Revisiting the Conflict-Resolving Problem from a Semantic Perspective

Jinhao Dong[*]
Key Laboratory of High Confidence Software Technologies (Peking University), MoE
School of Computer Science, Peking University
Beijing, China
dongjinhao@stu.pku.edu.cn

Jun Sun
School of Computing and Information Systems,
Singapore Management University
Singapore
junsun@smu.edu.sg

Yun Lin
Department of Computer Science and Engineering,
Shanghai Jiao Tong University
Shanghai, China
lin_yun@sjtu.edu.cn

Yedi Zhang, Murong Ma
School of Computing,
National University of Singapore
Singapore
yd.zhang@nus.edu.sg
murongma@u.nus.edu

Jin Song Dong
School of Computing,
National University of Singapore
Singapore
dcsdjs@nus.edu.sg

Dan Hao[†]
School of Electronic and Computer Engineering,
Peking University
Shenzhen, China
haodan@pku.edu.cn

ABSTRACT

Collaborative software development significantly enhances development productivity by enabling multiple contributors to work concurrently on different branches. Despite these advantages, such collaboration often increases the likelihood of causing conflicts. Resolving these conflicts brings huge challenges, primarily due to the necessity of comprehending the differences between conflicting versions. Researchers have explored various automatic conflict resolution techniques, including unstructured, structured, and learning-based approaches. However, these techniques are mostly heuristic-based or black-box in nature, which means they do not attempt to solve the root cause of the conflicts, i.e., the existence of different program behaviors exhibited by the conflicting versions.

In this work, we propose sMerge, a novel conflict resolution approach based on the semantics of program behaviors. We first give the formal definition of the merge conflict problem as well as the specific conditions under which conflicts happen and the criteria employed to select certain version as the resolution. Based on the definition, we propose to resolve the conflicts from the perspective of program behaviors. In particular, we argue that the key to resolving conflicts is identifying different program behaviors, and thus can be solved through targeted test generation. We conduct an extensive evaluation of sMerge using a comprehensive dataset of conflicts sourced from various projects. Our results show that sMerge can effectively solve the merge problem by employing different test generation techniques, including search-based, GPT-based, and manual testing. We remark that sMerge provides a way to understand the program behavior differences through testing, which not only allows us to solve the merge problem soundly but also enables the detection of incorrect ground truths provided by developers, thereby enhancing the reliability of the merge process.

[*]This work was done when Jinhao Dong was a visiting student in National University of Singapore (NUS) and Singapore Management University (SMU).
[†]Dan Hao is the corresponding author.

CCS CONCEPTS

• **Software and its engineering → Collaboration in software development**.

KEYWORDS

Behavior-Based Conflict Resolving, Targeted Test Generation

ACM Reference Format:
Jinhao Dong, Jun Sun, Yun Lin, Yedi Zhang, Murong Ma, Jin Song Dong, and Dan Hao. 2024. Revisiting the Conflict-Resolving Problem from a Semantic Perspective. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3694993

1 INTRODUCTION

As software systems grow increasingly complex, collaboration among developers has become essential for project completion, a practice called collaborative software development. Typically, multiple developers concurrently work on separate branches forked from the main repository, each working on respective tasks. Upon completion, developers submit pull requests to merge their branches into the main branch [18]. Although collaboration development enhances efficiency, they also introduce challenges, notably the merge problem—resolving conflicts between different branches. Conflicts will arise when two developers modify the same parts of code. It has been reported that 12% of the commits are aimed at merging code from different developers [18], and approximately 46% of these merges result in conflicts, highlighting the significant frequency of such issues in collaborative software development. Resolving these conflicts is a great challenge as it requires developers to understand the intentions behind each conflict version, which involves a meticulous examination of the code implementations and identification of the differences between the conflicting versions [7, 8, 11, 20, 28].

To alleviate the substantial efforts required to resolve conflicts, researchers have proposed various automatic techniques [5, 8, 10, 23, 25, 26]. Given the common ancestor $\mathcal{O}$ and the conflicting versions $\mathcal{A}$ and $\mathcal{B}$ produced by different developers, the goal of automatic conflict resolving techniques is to produce a resolved version $\mathcal{M}$ from these discrepancies. Existing techniques can be divided into three main categories: unstructured techniques [1, 19], structured techniques [5, 8, 10, 23, 25, 26, 34, 40], and learning-based techniques [12, 35]. Unstructured techniques, primarily text-based, are commonly utilized in version control systems (VCS). However, these methods are limited in resolving capabilities, typically reporting conflicts rather than resolving them when modifications occur at identical text locations. Structured techniques merge code based on abstract syntax tree (AST) matching and amalgamation. These techniques leverage structural information of specific programming languages (e.g., methods can be permuted safely in Java), which often leads to a higher merging rate compared to the unstructured methods. However, structured techniques are typically designed for particular languages and involve operations at the tree level, which causes a high computational complexity (at least cubic [25]). Learning-based techniques, currently achieves the SOTA performance, which employs deep learning models to produce the resolution strategies [35] or generate the resolution code directly [12, 13].

While various techniques have been proposed to resolve merge conflicts, they share a common limitation. *The perspectives from which existing techniques resolve conflicts are not from the root cause.* Fundamentally, a conflict arises when there are conflicting program behaviors. The program behaviors reveal the performance and effects of the software, which indicates the fundamental requirements from the developers. Thus, the key to resolving conflicts is identifying and understanding divergent program behaviors, and other perspectives will unavoidably compromise the accuracy of conflict resolution. Unstructured techniques, which resolve conflicts from a textual perspective [1], are simple and possess good generalization but suffer from low accuracy. Structured techniques, addressing conflicts based on syntax and structure [5, 8], offer higher accuracy than text-based methods by considering syntax. However, the static analysis has high computation complexity for object-oriented language (e.g., Java), which has complex data structures and long method call chains. Furthermore, they fall short of ensuring correctness when the conflicts are concerned with behavioral aspects. For example, if version $\mathcal{A}$ adds the line `x = y * 2` and version $\mathcal{B}$ inserts `x = y « 1` at the same location, both text-based and syntax-based techniques would flag this as a conflict, despite both versions being semantically equivalent implementations following the same specification. Structured-based techniques will also cause false positives [6], and the Java-targeted tool JDime [3] achieves a precision of only 26.3%. Learning-based methods, while innovative, also present limitations. These methods do not establish which version is preferable, nor do they set a specific optimization goal. Rather, they just learn strategies from historical data which usually leads to overfitting. Moreover, these techniques lack transparency in their decision-making processes, making it challenging to understand or justify the choices they make. Hence, we argue that understanding the conflicting behaviors is crucial for identifying the conflicts and the goal of resolution process should be: i) precisely identifying the conflicting program behaviors and ii) resolving the conflict based on the expected or desired program behavior.

In this work, we propose a novel merge technique from the perspective of program behaviors, i.e., sMerge. *Firstly, we formally define the merge problem, the condition under which conflicts happen, and the criteria for selecting certain version as the resolution.* Such formal definitions are crucial because they provide a solid foundation for justifying the choices made in resolving conflicts. *Secondly, we propose to resolve the conflicts by focusing on the differences in program behaviors, utilizing targeted test generation.* We identify the program behavior differences between two conflicting versions by executing them on both versions' tests. By analyzing the differences and relationships between their behaviors, we can strategically resolve the conflicts, and we categorize these into six different cases, including both unmergeable and mergeable scenarios. For example, one specific case is selecting the version whose behaviors subsume those of the other version as the resolution.

To effectively identify the behavioral differences between two versions, sMerge requires a comprehensive set of test cases for each version. Testing is an intuitive and accurate way to observe the program behaviors, and it is challenging for other ways (e.g., static analysis) to identify the behavioral differences. Therefore, the merge problem is reduced to a test generation problem. We employ various test generation techniques to depict the behaviors of the conflicting versions, including search-based test generation, GPT-based test generation, and manual testing. Specifically, we introduce an automatic behavioral-differences-identifying test generation technique based on GPT-4. This technique consists of two components and utilizes an iterative refinement process to enhance the tests' ability to pinpoint behavioral differences. Upon obtaining the test suite, we can further obtain the behavioral differences by validating each conflicting version against the tests of both versions, and then resolve the conflicts based on the behavioral differences. The general idea here is that the resolution needs to subsume the newly-introduced behaviors of both versions. Intuitively, we will choose the version whose passed tests cover the tests passed by the other version, meaning that the behaviors of one version subsume those of the other. For the scenario where neither $\mathcal{A}$ nor $\mathcal{B}$ can pass all the tests of the other, indicating no version is definitively superior, we will concatenate them or report a conflict based on whether their tests are conflicting. Executing tests is one stage of software development, so sMerge will not bring extra burden to the developers. By contrast, structured techniques and learning-based techniques will bring huge extra cost. In addition, sMerge does not need any language-specific or project-specific information, but structured techniques are language-specific.

We conduct a comprehensive evaluation of sMerge. We evaluate our technique on two popular datasets in the research community. Since sMerge solves the merge problem based on test cases that identify the behavior differences, we evaluate its performance with different set of test cases, and show the more sufficient the test cases, the more cases that sMerge solves. Firstly, we evaluate sMerge based on the existing tests. Through this experiment, we find that when the developers modify the code, they frequently neglect to add tests to verify their modifications. In addition, we find that the ground truth of some data is incorrect. sMerge achieves further improvement on the corrected dataset. Secondly, we expand the test suite with the automatic test generation tools EvoSuite and the LLM GPT-4. sMerge can resolve 78.13% (75/96) conflicts automatically and the precision exceeds 97%, indicating the soundness of sMerge. By contrast, the precision of SOTA technique is only close to 70%, which is unreliable and cannot justify the choices. Lastly, we further manually supplement the test suite for the remaining 21 conflicts that cannot be solved automatically. With the manually crafted test cases that reveal the behavior difference, sMerge can solve almost all the conflicts and the precision approaches 100%.

In summary, this paper makes the following contributions:

- **A behavior-difference-based perspective**, which defines the merge problem formally and resolves the conflicts by identifying the behavior differences of conflicting versions.
- **A testing-based implementation**, which leverages several test generation techniques to depict the program behaviors.
- **An GPT-based test generation technique** which iteratively refines the tests to identify the behavioral differences.
- **A sound resolving approach**, which solves the problem soundly and identifies wrong ground truths provided by the developers.

The data and replication scripts are available in the repository [2].

2 MOTIVATION

In this section, we will introduce the motivation, i.e., the importance of behavioral differences and testing for resolving conflicts.

2.1 Importance of Behavioral Differences

The root cause of conflicts is the behavioral differences between conflicting versions. We present an illustrative example in Figure 1. It is a conflict computed by `git`, where the code between `<<<<<<< a` and `=======` is from $\mathcal{A}$, and the code between `=======` and `>>>>>>> b` is from $\mathcal{B}$. Both versions intend to add a date-parsing function, which accepts a date string and extracts the "year", "month", and "day" from it. $\mathcal{A}$ splits the string with "/" and parse the split parts to date. However, this program can only process the date with the format of "yyyy/MM/dd". By contrast, $\mathcal{B}$ leverages the "`LocalDate`" library to parse the date string and supports various formats of date. When the input date string is with the format of "yyyy/MM/dd", $\mathcal{B}$ has the same behavior of $\mathcal{A}$; but when the date string is of other formats, only $\mathcal{B}$ can process. Therefore, the behaviors of $\mathcal{B}$ subsumes that of $\mathcal{A}$, we thus should choose $\mathcal{B}$ as the resolution. This conflict cannot be resolved if not from the perspective of behavior differences.

2.2 Importance of Test Cases

The previous example indicates the importance of behavioral differences in resolving conflicts. In order to identify the differences, test cases are necessary. In the previous example, without test cases, we cannot judge whether the two versions have the same behaviors only with text analysis or static analysis. We show another

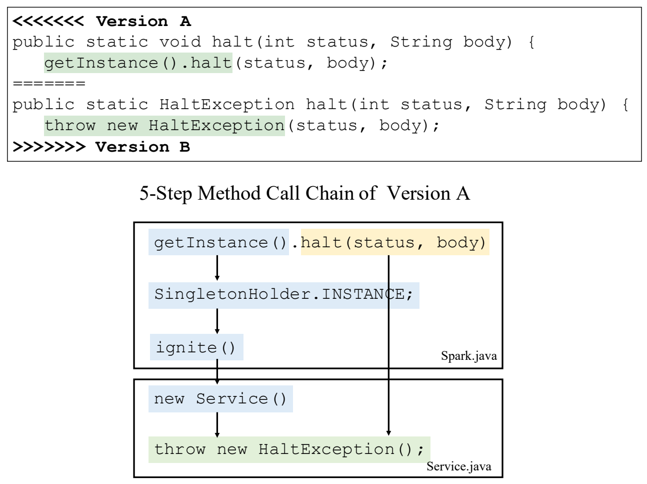
```
<<<<<<< Version A
public Map<String, Integer> parseDate(String dateString) {
   Map<String, Integer> dateMap = new HashMap<>();
   String[] parts = dateString.split("/");
   int[] values = new int[3];
   for (int i = 0; i < parts.length; i++) {
        values[i] = Integer.parseInt(parts[i]);
   }
   dateMap.put("Year", values[0]);
   dateMap.put("Month", values[1]);
   dateMap.put("Day", values[2]);
   return dateMap;
}
=======
public Map<String, Integer> parseDate(String dateString) {
   Map<String, Integer> dateMap = new HashMap<>();
   List<String> formats = Arrays.asList("yyyy-MM-dd",
   "dd/MM/yyyy", "MM-dd-yyyy", "yyyyMMdd", "MM/dd/yyyy");
   for (String format : formats) {
        try {
            DateTimeFormatter formatter = DateTimeFormatter.
            ofPattern(format);
            LocalDate date = LocalDate.parse(dateString,
            formatter);
            dateMap.put("Year", date.getYear());
            dateMap.put("Month", date.getMonthValue());
            dateMap.put("Day", date.getDayOfMonth());
            break;
        } catch (DateTimeParseException e) {
           continue;
        }
   }
   return dateMap;
}
>>>>>>> Version B
```

Figure 1: The motivating example showing the importance of behavioral differences

example in Figure 2 to show the importance of tests in revealing the behaviors. The top part shows the conflict, where two versions add an exception handler but the concrete implementations are different. The bottom part shows the method call chain of "`getInstance().halt()`" added by $\mathcal{A}$. It is a 5-step method call chain and finally calls a function in another class file, which also throws a "`HaltException`". This case is challenging for text or static analysis, because of the long call chain and involving multiple data structures. By contrast, by executing tests, we find that both versions throw a "`HaltException`" but the file from which the exception is thrown is different. Therefore, tests can clearly identify the behavioral similarity and difference of the two versions. Tests is a simple and intuitive way to observe the program behaviors.

3 PROBLEM DEFINITION

Each program has its specification $\mathcal{S}$, which refers to a set of requirements, standards, or constraints that the software must meet [22]. They are used to ensure the code is implemented correctly and ensure the behaviors satisfy the requirements. Therefore, the behaviors of the software should adhere to $\mathcal{S}$. In the evolutionary process of a program, the commits bring the program closer to $\mathcal{S}$ gradually. When conflicts happen, among those versions, in general, $\mathcal{O}$ is the farthest from $\mathcal{S}$, and $\mathcal{A}$ and $\mathcal{B}$ evolves closer to $\mathcal{S}$. However, the direction $\mathcal{A}$ and $\mathcal{B}$ evolve from $\mathcal{O}$ may be different, so their modifications may affect each other. We need to produce a resolution version $\mathcal{M}$ which involves both newly-added features of $\mathcal{A}$ and $\mathcal{B}$, and is closer to $\mathcal{S}$ than $\mathcal{A}$ and $\mathcal{B}$.



```
<<<<<<< Version A
public static void halt(int status, String body) {
   getInstance().halt(status, body);
=======
public static HaltException halt(int status, String body) {
   throw new HaltException(status, body);
>>>>>>> Version B
```

5-Step Method Call Chain of Version A
getInstance().halt(status, body)
SingletonHolder.INSTANCE;
ignite()
Spark.java
new Service()
throw new HaltException();
Service.java

Figure 2: The motivating example for the importance of tests

3.1 Definition of Software

First, we will give a definition to the software.

Definition 3.1. A software is in the evolution process, and it is gradually updated as the developers submit the commits. Therefore, a software is a finite set of commits $C = \{C_1, C_2, ..., C_n\}$.

For the software, let $\mathcal{S}$ be the ground-truth specification, which is the requirements the software needs to satisfy. $\mathcal{S}$ can be represented by a set of program behaviors. The commits are submitted to make the program behaviors satisfy $\mathcal{S}$. It is worth noting that we will first verify whether each commit can pass its own tests. If false, the commit should be fixed before sumbitted. If true, we assume the commit satisfies the expected behaviors defined by $\mathcal{S}$. If the commit is still incorrect, this is out of our scope. We cannot verify the correctness without neither sufficient tests nor specifications.

Each commit may introduce or update program behaviors and correspondingly should introduce new test cases to check the program behaviors. The test cases are the concrete manifestation of the program behavior. Let B and T be the program behaviors and test cases added or updated by the commit. We assume that $B \subseteq \mathcal{S}$ and $T \subseteq \mathcal{S}$, i.e., the newly introduced program behaviors and test cases are consistent with the ground-truth specification. In other words, the (implicit) ground-truth specification is assumed to be constant and the commits gradually complete the specification.

Therefore, each commit consists of three parts, that is, the implementation of code I, the program behaviors B, and the test cases T. The software can be defined as $C = \{C_1, C_2, ..., C_n\} = \{(I_1, B_1, T_1), (I_2, B_2, T_2), ..., (I_n, B_n, T_n)\}$. The implementation should meet the newly-added behaviors, that is, $I_i \models B_i$. This can be reflected by whether I_i passes the newly-added tests T_i, that is, $I_i \models T_i$. In the evolution process, the developers make the software approach $\mathcal{S}$ gradually. The merge problem can be defined as follows.

Problem 3.1. *Given two conflicting commits (I_1, B_1, T_1) and (I_2, B_2, T_2), find I such that $I \models B_1 \cup B_2$ and I is minimally different from I_1 and I_2.*

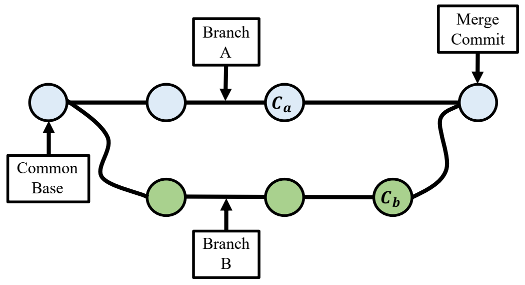


Figure 3: Illustrating of merging two branches

The resolution I should satisfy the behaviors of both conflicting versions. During implementation, because tests are the manifestation of behaviors, we find I such that $I \models T_1 \cup T_2$.

3.2 Multi-Branches Collaborative Development

Modern software is complex and evolves quickly, so it is impossible for a single developer to complete the whole software. Different developers will fork their own branches and work simultaneously, which can increase the development efficiency. As shown in Figure 3, two developers work concurrently in their respective branch. Assuming the number of developers is k, the commits C can be divided into k groups, i.e., $\{C_1, C_2, ..., C_k\}$. The group of commits C_i is from the i-th developer. The total set of commits is the union of the subset of commits from each developer, and the subset of commits from different developers have no intersection, i.e.,

$$C = \bigcup_{i=1}^{k} C_k \tag{1}$$

$$\forall i, j \in 1..k, C_i \cap C_j = \varnothing \tag{2}$$

Each group C_i consists of the commits from the i-th developer, $C_k = C_k^1, C_k^2, ..., C_k^{n_k}$, which are in chronological order. In the development process of one single developer (e.g., within branch A or branch B), the commits evolve consistently towards the same direction, and C_k^{i+1} is developed based on C_k^i, so there will not happen conflicts. However, when the commits from two branches need to be merged, the conflicts might happen. The merge commit in Figure 3 is the symbol of merging.

Therefore, we focus on the cases where the two commits to be merged are from different branches, which might cause conflicts. Existing VCS (e.g., `git`) all leverage text-based algorithm (i.e., `diff3` [1]) to identify the conflicts. Therefore, to combine with the widely-used VCS, sMerge first receives the textual conflicts returned by "`git merge`" as input, and then resolves the conflicts based on the behavior differences. The textual conflicts happen when two commits modify the same parts of code, i.e., $I_a \cap I_b \neq \varnothing$. The textual conflicts that are not conflicting from program behaviors can be resolved and those that are conflicting will be identified as behavioral conflicts, which will be introduced in Section 4.

4 APPROACH

sMerge consists of three stages, i.e., generating tests, executing tests, and resolving conflicts, and the overview is shown in Figure 4.

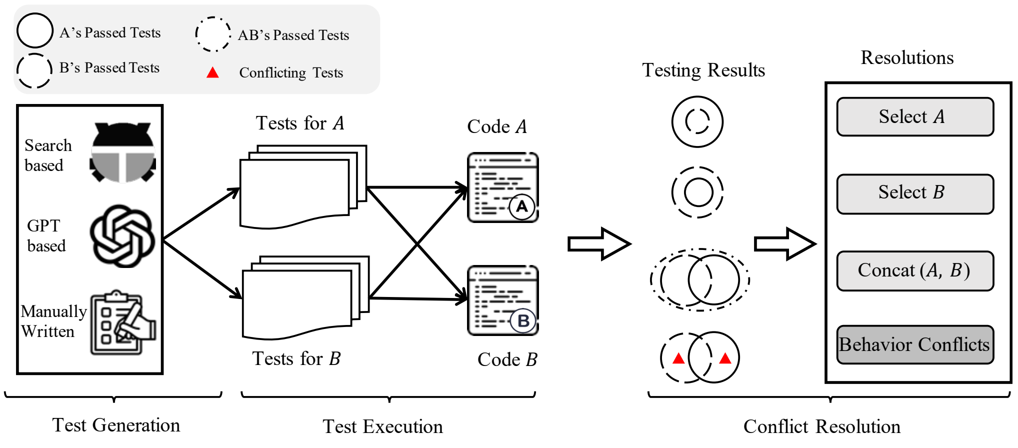


Figure 4: The overview of sMerge

```
protected Object encodeCommand(Channel channel, Command command) {
   switch (command.getType()) {
      ...
      case Command.TYPE_SET_TIMEZONE:
<<<<<<< Version A
         return formatTextCommand(channel, command, "LZ,{%s},{%s}",
         Command.KEY_LANGUAGE, Command.KEY_TIMEZONE);
=======
         return formatTextCommand(channel, command, "LZ,,%s",
         Command.KEY_TIMEZONE);
>>>>>>> Version B
      case Command.TYPE_SET_INDICATOR:
         return formatTextCommand(channel, command, "FLOWER,%s",
         Command.KEY_DATA);
      ...
```

Figure 5: Case 2: two versions have behavioral conflicts (traccar-8a696af)

Firstly, we produce tests automatically and manually (Section 4.4). Secondly, sMerge identifies the behavior differences by executing both versions' tests on each version. Finally, we resolve the conflicts based on the behavior differences. The general idea is that resolution needs to subsume the behaviors of both versions. The resolving process consists of three steps, including checking whether the commits are unmergable and how to make them mergable (Section 4.1), resolving the mergable (Section 4.2), and handing the rest with heuristics (Section 4.3).

4.1 When they are unmergable

The following shows three unmergable cases and how they can be identified.

- Case 1: $I_1 \not\models T_1$ or $I_2 \not\models T_2$, i.e., a commit is inconsistent with its own test cases. Checking whether this is the case is easy as we simply test whether each commit is consistent with its test cases. If it is not the case, we consult the author of the commit and ask him/her to solve the issue before committing.
- Case 2: $\exists t_1 \in T_1, t_2 \in T_2.\ t_1 \otimes t_2$ where $t_1 \otimes t_2$ means that the two test cases t_1 and t_2 are conflicting in behaviors. Figure 5 shows one such example. I_1 and I_2 modify the return statement differently and cause the behavioral conflicts. Checking whether the two commits fall in this case is relatively easy in practice. We exam T_1 and T_2 to see whether they are the same test cases with different assertions. In such a case, the authors of the two commits must be consulted as it is apparent that they have different understanding on how the system should behave.
- Case 3: $\exists t_1 \in T_1.\ I_2 \not\models t_1$ and $\exists t_2 \in T_2.\ I_1 \not\models t_2$. Intuitively, this case happens when each commit fails to satisfy some of the test cases of the other commit. For instance, Figure 6 shows one such example. I_1 (version $\mathcal{A}$) and I_2 (version $\mathcal{B}$) introduce new functions respectively, and they will not pass the tests of each other. While in theory, it is possible to apply program synthesis methods to synthesize a program based on I_1 and I_2 such that it satisfies $T_1 \cup T_2$, such an approach is still rather impractical due to the limited capacity and scalability of existing program synthesis techniques. We thus propose a candidate resolution I_{cat}, which concatenates I_1 and I_2 on the level of lines, and verify whether $I_{cat} \models T_1 \cup T_2$. If true, I_{cat} subsumes the behaviors of I_1 and I_2, and we output I_{cat} as the resolution. If false, the two conflicting versions are mutually influenced, and to make the results sound, we do not attempt to merge the commits automatically in such a case. We rather present the author of I_1 with those test cases in T_2 that are failed by I_1 (and vice versa), and decide whether to fix I_1 or discard those test cases from T_2. Because the three strategies ($\mathcal{A}$, $\mathcal{B}$, and concatenating) already occupy 98% resolutions in the open source projects [35], we only validate the resolution "concatenating" to reduce the execution overhead and improve the usage experience of users. On the other hand, any new strategy can be easily added to sMerge.

4.2 When they are mergable

The following shows two cases in which the commits can be merged straightforwardly.

- Case 4: If $I_1 \models T_1 \cup T_2$ and $I_2 \not\models T_1 \cup T_2$, the merging result is I_1. Intuitively, if I_1 passes all those test cases of T_2 and yet I_2 fails some test cases of T_1, I_1 subsumes the behaviors of I_2 and I_1 is closer to satisfy (S), and thus is objectively better. For instance, Figure 7 shows one such example. I_2 removes the space at the

```
<<<<<<< Version A
public static List<String> getEntityAuthorities(String entityName)
{
   List<String> authorities = new ArrayList<>();
   for (Permission permission : Permission.values())
       ...
=======
public static boolean isSessionExpired(HttpServletRequest request)
{
   return request.getRequestedSessionId() != null
&& !request.isRequestedSessionIdValid();
}
>>>>>>> Version B
```

Figure 6: Case 3: both versions fail to satisfy the test cases of the other version (molgenis-a20bee9)

```
<<<<<<< Version A
...
 for(String tableKey : tableConfigMap.keySet()){
 TableConfig itemConfig = tableConfigMap.get(tableKey);
 if((tableKey.equalsIgnoreCase(table) && itemConfig!=null)
|| tableKey.equals("*")){
    schemaConfig = schemaConfigMap.get(schemaKey);
    schema = schemaKey;
    tableConfig = itemConfig;
 ...

=======
...
for(String tableKey : tableConfigMap.keySet()){
TableConfig itemConfig = tableConfigMap.get(tableKey);
if(tableKey.equalsIgnoreCase(table) && itemConfig!=null){
   schemaConfig = schemaConfigMap.get(schemaKey);
   schema = schemaKey;
   tableConfig = itemConfig;
...

>>>>>>> Version B
```

Wrong Ground Truth:
Select version B
Corrected Ground Truth:
Select version A

Figure 7: Case 4: version $\mathcal{A}$ can pass tests of both versions (Mycat-Server-c42db00)

beginning of each line, which doesn't change the behavior of the original version. I_1 adds "disjunction" to the condition, and more cases can enter the `True` branch compared to the original version, which introduces new behaviors and provides new tests to verify the behavior. Therefore, the behaviors of I_1 subsume that of I_2, and I_1 can pass both T_1 and T_2, but I_2 cannot pass T_1. In addition, the ground truth of this example is incorrect and we will introduce in Section 6.1.2.

- Case 5: If $I_2 \models T_1 \cup T_2$ and $I_1 \not\models T_1 \cup T_2$, the merging result is I_2. This case is symmetric to Case 4.

4.3 When heuristics are applied

The last case is that when both I_1 and I_2 satisfy $T_1 \cup T_2$. Behavior-wise it does not matter which commit that we choose, i.e., either one is correct. Such a case takes place due to the conflicts on non-functionality parts (e.g., comments), which cannot cause the behavior differences. Since we have no way to tell the difference between I_1 and I_2, we resolve with heuristics. The following heuristics are adopted for comments. Note that we omit the symmetric case since it is obvious. (1) If both I_1 and I_2 add or modify the comments, $I_{cat} = concat(I_1, I_2)$ is returned. (2) If I_1 updates the comments, and

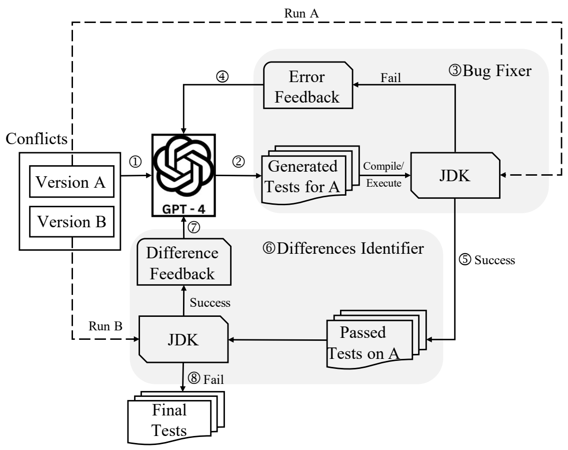


Figure 8: The overview of MergeTester

I_2 deletes the same comments, I_1 is selected. (3) If I_1 updates the code parts, and I_2 updates the comment parts, I_{cat} is returned.

4.4 Automatic Test Generation

Note that the above discussion makes it clear that the more test cases to reveal the behavioral differences between two versions, the better we are able to check whether the versions are mergable or how they should be merged. Ideally, each commit should be accompanied with test cases that reflect the change or capture the newly introduced feature. In practice, however, T_1 and T_2 are often limited or even non-exist, which causes that the introduced behaviors are unverified. Therefore, we propose two automatic ways of enriching the test cases for resolving the conflicts.

4.4.1 *Automatic Test Generation with EvoSuite.*

We first leverage the widely-used EvoSuite [17] to generate tests for the conflicting parts. EvoSuite is a search-based test generation for increasing the coverage, which leverages evolutionary algorithms [17]. It can capture the current behaviors, so the tests generated by EvoSuite can help depict the program behaviors of each version. We validate each version on the tests generated for both versions, and use the execution results to produce the resolutions based on the strategies introduced before. We also execute O on the generated tests. When running $\mathcal{A}$ and $\mathcal{B}$ on a specific test, the behavior that differs from O will be regarded as the oracle behavior. This decision logic is the same with `git`, which reserves the text different from O. However, if both versions' behaviors are different from O, the behavioral conflicts happen.

4.4.2 *Automatic Test Generation with GPT-4.*

EvoSuite is a search-based technique and focuses on increasing the coverage of code, which cannot understand the testing semantics and lack of general knowledge, and tends to build the naive and minimal objects [37]. Some branches are semantics-related and difficult for EvoSuite to achieve [24]. For example, a condition that accepts a certain program version string (e.g., "3.2.3") requires the comprehension of semantics. If we know it needs a version, we can easily produce a string with numbers separated by "."; however, it is difficult for a search-based technique to achieve the branch by mutating a random string. Moreover, a condition might require a complex object with nested structures or require the outcome of a long method call chain [37, 37], which needs the understanding of semantics. Furthermore, without the understanding of conflicting behaviors, even if the test covers the conflicting parts, it might also not identify differences. Therefore, besides EvoSuite, we also leverage the most powerful LLM GPT-4 [29] to generate tests for identifying the behavioral differences. LLM has a strong code understanding [27] and code generation [14, 16] ability, which is also applied to test generation and achieves good performance [37, 38].

We propose an automatic behavioral-differences-identifying test generation approach for resolving conflicts, i.e., MergeTester, and the overview is shown in Figure 8. We only show the process to generate tests for version $\mathcal{A}$, and that for $\mathcal{B}$ is the same. Existing LLM-based test generation techniques [37, 38] aim to improve the compilation rate or coverage. Different from them, MergeTester needs to generate tests for two versions and reflect their behavioral differences. MergeTester receives the conflicts as input, and generates the initial tests for both versions simultaneously, which is beneficial for GPT-4 to capture the their differences. After that, MergeTester iteratively refines the tests to be executable and reveal the behavioral differences, which is achieved by two components.

- *Bug Fixer* iteratively corrects the bugs of the generated tests based on the execution feedback. MergeTester executes the tests and collects the bug information, which is provided to MergeTester for further improvement. This component is similar to existing work [38].
- *Difference Identifier* will execute the tests on the other version if passed on current version. If the execution results are identical, which indicates that the tests cannot identify the behavioral differences, a "executable-but-undifferentiable" feedback will be provided to MergeTester and lets it generate another test.

5 EXPERIMENTAL SETUP

5.1 Research Questions

In this section, we will introduce the research questions that we aim to answer through experiments. We expand the test set and evaluate sMerge's performance with different set of test cases.

- **RQ1. Initial Performance** How does sMerge perform based on the existing test cases only?
- **RQ2. Automatically Generated Tests** How does sMerge perform when the test suite enhanced by automatic techniques?
- **RQ3. Behavior-Differences-Revealed Tests** How does sMerge perform when the tests revealing the differences are provided?

5.2 Dataset

We evaluate the performance of sMerge with two well-constructed datasets [33, 35] and merge them together. These datasets are mined from the top popular projects of GitHub and used for empirical study [33] or verify the conflict resolving techniques [13, 35]. Since the previous techniques leverage ASTs or deep learning models to process the dataset, they do not need to execute the projects. By contrast, sMerge needs to build and execute the projects to obtain the test execution results. To prepare the dataset for sMerge, we further process the datasets by static filter and dynamic execution, which is introduced below. Although sMerge requires test execution, it is one stage in normal development cycle so sMerge will not bring extra cost to the developers.

5.2.1 Static Filter. To adapt to our experimental setting, we use the following conditions to filter the conflicts. (1) We filter out the conflicts failing to recognize the resolution regions. (2) We filter out the conflicts not happening in ".java" files, (3) To make the build and execution easier, we only reserve the projects built with Maven. (4) We filter out the conflicts happening in tests, which means the conflicts in specifications and corresponds to Case 2 in Section 4.

5.2.2 Dynamic Execution. After we clone the projects, we use the command `git checkout` to switch to the two conflicting commits. We then compile them respectively. Because many of the commits are historical commits many years ago, some dependencies may not exist any longer. We filter out the conflicts whose two conflicting commits cannot be built successfully. Among them, 85.77% of these issues arise due to dependencies that cannot be found in the Maven Central Repository or cannot be resolved. After that, we will use the command `git merge` to merge the two conflicting commits and obtain the conflicting version C. We first try to merge the changes that can be merged directly. And then we build two intermediate versions $\mathcal{A}'$ and $\mathcal{B}'$ which are based on the conflicting version C and replace the conflicting parts with that part from $\mathcal{A}$ and $\mathcal{B}$. These two intermediate versions $\mathcal{A}'$ and $\mathcal{B}'$ are only different in the conflicting parts. The evaluation on the two intermediate versions will reflect the real conflicts between the two versions and avoid the influence of other different parts of two versions that are not conflicts. Then, we compile the two intermediate versions $\mathcal{A}'$ and $\mathcal{B}'$ and we filter out the failed conflicts. In addition, we filter out the data whose tests cannot be executed because of environments or dependencies. We further filter out the cases that cannot pass its own attached tests, which corresponds to Case 1 in Section 4. Finally, we remain **153** conflicting situations. The two versions $\mathcal{A}'$ and $\mathcal{B}'$ will be used to execute tests and resolve conflicts. We remark that the static and dynamic filtering are applied only for practical reasons (i.e., so that we can execute these projects) and they are not designed to harm the dataset's representativeness.

5.3 Compared Techniques

We compare sMerge against existing SOTA approaches in terms of merge precision and recall. The SOTA techniques MergeGen [13] and MergeBERT [35] are learning-based techniques and resolve conflicts by generation and classification respectively. In addition, we compare with the structured technique designed for Java, JDime. Compared with existing techniques, besides higher precision and recall, sMerge is sound and can justify the resolution results.

5.4 Evaluation Metrics

Following the previous works [12, 13, 35], we compute the precision and recall to evaluate sMerge. Precision refers to the percentage of correctly-resolved conflicts out of the conflicts the model can return a prediction. sMerge will not return if the test results cannot identify behavioral differences. Recall refers to the percentage of

Table 1: Performance of sMerge with only existing tests

Test Results	Strategy	Precision	Recall
$I_1 \models T_1 \cup T_2$ and $I_2 \not\models T_1 \cup T_2$	Select $\mathcal{A}$	66.67% (8/12)	27.59% (8/29)
$I_2 \models T_1 \cup T_2$ and $I_1 \not\models T_1 \cup T_2$	Select $\mathcal{B}$	77.78% (7/9)	41.18% (7/17)
$I_{cat} \models T_1 \cup T_2$ and $I_1, I_2 \not\models T_1 \cup T_2$	Concat($\mathcal{A}$, $\mathcal{B}$)	100% (24/24)	22.43% (24/107)
Ratio of cases sMerge can give resolutions: 29.41% (45/153)			
Overall	-	86.67% (39/45)	25.49% (39/153)

correctly-resolved conflicts out of all conflicts. As shown in Section 4, sMerge has six situations. Case 1 is filtered during the dataset construction stage, and Case 6 is using heuristics. Therefore, we will show the precision and recall of another four cases. Precision is more important in this task. High precision means that developers do not need to check the returned predictions.

5.5 Implementation

sMerge is established on the basis of `git merge` command, which is the most widely used merge technique in VCS (e.g., Git). We will resolve the conflicts detected by `git merge`. We use EvoSuite and GPT-4 to supplement the test suite. The version of EvoSuite we use is 1.0.6. For GPT-4, we leverage the official ChatGPT API [29], and we adopt the powerful gpt-4-turbo model. The temperature is set to 0.7 and the iterative rounds are set to 10. Moreover, the prompt is shown in our repository [2]. The implementation details about building and executing projects are already shown in Section 5.2.

6 RESULTS AND ANALYSIS

In this section, we present the performance of sMerge with existing tests only in Section 6.1, the performance with automatic test generation techniques in Section 6.2, and the performance when the remaining unsolved cases are provided manual tests in Section 6.3.

6.1 RQ1: sMerge with Existing Tests Only

We execute existing attached tests to evaluate the two conflicting versions, and produce resolutions based on the execution results. If the tests of both versions cannot identify the behavioral differences, sMerge regards it unsolvable. The execution results are shown in Table 1. We show the precision and the recall of three strategies corresponding to Case 3-5 in Section 4. sMerge doesn't find the conflicting cases with existing tests, so the results are omitted. For example, the precision of choosing $\mathcal{B}$ is 77.78%, that is, there are 9 cases that sMerge predicts as choosing $\mathcal{B}$, among which 7 is correct. Moreover, the overall precision is 86.67% and the overall recall is only 25.49%.

Table 2: Performance of sMerge with existing tests after correcting wrong ground truths.

Test Results	Strategy	Precision	Recall
$I_1 \models T_1 \cup T_2$ and $I_2 \not\models T_1 \cup T_2$	Select $\mathcal{A}$	83.33% (10/12)	31.25% (10/32)
$I_2 \models T_1 \cup T_2$ and $I_1 \not\models T_1 \cup T_2$	Select $\mathcal{B}$	100% (9/9)	56.25% (9/16)
$I_{cat} \models T_1 \cup T_2$ and $I_1, I_2 \not\models T_1 \cup T_2$	Concat($\mathcal{A}$, $\mathcal{B}$)	100% (24/24)	23.3% (24/103)
Number of identified wrong labels: 4			
Overall	-	95.56% (43/45)	28.1% (43/153)

The test suite attached by the developers is used to verify whether the code satisfies the specifications. It is important that developers maintain a quality test suite, and create/update the tests whenever a modification is made. However, developers often neglect to update tests. When the tests can reveal the specifications, sMerge is sound and the answers will be correct. Therefore, in terms of the poor results with the existing tests only, there are two possible reasons, i.e., (1) the test suite provided by the developers is insufficient; (2) the ground truths are incorrect, and we find that both situations happen. sMerge can help identify the problems that the key tests are neglected and the ground truths are incorrect.

6.1.1 *Insufficient Test Cases.*

With the existing tests, only 45 of 153 conflicts can be solved and given an answer by sMerge. The majority of the execution results on $\mathcal{A}$ and $\mathcal{B}$ are the same. This indicates that existing tests cannot identify the behavioral differences of conflicting versions. sMerge will not predict a resolution when tests cannot reveal differences. Therefore, the existing test suite provided by the developers is insufficient. The modified code is not verified and there is limited evidence that the modification is correct. We will solve this problem in the following.

Among the cases where sMerge can solve, the precision of sMerge is 86.67%. We carefully check the wrong conflicts, and found that among the 6 cases, 2 of them are caused by the insufficient tests of one version. Both versions introduce new behaviors but only one behavior is identified by the tests. This is solved after adopting test generation (see Section 6.2.1).

6.1.2 *Correct Wrong Ground Truths.*

Another 4 of the wrongly-predicted cases have the wrong ground truths, which are confirmed by the developers. This indicates that the resolutions conducted by the developers may also be incorrect and sMerge can help correct the wrong ground truths. The developers do not consider the behavior differences to resolve the conflicts, which causes the problems. We show the results of sMerge after correcting the wrong labels in Table 2. The precision increases from 86.67% to 95.56%, which is high. The recall is still low because of the insufficient tests. Precision is more important than recall. High precision means the developers do not need to check the returned correctness.

We show an example whose ground truth is incorrect and identified by sMerge, which is the same example in Figure 7 of the Approach section. The behaviors of $\mathcal{A}$ subsumes that of $\mathcal{B}$, so $\mathcal{A}$ should be selected. However, the ground truth is choosing $\mathcal{B}$. The existence of wrong ground truths also reflects the problems of existing techniques. The learning-based techniques use the wrong ground truths to train the model, which will harm the performance.

6.2 RQ2: sMerge with Generated tests

In RQ1, we find that developers often neglect to update or add the tests when they modify the code, which causes that the tests cannot identify the behavioral differences. Only 45 of 153 can be solved because of the insufficient test suite. In the following, we expand the test suite, and then we evaluate sMerge on the enhanced test suite. We leverage two automated approaches to generate the tests. Firstly, we adopt EvoSuite. After that, in terms of the cases where sMerge still cannot distinguish, we leverage GPT-4 to generate tests. Note

Table 3: Performance after supplemented by EvoSuite.

Test Results	Strategy	Precision	Recall
$I_1 \models T_1 \cup T_2$ and $I_2 \not\models T_1 \cup T_2$	Select $\mathcal{A}$	90% (18/20)	56.25% (18/32)
$I_2 \models T_1 \cup T_2$ and $I_1 \not\models T_1 \cup T_2$	Select $\mathcal{B}$	90.91% (10/11)	62.5% (10/16)
$I_{cat} \models T_1 \cup T_2$ and $I_1, I_2 \not\models T_1 \cup T_2$	Concat($\mathcal{A}$, $\mathcal{B}$)	100% (44/44)	42.72% (44/103)
Ratio of cases sMerge can give resolutions: 49.02% (75/153)			
Number of identified wrong labels: 6			
Overall	-	96% (72/75)	37.5% (72/153)

Table 4: Performance after further supplemented by GPT-4.

Test Results	Strategy	Precision	Recall
$I_1 \models T_1 \cup T_2$ and $I_2 \not\models T_1 \cup T_2$	Select $\mathcal{A}$	100% (21/21)	65.62% (21/32)
$I_2 \models T_1 \cup T_2$ and $I_1 \not\models T_1 \cup T_2$	Select $\mathcal{B}$	91.67% (11/12)	68.75% (11/16)
$I_{cat} \models T_1 \cup T_2$ and $I_1, I_2 \not\models T_1 \cup T_2$	Concat($\mathcal{A}$, $\mathcal{B}$)	100% (77/77)	74.76% (77/103)
$\exists t_1 \in T_1, t_2 \in T_2.\ t_1 \otimes t_2$	Report Conflicts	100% (1/1)	50% (1/2)
Ratio of cases sMerge can give resolutions: 72.55% (111/153)			
Number of identified wrong labels: 9			
Overall	-	99.1% (110/111)	71.9% (110/153)

that all the performance in this paper is an overall performance based on the tests supplemented by previous steps.

6.2.1 *Tests Generated by EvoSuite.*

We supplement tests for the conflicts whose existing tests of either version cannot identify the differences. The results are shown in Table 3. To save the space, we directly show the results after wrong labels are corrected. With the supplemented tests, sMerge can solve more conflicts, that is, 75 out of 153, accounting for 49.02%. This indicates that the more test cases that we have to reveal the behavioral differences, the better we are able to decide how the conflicts should be resolved. sMerge has a high precision (96%), so as long as sMerge gives a prediction, it is likely to be correct. The learning-based techniques can provide a revolution for any input, but the correctness cannot be guaranteed.

For the cases where sMerge produces wrong resolutions with existing tests, when combined with the tests generated by EvoSuite, the cases can be solved successfully. Here we give an example in Figure 6, the same example in the Approach section. $\mathcal{A}$ and $\mathcal{B}$ introduce a new function respectively, i.e., `getEntityAuthorities` and `isSessionExpired`. However, only $\mathcal{A}$ supplements a new test to evaluate `getEntityAuthorities`, and $\mathcal{B}$ doesn't supplement new tests. Only one behavior difference can be identified. $\mathcal{B}$ cannot pass the tests from $\mathcal{A}$, while $\mathcal{A}$ can pass all tests from $\mathcal{B}$, so $\mathcal{A}$ will be chosen. After EvoSuite supplements the tests suite, it successfully generates tests to evaluate the modifications introduced by $\mathcal{B}$. In this way, both two behavioral differences are identified and we should reserve both of them.

There are three cases where sMerge makes mistakes, and the reason is similar to the aforementioned case. The ground truths of are concatenating $\mathcal{A}$ and $\mathcal{B}$, but sMerge chooses certain version. The tests EvoSuite generates cover the conflicting parts of one version but fails to cover the other version. Therefore, sMerge wrongly selects certain version. EvoSuite cannot guarantee to cover the targeted parts because search-based algorithm lacks of semantics and knowledge, and they tend to build the naive and minimal objects [37]. Moreover, without the understanding of conflicting behaviors, even if the test covers the conflicting parts, it might also not identify differences. Therefore, we further leverage GPT-4 to understand the semantics of conflicts and further supplement the tests. And all the three cases are solved after supplementing tests with GPT-4. One example is that, $\mathcal{B}$ adds a private constructor to avoid instantiation, and the intention is difficult to understand. EvoSuite fails to generate a test revealing this behavior, but GPT-4 successfully reveals it by verifying whether the modifier of constructor is private. This is because GPT-4 can understand the semantics of private constructor and leverage the rich knowledge to solve it.

6.2.2 *Tests Generated by GPT-4.*

For the cases EvoSuite cannot generate tests that distinguish $\mathcal{A}$ and $\mathcal{B}$, we further leverage GPT-4 to supplement the tests. GPT-4 has a strong ability to understand code and can capture the semantic differences between conflicting versions, which can make up the semantic understanding drawback of EvoSuite. The results are shown in Table 4.

Plus the tests supplemented by GPT-4, sMerge can solve another 23.53% (36/153) cases, and can solve 72.55% (111/153) cases in total. sMerge achieves high precision for each category. We further find 3 cases whose ground truths are incorrect. In addition, we identify one behavior conflicting cases. Both $\mathcal{A}$ and $\mathcal{B}$ modify the throw message to different ones, and GPT-4 can generate tests to capture this conflicting behavior. The ground truth is incorrect that selects certain version, but in fact this case should involve developers to solve the behavioral conflicts. Furthermore, the three wrongly-predicted cases using EvoSuite can all be solved plus the tests introduced by GPT-4. Here we give an example that EvoSuite cannot produce the tests revealing the behaviors but GPT-4 can in Figure 9. This modification replaces "`zone_info`" with "`zoneinfo`" in the "`claims_supported`" list, which is further in the "entity" map. "entity" map also has an "issuer" item. This object is a complex nested structure so EvoSuite cannot generate. GPT-4 can understand the semantics and the construction process, so GPT-4 generates the correct test, which is shown in Figure 9b. GPT-4 correctly construct the object and depict the behaviors by verifying the replacement from "`zone_info`" to "`zoneinfo`".

However, there still exist some cases that GPT-4 cannot produce the resolutions. Among the 79 cases where GPT-4 tries to generate tests, 31.65% of the generated tests cannot be compiled or executed successfully, and 16.46% produce the same results on both versions. The uncompilable problem is caused by the hallucination of GPT-4, and sometimes even after multi-round feedback, it still cannot produce the executable tests. GPT-4 often generates the tests with the correct testing logic, but it may invoke the functions absent in this projects. The undistinguishable problem is because some branches have long call chains so that challenging to reach.

6.2.3 *Heuristic.*

After leveraging EvoSuite and GPT-4, we further leverage the heuristic rules proposed in Section 4.3. With our heuristic rules, we can handle 8 of 9 conflicting cases related to comments.

Therefore, in total, sMerge can automatically solve 78.43% (120/153) cases and 118 of 120 are correct, the precision is 98.33% and the recall is 77.12%. By comparison, the SOTA techniques MergeBERT and MergeGen achieve a precision of 63.9% and 69.2% respectively, and achieve a recall of 63.2% and 67.7% respectively. The total results on the automatically-solved cases are shown in Table 5. This

```java
public static Map<String, Object> buildOidcServer(final Model
model) {
   final Map<String, Object> m =
   Map.class.cast(model.asMap().get("entity"));
   final String baseUrl = m.get("issuer").toString();
   m.put("token_endpoint", baseUrl + "profile" +
TokenEndpoint.URL);
   m.put("userinfo_endpoint", baseUrl + "profile" +
UserInfoEndpoint.URL);
   m.put("jwks_uri", baseUrl + "profile" +
JWKPublishingEndpoint.URL);
+  final List claimsSupported = new ArrayList((List)
m.get("claims_supported"));
+  claimsSupported.remove("zone_info");
+  claimsSupported.add("zoneinfo");
+  m.put("claims_supported", claimsSupported);
   m.remove("service_documentation");
   m.remove("op_policy_uri");
   m.remove("op_tos_uri");
   return m;
}
```

(a) Conflicting code

```java
@Test
public void testBuildOidcServer()  {
   Model model = new BindingAwareModelMap();
   List<String> claimsSupportedList = new ArrayList<>();
   claimsSupportedList.add("zone_info");
   Map<String, Object> entityMap = new HashMap<>();
   entityMap.put("issuer", "http://example.com/");
   entityMap.put("claims_supported", claimsSupportedList);
   model.addAttribute("entity", entityMap);
   Map<String, Object> result = OidcUtils.buildOidcServer
(model);
   List<String> claimsSupported = (List<String>)
result.get("claims_supported");
   // check if "zone_info" is removed and "zoneinfo" is added.
   assertTrue("Should contain 'zoneinfo'",
claimsSupported.contains("zoneinfo"));
   assertFalse("Should not contain 'zone_info'",
claimsSupported.contains("zone_info"));
}
```

(b) Test generated by GPT-4

Figure 9: The example where EvoSuite cannot produce useful tests but GPT-4 can (shibboleth-oidc-3e183a0)

Table 5: Overall performance on the cases that can be automatically solved.

Test Results	Strategy	Precision	Recall
$I_1 \models T_1 \cup T_2$ and $I_2 \not\models T_1 \cup T_2$	Select $\mathcal{A}$	100.0% (22/22)	68.75% (22/32)
$I_2 \models T_1 \cup T_2$ and $I_1 \not\models T_1 \cup T_2$	Select $\mathcal{B}$	87.5%(14/16)	87.5% (14/16)
$I_{cat} \models T_1 \cup T_2$ and $I_1, I_2 \not\models T_1 \cup T_2$	Concat($\mathcal{A}$, $\mathcal{B}$)	100% (81/81)	78.64% (81/103)
$\exists t_1 \in T_1, t_2 \in T_2.\ t_1 \otimes t_2$	Report Conflicts	100% (1/1)	50% (1/2)
Ratio of cases sMerge can give resolutions: 78.43% (120/153)			
Number of identified wrong labels: 9			
Overall (sMerge)	-	98.33% (118/120)	77.12% (118/153)
JDime [3]	Structure-based	26.3%	21.6%
MergeBERT [35]	Learning-based	63.9%	63.2%
MergeGen [13]	Learning-based	69.2%	67.7%

results indicate that sMerge can automatically solve the majority of the cases and the results of sMerge are sound and trustworthy.

6.3 sMerge with Manually Crafted Tests

In RQ2, we introduce the results on the cases that can be automatically solved. For the rest of 33 unsolved cases, because of the hard-to-reach branches and the limitations of test generation tools, some cases cannot be identified the behavior differences or automatic tools cannot produce executable tests. We manually write tests to validate sMerge, and the total results are shown in Table 6.

Table 6: Performance of sMerge when further supplemented with manual tests.

Test Results	Strategy	Precision	Recall
$I_1 \models T_1 \cup T_2$ and $I_2 \not\models T_1 \cup T_2$	Select $\mathcal{A}$	100% (30/30)	93.75% (30/32)
$I_2 \models T_1 \cup T_2$ and $I_1 \not\models T_1 \cup T_2$	Select $\mathcal{B}$	88.89%(16/18)	100% (16/16)
$I_{cat} \models T_1 \cup T_2$ and $I_1, I_2 \not\models T_1 \cup T_2$	Concat($\mathcal{A}$, $\mathcal{B}$)	100% (102/102)	99.03% (102/103)
$\exists t_1 \in T_1, t_2 \in T_2.\ t_1 \otimes t_2$	Report Conflicts	100% (2/2)	100% (2/2)
Ratio of cases sMerge can give resolutions: 99.34% (152/153)			
Number of identified wrong labels: 12			
Overall	-	98.68% (150/152)	98.04% (150/153)

After we manually supplement the tests, sMerge can give answers to 152 of 153 cases. Although we cannot solve the 33 cases with automatic test generation, sMerge can produce the accurate resolutions when given the tests that reveal the behavioral differences. The precision and recall of each strategy are close to 100%. It is important that developers create/update the tests whenever a modification is made. There are only two cases for which sMerge produces wrong resolutions and one case cannot be predicted the resolution. All three cases are caused by function-irrelevant conflicts, which cannot be depicted by tests. One case is the conflicting comments we mention in Section 6.2.3. Another two cases are caused by the annotation, which adds the annotation "`@SuppressWarnings("unchecked")`" and "`@SuppressWarnings ("deprecation")`" respectively. The annotation will not affect the program behavior so that cannot be identified by tests. Additionally, sMerge is orthogonal to other approaches. When the two versions cannot be distinguished by tests and behaviors, more heuristic rules can be applied. Since these contents are function-irrelevant, they are not as important as the function-relevant contents.

7 DISCUSSION

Applicability of* sMerge*: In this work, sMerge is evaluated on a relatively small dataset. The reason is that we must filter those projects that we fail to execute for various reasons. In practice, we argue that it will not be a limitation since the developers will definitely build and run their projects successfully, and our approach can be integrated to the testing stage of development easily.

Quality of Tests Testing is a common way of evaluating the quality of the software. In this work, we leverage testing to guide merging and validate the semantics of the resolutions. The effectiveness of testing relies on the quality of the test suite. To increase the quality of tests, we adopt two complementary test generation techniques, i.e., Evosuite that focuses on increasing coverage and GPT-4 that aims to "understand" the behavioral difference and generate tests. Nonetheless, the tests may still not capture the key differences between two conflicting versions. It is possible to apply further approaches such as symbolic execution [31, 39] to improve the tests. However, symbolic execution is time-consuming and cannot apply to complex Java programs yet. We do not focus on test generation in this work and remark that our technique can still work to some extent even if the tests are incomplete. Improving the test generation for merging conflicts will be our future work.

Correctness of Commits The developers need to verify the correctness of the commits before submission. In case 1 of Section 4, we will first verify whether each version is consistent with its own tests. If false, the commit should be fixed before sumbitted. If true, we consider the commit correct, so its behaviors satisfy the specifications. Therefore, the test generation techniques can directly use the current behaviors as oracles and verify whether the other version has its behaviors with these oracles. If the submitted code is still incorrect, this is out of our scope. We cannot verify the correctness without neither sufficient tests nor specifications. If provided specifications, sMerge can still work and the general idea is the same, i.e., to find a resolution which can cover the specifications of both conflicting versions. We can generate tests based on specifications or if we have the formal specifications, we can use formal verification techniques. However, specifications are often lacked.

Threats to Validity *Threats to Internal Validity* mainly lie in the implementation accuracy. To mitigate this threat, sMerge is built upon mature and robust tools such as EvoSuite. To guarantee the correctness of the generated tests, we iteratively execute the tests and give the feedback to GPT-4. *Threats to external validity* mainly lie in the benchmark we adopt. To alleviate this, we leverage two benchmarks that are well-constructed in this area. There are many projects that cannot be compiled. This is because the conflicting commits are historical commits, some dated several years back, and the dependencies do not exist any more. To increase the success rate, we leverage multiple Java versions to build until it is compiled successfully. Another main *threat to external validity* lies in the quality of the tests. Our approach requires tests to reveal the behavioral differences between two conflicting versions. However, developers may neglect to update tests and the automatically generated tests may not reveal the key differences. To alleviate the threats, we leverage two automatic test generation techniques to augment the test suite. It has been shown that our approach becomes more effective with the increase of number and coverage of the tests.

Limitations Firstly, sMerge requires that the tests can depict the newly-introduced program behaviors. We believe it is important that developers maintain a quality test suite, and create/update the tests whenever a modification is made. However, developers often neglect to update the tests, which makes the modifications unreliable. We leverage automatic test generation techniques to supplement the tests, which can solve 78.13% of total cases. Actually, this limitation is brought by the non-standard development practice. Secondly, although sMerge is language-independent and can be applied to various languages, we currently focus our evaluation on Java. Evaluating the effectiveness on other languages is a future work to explore.

8 RELATED WORK

Existing conflict resolution techniques can be classified into three major categories, i.e., unstructured techniques, structured techniques, and learning-based techniques.

Unstructured Techniques. Unstructured techniques [1, 19] merge the code based on text and do not leverage other information. They are adopted in the VCS because of the good generalization ability. They can only merge the code not modifying the same locations, so they have limited resolving ability. We regard texts as a poor source of semantic information. To adapt to the widely-used VCS, sMerge resolves the conflicts reported by `git merge`.

Structured Techniques. Structured techniques propose to resolve the conflicts based on structure and syntax. They merge the code by tree AST matching and amalgamation. Moreover, they leverage the structural information of the specific language to help resolve the conflicts [3, 4]. Because structured techniques require operations on trees, the computational complexity is high (cubic at minimum) [3, 25, 26]. To enhance the efficiency, Apel et al. [3] suggest dynamically selecting between unstructured and structured techniques. In addition, structured techniques are language-specific [3, 4]. Structured techniques still revolve the conflicts not from semantics. Moreover, static analysis has many false positives and the precision of JDime (designed for Java) is only 26.3%.

Learning-based Techniques. Recently, learning-based technique [12, 13, 35] are proposed and achieve the SOTA performance. They directly use the deep learning models to learn the merging strategies from the historical data. The input is the conflicting code, and the output is resolution (generation-based) or the resolving strategies (classification-based). Although achieving good performance, the problem definition is unclear. They lack transparency in their decision-making processes, making it challenging to understand or justify the choices. On the contrary, sMerge gives a formal definition of resolving conflicts based on the root cause, which makes the results explainable and trustworthy.

Besides conflict-resolving techniques, there are also some conflict-identifying techniques. Different from them, sMerge focuses on resolving the textual conflicts from the semantic perspective. Several techniques design a representation (e.g., dependency graph [32] and AST [21]) for the dependencies and identify the modifications to the dependent program elements as the potential conflicts, which bring lots of false alarms. Pastore et al. [30] leverages Daikon [15] to discover the properties of function parameters and identify the conflicts, which cannot resolve conflicts. Besides, the properties Daikon discover are irrelevant to the program behavior differences. Some [9, 20, 36] run the build and test process after the code is successfully merged, to discover failures. However, these techniques only execute existing tests to verify the already merged code, which cannot resolve the conflicts. Instead, sMerge generates new tests to identify the behavior differences, which can resolve conflicts.

9 CONCLUSION

We propose a resolution technique from program behaviors. We first give a formal definition of merge problem, which can justify our choices. Furthermore, we resolve the conflicts by identifying the behavioral differences based on testing. Our evaluation shows that sMerge effectively solves this problem from a different perspective.

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China under Grant No. 62372005.

REFERENCES

[1] 2002. diff3. https://linux.die.net/man/1/diff3.

[2] 2024. Replication package. https://github.com/DJjjjhao/ase24-merge.

[3] Sven Apel, Olaf Leßenich, and Christian Lengauer. 2012. Structured merge with auto-tuning: balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 120–129.

[4] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured merge: rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 190–200.

[5] Sven Apel, Jörg Liebig, Christian Lengauer, Christian Kästner, and William R Cook. 2010. Semistructured Merge in Revision Control Systems.. In *VaMoS*. 13–19.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices* 49, 6 (2014), 259–269.

[7] Christian Bird and Thomas Zimmermann. 2012. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[8] 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.

[9] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2013. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1358–1375.

[10] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–27.

[11] Catarina Costa, Jair Figueirêdo, João Felipe Pimentel, Anita Sarma, and Leonardo Murta. 2019. Recommending participants for collaborative merge sessions. *IEEE Transactions on Software Engineering* 47, 6 (2019), 1198–1210.

[12] Elizabeth Dinella, Todd Mytkowicz, Alexey Svyatkovskiy, Christian Bird, Mayur Naik, and Shuvendu Lahiri. 2022. Deepmerge: Learning to merge programs. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1599–1614.

[13] Jinhao Dong, Qihao Zhu, Zeyu Sun, Yiling Lou, and Dan Hao. 2023. Merge Conflict Resolution: Classification or Generation?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1652–1663.

[14] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[15] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*. 213–224.

[16] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. LLM-based Test-driven Interactive Code Generation: User Study and Empirical Evaluation. *arXiv preprint arXiv:2404.10100* (2024).

[17] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[18] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and Andre Van Der Hoek. 2018. 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 (2018), 892–915.

[19] Git. 2023. Git-merge. https://git-scm.com/docs/git-merge.

[20] Mário Luís Guimarães and António Rito Silva. 2012. Improving early detection of software merge conflicts. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 342–352.

[21] Lile Hattori and Michele Lanza. 2010. Syde: A tool for collaborative software development. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 235–238.

[22] ASTM INTERNATIONAL. 2022. Form and Style for ASTM Standards, ASTM Blue Book. (2022).

[23] 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.

[24] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[25] Olaf Leßenich, Sven Apel, Christian Kästner, Georg Seibt, and Janet Siegmund. 2017. Renaming and shifted code in structured merging: Looking ahead for precision and performance. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 543–553.

[26] Tom Mens. 2002. A state-of-the-art survey on software merging. *IEEE transactions on software engineering* 28, 5 (2002), 449–462.

[27] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[28] 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 (2019), 2863–2906.

[29] OpenAI 2023. *GPT-4 is OpenAI's most advanced system, producing safer and more useful responses.* OpenAI. https://openai.com/index/gpt-4/

[30] Fabrizio Pastore, Leonardo Mariani, and Daniela Micucci. 2017. BDCI: Behavioral driven conflict identification. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 570–581.

[31] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2008. Test-suite augmentation for evolving software. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 218–227.

[32] Anita Sarma, David F Redmiles, and Andre Van Der Hoek. 2011. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering* 38, 4 (2011), 889–908.

[33] Bowen Shen, Muhammad Ali Gulzar, Fei He, and Na Meng. 2023. A characterization study of merge conflicts in Java projects. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–28.

[34] Marcelo Sousa, Isil Dillig, and Shuvendu K Lahiri. 2018. Verified three-way program merge. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[35] Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K Lahiri. 2022. Program merge conflict resolution via neural transformers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 822–833.

[36] Jan Wloka, Barbara Ryder, Frank Tip, and Xiaoxia Ren. 2009. Safe-commit analysis to facilitate team software development. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 507–517.

[37] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis. *arXiv preprint arXiv:2404.04966* (2024).

[38] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207* (2023).

[39] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. 2010. Test generation via dynamic symbolic execution for mutation testing. In *2010 IEEE international conference on software maintenance*. IEEE, 1–10.

[40] Fengmin Zhu and Fei He. 2018. Conflict resolution for structured merge via version space algebra. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.