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

Anonymous Authors

***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 involved artifacts. Although there is evidence that semistructured merge has significant advantages over unstructured merge, and that structured merge reports significantly less 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 on 24% of the scenarios with conflicts. Semistructured merge reports more false positives, whereas structured merge has more false negatives. Finally, we observe 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 integration effort and improve integration correctness. For merging software code artefacts, 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] are programming language specific and go beyond simple textual analysis by exploring the underlying syntactic structure and static semantics when integrating programs. Semistructured merge tools [8], [9] attempt to find 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, like method bodies in Java, semistructured merge tools simply apply unstructured merge textual analysis.

Although there is evidence that semistructured merge has significant advantages over unstructured merge [9], and that structured merge tools report significantly less conflicts than unstructured merge [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 conjecture that a structured tool reports less 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 work [8], [9] provide evidence that the extra structure exploited by semistructured merge is not only beneficial. It helps to eliminate certain kinds of spurious conflicts (false positives) reported by unstructured merge, but it might introduce others that can only be solved by a solution that further combines semistructured and unstructured merge. Similarly, 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 run both strategies on more than 40,000 merge scenarios (triples of base, and its two variants parent commits associated with a non-octopus merge commit[1]) 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 on individual tools implementation, 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.

Surprisingly, we found that 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 closer analysis reveals they differ when integrating changes that modified the same textual area in the body of a declaration, but the modifications involve different abstract syntax tree (AST) nodes in the structured merge representation of that body. 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.

We also found that semistructured merge reports false positives in more merge scenarios (36) than structured merge (4), whereas structured merge has more scenarios with false

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

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 would 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 strategies even closer as it approximates their behavior.

II. Semistructured and Structured Merge

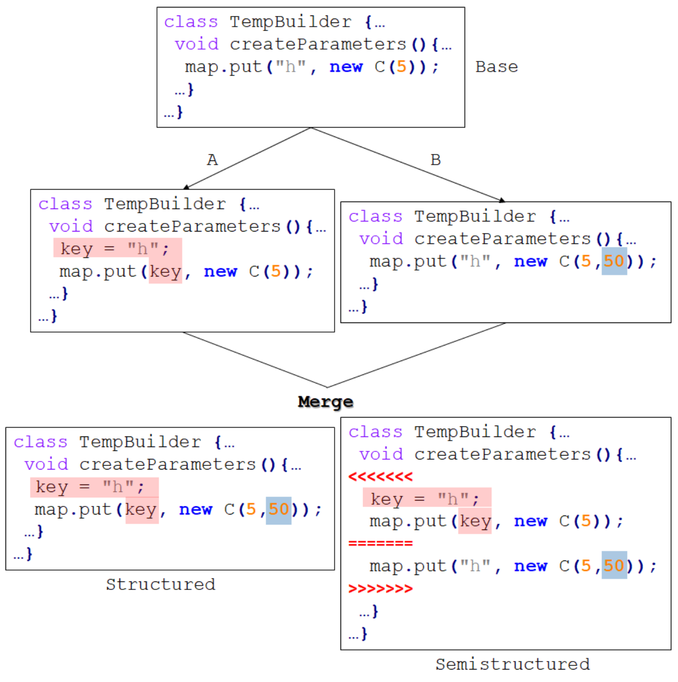
The most widely-used software merging tools are unstructured, which represent every software artifact as text. Although fast, these tools are imprecise. Alternatively, semistructured and structured merge tools incorporate information on the structure of the artifacts being merged. They represent classes 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 strings, and are merged by unstructured merge line-based algorithm.

We illustrate how this difference impacts merging in Figure 1, which shows different versions of part 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. Contrasting, 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 as in the illustrated example. The preference would be for the strategy that reports fewer conflicts. Given that merging contributions 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 a similar situation to Figure 1, 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 [10], and the behavior expected by *A* (new key with old value) and *B*

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

Figure 1: Merging with Semistructured and Structured Merge.



```
class TempBuilder {…
 void createParameters(){…
  map.put("h", new C(5));
 …}
…}
```

```
class TempBuilder {…
 void createParameters(){…
  key = "h";
  map.put(key, new C(5));
 …}
…}
```

```
class TempBuilder {…
 void createParameters(){…
  map.put("h", new C(5,50));
 …}
…}
```

```
class TempBuilder {…
 void createParameters(){…
  key = "h";
  map.put(key, new C(5,50));
 …}
…}
```

```
class TempBuilder {…
 void createParameters(){…
<<<<<<<
  key = "h";
  map.put(key, new C(5));
=======
  map.put("h", new C(5,50));
>>>>>>>
 …}
…}
```

(old key with new value) will not be observed when running the integrated code. In this case, the preference then would be for a semistructured tool, the tool that reports a conflict when integrating these changes.

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 building or execution. Such comparison should be based on the differences between the merge strategies. By construction, they 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 (triples of commits associated with a non-octopus merge commit, that is, a base commit and the two parents of the merge commit) from the development history of a number of software projects, while answering the following research questions.

RQ1: *How many conflicts arise from the use of 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. We then count the number of conflicting merge scenarios, which means 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, selects 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 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. This already corresponds to a large part of the conflicts. We also observed that conflicts occurring inside such declarations are exactly the same or contain slightly different text among 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.

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

We answer this question to better explain the differences quantitatively explored by the previous question. We manually inspect merge scenarios and the code merged with each strategy for a sample of scenarios that have conflicts reported by only one of the strategies, so we can understand the difference on strategies behavior that leads to diverging conflicts.

RQ4: *Which strategy 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. We name these as 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. [10], 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. 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 [10], [11], we rely on build and test information about the integrated code we analyze, and, when necessary, resort to manual analysis. Again, we focus on scenarios that have conflicts reported by only one of the strategies; so 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 are 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 a conflict.

RQ5: *Which strategy has fewer false negatives?*

A merge tool might also fail to detect a conflict (false negative). When this happens, the users 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 false negative for the strategy that yielded the clean merge.

It is important to emphasize that RQ4 and RQ5 consider only the differences between 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. So, we do not need to 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 strategies more similar?*

In the example of Figure 1, semistructured merge reports a conflict because developers *A* and *B* 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. This happens because the invoked unstructured merge reports a conflict whenever it cannot find a line that separates the 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 determine 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 assess 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. Empirical Evaluation

To answer our questions and compute the related metrics, we adopt a two-step setup: mining and execution. In the mining

[3]As illustrated in Figure 2.

step, we use tools that mine GitHub repositories of Java projects to collect merge scenarios— each scenario is composed by the three revisions involved in a three-way merge process associated with a non-octopus merge commit, that is, a base commit and the two parents of the merge commit.

In the execution step, we merge the selected scenarios with both semistructured and structured merge. For each resulting merge free of conflicts, we use a build manager to build the merged version and execute its tests, as this might helps us to find false positives and false negatives. We now detail these steps. All the scripts and data used in this study are available in our online appendix [12].

A. Mining Step

Regarding projects sampling, as our experiment relies both on the analysis of source code and build status information, for service popularity reasons [13] we opt for GitHub projects that use Travis CI for continuous integration. Besides, as the merge tool used in the execution step is language dependent, we consider only Java projects. Similarly, as parsing Travis CI's build log depends on the underlying build automation infrastructure, we analyze only Maven projects because we use its log report information for automatically filtering conflicts. Considering more languages, build systems, and CI services would demand significantly more implementation effort.

We start with the projects in Munaiah et al. [14] and Beller et al. [15] datasets, which include a large number of 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 and Maven configuration files, which indicates that the project is configured to use the Travis CI service, and that the project uses 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 project history, and therefore can be used to derive a merge scenario (the parents of such a commit, together with their most recent common ancestor). As most projects adopted Travis CI only later in project history, we consider only merge commits dated after the first project build on Travis. 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, as we would not be able to confirm whether a problem in the merged version was caused by conflicting changes, they could well have been inherited from the parents, in this case.

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], we argue that our sample has a considerable degree of diversity concerning 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 Jackson Databind has more than 100 KLOC. Moreover, the Web Magic project has 45 collaborators, while OkHttp has 195. We provide a complete list of the analyzed projects in our online appendix [12].

B. Execution Step

After collecting the sample 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, selects 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 adapt and improve previous and independent implementations of a semistructured [17] and a structured merge tool [18]. These tools take as input the three revisions that compose a merge scenario and try to merge their files.

For each merge scenario, we obtain two merged versions: a semistructured version and a structured version. For each file in such versions, 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]

With the number of conflicts in each merged version, 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 are equivalent conflicts, reported by the strategies in different ways, using different sets of markers and associated conflicting text. 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 2, in a merge scenario from project NEO4J-FRAMEWORK.[6]. In this example, 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. Contrasting, semistructured merge reports a single conflict for the entire declaration.

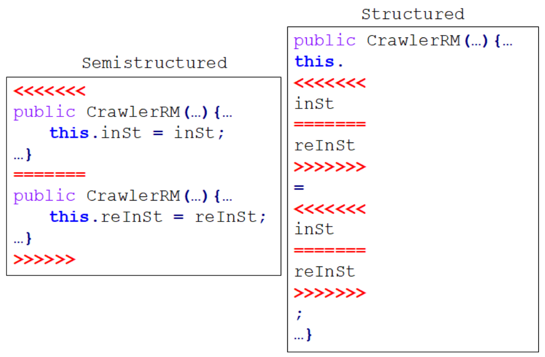
Having identified scenarios for which the strategies differ, we then collect evidence of false positives and false negatives. We

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

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

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

Figure 2: Equivalent conflicts with different granularity.



Semistructured

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

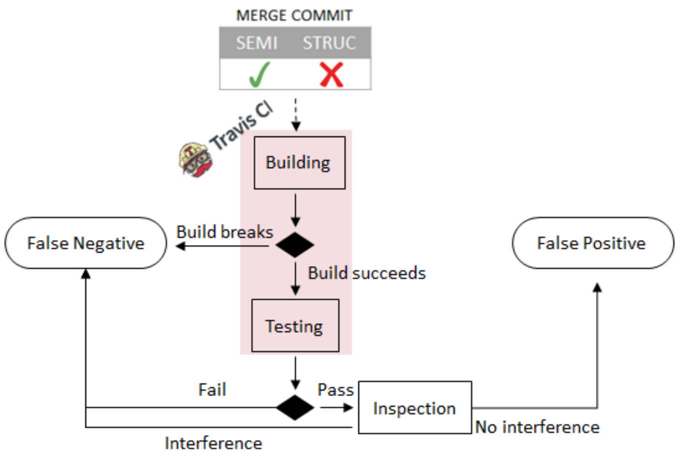
Structured

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

use Travis CI as our infrastructure for building and executing tests for each such scenario, as explained in Section III and illustrated in Figure 3. As Travis CI builds only the latest commit in the 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. Because of that, we use a script that forces build creation for one of the merged versions. Basically, we create a project fork, activate it on Travis CI, and clone it locally. Then, every push to our remote fork creates a new build on Travis CI. So, for each scenario for which the strategies differ (by definition one of the merged versions is clean and the other is conflicting), we create a merge commit with the clean merged version, and push it to our remote fork 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

Figure 3: Building and testing merge commits. A green check mark indicates no conflict with one strategy, a red cross indicates conflict with the other strategy.



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. So, we filter these cases as they do not reflect issues caused by conflicting contributions. 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.

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 conflict. However, whereas this provides precise guarantees for build issues, the guarantees for test issues are as good as the project 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 having potential false positives. In this manual analysis, two authors analyzed the first 5 conflicting files to consolidate the guidelines. Then, two other authors individually analyzed the remaining files. In case of divergence between authors' classification for the same file, another author reviewed that file. In 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, analysing whether they interfere, following the definition of interference in Section III. If one of the developers does not change program semantics, such as when refactoring or simply changing comments, we consider 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 when modifying related state, or when changing assignments to the same variable, we consider that there is interference. We then conclude 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.

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

V. Results

By executing the study design presented in the previous section, we analyze 43,509 merge scenarios from the development history of 508 Java projects. We compare semistructured and structured merge concerning a number of dimensions. In this section, we present the results, following the structure defined by our research questions. Detailed results for the analyzed projects, including tables and plots, are available in our online appendix [12].

A. How many conflicts arise from the use of semistructured and structured merge?

In our sample, we observed 4,732 conflicts when using semistructured merge, and 4,793 when using structured merge. This represents a reduction of 1.27% in the number of reported conflicts when using semistructured merge. Such result, at first, might be surprising to those that expect that more structure leads to conflict reduction. However, as pointed out in Section IV-B and illustrated in Figure 2, structured merge might report more conflicts due to its structure-driven and fine-grained approach for dealing with declarations bodies, including expressions and statements. This approach leads to conflicts that respect the boundaries of the language syntax, but 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 observe 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. In general, conflicts are not frequent when using both strategies.

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

We found 223 (0.51%) scenarios with conflicts reported *only* by semistructured merge, and 30 (0.07%) scenarios have conflicts 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 particularly small because most scenarios are free of conflicts even when using less sophisticated strategies such as unstructured merge. In fact, most of them involve only changes to disjoint sets of files, and could not possibly discriminate between merge strategies because there is no chance of conflict. So, it is important to consider the relative percentages of conflicting merge scenarios, which correspond to 2.28% of our sample scenarios. Overall, the strategies differ in 23.67% of the conflicting scenarios, with 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 next research question.

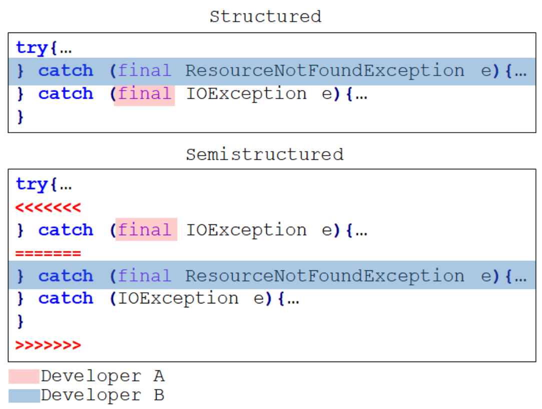
Summary: semistructured and structured merge substantially differ when applied to conflicting scenarios.

C. Why do semistructured and structured merge differ?

To better explain the differences between the merge strategies, we manually analyzed a random sample of 54 merge scenarios that have conflicts reported by only one of the strategies. 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 associate characteristics of the integrated changes with details of the strategy that lead to conflicts.

Starting with scenarios with semistructured merge conflicts, and a structured clean merge, consider the example in Figure 4. Developer *A* added the `final` modifier to the `IOException catch` right after the `try` block. Meanwhile, developer *B* added a new `catch` 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 the conflict illustrated in the figure. 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 a `try` node, and

Figure 4: Semistructured merge conflict from project GLACIERUPLOADER (from merge commit https://git.io/fjney).



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

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 earlier in this paper.

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

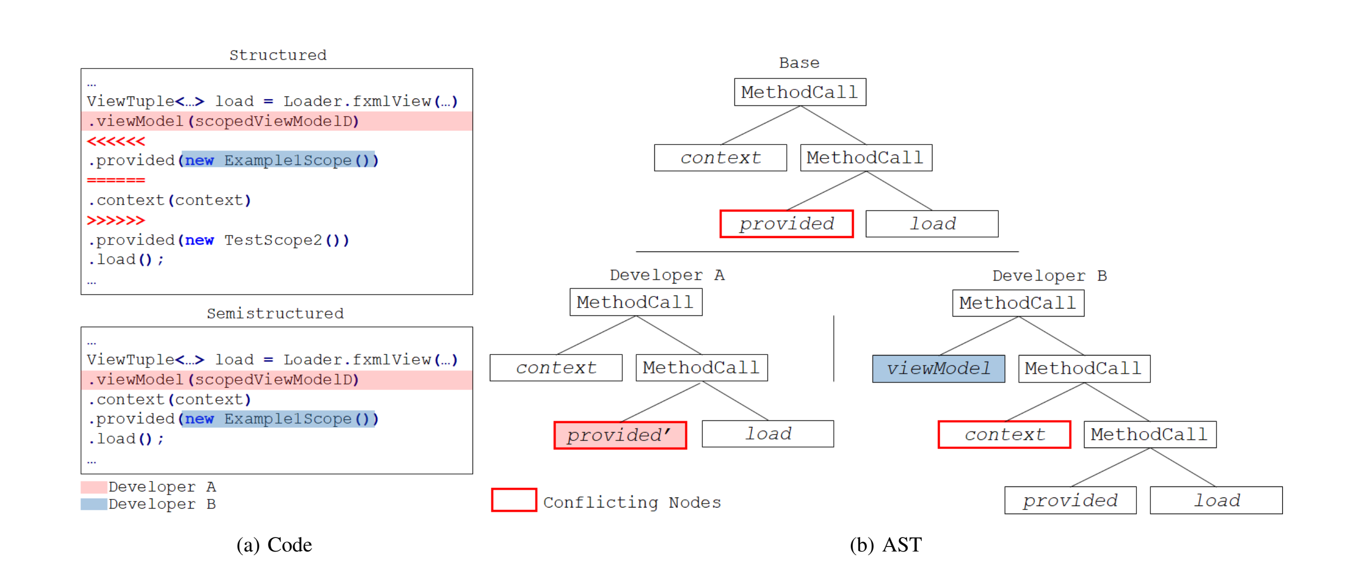
Moving now to scenarios with structured merge conflicts, and a semistructured clean merge, consider the example in Figure 5 (a). Developer *A* added a call to method `viewModel` to an existing method call chain. Developer *B* changed the argument of method `provided` in the same chain. Semistructured merge successfully integrates the changes because it detects 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 5 (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 call to `context` position 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 `provided` by a call to `context`. That 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).

Structured merge syntactic merge makes a difference in a second kind of situation, as illustrated in Figure 6. In this example, 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 because they are represented by nodes of different types. It correctly detects that the subtree of the `for` statement body was changed by one of the developers, but it incorrectly assumes that the whole `for` tree 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`.

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

D. *Which strategy reports fewer false positives?*

As explained in Section IV-B, we use Travis CI to build and test the resulting merged code of the 253 scenarios for which the strategies differ (one reports a conflict and the other cleanly merges the code). We found 44 scenarios with merged

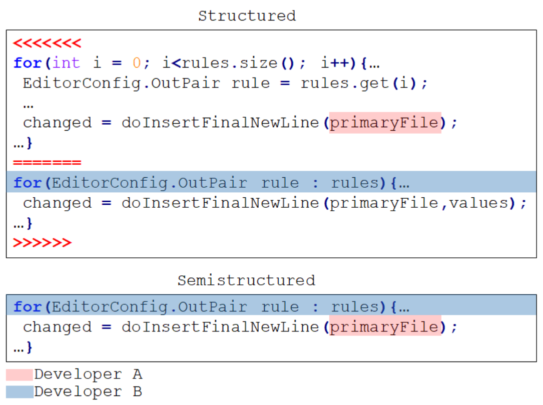

Structured
```
…
ViewTuple<…> load = Loader.fxmlView(…)
.viewModel(scopedViewModelD)
<<<<<<
.provided(new Example1Scope())
======
.context(context)
>>>>>>
.provided(new TestScope2())
.load();
…
```
Semistructured
```
…
ViewTuple<…> load = Loader.fxmlView(…)
.viewModel(scopedViewModelD)
.context(context)
.provided(new Example1Scope())
.load();
…
```
🟥 Developer A
🟦 Developer B

(a) Code

(b) AST

Figure 5: Structured merge conflict from project MVVMFX (from merge commit https://git.io/fjneD).

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


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

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 in these scenarios, there could still be other kinds of semantic conflicts, as unexpected interference between merged contributions might be missed by the 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 scenario contributions. The goal of is to identify possible interference between merged contributions.

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 actually have semistructured merge false positives. Only 3 scenarios were actual true positives of semistructured merge, and, as a consequence, false negatives of structured merge. Although the build was successful, and none of the tests failed, for these three scenarios, we could still observe interference between the merged contributions. 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 expects 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 a conflict. For example, suppose a test that checks whether the size of the list in $n+1$; if it passes in the developers' individuals versions, it will fail in the merged version, in which the size of the list will be $n+2$.

From the 5 scenarios having potential structured merge false positives, 4 of them were classified as actual false positives. Only 1 of the 5 scenarios was an actual true positive of structured merge, and a false negative of semistructured merge. The actual true positive is a scenario from project RESTY-GWT[9]. 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 developer, that checks the value of the mentioned parameter, and then enters into his added `if` block. 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.

E. Which strategy 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 observe 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 of the strategy that merged the code (the other having reported a conflict). Thus, we then focus on 40 scenarios with *errored* and *failed* status caused by the merged contributions; we confirm that 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 contributions.

From the analyzed 40 scenarios, we found only 4 scenarios having semistructured merge false negatives: 3 with *errored* status and 1 with *failed* status for the corresponding merge result produced by semistructured merge (structured merge having reported conflicts for these cases). In contrast, we found 36 scenarios having structured merge false negatives: 23 with *errored* status and 13 with *failed* status for the merge result produced by structured merge (semistructured merge having reported conflicts for these cases).

Although the merge strategies are somewhat different, we observed common causes for false negatives due to broken builds. For example, we found situations in clean merges from both strategies, in projects such as BLUEPRINTS and SINGULARITY, where one developer added a reference to a variable while the other developer deleted or renamed that variable. Consequently, the compiler could not build the file containing the reference to the removed or renamed element. We also observed situations, in projects such as NEO4J-RECO and VRAPTOR, where one developer changed the value passed as an argument, while the

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

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

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 instantiating the same object with different constructors in project DSPACE.

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

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

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

Summary: semistrutured merge reports more false positives, and structured merge misses more conflicts (has more false negatives).

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

Our results shows that our metrics slightly drop if a semistructured merge tool could resolve conflicts due to changes in consecutive lines.[10] In particular, the number of scenarios with semistructured merge conflicts is reduced by 3.38%. Besides, the number of scenarios in which semistructured and structured merge differ is reduced by 11.07%.

As we observed in projects such as QUICKML, SEJDA and SONARQUBE, 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 behavior to structured merge.

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 when they did not agree, another author acted as a mediator. We also asked contributions owner for clarification about the changes when they were not clear.

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

In addition, as we discard merge scenarios that we could not properly build on Travis, or that have broken or failed parents, we might have missed differences in the strategies behavior. We might have also missed that because we analyze only code integration scenarios that reach public repositories with merge commits; that is not the case, for example, of integrations with git rebase, or that were affected by git commands that rewrite history.

Finally, in this study 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 of them change disjoint sets of files, having no chance of leading to conflicts, not mattering 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 strategies differ in about 24% of the cases. This is, though, 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 differing 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%).

When the 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 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 granularity conflicts, as illustrated in Figure 2). Contrasting, structured merge 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, a large part of the observed false negatives correspond to dynamic semantics issues that can easily escape testing and end up affecting users. These are hard to detect and, when detected, are often hard to solve. A more rigorous analysis based on conflict detection and resolution timing data, in the spirit of Berry [19], could differently weight false positives and false negatives and better assess the benefits of the strategies.

Based on our findings regarding false positives and false negatives, and given the observed modest difference between the 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 tools [8]. Together with our consecutive lines result, this discussion points to 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 relation between structure and accuracy in non-semantic merge tools.

The derived observations from our study, especially the ones that explain when the 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 line, our manual analysis of false positives reveal opportunities for making the tool avoid a number of false positives. For example, by detecting straightforward semantic preserving changes, we could avoid 42% of the semistructured false positives in our sample.

Combining the two merge strategies in a similar fashion 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 than those associated with false negatives. Such a tool would eliminate structured merge false negatives, but would still have semistructured merge false negatives. Conversely, in the best case, when structured merge does detect conflicts, it would present structured merge false positives; and, in the worst case, the tool would present semistructured merge false positives. A less conservative combination, in which semistructured is used as long it does not detect conflicts, could also be explored by researchers.

VII. Related Work

A number of studies 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 disadvantages associated with traditional unstructured merge, structured [4], [5], [20]–[25] and semantic strategies have also been proposed [26]–[29].

For example, Apel et al.[4] propose and evaluate a tool that tunes the merging process on-line by switching between unstructured and structured merge, depending on the presence of conflicts. They also propose and evaluate semistructured merge, that takes advantage of underlying language syntactic structure and static semantics, but without the performance overhead associated with full structured merge [8]. Other studies [9], [30] confirm 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. 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 resolve conflicts caused by changes to consecutive lines. We also suggest that a combination of these strategies seems promising as it is able to reduce disadvantages of both strategies.

Souza et al. [29] propose and evaluate *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* is only able to analyze the class declaration associated with a modified method declaration, so it may suffer from its own false positives and false negatives as external callees from other classes might have been modified.

VIII. Conclusions

When integrating code contributions from software development tasks, one might have to deal with conflicting changes. While the state-of-practice still relies 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, to better detect and resolve conflicts. Previous studies provide evidence that semistructured merge has significant advantages over unstructured merge, and that structured merge reports significantly less conflicts than unstructured merge. However, it was known 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, especially when semistructured merge is able to resolve conflicts due to changes in consecutive lines of code. We also discuss that, when deciding which tool to use, a user should consider that semistructured merge reports more false positives, but structured merge miss more conflicts (false negatives). However, combining the two strategies seems promising as it is able to lessen disadvantages of both strategies. As future work, we should implement and evaluate such a combination of strategies to verify its actual benefits and drawbacks.

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 autotuning: 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] S. Horwitz, J. Prins, and T. Reps, “Integrating noninterfering versions of programs,” *ACM Transactions on Programming Languages and Systems*, 1989.

[11] V. Berzins, “On merging software extensions,” *Acta Informatica*, 1986.

[12] Anonymized, “Online appendix for the paper the impact of structure on software merging: Semistructured versus structured merge,” Hosted on http://delaevernu.github.io, 2019.

[13] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, “The impact of continuous integration on other software development practices: A large-scale empirical study,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE’17. IEEE, 2017.

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

[23] 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.

[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://spgroup.github.io/s3m, 2019.

[18] O. Lessenich, G. Seibt, and S. Apel, “Jdime - structured merge with auto-tuning,” Hosted on https://github.com/se-passau/jdime, 2019.

[19] 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

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

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

[22] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

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

[30] 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.