Semistructured Merge with Language-Specific Syntactic Separators

Guilherme Cavalcanti
Federal Institute of Pernambuco
Belo Jardim, Brazil

Paulo Borba
Leonardo dos Anjos
Jonatas Clementino
Federal University of Pernambuco
Recife, Brazil

ABSTRACT

Structured merge tools exploit programming language syntactic structure to enhance merge accuracy by reducing spurious conflicts reported by unstructured tools. By creating and handling full ASTs, structured tools are language-specific and harder to implement. They can also be computationally expensive when merging large files. To reduce these drawbacks, semistructured merge tools work with partial ASTs that use strings to represent lower level syntactic structures such as method bodies, and rely on unstructured tools to merge them. This, however, results in merge accuracy loss. To improve accuracy without compromising semistructured merge benefits, we propose a tool that leverages language-specific syntactic separators to infer structure without parsing. We still resort to an unstructured tool to merge lower level structures, but only after preprocessing the code so that text in between separators such as curly braces appear in separate lines. This way we emulate the capabilities of structured merge tools while avoiding their drawbacks. By comparing our tool with a robust implementation of semistructured merge, we find that our tool substantially reduces the number of spurious conflicts. We also observe significant but less substantial reductions on the overall number of reported conflicts, and of files with conflicts. However, similar to structured tools, our tool lets more merge conflicts go undetected. Our tool shows significant improvements over unstructured tools widely used in practice. Finally we observe that exploiting language-specific syntactic separators introduces unique textual alignment challenges.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; **Software configuration management and version control systems**.

KEYWORDS

Semistructured merge; Merge conflicts; Configuration management; Software evolution

1 INTRODUCTION

Unstructured merge tools have been in use for over 40 years [1], and are essential for team software development. These tools rely solely on textual analysis to detect and resolve merge conflicts [2]. They are fast and can handle any type of textual content, including source code in any language. However, they often report spurious merge conflicts, and fail to report conflicts that could lead to compilation and execution issues. For example, unstructured merge reports a conflict when asked to integrate rearrangeable declarations (such as method and field declarations) separately added by two developers to the same area of the source code [3]. Simply juxtaposing them in any order would be a correct merge, but an unstructured tool is not able to do that as it recognizes program lines instead of declarations.

To improve merge accuracy, researchers have proposed tools that consider the structure of the source code being merged [3–13]. Such structured merge tools go beyond simple textual, line-based, analysis by parsing the code and handling abstract syntax trees (ASTs). They employ tree matching and combination algorithms. Because of that, they are language-specific and demand significant implementation effort for each language they handle. They can also be computationally expensive when merging files with many or large declarations [9].

To reduce these drawbacks, researchers proposed semistructured merge tools [14–16] that attempt to hit a sweet spot between unstructured and structured merge by partially exploring the underlying programming language syntactic structure. The idea is to work with partial ASTs that use strings to represent lower level syntactic structures such as method bodies, statements, and expressions. To merge these parts of the code, semistructured tools simply rely on textual, line-based, analysis by invoking an unstructured tool. Although bringing significant benefits in comparison to unstructured merge, this, however, results in merge accuracy loss comparing with structured merge [17, 18].

To improve accuracy without compromising semistructured merge benefits, here we propose SESAME, a semistructured merge tool that leverages language-specific syntactic separators (like '{' and '}' in Java) to infer structure without parsing. We still resort to an unstructured tool to merge lower level structures, but only after preprocessing the code so that text in between separators appear in

separate lines. This way we more closely emulate the capabilities of structured merge tools while avoiding their drawbacks.

Whereas structured tools rely on the occurrence of specific token sequences to determine which AST node to create and latter match, our tool relies on separators to delimit areas of the code to match for textual merge. Thus, on lower level structures, Sesame actually matches sequences of characters delimited not only by '\n' (as in unstructured, line-based tools) but also by language-specific separators. Line-based tools solely rely on line breaks as the context delimiter for conflict detection, reporting conflicts whenever one tries to integrate changes made by two developers to the same or consecutive lines. In contrast, Sesame only reports conflicts when one tries to integrate changes made to the same text area delimited by the separators.

We evaluate Sesame by comparing it with a robust implementation of standard semistructured merge, on a dataset of merge scenarios— quadruples (*base*, *left*, *right*, *merge*), where *left* and *right* are the parent commits integrated by *merge*— from open-source projects gathered from GitHub. We replay the merge operation with the two tools on those scenarios to collect evidence on how often the tools report different results, merge conflicts, files with merge conflicts, false positives (conflicts incorrectly reported by a merge tool), and false negatives (actual conflicts missed by a merge tool). To also assess the benefits of our tool in relation to the standard practice, we also compare it with an unstructured merge tool. In particular, we answer the following research questions:

- **RQ1**: Does Sesame reduce the number of merge conflicts and files with merge conflicts compared to standard semistructured and unstructured merge tools?
- **RQ2**: Does Sesame reduce the number of merge conflicts identification errors compared to standard semistructured and unstructured merge tools?

Our results show that Sesame outperforms standard semistructured and unstructured merge in terms of conflict reduction; our tool reduces the number of merge conflicts by 41% and 13%, respectively. We observe a similar pattern in favour of our tool when considering the number of files with merge conflicts. Additionally our tool is far better in avoiding spurious conflicts (false positives); it substantially reduces them in comparison to unstructured and semistructured merge (91% and 88%, respectively). However, similar to structured tools and conforming to what is observed in previous studies [15–18], the increased ability to resolve merge conflicts comes at the price of missing actual, proper conflicts. Sesame is no different: we found no evidence that Sesame misses fewer actual conflicts (false negatives) than unstructured and standard semistructured merge. Based on these findings, Sesame makes a step forward on dealing with the merge inaccuracy of the state-of-practice textual merge tools without the implementation effort and runtime overhead of structured merge tools. We find evidence that our tool shows significant improvements over unstructured tools widely used in practice. Finally we observe that exploiting language-specific syntactic separators introduces unique textual alignment challenges.
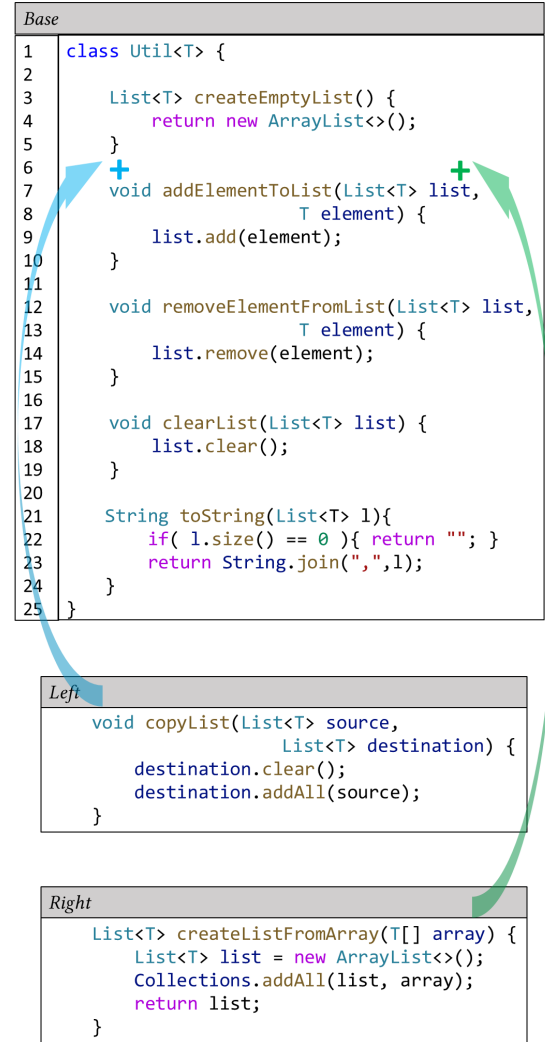

Figure 1: Independent changes to the same *Util* class.

2 MOTIVATION

To better explain the differences between the kinds of merge tools we discuss in the previous section, consider the example in Figure 1. On the top it shows the base version of the *Util* Java class. Consider that two developers, *Left* and *Right*, independently introduce declarations to that class. The first developer, adds the *copyList* method declaration right before the *addElementToList* declaration. Unaware of *Left*'s modification, *Right* adds the *createListFromArray* method declaration, at the same text area, but in its local copy of the file. Subsequently, the task at hand is to merge these disparate local file versions into a unified version of the *Util* class. The goal is to synthesize a consolidated version of the class that includes the changes introduced by both developers.

An unstructured merge tool contrasts the two modified files, line by line, relative to their shared common ancestor, identifying distinct sequences of lines known as "chunk". The tool then verifies whether the three file versions (*base*, *left*, *right*) share a common

text region that separates the contents of the modified chunks. If such a separator is not found, the tool reports a merge conflict. In our example, an unstructured merge tool reports the merge conflict illustrated in Figure 2. As both developers change line 6 of the *Base* version of the file, there is no common separator, so unstructured merge is unable to integrate the changes.

```
Unstructured Merge
1    class Util<T> {
...
6    <<<<<<<<<<<<
5        void copyList(List<T> source,
6                        List<T> destination) {
7            destination.clear();
8            destination.addAll(source);
9        }
10   ============
11       List<T> createListFromArray(T[] array) {
12           List<T> list = new ArrayList<>();
13           Collections.addAll(list, array);
14           return list;
15       }
16   >>>>>>>>>>>>
...  }
```

Figure 2: Unstructured merge of different versions of the same *Util* class resulted in merge conflict.

Note, however, that both changes could be safely integrated by a merge tool aware that the changes correspond to unrelated Java method declarations, whose order does not matter. Such a tool could simply juxtapose the declarations, no matter in which order. This motivated the development of syntax-based, structure aware merge tools, such as structured and semistructured merge tools [3, 9–15]. Such tools generate abstract syntax trees (ASTs) for each version of the files being integrated. These trees are then compared to identify common, introduced, or removed nodes in each tree. The tools then report merge conflicts when changes are related to the same tree node. As a result, semistructured and structured merge tools avoid spurious conflicts such as the one in Figure 2. In that example, the changes occur on the same text area, but each method declaration is attributed to a distinct AST node. Consequently, semistructured and structured merge tools interpret that these modifications are unrelated, and correctly accept both contributions, yielding the class version in Figure 3.

Now consider that besides adding the *copyList* and *createListFromArray* method declarations, the developers change the *toString* method in different ways. As illustrated in Figure 4, this method receives a list as argument and returns a string. *Left* changes the `if` statement condition to check whether the list is `null` or empty. Independently, *Right* extracted a constant from the condition body, which happens to be in the same line of the condition changed by *Left*.

A semistructured merge tool reports a conflict in this case because it invokes an unstructured tool to merge low level syntactic structures such as method bodies and statements. As the changes to be merged occur on the exact same line of code, we get the conflict shown in Figure 5. Once again this represents a spurious merge conflict. *Right*'s change is a refactoring, thus being semantically

```
Semistructured and Structured Merge
1    class Util<T> {
...
6        void copyList(List<T> source,
5                        List<T> destination) {
6            destination.clear();
7            destination.addAll(source);
8        }
9        List<T> createListFromArray(T[] array) {
10           List<T> list = new ArrayList<>();
11           Collections.addAll(list, array);
12           return list;
13       }
...
15   }
```

Figure 3: Structured and semistructured successful merge of different versions of the same *Util* class.

```
Left
String toString(List<T> l){
    if( l==null || l.isEmpty() ){ return ""; }
    return String.join(",",l);
}
```

```
Right
String toString(List<T> l){
    if( l.size() == 0 ){ return EMPTY; }
    return String.join(",",l);
}
```

Figure 4: Independent changes (highlighted in yellow) to the same *toString* method.

neutral and free of any interference with the changes made by *Left*. Semistructured merge, by only partially exploiting the syntactic structure of the code fails in this case. Structured merge, however, represents method bodies as ASTs and reports no conflict in this case because it detects that the changes occur in different AST nodes: one that represents the condition of the `if` statement, and the other that represents its body.

SESAME would yield the same result as structured merge in this case, as we will further discuss in the next section. Although our tool does not represent statements as ASTs, avoiding the major drawbacks of structured tools, it exploits the '{' separator that delimits the beginning of the `if` body.

3 SEMISTRUCTURED SEPARATOR-BASED MERGE

Previous studies demonstrate that semistructured merge offers significant advantages over traditional unstructured merge for resolving merge conflicts [14, 15, 17–19]. However, by design, semistructured merge inherits some of the inaccuracies associated with unstructured merge when merging lower level syntactic structures such as statements and expressions, as illustrated in the previous section. This occurs because typical semistructured merge tools
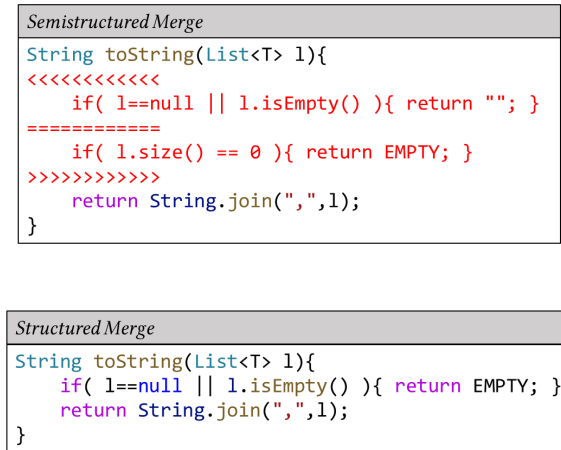
```
Semistructured Merge

String toString(List<T> l){
<<<<<<<<<<<<
    if( l==null || l.isEmpty() ){ return ""; }
============
    if( l.size() == 0 ){ return EMPTY; }
>>>>>>>>>>>>
    return String.join(",",l);
}
```

```
Structured Merge

String toString(List<T> l){
    if( l==null || l.isEmpty() ){ return EMPTY; }
    return String.join(",",l);
}
```

Figure 5: Structured and semistructured merge of different versions of the same *toString* method.

only create AST nodes up to class-level declarations in Java-like languages. So such tools will have nodes for declarations such as imports, fields and methods, but these nodes contents are not represented as ASTs. The remaining programming language elements are simply represented as plain text, and are merged using a standard unstructured merge algorithm.

To avoid the merge inaccuracies resulting from that design decision, one could increase the level of structure and detail in the ASTs generated by semistructured merge, aligning it more closely with structured merge. This, however, cannot be done without inheriting structured merge drawbacks of requiring an excessively language dependent implementation that requires substantial per language effort. Additionally, as the level of detail in ASTs increases, so does the computational costs associated with matching these trees with expensive algorithms. Although such costs are likely manageable for most merge scenarios [3, 9, 20], it is a factor that should be taken into consideration for more complex merge scenarios.

To balance these two opposing design dimensions, and mitigate the *inherited* inaccuracies of semistructured merge, we propose a technique that further explores the syntactic structure of source code while retaining the current level of detail of the generated ASTs of semistructured merge. We basically refine the unstructured merge process invoked by semistructured merge for merging low level syntactic structures. Our idea is to consider language-specific separators as delimiters for matching and conflict detection. So, instead of relying solely on line matching, we propose a character sequence matching process that takes into account not only '\n' as a separator, but also language-specific syntactic delimiters such as '{' and '}' in Java.

Line-based tools solely rely on line breaks as the context delimiter for conflict detection reporting conflicts whenever one tries to integrate changes made by two developers to the same or consecutive lines of the program text. In contrast, our technique only reports conflicts when one tries to integrate changes made to the same text area delimited by the separators. By doing so, we effectively avoid the false merge conflict illustrated in Figure 5. Our

technique considers '{' as a delimiter for text areas, thus recognizing that the changes to be integrated occur in distinct regions: before and after the '{' in the second line of the *toString* method declaration. The result is analogous to a full structured merge in that case, but without its associated drawbacks.

For most programming languages one can easily determine important separators for blocks, statements, and so on. However, as the exact set of separators used for a language can impact results, determining such a set might require running a few simple experiments and understanding how the separator-based merge algorithm works. For example, for languages like Python, it is slightly harder because indentation and line breaks work as separators.

We implement this technique in the Sesame tool, which builds upon both the widely used unstructured merge tool diff3 and the robust semistructured merge tool s3m [15], which is specific for merging Java code. We still resort to the unstructured tool to merge lower level structures, but only after preprocessing the code so that text in between separators appear in separate lines.

It works as follows:

(1) It inserts a new line before and after each language-specific separator (e.g. '{' and '}') in the different versions (*base*, *left*, *right*) of the lower level syntactic structures to be merged.
(2) It marks the newly added lines with a distinctive placeholder (we adopt a stream of '$' characters).
(3) It invokes unstructured merge on the marked versions of the lower level syntactic structures.
(4) It removes the new lines and placeholders from the result of unstructured merge.

To illustrate this process, consider again the example in Figure 4. Consider that Sesame is customized to use only curly braces ('{' and '}') as language-specific separators. In Java, these characters define the start and end of a code block, creating distinct text areas for analysis and matching. Initially, our tool inserts the new lines and placeholder both before and after every '{' and '}'. This action is carried out individually for each file to be merged by the tool, generating a total of three temporary shadow files, each one corresponding to distinct versions of the original file: the base, left the right versions, shown in Figure 6.

Next Sesame invokes unstructured merge on the three temporary files. The result of this step is then depicted in the bottom of Figure 6. Note that the merging process unfolds correctly, as evidenced by the successful accommodation of the changes, without merge conflicts. This success can be attributed to the fact that, now, the changes are placed in distinct text areas due to Sesame preprocessing: while changes originating from the left version are located in line 4, those originating from the right version appear in line 6. This distribution of the changes across different and non-consecutive lines enables unstructured merge to properly merge the changes. Moreover, the tool preserves the lines that remain unaffected by the changes from either the left or right versions, ensuring that they are in the final merged output.

Finally, in the last step, Sesame removes from the resulting file all the new lines and placeholders created before, generating its final result, which in this case is identical to the one yielded by a structured merge tool, shown before in Figure 5. In situations where conflicts arise during merging— such as when both the left and

right sides independently modify the condition of an `if` statement—Sesame employs the same process. However, in these conflicting scenarios, our tool reports the conflicting elements without the inclusion of the additional lines and placeholders introduced in its preprocessing phase.

Sesame can be used as a substitute for the merge algorithms adopted by widely used tools such as `git merge` and `rebase`. This can be done, for example, by implementing a *git merge driver*, a custom script that git uses to handle file merging. So when `git merge` is invoked, Git can be configured to delegate the merge to a specified driver. This allows for more specialized merge algorithms that are tailored to the specific needs of the project or the file types being merged, in our case, Java files. So this allows developers to leverage the specialized merging capabilities of Sesame directly within Git workflows.

4 EMPIRICAL EVALUATION

To assess Sesame's potential for code merging, we compare it with semistructured and unstructured merge tools. The comparison with the first allows us to evaluate our separator-based technique in isolation, understanding how it advances previous results on semistructured merge. The comparison with the latter provides insights into the benefits that our tool could bring to the standard practice.

In both cases, we evaluate the effectiveness of the merge tools using metrics that capture the tools' capacity to resolve merge conflicts while preserving the integrity and correctness of the merging process. This is essential to understand whether Sesame can increase development productivity— by reducing wasted time on resolving spurious conflicts— without compromising software quality— by letting actual conflicts escape. Our goal is to understand the strengths and weaknesses of Sesame compared to other merge tools, so that software development teams can use that as input when deciding to adopt merge tools.

In particular, we observe metrics such as the number of merge conflicts and the number of files with conflicts. This evidence, however, is not enough to justify adoption of a merge tool. So we also investigate false positives and false negatives. The challenge associated with such a comparison criteria is establishing ground truth for integration conflicts, as this is not computable in this context [21, 22]. Semantic approximations through static analysis or testing are imprecise and often expensive. Experts who understand the integrated code (possibly developers of each analyzed project) could determine truth, but not without the risk of misjudgment.

As these options would imply into a reduced sample and limited precision guarantees, we prefer to *relatively* compare the merge tools in pairs, regarding the occurrence of false positives and false negatives of one tool *in addition* to the ones of the other. We do that by simply analyzing when two merge tools report different results for the same merge scenario — a quadruple of commits, which we will call *base*, *left*, *right*, and *merge* commits, this last one being the final version of the merged code stored in the original project repository we analyze in our experiment. We identify files with merge conflicts reported by one tool but not by the other (*added false positive (aFP)*), and files with merge conflicts reported by one tool but missed by the other (*added false negative (aFN)*). Actual false positives and false negatives common to both tools

are ignored, as we observe only situations in which the tools differ. This is aligned with our interest in relatively comparing the tools.

4.1 Research Questions

In summary, we evaluate Sesame by addressing the following research questions. Further details are provided in Section 4.3.

4.1.1 **RQ1**: *Does Sesame reduce the number of merge conflicts and files with merge conflicts compared to standard semistructured and unstructured merge tools?* To answer this research question, we analyze the number of merge conflicts reported by each merge tool, and the files having these merge conflicts. This involves running the merge tools on the files associated with the base, left, and right commits of each merge scenario within our sample. Then, we count each conflict block reported by each tool, and we count the files in which these conflict blocks appear. A conflict block reported by these tools is a set of lines enclosed by the standard conflict markers ">>>>>>>" and "<<<<<<<" as ilustrated before in Figures 2 and 5.

4.1.2 **RQ2**: *Does Sesame reduce the number of merge conflicts identification errors compared to standard semistructured and unstructured merge tools?* To address this research question, we count added false positives and false negatives as indicators of potential errors in merge conflict identification. It is important to emphasize that our focus here lies in comparing merge tools in a relative context rather than determining their absolute accuracy regarding a broad definition of conflict. Therefore, we do not need to assess the frequency of false positives and negatives when the merge tools exhibit identical behavior.

4.2 Sampling

We evaluate the merge tools with merge scenarios from the GitHub open-source Java projects listed in Table 1. Our project list includes projects found in previous studies [9, 12, 15, 17–19]. However, we observed that many of the projects in those datasets are no longer relevant. Therefore, we decided to focus our study exclusively on widely-adopted Java projects and selected them from these other datasets. In particular, we selected these projects based on the following criteria:

- Popularity, based on their number of stars on GitHub. All projects have at least 700 stars;
- Number of collaborators, with at least 50 collaborators. A larger number of collaborators might be an indicative of concurrent work, increasing the likewise of changes that need to be merged;
- Presence of merge commits on projects' repository history. This is necessary so that we can derive merge scenarios to replicate the merge operation.

To conduct our comparative analysis of these merge tools, we initially compile a dataset of merge commits from our sample projects spanning the period from December 2019 to October 2021. Each merge commit represents a merge operation within the project's history and serves as the basis for deriving a merge scenario. We then exclude fast-forward merges from this dataset, as they cannot cause merge conflicts; and octopus merges, as Sesame current implementation only supports merging two branches.
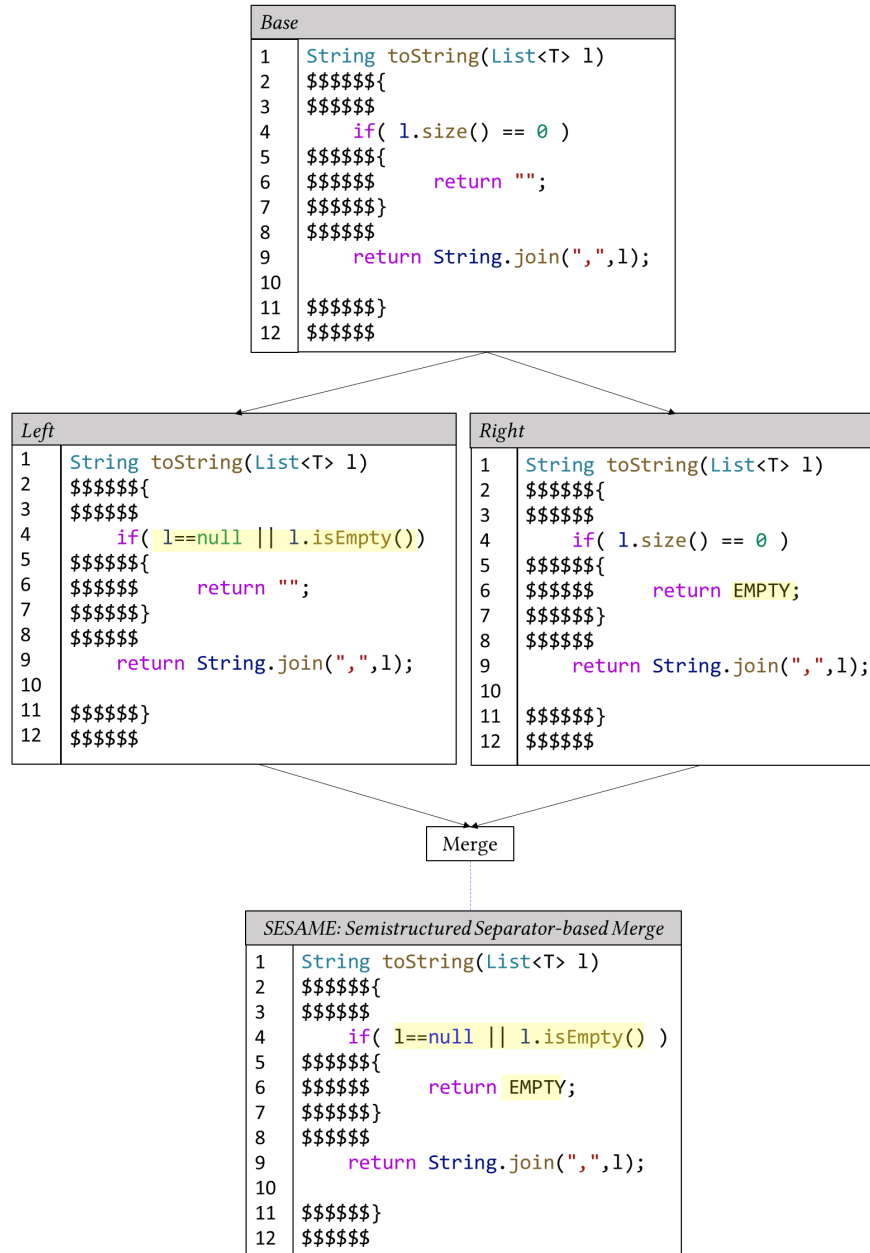
Figure 6: *toString* method code transformation by Sesame. Developers' changes highlighted in yellow.

For each merge commit, we identify the corresponding merge scenario, which consists of a quadruple of commits. This quadruple comprises the merge commit itself, representing the official merge result in the project's repository, and the commits associated with the three versions involved in a three-way merge: the base commit, the left commit, and the right commit. Given that each merge scenario contains four sets of files—one for each of the aforementioned commits — we use the merge tools to merge the files within the base, left, and right commits. We then store the results of these merge operations for each merge scenario. As consequence of this

step, we perform our study on 9510 sets of files from 1054 merge scenarios from our sample projects.

4.3 Methodology

To answer our research questions we gather evidence regarding the frequency of divergent results among the tools we compare, as well as instances of errors in conflict detection, namely added false positives and false negatives. We adopt the state-of-practice diff3 as standard unstructured merge tool, the state-of-art s3m [15] as standard semistructured merge tool, and our solution Sesame

Table 1: Overview of the sample projects.

GitHub Repository	Stars	Contributors
apache/accumulo	880	132
FasterXML/jackson-databind	2.9k	175
HubSpot/Singularity	799	80
spring-projects/spring-boot	56.3k	827
mockito/mockito	12k	221
spring-projects/spring-framework	43.6k	544
libgdx/libgdx	18.7k	511
jenkinsci/jenkins	17.6k	659
alibaba/fastjson	23.6k	173

configured with the following Java language-specific syntactic separators: '{', '}', ',', '(', ')' and ';'. We have experimented SESAME with a reduced set of separators ('{', '}' and ','), but we found that this led to a 16% increase in the conflict rate. This suggests that the fewer the separators, the closer SESAME behaves to pure semistructured merge, while the more separators, the closer SESAME behaves to structured merge. Therefore, we opted for the larger set in our study.

Following the merging process, we systematically collect data on the number of merge conflicts, the prevalence of conflicting files, added false positives, and added false negatives. A conflicting file is defined as a merged file containing at least one merge conflict. In cases where the merge results generated by the merge tools differ, we compare these results with the merge result in the repository, and collect evidence for further analysis, particularly concerning false positives and false negatives. We consider merge tools to differ when (1) they report a different number of conflicts for the same file, or (2) the number of merge conflicts is the same, but the content of the files, ignoring whitespace (as it is semantically neutral in Java), is different. In particular, when comparing two merge tools, M and N, we compute an occurrence of added false positives to M when a file has at least one merge conflict reported by M, while none are reported by N. Moreover, for these cases, N's result must align with the outcome in the original project repository (obtained via the merge commit), indicating N's successful and accurate resolution of merge conflicts in the file. Conversely, we compute an added false negatives to M when it fails to report any merge conflicts in a file, yet the merged file deviates from the result in the repository merge commit. Furthermore, we require tool N to report merge conflicts in the same file. In such instances, we consider that tool M inaccurately resolved the merge conflicts, resulting in syntactically or semantically incorrect code compared to the expected outcome.

Finally, to consolidate our metrics on merge conflict identification errors, we conducted a manual analysis of files that exhibited merge conflicts with one merge tool but not the other—specifically, files with potential false positives and false negatives. For these files, we compared each tool result with the original merge commit in the project repository (which is assumed to be the result desired by users) to check whether there was any actual interference between the integrated changes. The analyses were independently conducted by at least two authors, and special cases jointly discussed by three authors.

We provide the scripts and data associated with this study in a permanent publicly accessible repository.[1]

5 RESULTS AND DISCUSSION

Next, we present and discuss our findings, structured according to our the research questions.

RQ1: *Does SESAME reduce the number of merge conflicts and files with merge conflicts compared to standard semistructured and unstructured merge tools?*

Table 2 shows that embodying semistructured merge with language-specific syntactic separators, through the implementation of SESAME, resulted in a reduction not only in merge conflicts, but also in files having these merge conflicts. This applies when we compare SESAME to both the conventional unstructured merge diff3 tool, and the semistructured merge s3m tool. Specifically, SESAME reduced the number of merge conflicts by approximately 41% and the number of conflicting files by about 40% compared to diff3. Additionally, when comparing SESAME to s3m, it can be observed that SESAME managed to decrease the number of merge conflicts by around 13% and the number of conflicting files by approximately 21%.

	diff3	SESAME	s3m
Merge conflicts	2413	1413	1632
Conflicting files	1090	657	832

Table 2: Comparison of merge conflicts and conflicting files.

Our solution is driven by the aspiration to leverage the benefits of a fully structured merge tool, while simultaneously minimizing the costs typically associated with such tools. We aimed to strike a balance between the advantages in merge conflicts detection of structured merge and the practical considerations of implementation complexity and resource requirements. By comparing our findings with prior research, we have strong indications that our objective has been met. In particular, our results closely mirror those documented by Apel et al. [3] when they evaluated JDime, which is, to the best of our knowledge, the most representative structured tool for Java applications. They reported a 39% reduction in merge conflicts when comparing their structured tool against traditional unstructured merge. This alignment between our findings and prior research not only validates the efficacy of our tool but also reinforces the potential benefits of leveraging structure to reduce merge conflicts during the merging process.

Finally, the SESAME tool reduced the number of merge conflicts in our sample, but the specific merge conflicts reported by each merge tool are not necessarily identical. This discrepancy means that one tool might identify certain conflicts that another tool misses, and vice versa (as we will further discuss on the next research question), or a tool might report the same conflict in a different manner. Therefore, to gain a deeper understanding of the differences in the SESAME's behavior, it is essential to analyze not just the overall number of conflicts each merge tool reports, but also the frequency and nature of the instances where the tools diverge.

---

[1]https://doi.org/10.5281/zenodo.13368253

	diff3 vs.SESAME	diff3 vs.s3m	s3m vs.SESAME
Total	1145	1019	675
(%)*	12.04%	10.72%	7.10%

* In relation to the total number of merged files.

Table 3: How often the merge tools differ.

In that sense, Table 3 illustrates that when we textually compare the outputs produced by the merge tools, SESAME diverges from diff3 in approximately 12% of the 9510 merged files in our sample. Additionally, SESAME differs from s3m in about 7% of these files.

When inspecting the output produced by SESAME, we observed that the tool generates several new lines, one for each language-specific syntactic separator. This results in situations where different alignments emerge when running unstructured merge on these files as part of the SESAME process described in Section 3. These problematic alignments occur because SESAME identifies a greater match between files using the new, smaller lines created by the algorithm. This finer granularity introduced by the syntactic separators can sometimes lead to unexpected alignment issues, as the invoked unstructured merge tool may perceive the newly created lines as distinct changes. This phenomenon is a source of SESAME's added false positives, as we further elaborate next in our discussion of the second research question.

RQ2: *Does SESAME reduce the number of merge conflicts identification errors compared to standard semistructured and unstructured merge tools?*

Assessing the effectiveness of a merge tool involves more than just evaluating its merge conflict reduction capabilities; it also demands an analyzes of its conflict detection accuracy. Spurious merge conflict detection can significantly impede developers' productivity by leading them to spend valuable time resolving conflicts that are not genuine. Conversely, when a tool overlooks merge conflicts, it opens the door to potential syntactic and/or semantic errors in the source code, which can undermine the quality and stability of the software being developed. Therefore, a comprehensive evaluation of a merge tool should consider both its conflict reduction efficacy and its precision in identifying genuine conflicts. Thus, to answer our second research question, we relatively compare the merge tools in terms of added false positives and added false negatives as described in Section 4.3.

Starting with the false positives, Table 4 shows that SESAME managed to reduce the number of added false positives in relation to both diff3 (unstructured merge) and s3m (standard semistructured merge). In particular, SESAME exhibited a significant reduction in the number of added false positives by approximately 91% compared to diff3 and by approximately 88% compared to s3m. Note that for class-level declarations such as methods and fields, SESAME and s3m behave the same way by definition. This is because both tools handle class-level merge conflicts using the same semistructured approach. Conversely, at the statement level, s3m and diff3 exhibit the same behavior, as s3m relies on the traditional unstructured approach of diff3. However, SESAME is enhanced by language-specific syntactic separators, which allows it to handle statement-level conflicts more effectively. This distinction explains the comparable improvements

seen with SESAME when compared to the other tools. Specifically, SESAME reduces false positives from diff3 at the class level by leveraging its semistructured approach, such as merge conflicts caused by the (re)ordering of methods and field declarations as illustrated before in Section 2. Additionally, SESAME reduces false positives from both diff3 and s3m at the statement level due to its enhanced handling of language-specific syntax thanks to the separators. This improvement is particularly evident in cases involving changes to unrelated variable declarations or assignments on consecutive lines within a method declaration.

Nevertheless, SESAME approach leads to its own false positives. The occurrence of these false conflicts can be attributed to the way SESAME handles syntactic separators. By isolating these separators into new lines, SESAME alters the structure of the code as demonstrated before in Figure 6, which can sometimes lead to unexpected alignments. This textual misalignment occurs because SESAME, when performing the unstructured merge as part of its process, becomes "confused" as it attempts to identify and align the modified text areas from the left and right versions. Consequently, unstructured merge misinterprets these changes, resulting in false conflicts that would not have been present if the separators were not isolated into separate lines.

Regarding false negatives, SESAME follows a trend observed in prior studies [15–18]: the more a merge tool leverages the structure of the code being merged, the higher the likelihood that some merge conflicts will go undetected. Note in Table 4 that both SESAME and s3m show an increase in the number of added false negatives compared to diff3. Additionally, SESAME, though to a lesser extent, also shows an increase in added false negatives compared to s3m. This phenomenon occurs because structured merge tools, while being more sophisticated in understanding and integrating changes based on the syntactic and semantic structure of the code, can sometimes overlook conflicts that simpler, unstructured tools would

	diff3	SESAME
aFP	291	26
aFN	0	168

	s3m	SESAME
aFP	146	18
aFN	1	48

	diff3	s3m
aFP	169	31
aFN	4	124

Table 4: Comparison of merge conflicts identification errors in terms of false positives (aFP) and false negatives (aFN).

catch *accidentally* as they threat everything as lines of text. By focusing on the low-level, finer granularity, organization of the code, the structure-driven tools may miss subtle conflicts among statements or expressions, leading to an increase in false negatives.

For instance, consider the example taken from project SpringFramework in Figure 7. In that scenario, both developers moved the `return` statement to within an `if-else` statement. However, in the left version, the developer also changed the type of a variable from an array to a List. When merging these changes, both `diff3` and `s3m` report a merge conflict because part of the changes occurred on consecutive lines of code. In contrast, Sesame does not report merge conflict because the ";" in line 2 act as a separator of context. However, upon inspecting the original merged version in the merge commit (as illustrated in Figure 7 also), we observed that the output from Sesame did not match the original version: the integrator chose to keep the array from the base version instead of the List from the left version.

Additionally, the textual misalignment problem explained before, besides generating false conflicts, also causes Sesame to sometimes fail to generate a conflict during the merging process. To understand how the misalignment leads to an added false negative, consider a base file that has the line "`a().b(c).d();`" with chained method calls. Suppose that one of the developers modified the argument of method b, resulting in "`a().b(e).d();`". Meanwhile, the other developer replaced the call to method b with a call to g as follows: "`a().g(h(c)).d();`". When integrating these versions, Sesame, using the parentheses as separators, ends up aligning the part "`(c)`" from the base with the one from right. By adding a new line before and after each separator, Sesame interprets these lines as forming the largest alignment and, because of this, it accepts both modifications without conflict, resulting in "`a().g(h(e)).d();`". However, if there are semantic dependencies between the modifications, Sesame would let merge conflict goes undetected.

Classifying merge conflicts as either false positives or false negatives is a delicate task because it requires understanding whether and how the changes are related. At the statement and expression levels, this task might involve comprehending the behavior of the code and analyzing other files, which extends beyond the scope of the studied tools. As a result, a compromise exists: unstructured tools tend to be overly conservative, treating everything uniformly as text, whereas structure-driven tools are more optimistic, assuming that changes to different code structure elements are safe to merge. This trade-off reflects the challenges inherent in balancing precision and comprehensiveness in merge conflict identification and resolution.

Given that, in our context, false positives might lead to a reduced productivity [23], increasing the chances of resolving more false positives, even at the expense of introducing new false negatives, can be a viable design option for a new merge tool. In practice, stricter, conservative tools, such as unstructured merge tools, that report more conflicts might be preferable in contexts where developer effort is less critical for handling false positives, and where automated tests are not robust enough to reliably catch false negatives. On the other hand, tools that prioritize reducing spurious conflicts may be more suitable in environments where developers are frustrated by false positives, but strong test suites provide confidence that most false negatives will be caught during build and

test processes. Nevertheless, adopters should consider the potential benefits of combining different tools [11, 24–28] to create a more seamless and less disruptive merging process, as we further discuss in Section 6.3. By leveraging the strengths of multiple tools, it may be possible to mitigate the limitations of any single approach, improving the efficiency and accuracy of merge conflict resolution. As merging strategies and project requirements can vary, using a hybrid approach may offer a more adaptable and effective solution for handling diverse scenarios.

5.1 Threats to Validity

The generalizability of our results may be limited by the specific programming language and types of projects included in our evaluation. Our approach leverages Java language-specific syntactic separators, so its effectiveness may vary across different languages and coding styles. So, further studies are necessary to confirm the tool's applicability in other contexts.

Besides that, the accuracy of our merge conflicts identification errors metrics, in particular false negatives, is a potential threat. There is the possibility that the developer modified the code during the merge process. So we cannot effectively identify false negatives. Therefore, we conducted a manual analysis of 93 merged files as explained in Section 4.3. Given that this analysis was manual, there is a risk of incorrect judgment in evaluating these cases. However, we tried to mitigate these risks by involving multiple authors in the process.

Finally, our sample represents only a subset of the actual code integration scenarios within the analyzed projects. Specifically, we only analyzed scenarios that culminate in public repository merge commits. This excludes integrations performed using `git rebase` or other Git commands that rewrite history, which do not result in merge commits and therefore are not captured in our analysis. We also excluded octopus merges since Sesame current implementation only supports merging two branches. While merging more than two branches could be done pairwise, it would create multiple merge commits, which octopus merges aim to avoid. We believe no technical decision in our approach precludes one from implementing an octopus strategy on top of it, but we have not explored this, as the strategy is not commonly used. Other research merge tools also do not implement octopus as far as we know.

6 RELATED WORK

The challenge of efficiently detecting and resolving merge conflicts in collaborative software development has led to significant research in the field of version control systems.

6.1 Semistructured merge

Apel et al. [14] proposed semistructured merge, a merge approach that *partially* leverages the syntactic structure and static semantics of the underlying language, but without the performance overhead associated with fully structured merge. They implemented semistructured merge in the tool `FSTMerge` for C#, Java, and Python. Tavares et al. [16] later extended `FSTMerge` to support JavaScript, finding that the benefits of semistructured merge might be limited for languages that support both commutative and non-commutative declarations at the same syntactic level, such as JavaScript.
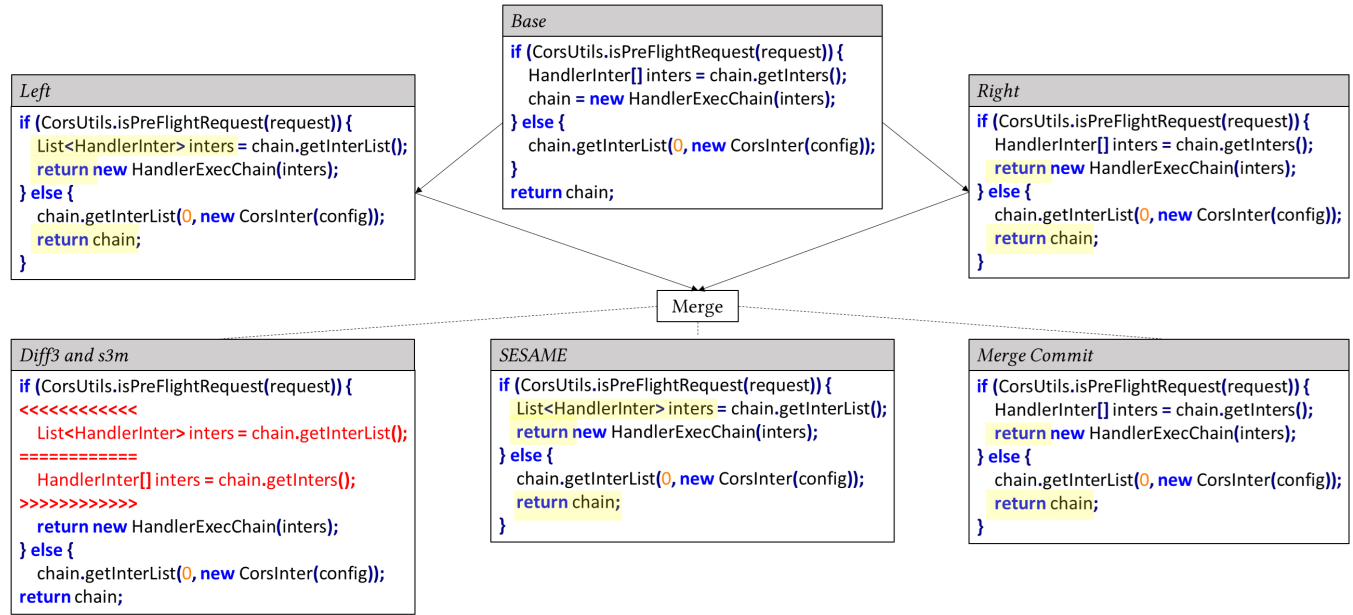
Figure 7: Merging *AbstractHandlerMapping.java* from project SpringFramework (changes highlighted in yellow).

Studies by Cavalcanti et al. [15, 19] provide evidence that semistructured merge can reduce the number of reported conflicts compared to traditional unstructured merge, though this benefit is not consistent across all projects and merge scenarios. Additionally, Cavalcanti et al.[15] found that semistructured merge significantly reduces the number of false positives, but does not necessarily lead to fewer false negatives. Cavalcanti et al. [17] further compared semistructured merge with structured merge. They found that semistructured and structured merge differ in 24% of the merge scenarios with merge conflicts, semistructured merge reports more false positives, whereas structured merge has more false negatives. In our work, we found that semistructured merge effectively reduces merge conflicts and false positives. However, it also exhibits a tendency towards false negatives, a tendency amplified by our approach that aims to mimic structured merge.

Seibt et al. [18], in a comprehensive evaluation of various merge algorithms (ranging from unstructured to semistructured to structured, as well as combinations of them), found that combined strategies offer the best of both worlds. Their evidence suggests that these hybrid approaches can resolve as many merge conflicts as structured merge algorithms while achieving significantly lower runtimes per merge scenario. Our work follows this direction by further combining unstructured and semistructured in a merge tool designed to mimic structured merge.

6.2 Structured merge

Apel et al. [3] introduced `JDime`, a state-of-the-art structured merge tool for Java applications. `JDime` dynamically switches between unstructured and structured merge depending on the presence of conflicts, optimizing the merging process. This work has inspired numerous recent advancements in the field. For instance, Lessenich

et al. [9] enhanced `JDime` by incorporating a syntax-specific lookahead to detect refactorings, renamings, and shifted code. Their approach significantly improved matching precision, achieving a 28% increase while maintaining performance.

Zhu et al. [29] built upon `JDime` with `AutoMerge`, which uses an adjustable *quality function* to match nodes. This method maximizes the quality function to prevent the matching of logically unrelated nodes, thereby reducing false positive conflicts. Their findings indicate that `AutoMerge` successfully reduced the number of reported conflicts compared to the original `JDime`, albeit with a slight decrease in speed. Zhu et al. [12] also proposed `Mastery`, a structured merge tool that utilizes both top-down and bottom-up traversal of abstract syntax trees (ASTs) to handle shifted code more effectively and elegantly compared to `JDime`, which relies solely on a top-down traversal restricted to level-wise AST node matching. Their approach demonstrates to be 2.4 times faster than `JDime`, and achieves a merge accuracy of over 80% relative to the expected output of the merging process.

Larsen et al. [13] proposed `Spork`, a structured merge tool tailored to the Java programming language, designed to preserve the formatting of merged code by reusing the source code from the input files during pretty-printing. `Spork` builds upon the merge algorithm of the 3DM merge tool for XML documents, as they found its core principles are applicable to Java. Their evaluation demonstrates that `Spork` significantly improves formatting preservation in over 90% of the merged files compared to `JDime`, while also being 51% faster.

All of these works recognize the inaccuracy of unstructured merge, as well as the performance cost of structured tools. Here, we propose a tool that offers the benefits of structured tools without the inherent development complexity and performance cost.

6.3 Other kinds of tools

Shen et al. [11] introduced IntelliMerge, a graph-based refactoring aware merging tool for Java programs. Their findings demonstrate that IntelliMerge reduces the number of merge conflicts by about 59% compared to unstructured merge and by about 12% compared to a semistructured merge tool. Additionally, IntelliMerge achieves a precision of around 88% and a recall of 90%, using the manually resolved version as the ground truth. Ellis et al .[25] proposed RefMerge, an operation-based refactoring-aware merging tool. Operation-based based tools record the types of code changes (operations) instead of simple textual changes. Built on top of Git, RefMerge supports 17 refactoring types, including Extract Method and Inline Method. Their research indicates that RefMerge can reduce the number of false positives while eliminating false negatives compared to IntelliMerge. However, they also noted that RefMerge can sometimes introduce new conflicts of its own. Our solution does not specifically target refactorings, so it can work in conjunction with these tools to enhance the accuracy of the merging process, as they have demonstrated that refactoring is a significant source of merge conflict identification errors.

Other works propose tools to address the consequences of merge tools' missed conflicts (false negatives) in the merging process. For instance, Souza et al. [24] proposed SafeMerge, a semantic tool that checks whether a merged program does not introduce new unwanted behavior. They achieve that by combining lightweight dependence analysis for shared program fragments and precise relational reasoning for the modifications. They found that their proposed approach can identify behavioral issues in problematic merges that are generated by unstructured tools. Zhang et al. [26] introduced GMERGE, a tool that leverages k-shot learning with a large language model (GPT-3) to automatically suggest merge conflict resolutions based on the input merge conflict and merge histories. GMERGE achieved an overall accuracy of approximately 64% in resolving semantic merge conflicts in Microsoft Edge, using the developer's actual fixes as the ground truth. Towqir et al. [27] proposed Bucond, a tool that models each version in a merge scenario as a graph and compares these graphs to extract entity-related edits, such as class renaming, to avoid post-merge source code build errors. Their evaluation across three datasets demonstrated that Bucond detects conflicts with high precision and recall, although it missed some conflicts that were identified through automatic builds or manual inspection. We believe that our proposed tool, SESAME, could be used in conjunction with these tools by applying them to the results generated by SESAME. This approach has the potential to mitigate the observed tendency to let conflicts go undetected, which can result in build and semantic errors in the merged code.

7 CONCLUSION

In this paper, we introduced a novel semistructured merge approach, implemented in the SESAME tool, which leverages language-specific syntactic separators to better detect and resolve merge conflicts. Our approach aims to combine the strengths of structured merge tools, which utilize the syntactic structure of code, with the simplicity and performance of traditional unstructured tools. By incorporating language-specific syntactic separators, SESAME mirrors the capabilities of fully structured merge tools. This allows it to more accurately

detect and resolve merge conflicts, thereby reducing the number of false positives.

Our evaluation demonstrated that SESAME improves conflict detection over standard semistructured and unstructured merge tools, achieving gains comparable to fully structured merge tools without incurring the same development and runtime overhead. In particular, our tool reduced the number of merge conflicts by approximately 41% compared to the state-of-practice unstructured merge tool, and by 13% compared to the state-of-the-art semistructured merge tool, while nearly eliminating their spurious conflicts, with reductions of 91% and 88%, respectively. However, we also identified a drawback shared with fully structured tools: a tendency to fail in detecting actual, true merge conflicts. Additionally, we observed that incorporating language-specific syntactic separators can sometimes cause textual alignment errors, leading to different types of merge conflicts identification errors. Future work will focus on further refining our approach, with a particular emphasis on addressing the textual alignment errors observed in the current implementation.

REFERENCES

[1] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, 2002.
[2] S. Khanna, K. Kunal, and B. C. Pierce, "A formal investigation of diff3," in *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'07, Springer-Verlag, 2007.
[3] S. Apel, O. Lessenich, and C. Lengauer, "Structured merge with auto-tuning: Balancing precision and performance," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE'12, ACM, 2012.
[4] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proceedings of the 3rd International Workshop on Software Configuration Management*, SCM'91, ACM, 1991.
[5] J. E. Grass, "Cdiff: A syntax directed differencer for c++ programs," in *Proceedings of the USENIX C++ Conference*, USENIX Association, 1992.
[6] D. Binkley, S. Horwitz, and T. Reps, "Program integration for languages with procedure calls," *ACM Transactions on Software Engineering and Methodology*, 1995.
[7] J. Buffenbarger, "Syntactic software merging," *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, 1995.
[8] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Jdiff: A differencing technique and tool for object-oriented programs," *Automated Software Engineering*, 2007.
[9] O. Lessenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund, "Renaming and shifted code in structured merging: Looking ahead for precision and performance," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, IEEE, 2017.
[10] F. Zhu and F. He, "Conflict resolution for structured merge via version space algebra," *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2018.
[11] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "Intellimerge: a refactoring-aware software merging technique," *Proceedings of the ACM on Programming Languages*, no. OOPSLA, 2019.
[12] F. Zhu, X. Xie, D. Feng, N. Meng, and F. He, "Mastery: Shifted-code-aware structured merging," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pp. 70–87, Springer, 2022.
[13] S. Larsén, J.-R. Falleri, B. Baudry, and M. Monperrus, "Spork: Structured merge for java with formatting preservation," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 64–83, 2022.
[14] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "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*, ESEC/FSE'11, ACM, 2011.

[15] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2017.

[16] A. T. Tavares, P. Borba, G. Cavalcanti, and S. Soares, "Semistructured merge in javascript systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE'19, IEEE, 2019.

[17] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, "The impact of structure on software merging: semistructured versus structured merge," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE'19, IEEE, 2019.

[18] G. Seibt, F. Heck, G. Cavalcanti, P. Borba, and S. Apel, "Leveraging structure in software merge: An empirical study," *IEEE Transactions on Software Engineering*, 2021.

[19] G. Cavalcanti, P. Accioly, and P. Borba, "Assessing semistructured merge in version control systems: A replicated experiment," in *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement*, ESEM'15, ACM, 2015.

[20] O. Lessenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering: An International Journal*, 2015.

[21] V. Berzins, "On merging software extensions," *Acta Informatica*, 1986.

[22] S. Horwitz, J. Prins, and T. Reps, "Integrating noninterfering versions of programs," *ACM Transactions on Programming Languages and Systems*, 1989.

[23] D. M. Berry, "Evaluation of tools for hairy requirements engineering and software engineering tasks," *School of Computer Science, University of Waterloo, Tech. Rep*, 2017.

[24] M. Sousa, I. Dillig, and S. K. Lahiri, "Verified three-way program merge," *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2018.

[25] M. Ellis, S. Nadi, and D. Dig, "Operation-based refactoring-aware merging: An empirical evaluation," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2698–2721, 2022.

[26] 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 *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 77–88, 2022.

[27] S. S. Towqir, B. Shen, M. A. Gulzar, and N. Meng, "Detecting build conflicts in software merge for java programs via static analysis," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–13, 2022.

[28] Q. Zhang, J. Li, J. Lin, J. Ding, L. Lin, and C. Qian, "Wizardmerge–save us from merging without any clues," *arXiv preprint arXiv:2407.02818*, 2024.

[29] F. Zhu, F. He, and Q. Yu, "Enhancing precision of structured merge by proper tree matching," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE'19, IEEE, 2019.