The Impact of Structure on Software Merging: Semistructured versus Structured Merge

Guilherme Cavalcanti*, Paulo Borba*, Georg Seibt+ and Sven Apel£
*Federal University of Pernambuco, Recife, Brazil
+University of Passau, Passau, Germany
£Saarland University, Saarbrücken, Germany

***Abstract*—Merge conflicts often occur when developers concurrently change the same code artifacts. While state of practice unstructured merge tools (e.g. Git merge) try to automatically resolve merge conflicts based on textual similarity, semistructured and structured merge tools try to go further by exploiting the syntactic structure and semantics of the artifacts involved. Although there is evidence that semistructured merge has significant advantages over unstructured merge, and that structured merge reports significantly fewer conflicts than unstructured merge, it is unknown how semistructured merge compares with structured merge. To help developers decide which kind of tool to use, we compare semistructured and structured merge in an empirical study by reproducing more than 40,000 merge scenarios from more than 500 projects. In particular, we assess how often the two merge strategies report different results: we identify conflicts incorrectly reported by one but not by the other (false positives), and conflicts correctly reported by one but missed by the other (false negatives). Our results show that semistructured and structured merge differ in 24% of the scenarios with conflicts. Semistructured merge reports more false positives, whereas structured merge has more false negatives. Finally, we found that adapting a semistructured merge tool to resolve a particular kind of conflict makes semistructured and structured merge even closer.**

***Index Terms*—software merging, collaborative development, code integration, version control systems**

I. Introduction

To better detect and resolve code integration conflicts, researchers have proposed tools that use different strategies to decrease effort and improve correctness of the integration. For merging source code artifacts, unstructured, line-based merge tools are the state of practice [1]–[3], relying on purely textual analysis to detect and resolve conflicts. Structured merge tools [4]–[7] go beyond simple textual analysis by exploring the underlying syntactic structure and static semantics when integrating programs. Semistructured merge tools [8], [9] attempt to hit a sweet spot between unstructured and structured merge by *partially* exploring the syntactic structure and static semantics of the artifacts involved. For program elements whose structure is not exploited (e.g., method bodies), semistructured merge tools simply apply unstructured merge textual analysis.

Although there is evidence that semistructured merge has significant advantages over unstructured merge (semistructured merge reports fewer conflicts, fewer false positives, and its false positives are easier to analyze and resolve)[9], and that structured merge tools report significantly less conflicts than unstructured merge (average reduction of 59% on the number of reported conflicts) [4], it is unknown how semistructured merge compares with structured merge. Apel et al. [4] argue that structured tools are likely more precise than semistructured tools, and they conjecture that a structured tool reports fewer conflicts than a semistructured tool. However, the reduction of reported conflicts alone is not enough to justify industrial adoption of a merge tool, as the reduction could have been obtained at the expense of missing actual conflicts between developers changes.

In fact, although one might expect only accuracy benefits from the extra structure exploited by structured merge, we have no guarantees that this is the case. Previous works [8], [9] provide evidence that the extra structure exploited by semistructured merge is not only beneficial: while it helps to eliminate certain kinds of spurious conflicts (false positives) reported by unstructured merge, it might introduce others that can only be solved by algorithms that further combine semistructured and unstructured merge. Likewise, the extra structure helps semistructured merge to detect conflicts that are missed (false negatives) by unstructured merge, but it unfortunately comes with new kinds of false negatives. So, it is imperative to investigate whether the same applies when comparing semistructured and structured merge, as this is essential for deciding which kind of tool to use in practice.

To compare and better understand the differences between semistructured and structured merge, we apply both strategies to more than 40,000 merge scenarios (triples of base commit, and its two variants parent commits associated with a non-octopus[1] merge commit) from more than 500 GitHub open-source Java projects. In particular, we assess how often the two strategies report different results, and we identify false positives (conflicts incorrectly reported by one strategy but not by the other) and false negatives (conflicts correctly reported by one strategy but missed by the other). To control for undesired variations arising from implementation details, we have implemented a single tool that can be configured to use semistructured or structured merge. This way, we guarantee that structured merge behaves exactly as semistructured merge except for merging the body of method, constructor, and field declarations.

We found that, overall, the two strategies rarely differ for the scenarios in our sample. Considering only scenarios with conflicts, however, the tools differ in about 24% of the cases. A

[1] An octopus merge commit represents the merging of more than two variants.

closer analysis reveals that they differ when integrating changes that affect the same textual area in the body of a declaration, but the modifications involve different abstract syntax tree (AST) nodes in the structural representation. Correspondingly, they also differ when changes in the same AST node correspond to different text areas in the semistructured merge representation of the same declaration body.

Furthermore, we found that semistructured merge reports false positives in more merge scenarios (36) than structured merge (4), whereas structured merge has more scenarios with false negatives (39) than semistructured merge (5). Based on our findings regarding false positives and false negatives, and the observed performance overhead associated with structured merge, semistructured merge appears to be a better match for developers that are not overly concerned with false positives. Finally, we observe that adapting a semistructured merge tool to report textual conflicts only when changes occur in the *same* lines (resolving conflicts caused by changes to *consecutive* lines) would make the two strategies report different results in fewer merge scenarios.

All the scripts and data used in this study are available in our online appendix [10].

II. Semistructured and Structured Merge

The most widely used software merging tools are unstructured: every software artifact is represented as text. Although fast, unstructured merge tools are imprecise [4], [8], [9], [11]. Alternatively, semistructured and structured merge tools incorporate information on the structure of the artifacts being merged. They represent classes and class level declarations as AST nodes. This way, they avoid typical false positive conflicts of unstructured merge [8], [9], such as when developers add declarations of different and independent methods to the beginning of a class. They differ only on how they represent the bodies of method, constructor, and field declarations. In a structured tool, such bodies are also represented as AST nodes; in a semistructured tool, they are represented as text, and are merged in an unstructured way.

We illustrate how this difference affects merging in Figure 1, which shows different versions of a method body.[2] The *base* version at the top shows a method call that adds a new key-value entry to a map. The *structurally merged* version at the bottom highlights, in red, the changes made by developer *A*, who simply refactored the code by extracting `key`. It also highlights, now in blue, the changes made by developer *B*, who added an extra argument to the constructor call. As the two developers changed different AST nodes from the *base* version, corresponding to different arguments of the method call, structured merge successfully integrates their changes. In contrast, semistructured merge reports a conflict because the two developers changed the same line of code in the method body.

To compare semistructured and structured merge, we could simply measure how often they are able to merge contributions

[2]Based on method `createDefaultParametersToOptimized` merged in merge commit https://git.io/fjneH from our sample.

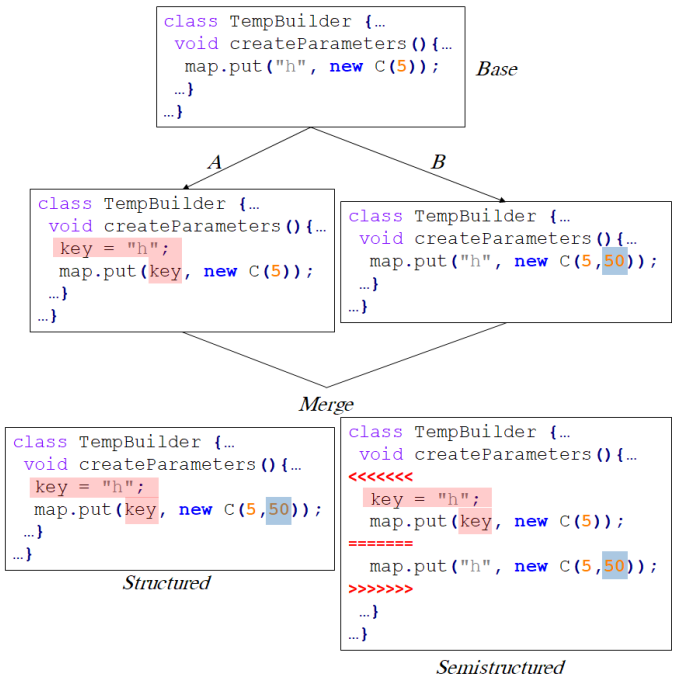


Figure 1: Merging with semistructured and structured merge (false positive).

as in the illustrated example. The preference would be for the strategy that reports fewer conflicts. Given that merging code is the main goal of any merge tool, in principle that criterion could be satisfactory. However, in practice, merge tools go beyond that and detect other kinds of integration conflicts that do not preclude the generation of a valid program, but would lead to build or execution failures. For instance, consider the situation illustrated on Figure 2, where developer *A*, besides extracting the `key` variable, also changed its value to `"j"`. The merge tools would behave exactly as in the original example. In this case, however, the changes interfere [12], and the behavior expected by *A* (new key with old value) and *B* (old key with new value) will not be observed when running the integrated code. In this case, the preference would be for a semistructured tool—the tool that reports a conflict when integrating these changes.

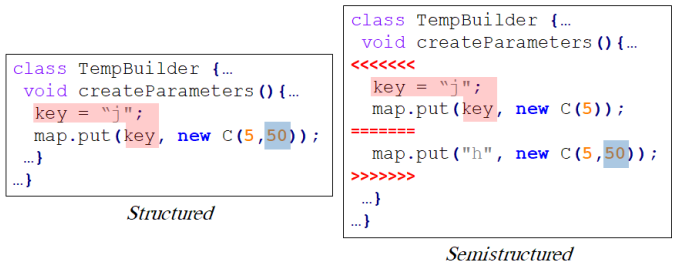


Figure 2: Merging with semistructured and structured merge (true positive).

Whereas the original example in Figure 1 illustrates semistructured merge reporting a *false positive* (incorrectly

reported conflict), the modified example illustrates a structured merge *false negative* (missed conflict). This shows that our comparison criteria should go beyond comparing the number of reported conflicts. We should also consider the number of false positives and false negatives, that is, the possibility of missing or early detecting conflicts that could appear during build or execution. Such comparison should be based on the differences between the merge strategies. By construction, semistructured and structured merge differ only when merging the bodies of method, constructor, and field declarations.

III. Research Questions

To quantify the differences between semistructured and structured merge, and to help developers decide which strategy to use, we analyze merge scenarios from the development history of a number of software projects, while answering the following research questions.

RQ1: *How many conflicts arise when using semistructured and structured merge?*

To answer this question, we integrate the changes of each merge scenario with semistructured and structured merge. For the results of each strategy, we count the total number of conflicts, that is, the number of conflict markers[3] in the files integrated by each strategy. Based on this, we count the number of conflicting merge scenarios, that is, scenarios with, at least, one conflict with semistructured or structured merge. To control for undesired variations on individual tools implementation, we have implemented a single configurable tool that, via command line options, applies a semistructured or structured merge strategy.

RQ2: *How often do semistructured and structured merge differ with respect to the occurrence of conflicts?*

We answer this question by measuring the number of merge scenarios having conflicts reported by only one of the two strategies. In principle, the strategies could still differ when they both report conflicts for the same scenario, as the reported conflicts might be different. However, by construction, both strategies report the same conflicts occurring outside of method, constructor, and field declarations. In dry runs, we observed that this already corresponds to a large fraction of the conflicts. We also observed that conflicts occurring inside such declarations are exactly the same or contain slightly different text in between conflict markers, but are essentially the same conflict in the sense that they report the same issue. Similarly, we observed equivalent conflicts that are reported with a single marker by semistructured merge, but involve a number of markers in structured merge, as illustrated later in this paper.

[3]As illustrated in Figure 3.

RQ3: *Why do semistructured and structured merge differ?*

We answer this question by inspecting merge scenarios and the code merged with each strategy for a sample of scenarios that have conflicts reported by only one of the strategies. This way we can understand the difference on strategies behavior that leads to diverging results.

RQ4: *Which of the two strategies reports fewer false positives?*

A merge tool might report spurious conflicts in the sense that they do not represent a problem and could be automatically solved by a better tool. These are false positives, which lead to unnecessary integration effort and productivity loss, as developers have to manually resolve them. To capture true positives, we rely on the notion of interference by Horwitz et al. [12], who state that two contributions (changes) to a base program interfere when the specifications they are individually supposed to satisfy are not jointly satisfied by the program that integrates them. This often happens when there is, in the integrated program, data or control flow between the contributions. We then say that two contributions to a base program are conflicting when there is not a valid program that integrates them and is free of unplanned interference.

As interference is not computable in our context [12], [13], we rely on build and test information about the integrated code that we analyze, and, when necessary, we resort to manual analysis. Again, we focus on scenarios that have conflicts reported by only one of the strategies; so when only one of the strategies produced a clean merge. We attempt to build the clean merge and run its tests. If the build is successful and all tests pass, we manually analyze the clean merged code to make sure the changes do not interfere; passing all tests is a good approximation, but no guarantee that the changes do not interfere, as a project's test suite might not be strong enough, or even do not cover the integrated changes. If we find no interference in the clean merge, we count a scenario with false positive for the strategy that reported the conflict.

RQ5: *Which of the two strategies has fewer false negatives?*

A merge tool might also fail to detect a conflict (false negative). When this happens, a user would be simply postponing conflict detection to other integration phases such as building and testing, or even letting conflicts escape to operation. So false negatives lead to build or behavioral errors, negatively impacting software quality and the correctness of the merging process. Similarly to RQ4, we rely on build and test information to identify false negatives. We attempt to build the clean merge and run its tests. If the build breaks or, at least, one test fails due to developers changes (when the base version and the

integrated variants do not present build or test issues, but the merge result has issues, so the changes cause the problem), the strategy responsible for the clean merge has actually missed a conflict (false negative). Thus, we count a scenario with a false negative for the strategy that yielded the clean merge.

It is important to emphasize that RQ4 and RQ5 consider only the differences between the semistructured and structured merge strategies. Our interest here is to relatively compare both strategies—not to establish how accurate they are in relation to a general notion of conflict (we do not have the ground truth). So we do not measure the occurrence of false positives and negatives when both strategies behave identically.

RQ6: *Does ignoring conflicts caused by changes to consecutive lines make the two merge strategies more similar?*

In the example of Figure 1, semistructured merge reports a conflict because developers *A* and *B* have changed the same line in a method body. However, even if *A* had simply added a single line (even a comment like `//updating the map`) before the method call, semistructured merge would report a conflict too. This happens because the invoked unstructured merge algorithm reports a conflict whenever it cannot find a line that separates developers changes. As in the example, structured merge would successfully integrate the changes. Assuming that changes to the same line are often less critical than changes to consecutive lines, it would be important to know whether a semistructured tool that resolves consecutive lines conflicts would present closer results to a structured tool. So, to answer this question, we check whether a semistructured merge conflict is due to changes in consecutive lines of code, that is, there is no intersection between the sets of lines changed by each developer, but one of them changes line n and the other changes line $n+1$. Then, for each merge scenario, we check the number of reported conflicts by semistructured merge, and how many of these conflicts are in consecutive lines. Finally, answering this research question consists of revisiting previous research questions contrasting results with and without consecutive lines conflicts.

IV. Study Setup

Answering our research questions involves two steps: mining and execution. In the mining step, we implemented scripts to mine GitHub repositories of Java projects and collect information on merge scenarios—each scenario consists of the three revisions involved in a three-way merge.

In the execution step, we merge the selected scenarios with both semistructured and structured merge. For each merge without conflicts, we use a build manager to build the merged version and execute its tests (to find false positives and false negatives; see Section III). In the remaining of the section, we describe the two steps in detail.

A. Mining Step

Our study relies both on the analysis of source code and build status information, so we opt for GitHub projects that use Travis CI for continuous integration. As the merge tool used in the execution step is language dependent, we consider only Java projects. As parsing Travis CI's build log depends on the underlying build automation infrastructure, we consider only Maven projects because we use its log report information for automatically filtering conflicts.

We start with the projects in the datasets of Munaiah et al. [14] and Beller et al. [15], which include numerous carefully selected open-source projects that adopt continuous integration. From these datasets, we select Java projects that satisfy two criteria: first, the presence of Travis CI and Maven configuration files, which indicates that the project is configured to use the Travis CI service, and that the project uses the Maven build manager;[4] second, the presence of, at least, one build process in the Travis CI service, and confirmation of its active status, which indicates the project has actually used the service.

After selecting the project sample, we execute a script that locally clones each project and retrieves its non-octopus merge commit list—a merge commit represents a merge in the subject project's history and therefore can be used to derive a merge scenario. As most projects adopted Travis CI only later in project history, for each project, we consider only the merge commits dated after the project's first build on Travis CI. For each scenario derived from these merge commits, we check the Travis CI status of the scenario's three commits. If any of them has an *errored* (indicates a broken build) or *failed* status (indicates failure on tests), we discard the scenario. The reason is that we would not be able to confirm whether a problem in the merged version was caused by conflicting changes—the problem could well have been inherited from the parents.

As a result of the mining step, we obtained 43,509 merge scenarios from 508 selected Java projects. Although we have not systematically targeted representativeness or even diversity [16], our sample exhibits a considerable degree of diversity along various dimensions. Our sample contains projects from different domains, such as APIs, platforms, and network protocols, varying in size and number of developers. For example, the TRUTH project has approximately 31 KLOC, while HIVE has more than 1 KKLOC. The WEB MAGIC project has 45 collaborators, while OKHTTP has 195. We provide a complete list of the analyzed projects in our online appendix [10].

B. Execution Step

After collecting the subject projects and merge scenarios, we merge the selected scenarios with both semistructured and structured merge. To control for undesired variations, we have implemented a single configurable tool that, via command line options, applies semistructured or structured merge. This way we guarantee that structured merge behaves exactly as semistructured merge except for merging the body of method, constructor, and field declarations. The new implementation adapts and improves previous and independent implementations of a semistructured [17] and a structured merge tool [18].

[4]We check whether the repository contains both Travis CI and Maven configuration files: `travis.yml` and `pom.xml`.

In particular, our tool is built on top of the semistructured tool. While a standard semistructured merge tool invokes unstructured merge for bodies of declarations, our configurable tool also allows structured merge to be invoked instead. When configured to use semistructured merge, the tool invokes the standard and widespread *diff3* unstructured algorithm at declarations level. When configured as structured merge, the tool invokes the most mature and extensively evaluated structured implementation [4], [19]–[22].

For each merge scenario, our infrastructure generates two merged versions: a semistructured version and a structured version. For each file, we count the number of reported conflicts. For the semistructured merge versions, we also count the number of conflicts that are due to changes in consecutive lines. To do so, we check whether the sets of changed lines in the variants are disjoint, and whether the numbers of the contribution lines in the conflict text are consecutive.[5]

Based on the number of conflicts, we select scenarios having conflicts reported by only one of the strategies. The strategies could also differ by reporting different conflicts for the same scenario, as discussed earlier in Section III. In our sample, however, we verified that whenever semistructured and structured merge report conflicts in the same scenario, these conflicts are in the same file. Even so they could still report different conflicts in the same file. We have, in fact, observed such cases, but they actually refer to equivalent conflicts, reported by the strategies in different ways, using different sets of markers and associated conflicting code. So we can consider them to be the same conflict, but with different textual representations derived from the difference in the exploited syntax granularity. This is illustrated in Figure 3, in a merge scenario from project NEO4J-FRAMEWORK:[6] both developers added different declarations for the same constructor. As this constructor is not declared in the base version, both strategies report conflicts. Structured merge reports a conflict for any two syntactic level differences between the versions, resulting in several small conflicts. Semistructured merge reports a single conflict for the entire declaration.

Having identified scenarios for which the strategies differ, we collect information on false positives and false negatives. We use Travis CI as our infrastructure for building and executing tests for each scenario, as explained in Section III and illustrated in Figure 4. As Travis CI builds only the latest commit in a push command or pull request, not all commits in a project have an associated build status on Travis CI. The generated semistructured and structured merged versions certainly do not have a Travis CI build, as they are generated by our experiment. So we use a script that forces build creation in such cases. Basically, we create a project fork, activate it on Travis CI, and clone it locally. Then, every push to our remote fork

[5]We use GNU's `diff` command passing the base version and each variant separately.

[6]https://git.io/fjne9

Semistructured

```
<<<<<<<
public CrawlerRM(…){…
    this.inSt = inSt;
…}
=======
public CrawlerRM(…){…
    this.reInSt = reInSt;
…}
>>>>>>>
```

Structured

```
public CrawlerRM(…){…
this.
<<<<<<<
inSt
=======
reInSt
>>>>>>>
=
<<<<<<<
inSt
=======
reInSt
>>>>>>>
;
…}
```

Figure 3: Equivalent conflicts with different granularity.

to trigger a Travis CI build. Note that we are only able to build and test code without conflicts, as the conflicts markers invalidate program syntax.

If the build status on Travis CI of the resulting merge commit is *errored* (when the build is broken) or *failed* (when the build is ok, but, at least, one of the tests failed), we consider that the corresponding merge scenario has a false negative from the strategy that did not report a conflict—therefore a true positive reported by the other strategy. However, it is possible that a build breaks or a test fails due to external configuration problems, such as trying to download a dependency that is no longer available or exceeding the time to execute tests. We filter these cases as they do not reflect issues caused by conflicting code. To do so, we analyze, for each generated build, its Maven log report seeking for indicative message errors. Finally, since we have also filtered merge scenarios having problematic parents (see Section IV-A), if the new merge commit still has build or test issues, we can conclude that this is because developers changes interfere.

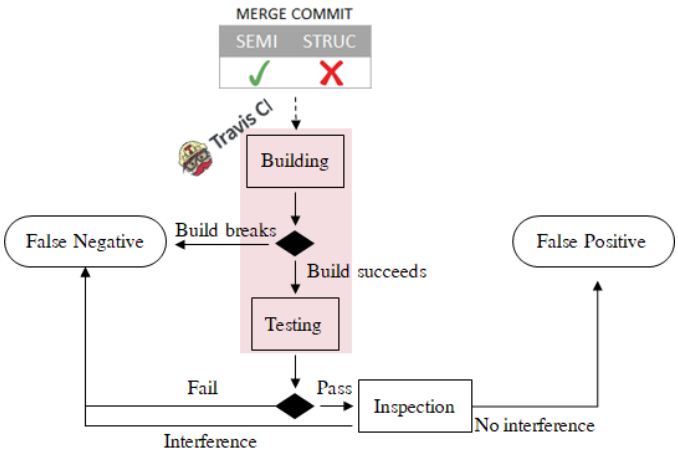


Figure 4: Building and testing merge commits. A green check mark indicates no conflict with one strategy, a red cross indicates conflict with the other strategy.

In case the resulting merge commit build status on Travis CI is *passed*, we are sure that the merged version has no build error, and all tests pass. So this is a candidate false positive of the strategy that reported the conflict. However, whereas this provides precise guarantees for build issues, the guarantees for test issues are as only good as the project's test suite. Even for projects with strong test suites, unexpected interference between merged contributions might be missed by the existing tests. So, to complement test information, we manually inspect all conflicting files from all merged versions with potential false positives. In this manual analysis, two of the authors analyzed the first 5 conflicting files to consolidate the guidelines. Then, two other authors individually analyzed the remaining files. In the case of divergence between authors' classification for the same file, a third author reviewed that file. In the case of uncertainty regarding the contributions, a message was sent to the original committers to clarify the changes.[7]

During this manual analysis, we check the changes made by each developer, analyzing whether they interfere, following the definition of interference of Section III. If one of the developers simply changes spacing and comments, or extracts a variable or a method, we conclude that there is no interference. The corresponding merge scenario is then confirmed as having false positives. The same applies when the developers change unrelated state, or when they change assignments to unrelated local variables. Conversely, if both developers change program semantics, such as modifying related state or changing assignments to the same variable, we conclude that there is interference. We then confirm that the corresponding merge scenario has a false negative. As discussed in Section II, the same applies to the variation of the example illustrated in Figure 1. For each merge scenario we find interference in the merged version, we add explanation and discuss a test case that fails in the base commit, passes in one of the parent commits, and fails in the merged version. This is further evidence that the changes made by the considered parent commit were affected by the changes of the other parent commit.

V. Results

We use our study design to analyze 43,509 merge scenarios from the development histories of 508 Java projects. In what follows, we present our results, following the structure defined by our research questions. More details, including tables and plots, are available in our online appendix [10].

A. How many conflicts arise when using semistructured and structured merge?

In our sample, we found 4,732 conflicts using semistructured merge, and 4,793 when using structured merge. This is a reduction of 1.27% in the number of reported conflicts when using semistructured merge. This results at first might be surprising to those who expect that more structure leads to conflict reduction. However, as pointed out in Section IV-B and illustrated in Figure 3, structured merge might report more conflicts due to its structure-driven and fine-grained approach. This leads to conflicts that respect the boundaries of the language syntax, which might result in many small conflicts that are reported as a single conflict by semistructured merge.

To control for the bias of conflict granularity, we consider also the number of merge scenarios with conflicts: 1,007 (2.31% of the scenarios) using semistructured merge, and 814 (1.87%) using structured merge. This time we observe a reduction of 19.17% in the number of scenarios with conflicts when using structured merge. In a per-project analysis, we found similar results: 2.25 $\pm$ 4.58% (average $\pm$ standard deviation) of conflicting scenarios with semistructured merge, and 1.8 $\pm$ 3.92% with structured merge.

Summary: Semistructured and structured merge report similar numbers of conflicts, but the number of merge scenarios with conflicts is reduced using structured merge (by about 19%). In general, conflicts are not frequent when using both strategies (in about 2% of the scenarios).

B. How often do semistructured and structured merge differ with respect to the occurrence of conflicts?

Overall, we found 223 (0.51%) scenarios with conflicts reported *only* by semistructured merge, and 30 (0.07%) reported *only* by structured merge. So the two strategies differ in 0.58% (253) of the scenarios in our sample; a per-project analysis gives a similar result: on average, the strategies differ on 0.52 $\pm$ 2.06% of the scenarios.

The reported percentages are comparatively small because most scenarios are free of conflicts even when using less sophisticated strategies such as unstructured merge. In fact, most scenarios involve only changes to disjoint sets of files, so they cannot possibly discriminate between merge strategies because there is no chance of conflict. So it is more reasonable to consider the relative percentages for conflicting merge scenarios, which correspond to 2.28% of our sample scenarios. Overall, semistructured and structured merge differ in 23.67% of the conflicting scenarios (an average of 23.22 $\pm$ 44.45% in a per-project analysis). The observed error bounds are explained by some projects having low rates of merge scenarios with conflicts. For instance, projects such as CLOCKER, WIRE and LA4J had only one conflicting merge scenario, and, for this single scenario, the strategies differ as a result of the reasons we explain on the next research question.

Summary: Semistructured and structured merge substantially differ in terms of reported number of conflicts when applied only to conflicting scenarios of our sample (they differ in about 24% of these scenarios).

C. Why do semistructured and structured merge differ?

To better understand the differences between the merge strategies, we manually analyzed a random sample of 54 merge scenarios that have conflicts reported by only one of

[7]We provide a sheet with the detailed analysis of all files in our online appendix.

the strategies, guided by power and sample size estimation statistics [23]. This includes 44 scenarios with conflicts reported only by semistructured merge, and 10 scenarios with conflicts reported only by structured merge. For each scenario, we analyzed developers' changes, the code merged by one of the strategies, and the conflict reported by the other strategy. This way, we can relate characteristics of the integrated changes with the strategy that reported the conflicts.

We begin with scenarios having semistructured merge conflicts, and a structured clean merge. Consider the example in Figure 5. Developer *A* added modifier `final` to the `IOException catch` clause right after the `try` block. Meanwhile, developer *B* added a new `catch` clause to `ResourceNotFoundException`, also right after the `try` block. As no line separates these changes in two distinct areas of the text, semistructured merge—which invokes unstructured merge to integrate method bodies— reports a conflict. Developers then have to manually act and decide which `catch` should appear right after the `try` block. In contrast, structured merge detects that the changes affect different child nodes of the `try` node, and successfully integrates the changes by including the new child node (*B*'s contribution) and the existing changed node (*A*'s contribution). We observed the same kind of situation in every scenario that leads only to semistructured merge conflicts, including the motivating example illustrated in Section II.

Structured

```
try{…
} catch (final ResourceNotFoundException e){…
} catch (final IOException e){…
}
```

Semistructured

```
try{…
<<<<<<<
} catch (final IOException e){…
=======
} catch (final ResourceNotFoundException e){…
} catch (IOException e){…
}
>>>>>>>
```

Developer A
Developer B

Figure 5: Semistructured merge conflict from project GLACIERUPLOADER (from merge commit https://git.io/fjney).

Summary: Semistructured and structured merge differ when changes occur in overlapping text areas that correspond to different AST nodes.

Next, we consider scenarios with structured merge conflicts, and a semistructured clean merge. In the example of Figure 6, developer *A* deletes an argument from the call to method `doInsertFinalNewLine` inside a `for` statement. Developer *B* converts the same `for` statement into a `for`-each statement. Since these changes occur in non-overlapping text areas, semistructured merge successfully integrates the contributions. Structured merge reports a conflict because it is unable to match the new `for`-each with the previous `for` statement—they are represented by nodes of different types. It correctly detects that the subtree of the body of the `for` statement was changed by one of the developers, but it incorrectly assumes that the whole `for` statement was deleted by the other developer. As a consequence, structured merge does not proceed merging the child nodes from these iteration statements, and reports a single conflict for the entire statements. Note that the changed method call `doInsertFinalNewLine` is accidentally included in this deletion as it is not matched with the corresponding version in the `for`-each statement.

Structured

```
<<<<<<<
for(int i = 0; i<rules.size(); i++){…
 EditorConfig.OutPair rule = rules.get(i);
 …
 changed = doInsertFinalNewLine(primaryFile);
…}
=======
for(EditorConfig.OutPair rule : rules){…
 changed = doInsertFinalNewLine(primaryFile,values);
…}
>>>>>>>
```

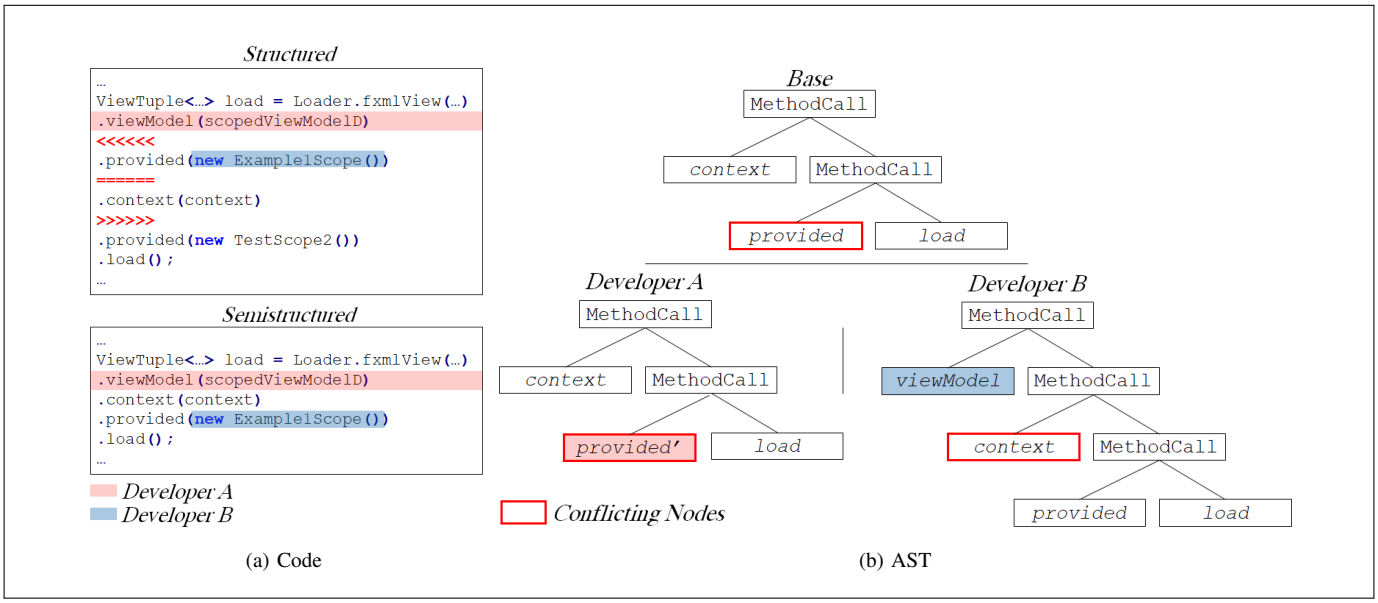
Semistructured

```
for(EditorConfig.OutPair rule : rules){…
 changed = doInsertFinalNewLine(primaryFile);
…}
```

Developer A
Developer B

Figure 6: Structured merge conflict from project EDITORCONFIG-NETBEANS (from merge commit https://git.io/fjneX).

Structured merge differs in a second kind of situation, as illustrated in Figure 7 (a). In this example, developer *A* added a call to method `viewModel` to an existing method call chain. Developer *B* changed the argument of method `provided` in the same chain. Semistructured merge successfully integrates the changes because it detects that they occur in non-overlapping text areas: the line that calls method `context` act as a separator between the areas. Structured merge reports a conflict because, by analyzing and matching the base AST with the developers' ASTs (see Figure 7 (b)), it incorrectly concludes that the left child of the second `MethodCall` node was changed by both developers. Indeed, as marked in red in the figure, the three nodes in this position are different. Developer *B* has not actually changed the call to `provided`, but changed the level of the call to `context` in the AST by adding a new method call to `viewModel`. As tree matching is top-down and mostly driven by `MethodCall` nodes (in this case) [4], structured merge is not able to correctly match the calls, and assumes that Developer *B* changed the call to



(a) Code (b) AST

Figure 7: Structured merge conflict from project MVVMFX (from merge commit https://git.io/fjneD).

`provided` by a call to `context`. This is why the reported conflict involves these two method calls; the second in the conflict text corresponds to a base node not changed by the developers (`context` call). The text does not refer to the AST node that actually caused the conflict (`viewModel` call).

Summary: Semistructured and structured merge differ when changes occur in non-overlapping text areas that correspond to *(a)* different but incorrectly matched nodes and to *(b)* the same node.

D. Which of the two strategies reports fewer false positives?

As explained in Section IV-B, we use Travis CI to build and test the merged code of the 253 scenarios for which the strategies differ. We found 44 scenarios with merged code that successfully builds and for which all tests pass; their Travis CI status is *passed*. Although this status provides precise guarantees that there are no build and test conflicts, there could still be other kinds of semantic conflicts, as unexpected interference between merged contributions might be missed by existing tests. These 44 scenarios are then potential false positives of the strategy that reported a conflict, but we have to confirm this with a manual inspection of the merged code and the individual code contributions. As explained in Section IV-B, these scenarios were analyzed by two authors separately. In 3 scenarios, there was disagreement between the authors, so the review of another author was necessary. Besides, in only 1 scenario the contributions were not clear, so we asked the actual contributors for clarification by commenting the original merge commit.

From the 44 potential scenarios with false positives, 39 are related to semistructured merge; they were successfully merged by structured merge and have a *passed* status in Travis CI. Conversely, only 5 scenarios are potential false positives of structured merge. The manual analysis revealed that 36 of the 39 scenarios were actually false positives produced by semistructured merge. Only 3 scenarios were actual true positives, and, as a consequence, false negatives of structured merge. For instance, in a merge scenario from project SWAGGER-MAVEN-PLUGIN,[8] both developers added elements to the same list. As a consequence, each developer expected different resulting lists, which are themselves different from the list that will be obtained by executing the merged code. None of this project's tests exercises these contributions, but it is not hard to come up with a test that passes in the developers versions but fails in the merged version, revealing the conflict.[9]

From the 5 structured merge scenarios having potential false positives, 4 of them were classified as actual false positives. Only 1 was an actual true positive produced by structured merge, and a false negative of semistructured merge. The actual true positive is a scenario from project RESTY-GWT.[10] In this scenario, one of the developers edited the condition and block of an existing `if` statement, while the other added another `if` statement after the previous `if` statement. Both `if` statements return different values based on the value of the same method parameter. However, the first developer's edited condition now satisfies both developers conditions, affecting the method result expected by the other developer, and no test of the mentioned project captures this interference. A test that captures this interference could be one, added by the second

[8]https://git.io/fjneS

[9]Suppose a test that checks whether the size of the list is *n+1*; if it passes in the developers' individuals versions, it will fail in the merged version, in which the size of the list will be *n+2*.

[10]https://git.io/fjneF

developer, that checks the value of the mentioned parameter, and then enters into his added `if` block.[11]

E. Which of the two strategies has fewer false negatives?

We found 209 scenarios with merged code that either cannot be successfully built (Travis CI *errored* status) or can be properly built but, at least, one of the tests do not pass (Travis CI *failed* status). By performing a Travis CI log report analysis, we found that most scenarios (169) *errored* and *failed* status are due to a number of reasons (Travis CI timeout, unavailable dependencies, etc.) unrelated to the contributions being merged, and that suggest these are older scenarios that would be hard to compile and build anyway. So we cannot automatically classify these as false negatives. We then focus on 40 scenarios with *errored* and *failed* status; we confirm the status by parsing Travis CI log messages and checking that they are compiler or test related. Since our sample does not include scenarios having broken or failing parents (see Section IV-A), if the resulting merged code presents build or test issues, we conclude this is due to interference between the merged code contributions.

From the analyzed 40 scenarios, we found only 4 scenarios that are false negatives produced by semistructured merge: 3 with *errored* and 1 with *failed* status.[12] In contrast, we found 36 scenarios that are false negatives produced by structured merge: 23 with *errored* status and 13 with *failed* status for the merge.[13]

Although the two merge strategies are somewhat different, we identified some common causes for false negatives due to broken builds. For example, we found situations in clean merges from both strategies (e.g., in projects BLUEPRINTS and SINGULARITY), where one developer added a reference to a variable while the other developer deleted or renamed this variable. Consequently, the compiler could not build the file. We also observed situations (e.g., in projects NEO4J-RECO and VRAPTOR), where one developer changed the value passed as an argument, while the other developer changed the corresponding parameter's type. After the merge, there is a compilation error reported due to the mismatch between expected and passed argument.

Regarding test failures causing false negatives, the only failed scenario from semistructured merge was in project CLOSURE-COMPILER, where the developers' changes are responsible to update the same list. Conversely, on failed scenarios from structured merge, we observed, for example, developers inadvertently changing the same connection creation in project JEDIS, or assigning different objects to the same variable in project DSPACE.

Table I summarizes our findings for false positives and false negatives after all analyses.

Summary: Semistrutured merge reports more false positives (9 times more scenarios with false positives), and structured merge misses more conflicts (has more false negatives; 8 times more scenarios with missed conflicts).

Table I: Numbers for merge scenarios with false positives and false negatives.

	Semistructed Merge	Structured Merge
False Positives	36	4
False Negatives	5	39

F. Does ignoring conflicts caused by changes to consecutive lines make the two strategies more similar?

Our results show that our metrics on number of reported conflicts and on when the two strategies differ slightly drop if a semistructured merge tool could resolve conflicts due to changes in consecutive lines.[14] In particular, the number of scenarios with semistructured merge conflicts is reduced by 3.38%, and the number of scenarios in which semistructured and structured merge differ is reduced by 11.07%.

In projects such as QUICKML, SEJDA and SONARQUBE, we found that this happens because changes to consecutive lines often correspond to changes to different AST nodes. In such situations, structured merge does not report conflicts. Thus, when semistructured merge is able to resolve consecutive lines conflicts, it might avoid conflicts due to changes to different AST nodes, similar to structured merge.

Summary: A semistructured merge tool that can resolve consecutive lines conflicts would present even closer number of scenarios with conflicts to structured merge, and fewer scenarios in which the two strategies differ.

G. Threats to Validity

We rely on manual analysis to identify interference between merged contributions, so there is a risk of misjudgment. To mitigate this threat, every scenario was analyzed separately by two authors, and in case of disagreement, another author acted as a mediator. We also asked the actual contributors for clarification of the changes when they were not clear. As mentioned in Section V-D, this was only necessary in one occasion.

We opted for a single merge tool that can be configured to apply semistructured and structured merge. This was necessary to ensure that we have a structured merge tool working as expected. This single tool is basically an extension of the semistructured merge tool able to invoke a structured merge tool on declarations. To the best of our knowledge, these tools are the most mature and evaluated tools available.

In addition, as we discard merge scenarios that we could not properly build on Travis CI, or that have broken or failed

[11]The test passes on second developer's version, and fails on the merged version because now it would enter on first developer's `if` block, returning a different value.

[12]Structured merge having reported conflicts for these cases

[13]Semistructured merge having reported conflicts for these cases.

[14]We only count consecutive lines conflicts, we actually do not resolve them. Thus, we dot not have numbers for false positives and false negatives.

parents, we might have missed differences in the strategies' behavior. We might have also missed them because we analyze only code integration scenarios that reach public repositories with merge commits; this is not the case, for example, for integrations with Git rebase, or that were affected by Git commands that rewrite history.

Finally, we focus on open-source Java projects hosted on GitHub, using Travis CI and Maven. Thus, generalization to other platforms and programming languages is limited. Such requirements were necessary because the merge tools are language specific, and to reduce the influence of confounds, increasing internal validity.

VI. Discussion

Our results show that, overall, the two merge strategies rarely differ for the scenarios in our sample, as most of them are free of conflicts. Many merge scenarios affect disjoint sets of files, having no chance of leading to conflicts, no matter which merge strategy is adopted by the tool one uses. However, for scenarios that reflect more complicated merge situations, we do observe that the choice of the merge strategy makes a difference: considering scenarios with conflicts, the two strategies differ in about 24% of the cases. This is maybe surprisingly low given that most code and changes occur inside (method, constructor, etc.) declarations exploited by the significant extra structure considered by structured merge. In terms of conflicting scenarios with diverging behavior, structure plays a similar role when moving from unstructured merge to semistructured merge (27%) [9], and when moving from semistructured to structured merge (24%).

In cases where two strategies differ, semistructured merge reports false positives in more merge scenarios than structured merge, whereas structured merge has more scenarios with false negatives than semistructured merge. The extent of the difference in the false positive and false negative rates are quite similar. Semistructured merge's false positives are not hard to resolve: the fix essentially involves removing conflict markers. Analyzing the changes before removing the markers might be expensive, but certainly not as in unstructured merge (with its crosscutting conflicts [9]), or as in structured merge (with its fine-graned conflicts, as illustrated in Figure 3). In contrast, structured merge's false negatives might be hard to detect and resolve. Most of the observed false negatives actually correspond to compilation and static analysis issues that escape the merging process but cannot escape the building phase. These are always detected and are often easy to resolve. However, part of the observed false negatives are related to dynamic semantics issues that can easily go unnoticed by testing and end up affecting users. These are hard to detect and, when detected, are often hard to resolve. A more rigorous analysis based on conflict detection and resolution timing data could differently weight false positives and false negatives, in the spirit of Berry [24], and better assess the benefits of the two strategies.

Based on our findings regarding false positives and false negatives, and given the observed modest difference between the two merge strategies, we conclude that semistructured merge would be a better match for developers that are not overly concerned with false positives. This is reinforced by considering the observed performance overhead associated with structured merge, and the extra effort needed to develop structured merge tools [8]. Together with our findings about consecutive lines conflicts, this discussion suggests the development of a tool that adapts semistructured merge to report textual conflicts only when changes occur in the *same* lines (resolving conflicts caused by changes to *consecutive* lines). Such a tool could hit a sweet spot in the tension between structure and accuracy in merge tools.

Our observations, especially the ones that explain when the two strategies differ, shall help researchers and merge tool developers to further explore improvements to merge accuracy and the underlying tree matching algorithms. In the same vein, our manual analysis of false positives reveal opportunities for making merge tools avoid a number of false positives. For example, by detecting straightforward semantic preserving changes, we could avoid 42% of false positives reported by semistructured merge in our sample.

Combining the two merge strategies as suggested by Apel et al. [4] seems also promising. One idea is to invoke structured merge, and when it does not detect conflicts, invoke semistructured merge and return its result, which would reduce the chances of false negatives. This is a conservative approach, which considers the costs associated with false positives to be inferior to those associated with false negatives. Such a tool would eliminate structured merge's false negatives, but would still have semistructured merge's false negatives. Conversely, in the best case, when structured merge does detect conflicts, it would present structured merge's false positives; and, in the worst case, the tool would present semistructured merge's false positives. A less conservative combination, in which semistructured is used as long as it does not detect conflicts, is also worthwhile to explore.

VII. Related Work

Several researches propose development tools and strategies to better support collaborative development environments. These tools try to both decrease integration effort and improve correctness during code integration. For instance, to overcome weaknesses associated with traditional unstructured merge, structured [4], [5], [20], [25]–[29] and semantic merge strategies have also been proposed [30]–[33].

For example, Apel et al.[4] developed *JDime*, the structured tool used in this study, also capable of tuning the merging process on-line by switching between unstructured and structured merge, depending on the presence of conflicts. They also proposed semistructured merge, which takes advantage of the underlying language's syntactic structure and static semantics, but without the performance overhead associated with full structured merge [8]. Studies [9], [11] provide evidence that semistructured merge might reduce the number of reported conflicts in relation to traditional unstructured merge, but not for all projects and merge scenarios. Cavalcanti et al. [9] go further and provide evidence that the number of false positives

is significantly reduced when using semistructured merge. However, they do not find evidence that semistructured merge leads to fewer false negatives. Lessenich et al.[20] attempt to improve *JDime* by employing a syntax specific lookahead to detect renamings and shifted code. They demonstrate that their solution can significantly improve matching precision in 28% while maintaining performance. Zhu et al.[22] built *AutoMerge*, on top of *JDime*, that matches nodes based on an adjustable so-called *quality function*. Their goal is to find a set of matching nodes that maximizes the *quality function*, preventing the matching of logically unrelated nodes, and, as consequence, false positives conflicts. They found that *AutoMerge* was able to reduce the number of reported conflicts compared to original *JDime*, being slightly slower. We complement these prior studies by comparing semistructured and structured merge, not only in terms of reported conflicts, but also in terms of false positives and false negatives. We conclude that semistructured merge would be a better match for developers that are not overly concerned with false positives, especially when a semistructured merge tool resolves conflicts caused by changes to consecutive lines. We also suggest that a combination of these two strategies seems promising as it is able to reduce weaknesses of both strategies.

Souza et al. [33] propose *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 the proposed approach can identify behavioral issues in problematic merges that are generated by unstructured tools. This tool needs as input a merged program besides the three versions present in a merge scenario, so it could be used in combination with a semistructured or structured merge tool, or even our suggested tool that further combines these two strategies, to reduce their behavioral false negatives. However, *SafeMerge* only analyzes the class file associated with the modified method declarations, so it may suffer from both false positives and false negatives too. In particular, their analysis results are only sound under the assumption that the external callees from other classes have not been modified.

Other empirical studies provide evidence about the occurrences and effects of conflicts and their associated causes [2], [34]–[41]. For example, Brun et al. [34] and Kasi et al. [35] reproduce merge scenarios from different GitHub projects with the purpose of measuring the frequency of merge scenarios that resulted in conflicts. Zimmermann [2] conducted a similar analysis reproducing integrations from CVS projects instead. They all conclude that conflicts are frequent. Adams and McIntosh [37], and Henderson[38] even report that companies have migrated to single-branched repositories to avoid merge problems. Our work complements these studies providing evidence of conflict frequency depending on the use of different merge strategies. Finally, Menezes et al.[39] analyze merge scenarios from open-source Java projects to investigate the nature of merge conflicts. in terms of what conflicts look like, what kinds of conflicts occur, how developers fix them, and more. Based on their results, they argue that it is difficult to envision a single generic merge strategy that can automatically resolve all possible conflicts, because the diversity in conflicts is simply too large. Still, they believe it is possible to improve over the existing tools to better resolve conflicts, for instance, in the form of plug-ins that can automatically handle specific kinds of conflicts. Accioly et al. [41] derive a catalog of conflict patterns expressed in terms of the structure of code changes that lead to merge conflicts. Their results show that most conflicts occur because developers independently edit the same or consecutive lines of the same method. However, the probability of creating a merge conflict is approximately the same when editing methods, class fields, and modifier lists. Similarly, [36] investigate how conflicts on method declarations are resolved on open source Java projects. They found that most part of them is resolved by adopting one of the versions, then discarding the other. These findings about conflicts characteristics might be adapted by a merge tool as strategies for resolving conflicts.

VIII. Conclusions

When integrating code contributions from software development tasks, one often has to deal with conflicting changes. While state of practice tools still rely on an unstructured, lined-based strategy to merging, recent developments demonstrate the merits and prospects of advanced merge strategies, in particular semistructured and structured merge. Previous studies provide evidence that semistructured merge has significant advantages over unstructured merge, and that structured merge reports significantly fewer conflicts than unstructured merge. However, it was unknown how semistructured merge compares with structured merge. In this paper, we compared semistructured and structured merge by reproducing 43,509 merge scenarios from 508 GitHub Java projects. Our results show that users should not expect much difference when using a semistructured or a structured merge tool: they differ substantially only when applied to conflicting scenarios (in about 24% of them), which corresponds to only about 2% of our subject scenarios. When semistructured merge is able to resolve conflicts due to changes in consecutive lines of code, the two strategies differ in about 22% of the conflicting scenarios instead, and the number of scenarios in which they differ is reduced by about 11%. When deciding which kind of tool to use, a user should consider that semistructured merge reports more false positives (9 times more scenarios with false positives), but structured merge misses more conflicts (false negatives; 8 times mores scenarios with missed conflicts). Combining the two strategies seems promising as it is able to mitigate the weaknesses of both strategies. As future work, we shall implement and evaluate such a combination of strategies to verify its actual benefits and drawbacks.

Acknowledgments

We thank Leuson Silva for providing support on Travis CI infrastructure. We also thank INES 2.0, FACEPE grants APQ-0399-1.03/17 and IBPG-0546-1.03/15, CAPES grant 88887.136410/2017-00, CNPq grant 465614/2014-0 and the German Research Foundation (AP 206/11) for partially funding this research.

REFERENCES

[1] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, 2002.

[2] T. Zimmermann, "Mining workspace updates in cvs," in *Proceedings of the 4th International Workshop on Mining Software Repositories*, ser. MSR'07. IEEE, 2007.

[3] 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*, ser. FSTTCS'07. Springer-Verlag, 2007.

[4] 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*, ser. ASE'12. ACM, 2012.

[5] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Jdiff: A differencing technique and tool for object-oriented programs," *Automated Software Engineering*, 2007.

[6] J. E. Grass, "Cdiff: A syntax directed differencer for c++ programs," in *Proceedings of the USENIX C++ Conference*. USENIX Association, 1992.

[7] B. Westfechtel, "Structure-oriented merging of revisions of software documents," in *Proceedings of the 3rd International Workshop on Software Configuration Management*, ser. SCM'91. ACM, 1991.

[8] 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*, ser. ESEC/FSE'11. ACM, 2011.

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

[10] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, "Online appendix for the paper the impact of structure on software merging: Semistructured versus structured merge," Hosted on https://spgroup.github.io/s3m/svj.html, 2019.

[11] 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*, ser. ESEM'15. ACM, 2015.

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

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

[14] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, 2017.

[15] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR'17. IEEE, 2017.

[16] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE'13. ACM, 2013.

[17] G. Cavalcanti, P. Accioly, and P. Borba, "Jfstmerge - a semistructured merge tool for java applications," Hosted on https://github.com/guilhermejccavalcanti/jFSTMerge, 2019.

[18] O. Lessenich, "Jdime - structured merge with auto-tuning," Hosted on https://github.com/se-passau/jdime, 2019.

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

[20] 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*, ser. ASE 2017. IEEE, 2017.

[21] F. Zhu and F. He, "Conflict resolution for structured merge via version space algebra," *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2018.

[22] 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*, ser. ICSE'19. IEEE, 2019.

[23] J. Eng, "Sample size estimation: how many individuals should be studied?" *Radiology*, 2003.

[24] D. M. Berry, "Evaluation of tools for hairy requirements engineering and software engineering tasks," 2017. [Online]. Available: https://cs.uwaterloo.ca/~dberry/FTP_SITE/tech.reports/EvalPaper.pdf

[25] J. Buffenbarger, "Syntactic software merging," *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, 1995.

[26] T. N. Nguyen, "Object-oriented software configuration management," in *Proceedings of the 22th International Conference on Software Maintenance*, ser. ICSM'06. IEEE, 2006.

[27] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen, "Effective software merging in the presence of object-oriented refactorings," *IEEE Transactions of Software Engineering*, 2008.

[28] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. ACM, 2014.

[29] D. Asenov, B. Guenat, P. Müller, and M. Otth, "Precise version control of trees with line-based version control systems," in *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering*. Springer, 2017.

[30] V. Berzins, "Software merge: Semantics of combining changes to programs," *ACM Transactions on Programming Languages and Systems*, 1994.

[31] D. Jackson and D. A. Ladd, "Semantic diff: A tool for summarizing the effects of modifications," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM'94. IEEE, 1994.

[32] D. Binkley, S. Horwitz, and T. Reps, "Program integration for languages with procedure calls," *ACM Transactions on Software Engineering and Methodology*, 1995.

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

[34] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE'11. ACM, 2011.

[35] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *Proceedings of the 35th International Conference on Software Engineering*, ser. ICSE'13. IEEE, 2013.

[36] R. Yuzuki, H. Hata, and K. Matsumoto, "How we resolve conflict: an empirical study of method-level conflict resolution," in *2015 IEEE 1st International Workshop on Software Analytics*, ser. SWAN'17. IEEE, 2015.

[37] B. Adams and S. McIntosh, "Modern release engineering in a nutshell – why researchers should care," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER'16. IEEE, 2016.

[38] F. Henderson, "Software engineering at google," *CoRR*, 2017.

[39] G. G. L. Menezes, L. G. P. Murta, M. O. Barros, and A.Van Der Hoek, "On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github," *IEEE Transactions on Software Engineering*, 2018.

[40] P. Accioly, P. Borba, L. Silva, and G. Cavalcanti, "Analyzing conflict predictors in open-source java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR'18. ACM, 2018.

[41] P. Accioly, P. Borba, and G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source java projects," *Empirical Software Engineering*, 2018.