p. 243–246, Association for Computing Machinery, 2010.

[46] A. Sarma, D. F. Redmiles, and A. van der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Trans. Softw. Eng.*, vol. 38, no. 4, pp. 889–908, 2012.

[47] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, p. 235–238, Association for Computing Machinery, 2010.

[48] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *34th Int. Conf. Softw. Eng.*, pp. 342–352, 2012.

[49] T. Berlage and A. Genau, "A framework for shared applications with a replicated architecture," in *Proc. 6th annual ACM symp. User interface softw. technology*, pp. 249–257, 1993.

[50] G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for p2p collaborative editing," in *Proc. 20th Anniversary Conf. Computer Supported Cooperative Work*, p. 259–268, Association for Computing Machinery, 2006.

[51] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Jdiff: A differencing technique and tool for object-oriented programs," *Automated Software Engineering*, vol. 14, pp. 3–36, 2007.

[52] J. Hunt and W. Tichy, "Extensible language-aware merging," in *International Conference on Software Maintenance, 2002. Proceedings.*, pp. 511–520, 2002.

[53] H. Shen and C. Sun, "Syntax-based reconciliation for asynchronous collaborative writing," in *Int. Conf. Collaborative Computing: Networking, Applications and Worksharing*, pp. 10 pp.–, 2005.

[54] O. Leßenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, 2015.

[55] B. Shen, W. Zhang, A. Yu, Y. Shi, H. Zhao, and Z. Jin, "Somanyconflicts: Resolve many merge conflicts interactively and systematically," in *36th IEEE/ACM Int. Conf. Automated Softw. Eng.*, pp. 1291–1295, 2021.