Evaluating and Improving Semistructured Merge

ANONYMOUS AUTHOR(S)

While unstructured merge tools rely only on textual analysis to detect and resolve conflicts, semistructured merge tools go further by partially exploiting the syntactic structure and semantics of the involved artifacts. Previous studies compare these merge approaches with respect to the number of reported conflicts, showing, for most projects and merge situations, reduction in favor of semistructured merge. However, these studies do not investigate whether this reduction actually leads to integration effort reduction (productivity) without negative impact on the correctness of the merging process (quality). To analyze this, and better understand how merge tools could be improved, in this paper we reproduce more than 30,000 merges from 50 open source projects, identifying spurious conflicts reported by one approach but not by the other (false positives), and interference reported as conflict by one approach but missed by the other (false negatives). Our results and complementary analysis indicate that, in our sample, the number of false positives is significantly reduced when using semistructured merge, and we find evidence that its added false positives are easier to analyze and resolve than those reported by unstructured merge. However, our evidence shows that semistructured merge might lead to more false negatives, and we argue that they are harder to detect and resolve than unstructured merge false negatives. Driven by these findings, we propose an improved semistructured merge tool that further combines both approaches to reduce the false positives and false negatives of semistructured merge. We find evidence that the improved tool, when compared to unstructured merge, reduces the number of reported conflicts by half, has no extra false positives, has at least 8% less false negatives, and is not prohibitively slower.

CCS Concepts: •**General and reference** → *Empirical studies;* •**Software and its engineering** → **Software configuration management and version control systems;**

Additional Key Words and Phrases: software merging, collaborative development, version control systems, empirical studies

ACM Reference format:
Anonymous Author(s). 2017. Evaluating and Improving Semistructured Merge. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 21 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In a collaborative software development environment, developers often perform tasks in an independent way, using individual copies of project files. So, when integrating contributions from each task, one might have to deal with conflicting changes, and dedicate substantial effort to resolve them. These conflicts might be detected during merging, building, and testing, impairing development productivity, since understanding and resolving conflicts often is a demanding and tedious task (Bird and Zimmermann 2012; Brun et al. 2011; Kasi and Sarma 2013; Zimmermann 2007). Perhaps worse, conflicts might not be detected during integration and testing, escaping to production releases and compromising correctness.

To deal with these problems, researchers have proposed tools that use different strategies to decrease integration effort and improve integration correctness. For example, unstructured merge tools, which are widely used in industry, rely on purely textual analysis to detect and resolve conflicts (Khanna et al. 2007). Alternatively, semistructured merge tools go further by partially exploiting the syntactic structure and semantics of the involved

A note.
2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

artifacts. For program elements whose structure is not exploited, semistructured merge tools simply apply the usual textual resolution adopted by unstructured merge (Apel et al. 2011).

Previous studies compare these two merge approaches with respect to the number of reported conflicts, showing, for most but not all merges and projects, reduction in favor of semistructured merge. This reduction is mainly due to the automatic resolution of obvious unstructured merge spurious conflicts, which are reported when, for example, developers add different methods to the same file text area. In merge situations where semistructured merge reduces the number of reported conflicts, Apel et al. (2011) show an average reduction of 34% compared to unstructured merge. In a replication of this study, Cavalcanti et al. (2015) find an even greater average reduction of 62%, again in favor of semistructured merge.

This evidence, however, is not enough to justify industrial adoption of semistructured merge. The problem is that previous studies do not investigate whether the observed conflict reduction actually leads to integration effort reduction (productivity) without negative impact on the correctness of the merging process (quality). In fact, the observed reduction of spurious conflicts (*false positives*) could have been obtained at the expense of missing actual interference between developers changes (*false negatives*). If that is the case, we would be simply postponing conflict detection to other integration phases such as building and testing, or even letting conflicts escape to users. Moreover, given that the set of conflicts reported by semistructured merge in previous studies is often smaller but not a subset of the set reported by unstructured merge, semistructured merge could even be introducing other kinds of false positives that might be harder to resolve than the ones it eliminates. As even a minor disadvantage of a new tool can become a huge barrier for its adoption in practice, if we want to move forward on the state of the practice on merge tools, it is important to have solid evidence and further knowledge about false positives and false negatives resulting from these two merge approaches.

So, to better compare those two merge approaches and understand how merge tools could be improved, in this paper we reproduce 34,030 merges from 50 GitHub projects. We collect evidence about spurious conflicts reported by unstructured merge but not reported by semistructured merge (and vice versa). We also evaluate interference missed by semistructured merge but reported by unstructured merge (and vice versa). In particular, we investigate the following main research questions:

- **RQ1** *When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort by reporting fewer spurious conflicts?*
- **RQ2** *When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more non spurious conflicts?*

In summary, in our sample, the number of false positives is significantly reduced when using semistructured merge, and we find evidence that its added false positives are easier to analyze and resolve than those reported by unstructured merge. On the other hand, our evidence shows that semistructured merge might lead to more false negatives, and we argue that they are harder to detect and resolve than unstructured merge false negatives. Although our comparison process favors unstructured merge whenever we are not able to precisely classify a conflict, this last finding might justify non adoption of semistructured merge in practice. Nevertheless, our findings shed light on how merge tools can be improved. So we benefit from that and propose an improved semistructured merge tool that further combines both merge approaches to reduce the false positives and false negatives of semistructured merge. We find evidence that the improved tool, when compared to unstructured merge, reduces the number of reported conflicts by half, has no extra false positives, has at least 8% less false negatives, and is not prohibitively slower.

2 COMPARING MERGE APPROACHES

Although version control systems (VCSs) have evolved over the years, merge tools have not evolved much. The state of the practice still is textual, line-based, **unstructured merge**, such as GNU merge (Mens 2002). These

tools are based on the so-called *diff3* algorithm. When merging files, unstructured merge tools typically compare, line by line, two modified files in relation to their common ancestor (the version from which they have been derived) and detect sets of differing lines (*chunks*), in a process called *three-way merge*. For each chunk, the tool checks whether the three revisions have a common text area that separates the chunk content. If such separator is not found, the tool reports a conflict (Khanna et al. 2007).

Contrasting, **semistructured merge** (Apel et al. 2011) understands part of the language syntax and semantics. Such tools represent part of the program elements as trees and rely on information about how nodes of certain types (methods, classes, etc.) should be merged. Trees are merged recursively, through superimposition, which matches nodes based on structural and nominal similarities (Apel and Lengauer 2008). Such trees include some but not all syntactic structural information. Concerning Java, for example, classes, methods and fields appear as nodes in the tree, whereas statements and expressions in method (or constructor) declarations appear as plain text in tree leaves.[1] To merge leaves, semistructured merge simply calls unstructured merge. Considering all language elements as nodes, as in structured merge, can considerable impact performance. So we opt for a comparison with semistructured merge, which is likely more adoptable in practice.

To compare these two merge approaches, we could simply measure how often they are able to merge contributions. The preference would be for the approach that most often generates a syntactically valid resulting program that integrates both contributions. Under this criteria, semistructured merge would be superior because it often reports less conflicts, as reported in previous studies. Given that merging contributions is the main goal of any merge tool, in principle that criteria could be satisfactory. However, in practice merge tools go slightly beyond that and might detect other kinds of integration conflicts that don't preclude them from generating a valid program, but would lead to build or execution failures. So, from a developer's perspective, it is important to use a comparison criteria that considers not only the capacity of generating a merged program, but also the possibility of missing or early detecting conflicts that could appear during building or execution.

The challenge for using such a more comprehensive comparison criteria is establishing ground truth for integration conflicts (and therefore false positives and false negatives), or more generally interference between development tasks. In this context, interference is not computable (Berzins 1986). Semantic approximations through static analysis or testing are imprecise and often too expensive, in the case of information flow analysis. Experts who understand the integrated code (possibly developers of each analyzed project) could determine truth, but not without the risk of misjudgment. As these options would imply into a reduced sample and limited precision guarantees, we prefer to relatively compare the two merge approaches with regard to the *added* occurrence of false positives and false negatives associated to each approach. We do that by simply analyzing when the merge approaches report different results for the same merge scenario— each scenario contains the revisions involved in a three-way merge. We identify spurious conflicts reported by one approach but not by the other (false positives), and interference reported as conflict by one approach but missed by the other (false negatives).

To guide our relative comparison, we first tried to understand the differences in the behavior of representative tools of both approaches. As semistructured merge tool, we use the original implementation of `FSTMerge` (Apel et al. 2011), with the annotated Java grammar they provide, following previous studies (Apel et al. 2011; Cavalcanti et al. 2015). Besides that, we arbitrarily chose the `Kdiff3` tool, which is one of the many unstructured merge tools available but is a representative implementation of the *diff3* algorithm.[2]. We systematically analyzed their implemented algorithms, and empirically assessed a small sample of Java merge scenarios to observe how and when they behave differently, and how this might lead to false positives and false negatives. In particular, for each conflict reported by unstructured merge, we checked whether semistructured merge also reported that

[1] From now on, we use method declarations to refer both to method and constructor declarations.
[2] http://kdiff3.sourceforge.net/

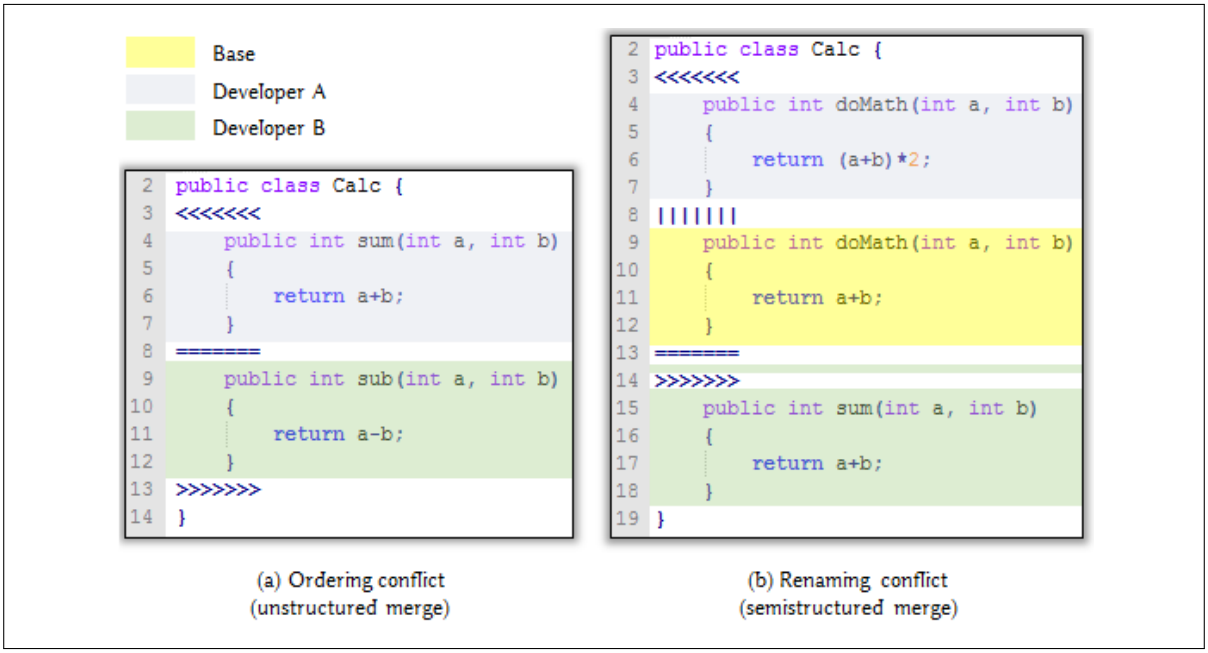

Fig. 1. False positives added by unstructured and semistructured merge. Conflict markers in blue. We show developers contributions together due to space limitations.

conflict, and vice versa. In case of divergence, we judged if the conflict was a false positive or a false negative. In the following sections, we describe the observed kinds of added false positives and false negatives of each merge approach. Although we use toy examples for simplicity, the inspiration comes from concrete merge scenarios from non trivial open source projects.

2.1 False Positives Added by Unstructured Merge

One of the main weaknesses of unstructured merge is its inability to detect rearrangeable declarations. In Java, for instance, a change in the order of method and field declarations has no impact on program behavior, but unstructured merge might report false positives— the so-called *ordering conflicts*— when developers add declarations of different elements to the same part of the text. Figure 1(a) illustrates this situation: a conflict reported because different developers added two different methods (`sum` and `sub`, separated by typical conflict markers) to the same text area. In contrast, this is not reported as a conflict by semistructured merge. Due to superimposition, semistructured merge identifies commutative and associative declarations, and understands that the changes are related to different nodes. As discussed later, not all ordering conflicts are false positives added by unstructured merge: import declarations are often, but not always, rearrangeable.

2.2 False Positives Added by Semistructured Merge

Renamings challenge semistructured merge, as illustrated by Figure 1(b), which shows a false positive *renaming conflict*: one of the developers renamed the `doMath` method to `sum`, whereas the other developer kept the original signature but edited the method body. Semistructured merge reports this conflict because its algorithm interprets method renaming as method deletion, and assumes that a developer deleted the method changed by the other.

In particular, to check whether a base declaration was changed by both developers, the merge algorithm tries to match nodes by the type and identifier of the corresponding declaration. In Java, for instance, a method is identified by its name and the types of the formal parameters. So, when an element is renamed, the merge algorithm is not able to map the base element to the new elements in the changed version of the file, and assumes the element was deleted. Note that the `sum` method does not appear in the conflict; that is, it is not surrounded by the conflict markers (the vertical bars and equal signs separate base code from code to be merged).

In the illustrated case, merging the new signature with the new body would be a safe valid integration; there is no interference because the changes do not affect the developers' expectations: the method will behave as desired by one developer, and will be called as wanted by the other developer. This is exactly what is done by unstructured merge. It does not report a conflict because the changes occur in distinct text areas; in the example, the "{" in line 10 separates the two changed areas. Note that renaming conflicts involving field declarations are also possible, but they could only be a semistructured merge added false positive in uncommon cases where the field declaration is slip into multiples lines of code.

Renaming conflicts are often false positives, but they might be true positives too. For example, consider the similar case where one of the developers renames a method, and the other developer not only changes this method body but also adds new calls to it. Merging the new signature with the new body would lead to an invalid program that calls an undeclared method. Semistructured merge would soundly not perform the merge and report a conflict, whereas unstructured merge would unsoundly merge the contributions provided there is a separator as in the illustrated example.

2.3 False Negatives Added by Unstructured Merge

Unstructured merge added false negatives are mostly caused by failing to detect that the contributions to be merged add duplicated declarations. For example, unstructured merge reports no conflict when merging developers contributions that add declarations with the same signature to different areas of the same class. This leads to an invalid resulting program with a compiler *duplicated declaration error* and a build issue. Figure 2(a) illustrates this situation: both developers added methods with the same signature but with different bodies, in different areas, not leading to a conflict by unstructured merge. As semistructured merge matches elements to be merged by their type and identifier, it would be able to detect the problem and correctly report a conflict.

Besides that, renaming conflicts can be added false negatives by unstructured merge. As explained in the previous section, renaming conflicts detected by semistructured merge might be true positives. But these would only be detected by unstructured merge in case the changes occur in the same text area. For instance, consider a similar example to the one in Figure 1(b), but where developer A also added a call to method `doMath` in an area not changed by developer B. As there are separators between the developers changes, unstructured merge would erroneously not report a conflict.

2.4 False Negatives Added by Semistructured Merge

As mentioned in Section 2.1, ordering conflicts involving import declarations are often, but not always, false positives. In fact, merging developers contributions that add import declarations to the same text area might lead to false negatives added by semistructured merge, as it assumes that such declarations are rearrangeable. For example, this might lead to a *type ambiguity error* (see Figure 2(b)) and build issues when the import declarations involve members with the same name but different packages. In the illustrated case, both imported packages have a `List` class. As the import declarations appear in the same or adjacent lines of code, an unstructured tool would correctly report a conflict, whereas the semistructured tool would let the conflict escape. In other situations, semistructured merge's assumption about import declarations might lead to behavioral errors instead. In the illustrated example, suppose that developer A had written `import java.awt.List`. As the ambiguous members might share methods with the same signature but different behavior, the presence of both declarations

Fig. 2. False negatives added by unstructured and semistructured merge.

might affect class behavior. In the example, both `List` members have `add` methods behaving differently. Again, semistructured merge would miss the conflict, which would be reported by unstructured merge.

Besides this issue with import statements, semistructured merge also adds false negatives due to the way it handles *initialization blocks*. Since it uses elements type and identifier to match nodes, the algorithm is unable to match nodes without identifiers. This leads to problems like the one illustrated in Figure 2(d); as the initialization elements have no identifier, semistructured merge cannot match them, and therefore creates duplicates. A conflict should have been reported so that the developers could negotiate an adequate solution for both of them. This is what is done by unstructured merge in case the initialization blocks were added to the same text area.

Other kinds of semistructured merge added false negatives do not conform to a small set of recurring syntactic patterns, but they all result from unstructured merge *accidentally* detecting conflicts that would otherwise escape if changes were performed in slightly different text areas. For example, this might occur when developers change or add, in the same text area, different but dependent elements. In particular, we observed semistructured merge added false negatives in cases where one developer adds a new element that references an existing one that is edited by the other developer. In such cases, the developer who added the new element might not be expecting the changes made to the referenced element, possibly leading to negative impact on the merged program behavior. This is illustrated in Figure 2(c), where we see that the new method `composed` references the method `doMath`, which was changed by the other developer. Although one might assume that methods provide the lowest level of information hiding and modularity, having its signature as interface, in practice an unchanged method interface is not sufficient to ensure that the method behavior is also unchanged. So we consider that could be an added false negative. In fact, as the changes correspond to different elements (technically, different nodes in the semistructured merge tree), semistructured merge reports no conflict. Contrasting, unstructured merge might accidentally detect such conflicts when the changes are made in the same text area.

2.5 Common False Positives and False Negatives

The kinds of false positives and false negatives described in the previous sections correspond only to the differences between semistructured and unstructured merge algorithms. As our interest here is to compare both approaches— not to establish how accurate they are in relation to a general notion of interference— we do not need to measure the occurrence of false positives and negatives when both approaches behave identically. For example, when processing changes inside method bodies, semistructured merge actually calls unstructured merge, so they present common false positives and false negatives. As an example of a common false positive consider that developers edit consecutive lines in a method, but one of them does not change behavior (simply refactors or changes spacing). A common false negative would be edits to different method lines, in different areas of the body, but with a data flow dependency between the statements in such lines. Besides that, it is important to note that unstructured merge might accidentally report renaming false positives in case changes occur in the same area, or miss the same kinds of false negatives reported in the previous section, in case changes are not in the same area. But these cases don't interest us because both merge tools would behave identically. Although important for establishing accuracy in general, these are not useful for relatively comparing merge approaches.

3 EMPIRICAL EVALUATION

Our evaluation investigates whether semistructured merge reduction in the number of conflicts in relation to unstructured merge (Apel et al. 2011; Cavalcanti et al. 2015) actually leads to integration effort reduction (productivity) without negative impact on the correctness of the merging process (quality). We do that by reproducing merges from the full development history of different GitHub projects, and collecting evidence about the occurrence of conflicts, and the kinds of diverging false positives and false negatives described in the previous section. In particular, we investigate the following research questions:

- **RQ1** *When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort by reporting fewer spurious conflicts?*
- **RQ2** *When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more non spurious conflicts?*

To answer **RQ1**, we compute the *overestimated number of false positives added by semistructured merge— aFP(SS)* (spurious conflicts reported by semistructured merge and not reported by unstructured merge) metric. We also compute the *underestimated number of false positives added by unstructured merge— aFP(UN)* (spurious conflicts reported by unstructured merge and not reported by semistructured merge) metric. As the metrics names suggest,

they are approximations. To consider a large sample, as discussed in Section 2, we basically evaluate the merge approaches by computing the number of false positives added and removed by one merge approach in comparison to the other. However, precisely computing the number of the described false positives (and false negatives) is hard, as discussed later in this section. Nevertheless, if we find that an upper bound value is inferior to a lower bound one, we can conclude that the exact value underlying the *overestimated* number is lower than the exact value underlying the *underestimated* one. This was indeed observed in dry runs of our study, giving us confidence that we could adopt this design. As different conflicts might demand different resolution effort (Mens 2002; Prudêncio et al. 2012; Santos and Murta 2012), comparing conflict numbers might not be enough for understanding the impact on integration effort. So, to better understand the effort needed to resolve different kinds of conflicts, we conduct a number of complementary analyses to estimate the impact on integration effort. Our goal with these analyses is to simply check that the computed metrics are not obviously bad choices as proxies for integration effort.

For answering **RQ2**, we compute the *overestimated number of false negatives added by semistructured merge— aFN(SS)* (conflicts missed by semistructured merge and correctly reported by unstructured merge), and the *underestimated number of false negatives added by unstructured merge— aFN(UN)* (conflicts missed by unstructured merge and correctly reported by semistructured merge) metrics. We also discuss the impact of the kinds of false negatives of both approaches, but this does not require further elaborated analyses.

To answer these questions and compute the related metrics, we adopt a three-step setup: mining, execution and analysis. In the *mining* step, we use tools that mine GitHub repositories to collect merge scenarios— each scenario contains the three revisions involved in a three-way merge. In the *execution* step, we use an unstructured and a semistructured merge tool to merge the selected merge scenarios and to find added false positives and false negatives candidates. In the *analysis* step, we filter these candidates to confirm their occurrence. Finally, we use R scripts to analyze the results. We now detail these steps, explaining together the execution and analysis steps for simplicity.

3.1 Mining Step

To select more meaningful projects, we first searched for the top 100 Java projects with the highest number of stars in GitHub's advanced search page.[3] From this search result, we selected 50 projects having degree of diversity (Nagappan et al. 2013) with respect to a number of factors described later. We restricted our sample to Java projects because the execution and analysis steps demand language dependent tool implementation and configuration. After selecting the sample projects, we use tools to mine their GitHub repositories and collect merge scenarios from their full histories. In particular, we use the GitMiner tool to convert the entire development history of a GitHub project into a graph database.[4] Subsequently, we implemented scripts to query this database and retrieve a list of the identifiers of all merge commits— commits that were created by a `git merge` command— and their parents. As a result, we obtained 34,030 merge scenarios from the 50 selected Java projects. Given that part of the execution and analysis steps are language dependent, we process only the Java files in these scenarios, missing conflicts in non-Java files (we discuss this threat later in Section 5).

Although we have not systematically targeted representativeness or even diversity, we believe that our sample has a considerable degree of diversity with respect to, at least, the number of developers, source code size and domain. It contains projects from different domains such as databases, search engines and games. They also have varying sizes and number of developers. For example, retrofit, an HTTP client for Android, has only 12 KLOC, while OG-Platform, a solution for financial analytics, has approximately 2,035 KLOC. Moreover, mct has 13 collaborators, while dropwizard has 141. Besides that, our sample projects includes projects such as cassandra,

[3] https://github.com/search/advanced
[4] https://github.com/pridkett/gitminer

Junit and Voldemort, which are analyzed in previous studies (Brun et al. 2011; Cavalcanti et al. 2015; Kasi and Sarma 2013). The list of the analyzed projects, together with the tools we used, is in our online appendix (Appendix 2017).

3.2 Execution and Analysis Steps

After collecting the sample projects and merge scenarios, we use the `KDiff3` and the `FSTMerge` tools described in Section 2, respectively, as our unstructured and semistructured tools to merge the selected scenarios. We then identify and compare the occurrence of the added false positives and false negatives described in Section 2. These tools take as input the three revisions that compose a merge scenario (here we call them as *base*, *left*, and *right* revisions) and try to merge their files. To identify false positives and false negatives candidates, we intercept `FSTMerge` during its execution. Given that the tool is structure-driven, we are able to inspect the source code and the conflicts in terms of the syntactic structure of the underlying language elements. This would not be possible with a textual tool. To confirm the occurrence of the false positives and false negatives, we use a number of scripts; some of them rely on the parsing and compiler features of the Eclipse JDT API.[5] For brevity, here we overview how we compute the metrics, and leave the detailed explanation to the online appendix (Appendix 2017), where we also point to the version of `FSTMerge` that contains the interceptors we need.

3.2.1 Computing the Overestimated Number of False Positives Added by Semistructured Merge— aFP(SS). The semistructured merge added false positives, as explained in Section 2.2, are due to renamings. To identify these false positives, we first intercept conflicts detected by `FSTMerge` and check whether the involved triple of base, left and right elements contains a nonempty base, and an empty left or right element. Not having the left or right version, represented by the red empty string in Figure 3, indicates that the semistructured merge algorithm could not map the element (method in this case) to its previous version, either because the element was renamed or deleted. Since we cannot precisely guarantee there was a deletion (the best option would be a similarity analysis on method bodies), we conservatively analyze all such cases.

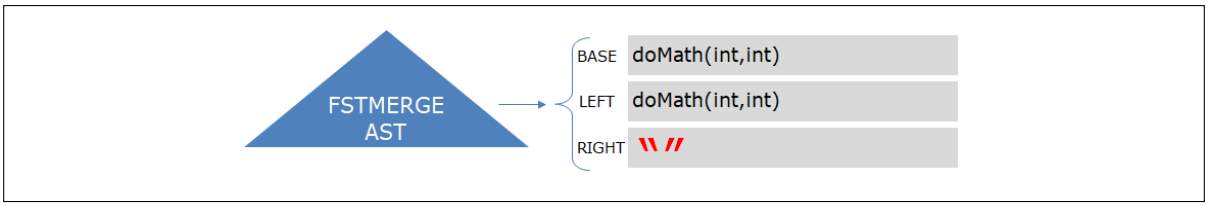

Fig. 3. Intercepting `FSTMerge` tool to find *renaming* false positives (*aFP(SS)*) candidates.

To filter out obvious renaming true positives, we consider as added false positives only cases with no remaining references to the renamed or deleted element. For efficiency and simplicity, we conservatively check that by parsing the merged revision files and looking for nodes that refer to the original signature of the renamed or deleted element. More precise analyses with call graphs, and verifying that the existing references are not part of dead code, would still be incomplete and would compromise our sample size because of both performance overhead and the need for building projects. That is why we opt for computing the *overestimated* number of false positives added by semistructured merge.

[5] http://www.eclipse.org/jdt/

3.2.2 Computing the Underestimated Number of False Negatives Added by Unstructured Merge— `aFN(UN)`. The main pattern of unstructured merge added false negative occurs, as explained in Section 2.3, when developers introduce duplicated declarations in different areas of the program text. To identify such situations, we intercept `FSTMerge` when it matches triples of elements with an empty base and nonempty left and right elements. This indicates the addition of two new elements with the same signature. To confirm that there is a duplicated declaration error, we merge the files with the unstructured tool and verify if there is no conflict involving the duplicated candidate. In case there is an unstructured merge conflict, the candidate is discarded. If there is no conflict, we try to compile the resulting file, and we search the compiler output for compilation problems corresponding to duplicated declarations related to the observed candidate.

Whereas we are able to precisely compute the number of duplicated declaration errors, it would be harder to precisely compute the other kinds of false negatives— such as true renaming conflicts— missed by unstructured merge. So we actually compute the *underestimated* number of false negatives added by unstructured merge.

3.2.3 Computing the Overestimated Number of False Negatives Added by Semistructured Merge— `aFN(SS)`. As explained in Section 2.4, the false negatives added by semistructured merge are related to three major causes: reordering import statements that involve types with the same identifier, not matching initialization blocks, and unstructured merge accidental conflict detection.

To compute the first kind of false negative, we use `FSTMerge` to identify attempts to merge trees that contain at least a pair of introduced or modified import declaration nodes. Afterwards, we check in the corresponding file merged by unstructured merge if there is a conflict involving the pair of imports statements (otherwise it would be a common negative of both approaches). In case there is a conflict involving the import statements reported by unstructured merge, to check if the import declarations lead to type ambiguity errors, we compile the resulting merged file by semistructured merge and search for type ambiguity compilation errors. To check if the import statements might lead to behavioral issues, we search for the name of the member imported by one developer in the changes introduced by the other, and vice versa. With a positive result in one of the checks, we conservatively consider the pair of import statements as a conflict missed by semistructured merge (false negative).

To identify the second kind of false negative, we select all nodes representing initialization blocks in the three versions of the trees representing the files being merged. Using textual similarity, based on the *Levenshtein distance algorithm* with 80% of degree of similarity, we group triples of similar initialization nodes in the different tree versions. Lastly, we merge these triples with unstructured merge. If they conflict with unstructured merge, we conservatively classify the conflict as a semistructured merge added false negative.

All other cases of conflicts detected by unstructured merge but not by semistructured merge are conservatively classified as added false negatives except in the following two cases when the unstructured merge conflict can be parsed. First, when the conflict contains only field declarations that do not reference each other. Second, when the conflict resolution keeps all changes from both left and right revisions, and adds no new code; we check that by parsing and inspecting the original merge commit in the project repository. We assume that the developer correctly analyzed the conflict and decided there was no interference; so that would be a unstructured false positive, not a semistructured false negative. More precise analyses, such as testing or information flow, could possibly reduce our upper bound of semistructured merge false negatives, but would still be imprecise and reduce the analyzed sample, as explained earlier.

3.2.4 Computing the Underestimated Number of False Positives Added by Unstructured Merge— aFP(UN). The false positives added by unstructured merge are due to its inability to perceive that some declarations are commutative and associative— the ordering conflicts (see Section 2.1). We found no specific patterns of ordering conflicts that would allow us to identify them by systematically inspecting the reported conflicts. We can, however, compute this metric in terms of the others. Figure 4 illustrates the set of conflicts reported by unstructured and

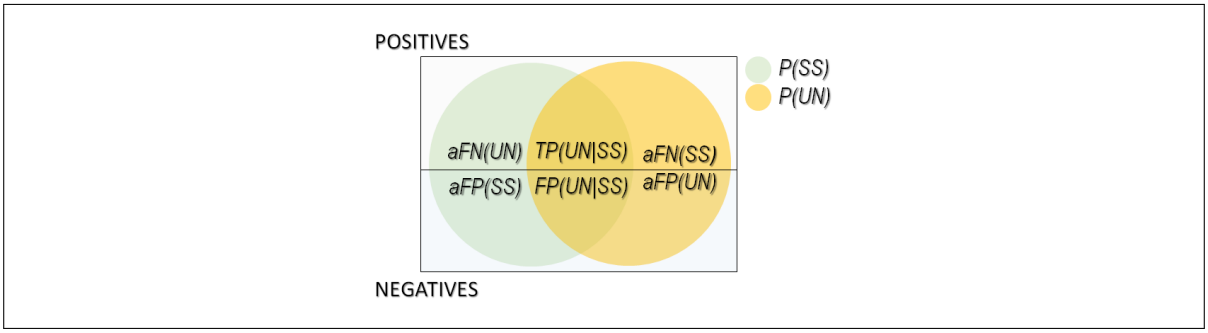

Fig. 4. Set of conflicts reported by unstructured and semistructured merge

semistructured merge. For instance, unstructured merge set includes its own false positives (*aFP(UN)*). The sets also include true and false positives common to both approaches, denoted by *FP(UN|SS)* and *TP(UN|SS)*. Based on the diagram, we can infer that

$$aFP(UN) = P(UN) - (FP(UN|SS) + TP(UN|SS)) - aFN(SS)$$

Observe that we can estimate $FP(UN|SS) + TP(UN|SS)$ in terms of *P(SS)* (in green in the diagram). To compute a lower bound of *aFP(UN)*, we need an upper bound of $FP(UN|SS) + TP(UN|SS)$ because it is a subtractive factor. This upper bound is reached when *aFP(SS)* is at its minimum (zero). Finally, we can derive the underestimated number of false positives added by unstructured merge as follows:[6]

$$aFP(UN) \geqslant P(UN) - P(SS) + aFN(UN) - aFN(SS)$$

Note that *P(SS)* and *P(UN)* can be simply computed by running the merge tools and observing the reported conflicts.

4 RESULTS AND DISCUSSION

By analyzing a total of 34,030 merge scenarios from 50 Java projects, we identified 19,238 conflicts when using unstructured merge, and 14,544 using semistructured merge, representing a reduction of approximately 24% in the total number of reported conflicts. Besides that, 8.80% of the sample merge scenarios had conflicts when using unstructured merge, compared to 7.11% with semistructured merge. We observed that semistructured merge was able to reduce the number of reported conflicts in 54.60% of the merge scenarios. This is similar to the studies of Apel et al. (2011) and Cavalcanti et al. (2015), differing at most by 5%. This reinforces our assumption that studying only numbers of conflicts is not enough for justifying the adoption of a merge tool. [7] In these scenarios, the observed reduction in the total number of conflicts was of 71.02 ± 29.88% (average ± standard deviation), compared to 62 ± 24% in the study of Cavalcanti et al. (2015), and 34 ± 21% in the study of Apel et al. (2011). Considering the merge scenarios having conflicts, regardless the merge approach, we found that in 27.07% of them one approach detected at least one conflict, and the other none. In 17.07% the approaches reported the same conflicts, and in 25.88% the approaches detected only different conflicts. So they differ more substantially than we originally expected. In the remaining of the section, we further present descriptive statistics, structured according to our research questions, and discuss their implications. Detailed results for the analyzed projects are available in the online appendix (Appendix 2017).

[6]We formally derive the formula in the online appendix.
[7]Considering merge scenarios in which either unstructured or semistructured merge reported at least one conflict, as in the previous studies.

4.1 When compared to unstructured merge, does semistructured merge reduce unnecessary integration effort by reporting fewer spurious conflicts?

To answer RQ1, we compare the number of false positives added by each merge approach. Our results show that, in our sample, when using an unstructured merge tool, 6.58 ± 6.07% of the merge scenarios have at least one estimated added false positive (*aFP(UN)*). Moreover, 43.47 ± 19.01% of the reported conflicts are false positives according to our metric (*aFP(UN)*). This is bigger than the percentage of false positives added by semistructured merge (*aFP(SS)*): 30.21 ± 20.68%. In addition, only 3.12 ± 3.55% of the merge scenarios have at least one added false positive (*aFP(SS)*). So, considering the aggregated scenarios of all projects, we conclude that semistructured merge has fewer added false positives and fewer scenarios with added false positives. In practice, we should expect a bigger difference in favor of semistructured merge since *aFP(UN)* is underestimated and *aFP(SS)* is overestimated. However, these findings do not uniformly hold across projects: *aFP(SS)* is greater than *aFP(UN)* in 26% of our sample projects. Contrasting, in only 2% of the projects semistructured merge had more merge scenarios with added false positives. We observed that these semistructured merge loss was caused by renaming of directories. In such cases, due to semistructured merge finer granularity, the conflicts are reported per method or constructor in the files of the renamed directory, substantially increasing the number of reported conflicts.

Given that our data is paired, and deviates from normality, we analyze differences in the computed metrics with the paired Wilcoxon Signed-Rank test. It shows that the merge approaches present statistically significant different means of percentages of merge scenarios with false positives (*p-value* = 1.13e-09 <0.05). Besides that, we observed a large effect size (r = 0.82>0.5). There is also significant difference between the percentages of added false positives (*p-value* = 0.001047<0.05), with medium effect size (r = 0.46>0.3). This tendency can also be observed in the box plots of Figure 5. Note, for instance, that in both cases (merge scenarios and conflicts) the 3rd quartile in the box plots of the semistructured approach is inferior to the median in the box plots of unstructured merge.

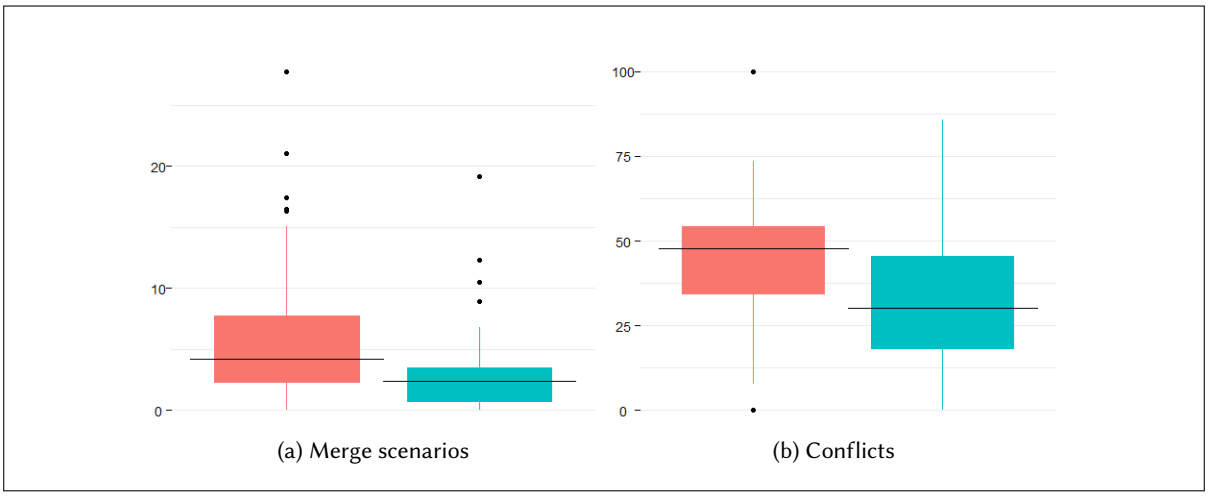

Fig. 5. Box plots describing the percentage, per project, of added false positives in terms of merge scenarios and conflicts. Unstructured merge in red, semistructured in blue.

4.1.1 Semistructured Merge False Positives Easier to Analyze and Resolve. Although semistructured merge reduction in the number of added false positives for most projects suggests that it might reduce integration effort,

a more accurate comparison would measure the actual effort required for analyzing and discarding false positives. This is important because different conflicts often demand different effort to be analyzed, and then discarded or resolved. Thus, to better understand the impact of false positives on integration effort, we first manually analyzed 100 merge scenarios: 50 containing semistructured merge renaming conflicts (likely false positives), and 50 with unstructured merge ordering conflicts (likely false positives). Each block of 50 contains one randomly selected scenario from each project in our sample.

For each scenario, we observed the reported conflicts to understand how easily they could be analyzed and discarded. We also observed the corresponding merge commit in the project history, to understand how each developer contribution was integrated. Similarly to Menezes (2016), we assume that a merge commit that contains only code coming from the integrated contributions results from a merge process, and eventual conflict resolution, that requires less effort than when new code (including combination of contributed code) appears in the merge commit.

This manual analysis revealed that semistructured merge false positives (a fraction of the renaming conflicts) are easy to analyze and resolve. As explained before and illustrated in Figure 1(b), this kind of conflict shows the original element name with its new body (as left, for example), and its original body (as base). The integrator can then easily find a corresponding element with a new name and original body.[8] Resolution basically consists of declaring a single element, with the new name and the new body. But we have also observed cases where the integrator discarded the new name or the new body. In retrospect, we noticed that a simple analytical analysis of how semistructured merge reports renaming conflicts would be sufficient to reach the same conclusion, demanding no further empirical analysis.

With respect to unstructured merge false positives (a fraction of the ordering conflicts), only part of the manually analyzed cases was easy to analyze and resolve. We believe that conflicts caused by the introduction of declarations (methods or fields) in the same text area can often be analyzed and resolved with little effort because the integrator simply has to choose one of the declarations, or decide to keep them all. However, we also observed a challenging kind of ordering conflict that does not respect the boundaries of Java syntactic structures. These *crosscutting conflicts* mix parts of different language elements, as illustrated in Figure 6, observed in a merge scenario of project cassandra. Note that parts of the `getColumn` and `validateMemtableSetting` methods conflict because the changes occurred in the same text area. Such conflicts are more difficult to analyze and resolve because they demand one to map code chunks to corresponding syntactic structures; in the illustrated example, it is not clear to which method the `for` and `if` statements belong. As such kind of conflict involves different syntactic elements (and then different nodes in a parse tree), semistructured merge automatically resolves them.

To understand how these findings are related to our entire sample, we carried on further automatic analysis. By trying to parse code of the unstructured merge added false positives, we found that 44.81% of the conflicts are crosscutting; that is, we could not parse the conflict text because it does not correspond to a single valid language element. When analyzing how these conflicts were resolved, we found that 92.76% of the resolutions involved no new code. This suggests that a significant part of unstructured merge false positives might be hard to analyze, but their resolution is rarely hard. In fact, conflict analysis might be so hard that resolution might simply correspond to discarding one of the contributions.

> In our sample, though not uniformly across projects, semistructured merge reduced the overall number of reported conflicts, and introduced fewer false positives than unstructured merge. Furthermore, we argue that semistructured merge added false positives are easier to understand and resolve.

[8] Not finding such an element indicates deletion (instead of renaming), which implies into a true positive, not being useful for our analysis.

Fig. 6. Crosscutting ordering conflicts.

4.2 When compared to unstructured merge, does semistructured merge compromise integration correctness by missing more non spurious conflicts?

To answer RQ2, we compare the number of false negatives added by each merge approach. Our results show that the number of false negatives added by semistructured merge (*aFN(SS)*) is 20.60 ± 21.30% with respect to the total number of reported conflicts, compared to 9.62 ± 16.29% of unstructured merge (*aFN(UN)*). We also observed that 4.42 ± 5.53% of the merge scenarios have at least one false negative added by semistructured merge (*aFN(SS)*), compared with 0.88 ± 1.08% of unstructured merge (*aFN(UN)*). So, considering the aggregated merge scenarios of all projects, semistructured merge introduced more false negatives and has more scenarios with added false negatives. However, we should expect a lower advantage in favor of unstructured merge since *aFN(SS)* is overestimated and *aFN(UN)* is underestimated. Besides, as for RQ1, this does not uniformly hold for each project: *aFN(UN)*>*aFN(SS)* in 18% of the sample projects. Additionally, in only 2 projects (closure-compiler and Essentials), unstructured merge had more merge scenarios with added false negatives (*aFN(UN)*).

Wilcoxon Signed-Rank tests show that there is statistically significant difference when comparing the two approaches, both in terms of merge scenarios and in terms of conflicts (*p-value* equals to, respectively, 4.18e-09 and 5.54e-06<0.05). We also observed a large effect size in terms of merge scenarios (r = 0.8>0.5), and in terms of conflicts (r = 0.57>0.5). This tendency can be observed in the box plots of Figure 7. In the case of merge scenarios with added false negatives (Figure 5(a)), observe that the maximum whisker of the unstructured merge box plots is inferior to the median of the semistructured merge box plot. Besides, in terms of conflicts (Figure 5(b)),

the 3rd quartile in the box plots of the unstructured approach is inferior to the 1st quartile in the box plots of semistructured merge.

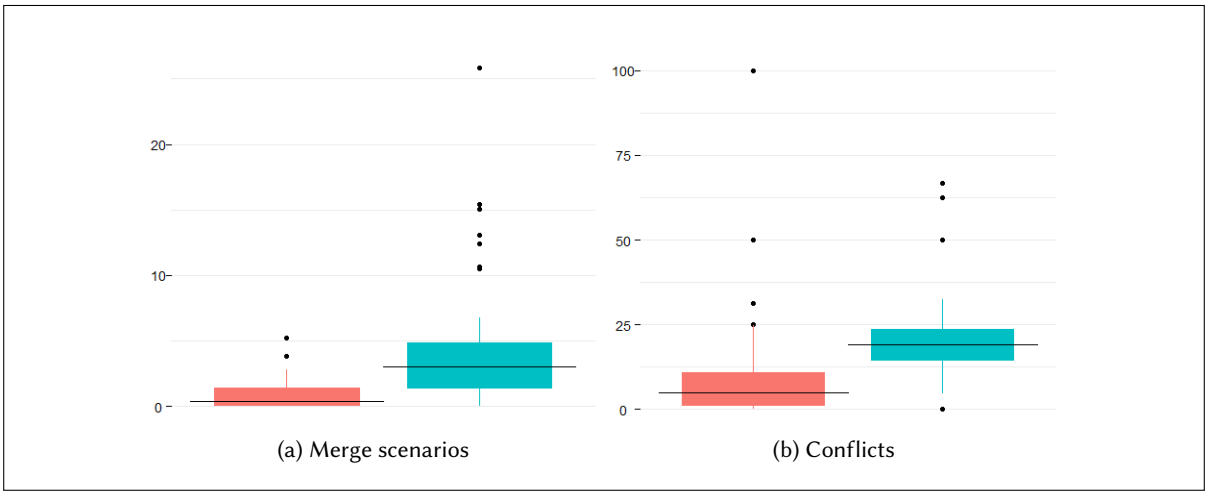

Fig. 7. Box plots describing the percentage, per project, of the added false negatives in terms of merge scenarios and conflicts. Unstructured merge in red, semistructured in blue.

We were expecting a numerical advantage of unstructured merge due to the imprecision of our metric (*aFN(SS)*), but not at the observed level. When distinguishing among the kinds of semistructured merge added false negatives (see Section 2.4), we found that only 0.46% of them are type ambiguity errors, 6.78% are due to initialization blocks, and 92.76% are unstructured merge accidental conflict detection. So, to understand how high our upper bound could be, we manually analyzed 50 randomly selected possible semistructured merge added false negatives to check if they indeed represent missed interference. From the 50 analyzed conflicts,[9] only 6 were confirmed false negatives. Among these, consider the conflict illustrated in Figure 8(a), where both developers added parameters to the same method; as the developers might not be expecting the extra parameter, the conflict is appropriate since this will likely affect the build. Semistructured merge is unable to detect this conflict because the method signature was changed; it assumes both developers deleted the original method and added two new methods with different signatures. Contrasting, Figure 8(b) illustrates a correctly non reported conflict (true negative) by semistructured merge. In this case, Developer A added a comment to the `getTable` declaration, while Developer B added an access modifier. These changes clearly do not represent interference. Besides, as they correspond to different parts of the same node representing the `getTable` declaration, semistructured merge does not report conflict. Uunstructured merge does report an spurious conflict because the changes occurred in the same text area.

4.2.1 Semistructured Merge Added False Negatives Harder To Detect an Resolve. When comparing false negatives, at first thought, the reasoning seems straightforward because the greater the number of false negatives, the greater the number of post-merge build and behavioral errors. Therefore, the weaker the correctness guarantees of the merging process. In that sense, the achieved results suggest that unstructured merge beats semistructured merge for most, but not all, scenarios and projects. However, we cannot ignore that some bugs are more critical than others, and that build problems can be automatically detected, while behavioral problems are often hard

[9]The analyzed conflicts are in the online appendix.

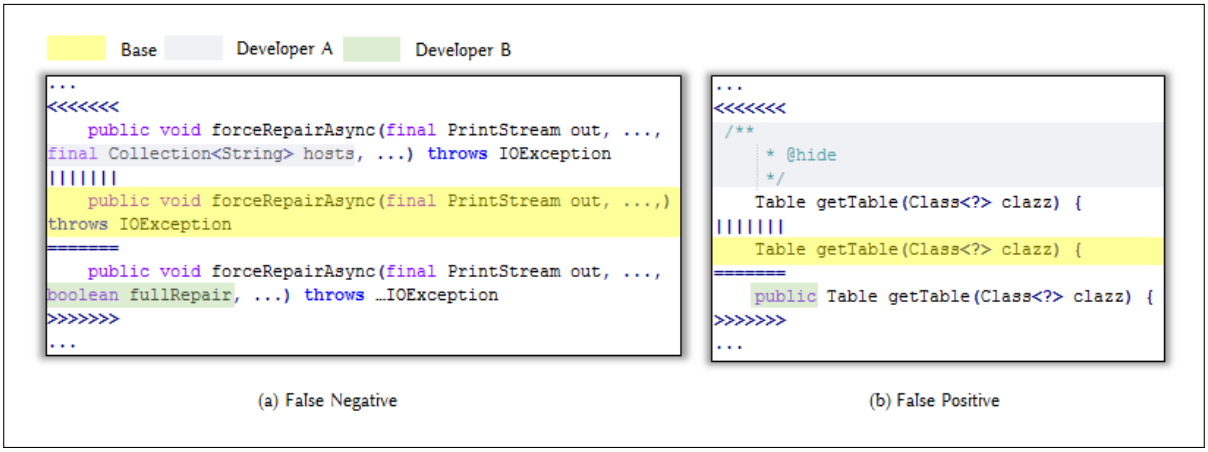

Fig. 8. Unstructured merge conflicts classified as semistructured merge added false negatives.

to detect. In particular, the false negatives added by unstructured merge cause compilation errors, guiding the developers toward the location and cause of the problem. Conversely, the added false negatives by semistructured merge might involve subtle errors. For instance, when one developer adds a new call to a method edited by the other developer, there is no compilation error, but there might be a behavioral issue. That is, the changes made by one developer might affect the behavior expected by the other. We believe that in such cases the detection and resolution of the issue is likely more difficult and demands more effort.

> In our sample, semistructured merge introduced more false negatives, in most but not all projects and scenarios. Moreover, we argue that semistructured merge false negatives are harder to detect and resolve.

5 THREATS TO VALIDITY

Our empirical analyses and evaluation naturally leave open a set of potential threats to validity, which we explain in this section.

Construct Validity. Our analysis of integration effort is based on the number of false positives reported by the merge approaches, the nature of renaming and crosscuting conflicts, and how contributions were integrated in the project repositories. Further analysis involving integrators would be important.

As our metrics are approximations, one can argue that we perhaps could not compare them properly, but the achieved results allowed us to make useful comparisons, especially in the case of false positives, and to a lesser extent in the case of false negatives, where the observed difference is not substantially large. The main issue is related to the semistructured merge added false negatives metric, because the upper bound is too high. To confirm that, we manually inspected a small sample of merged code. As a consequence of the approximations we use, the obtained percentages should not be interpreted as the expected percentages of added false positives and false negatives, but only as sufficient evidence to support the expected advantage of using semistructured merge or unstructured merge in most scenarios and projects.

Internal Validity. A potential threat to the internal (and external) validity of our study is our approach to collect merge scenarios. We analyze public Git repositories, which supports commands such as `rebase` and `cherry-pick` that rewrite the development history. Consequently, depending on the development practices of

each project, we may have lost merge scenarios where developers had to deal with merge conflicts, but that do not appear on Git history as merge commits. When those commands are used in a systematic way it dramatically decreases the number of merge commits. Consequently, to analyze all merge scenarios, we would need to have access to developers' individual repositories. Therefore, our sample actually corresponds to part of the conflicts that actually happened in the analyzed projects. The impact of an increased sample on the results presented here is hard to predict, but we are not aware of factors that could make the missed conflicts different from the ones we analyzed.

Additionally, we had to discard Java files that could not be parsed by the semistructured tool used in the study. So one could argue that we bias the results in favor of the semistructured merge approach because we actually miss the false positives and false negatives present in the discarded files. However, we found that this corresponds only to 0.16% of the total number of Java files. Therefore, we believe that such issue has insignificant impact on our results. We also discarded non Java files from the analyzed projects, which corresponds to 15.33 ± 19.46% of all the files in the analyzed projects. Nevertheless, in both cases, as semistructured merge invokes the unstructured merge tool to handle methods body, it could do the same to merge these files. As a consequence, the approaches would behave identically (and therefore present the same numbers).

External Validity. Although the semistructured merge tool used in this study supports more languages, we restricted our sample to Java projects because our setup demands language dependent tool implementation and configuration. However, all the false positives and false negatives analyzed here are also likely to happen in projects written in other class based languages besides Java. Besides, although our sample has a considerable degree of diversity (see Section 3.1), it would be important to systematically approach that in further studies, including new dimensions such as programming languages. Finally, we only explored open-source projects, but we are not aware of factors that could make conflicts present in projects of different nature unlike the ones we analyzed.

6 AN IMPROVED SEMISTRUCTURED MERGE TOOL

Even considering that our comparison process favors unstructured merge whenever we are not able to precisely classify a conflict, our findings about conflicts and false positives reduction are hardly sufficient to justify adoption of semistructured merge in practice. Practitioners might still be reluctant to adopt semistructured merge because of the risk of loss in part of the merge scenarios, and also because of the extra risk and complexity associated to its false negatives. In fact, if renaming is a common practice in a project, developers might have to deal with too many spurious renaming conflicts. If changes often occur in the same text area, conflicts might escape the merging process. Our findings, nevertheless, shed light on how merge tools can be improved. They help us to better understand the technical justification that might prevent the adoption of semistructured merge tools in practice, and motivates us to propose an improved tool. So we benefit from that and propose a merge tool that further combines both merge approaches to reduce the false positives and false negatives of semistructured merge.

With that aim, our improved merge tool[10] implements the algorithms underlying the scripts we used to detect false positives and false negatives (see Section 3.2) in our empirical analysis. On top of the `FSTMerge` tool (Apel et al. 2011), we add a module (or *handler*) for semistructured merge added false positives and three kinds of added false negatives. After the merged tree is constructed, each handler updates the tree according to specific analyses based on information gathered during tree construction. In particular, the *renaming handler* first uses the Levenshtein distance algorithm to map renamed elements and to fill renaming conflicts with information of the mapped element. Afterwards, the handler looks for references to the edited element with original name. This way we can detect cases that are certainly not false positives, in the presence of references. If the handler

[10]Publicly available at https://goo.gl/jyI8pa.

does not find references, suggesting that the renaming conflict is a false positive, the conflict is only reported if unstructured merge has reported similar conflict. This is a major concern for the design of our improved tool: ensuring that, whenever possible, it is not worse than an unstructured merge tool, by invoking unstructured merge where the underlying algorithms are not precise. This ensures that our improved tool eliminates all added false positives of the original tool.

To reduce false negatives, the *type ambiguity error handler* uses compiler features to search for compilation problems related to import statements, avoiding all extra false negatives of this kind. The *new element referencing edited one handler* checks whether added elements directly reference edited ones. If that is the case, and if unstructured merge also reports a conflict, our tool also reports a conflict, eliminating a possible false negative, but making sure it is not adding a false positive in relation to unstructured merge. The *initialization blocks handler*, uses the Levenshtein distance algorithm described above to match elements of this type, and invokes unstructured merge to integrate the matched elements, reporting the conflicts it reports. In case there is no matching among initialization blocks, no conflict is reported.

The tool is universally applicable by simply calling unstructured merge for files it cannot process (invalid Java files or non-Java files). This way, the tool can be used wherever unstructured merge is used. Annotating other languages grammars, and implementing specific false positive and false negative handlers for these languages would make semistructured merge benefits more widely applicable. Finally, our tool can be configured to resolve false positives due to code indentation: it compares elements content ignoring spacings.

Based on the sample and results of our empirical evaluation, Table 1 summarizes how the improved tool compare to unstructured merge and the original semistructure merge tool. Considering the aggregated scenarios of all projects, the improved tool reduces the number of reported conflicts in approximately 51% in relation to unstructured merge, and in 36% compared to the original semistructured tool; exploring another viewpoint, the table shows increasing rates. A similar reduction pattern, but with less intensity, can be observed for the number of merge scenarios with conflicts; the table shows the percentages in relation to the total number of scenarios followed by the increasing rates. We can also see that the improved tool completely eliminates semistructured merge added false positives. It also eliminates a few kinds of false negatives, leading to a reduction of approximately 24% in the number of added false negatives in relation to the original tool. This way, the improved tool is superior to unstructured merge with respected to the overall number of added false negatives: it misses at least 8% less false negatives than unstructured merge. This does not uniformly hold across projects, but most projects follow this pattern.

Table 1. Comparing unstructured, semistructured and improved tools. Arrows indicate whether the number is underestimated (↑, meaning the numbers should be bigger in practice) or overestimated (↓).

	Unstructured tool	**Semistructured tool**	**Improved tool**
Reported Conflicts	19,238 (206%)	14,544 (156%)	9,343 (100%)
Merge Scenarios with Conflicts	2,995 (8.8% / 155%)	2,420 (7.1% / 125%)	1,935 (5.7% / 100%)
Added False Positives	7,958 ↑	5,201 ↓	-
Added False Negatives	2,714 ↑	3,260 ↓	2,489 ↓

Regarding usability of the new tool, after installation, it is entirely integrated with git version control system (integration with other VCS is on the way); every time a user calls the `git merge` command, the tool is automatically invoked, generating results in the same format as `git merge`. The tool can also be used standalone, likewise available *diff3* tools. Regarding performance, we evaluated the improved tool on a subsample of 1731

merge scenarios from 25 projects. Every scenario in this sample has at least one reported conflict regardless of the merge approach. For each merge scenario, we invoked both the improved tool and unstructured merge (we did not include the original tool in the comparison due to severe performance issues it has due to its prototype nature), 5 times each, measuring execution time. We conducted the evaluation on a desktop machine (Intel Core i5, with 4 cores @4.0 GHz, and 16 GB RAM) with Windows 10 64 bits. Taking the median of the measured times, the improved tool took approximately 24 minutes to merge the entire sample, compared to only 45 seconds of the unstructured merge tool. This large difference could be reduced by an industrial strength implementation of our tool. In fact, our implementation could be optimized in a number of ways. For example, we explore no parallelization, and merge files sequentially. However, due to the handlers and complexity of the tree merging algorithm we use, we expect an optimized tool would still be much slower than unstructured merge. But we also observed that, in practice, this performance difference is often non prohibitive. In more than 80% of the scenarios, our tool took less than 1 second to merge the involved files. It took more than 5 seconds in only 35 scenarios, with a maximum of 67 seconds in a lucene-solr scenario that merges 303 files, resulting in 618 conflicts and 17,567 KLOC of conflicting code. For the same scenario, unstructured merge took 6 seconds, resulting in 866 conflicts and 27,397 KLOC of conflicting code. In our sample both approaches spent, on average, less than 1 second per merge scenario (0,83 ± 2,47 seconds with the improved tool, compared to 0,03 ± 0,09 seconds with unstructured merge).

7 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 task integration. For instance, Apel et al. (2011) propose and evaluate `FSTMerge`, the semistructured merge tool used here. We confirm previous evidence (Apel et al. 2011; Cavalcanti et al. 2015) that `FSTMerge` might reduce, but not for all projects and scenarios, the number of reported conflicts. However these studies do not investigate whether the obtained reduction is achieved at the expense of extra false negatives, or new kinds of false positives that are harder to resolve. In fact the set of conflicts reported by `FSTMerge` is not a subset of the conflicts reported by unstructured merge. Here we go further by analyzing the relatively added false positives and false negatives of each tool. We also propose an improved tool, which is essential for justifying industrial adoption of more advanced merge tools. Whereas our improved semistructured merge tool implements a basic syntactic-based renaming handler to detect renamings, a more advanced semantic-based refactoring detection module (Dig et al. 2006) could further improve precision. Meanwhile, we avoid such renaming false positives in the improved tool we propose by using unstructured merge result whenever they differ.

Structured and semantic merge approaches have also been proposed. Westfechtel (1991) and Buffenbarger (1995) propose tools that incorporate extra structural information, such as the context-free and context-sensitive syntax, in the merging process. Researchers have also proposed a wide variety of structural comparison and merge tools including tools specific to Java (Apiwattanapong et al. 2007) and C++ (Grass 1992). Apel et al. (2012) also propose a tool that tunes the merging process on-line by switching between unstructured and structured merge, depending on the presence of conflicts. Some tools even use additional language semantic information (Binkley et al. 1995), or require a formal semantics for the documents to be merged (Berzins 1994; Jackson and Ladd 1994). Nguyen (2006) proposes `Molhado`, a structure versioning framework that facilitates the construction of structure-oriented difference tools for various types of software artifacts. Dig et al. (2008) uses `Molhado` to build `MolhadoRef`, a operation-based approach that records change operations (refactorings and edits) used to produce one version and replays them when merging versions. However, the semantic and structured techniques are quite expensive and not practical yet.

Regarding the concept of integration effort, Prudêncio et al. (2012) suggest that it can be measured as the number of extra actions (additions, deletions or modifications) that a developer has to perform during code

integration to conciliate the changes made in the contributions to be merged. Furthermore, Santos and Murta (2012) correlate the number of conflicts to that metric, suggesting that conflict reduction imply effort reduction. We opted for a qualitative analysis because this metric only estimates the fraction of the time taken by a developer to edit code, not taking into account the time that the developer took reasoning about how to resolve the conflict. This way, this editing time amounts to only part (perhaps the minor part) of the total integration effort time. Kasi and Sarma (2013) measure integration effort based on the number of days that the conflict persisted in a project repository. However, they assume that, during this period, the developers exclusively worked to resolve that conflict. As the precise fixing period might be hard to find, and we believe that is often not the case that developers exclusively work to resolve conflicts when they happen, we opted for a qualitative analysis. In our study we have analyzed conflicts that do not represent interference between development tasks (false positives), and, as such, resolving them is an unnecessary integration effort. So, implementing mechanisms to automate the resolution of such conflicts, as in the proposed new tool, might decrease integration effort.

Finally, other empirical studies provide evidence about the frequency and impact of conflicts, and their associated causes. For example, Kasi and Sarma (2013), and Brun et al. (2011) reproduce merge scenarios from different GitHub projects with the purpose of measuring the frequency of merge scenarios that resulted in conflicts. Moreover, Zimmermann (2007) does a similar analysis, but with a different metric, since the author reproduces file integration from CVS projects. They all conclude that conflicts are frequent. Our work complements these studies by bringing evidence about the frequency of integrations that had certain types of false positives and false negatives, meaning how often integrations demands unnecessary integration effort and had interference undetected by unstructured and semistructured merge tools. In addition, Kasi and Sarma (2013), and Brun et al. (2011) also study the frequency of merge scenarios that had build or test failures, which can be seen as a consequence of the false negatives in the merging process. In this respect, we have explored specific types of false negatives that cause build or test failures.

8 CONCLUSIONS

Previous studies provide evidence that semistructured merge reduces, for most but not all projects and scenarios, the number of conflicts in relation to unstructured merge (Apel et al. 2011; Cavalcanti et al. 2015). However, practitioners would hardly adopt a new merge tool without knowing whether this reduction actually leads to merge effort reduction without compromising the correctness of the merging process. So, in this paper, by reproducing 34,030 merges from 50 Java projects, we relatively compare these merge approaches with respect to the resulting occurrences of false positives and false negatives. In particular, false positives represent unnecessary integration effort, which decreases productivity, because developers have to resolve conflicts that actually do not represent interference. Besides that, false negatives represent build or behavioral errors, negatively impacting software quality and correctness of the merging process. For most merges and projects, we observed that semistructured merge not only reduces the number of reported conflicts, but also introduces fewer false positives when compared to unstructured merge. Furthermore, we find evidence that semistructured merge added false positives are easier to analyze and resolve than those reported by unstructured merge. However, we found that semistructured merge introduced more false negatives than unstructured merge, and we argue that they are harder to detect and resolve. Driven by these findings, we propose an improved semistructured merge tool that further combines both approaches to reduce the false positives and false negatives of semistructured merge. We find evidence that the improved tool, when compared to unstructured merge, reduces the number of reported conflicts by half, has no extra false positives, has at least 8% less false negatives, is not prohibitively slower, and presents no extra usability barriers in relation to state of the art merge tools.

REFERENCES

Sven Apel and Christian Lengauer. 2008. Superimposition: A Language-independent Approach to Software Composition. In *Proceedings of the 7th International Conference on Software Composition (SC'08)*. Springer-Verlag.

Sven Apel, Olaf Lessenich, and Christian Lengauer. 2012. Structured Merge with Auto-tuning: Balancing Precision and Performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*. ACM.

Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM.

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2007. JDiff: A Differencing Technique and Tool for Object-oriented Programs. *Automated Software Engineering* (2007).

Online Appendix. 2017. Hosted on http://goo.gl/u3IBVP. (2017).

Valdis Berzins. 1986. On merging software extensions. *Acta Informatica* (1986).

Valdis Berzins. 1994. Software Merge: Semantics of Combining Changes to Programs. *ACM Trans. Program. Lang. Syst.* (1994).

David Binkley, Susan Horwitz, and Thomas Reps. 1995. Program Integration for Languages with Procedure Calls. *ACM Transactions on Software Engineering and Methodology* (1995).

Christian Bird and Thomas Zimmermann. 2012. Assessing the Value of Branches with What-if Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*. ACM.

Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM.

Jim Buffenbarger. 1995. Syntactic Software Merging. In *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*. Springer-Verlag.

Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. 2015. Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment. In *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement (ESEM'15)*. ACM.

Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Springer-Verlag.

Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. 2008. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Transactions of Software Engineering* (2008).

Judith E. Grass. 1992. Cdiff: A Syntax Directed Differencer for C++ Programs. In *Proceedings of the USENIX C++ Conference*. USENIX Association.

Daniel Jackson and David A. Ladd. 1994. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In *Proceedings of the International Conference on Software Maintenance (ICSM'94)*. IEEE.

Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. IEEE.

Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. 2007. A Formal Investigation of Diff3. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*. Springer-Verlag.

Gleiph Menezes. 2016. *On the Nature of Software Merge Conflicts*. Ph.D. Dissertation. Federal Fluminense University.

T. Mens. 2002. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering* (2002).

Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in Software Engineering Research. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. ACM.

T. N. Nguyen. 2006. Object-Oriented Software Configuration Management. In *Proceedings of the 22th International Conference on Software Maintenance (ICSM'06)*. IEEE.

João Gustavo Prudêncio, Leonardo Murta, Cláudia Werner, and Rafael Cepêda. 2012. To Lock, or Not to Lock: That is the Question. *Journal of Systems and Software* (2012).

Rafael de Souza Santos and Leonardo Gresta Paulino Murta. 2012. Evaluating the Branch Merging Effort in Version Control Systems. In *Proceedings of the 26th Brazilian Symposium on Software Engineering (SBES'12)*. IEEE Computer Society.

Bernhard Westfechtel. 1991. Structure-oriented Merging of Revisions of Software Documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management (SCM'91)*. ACM.

Thomas Zimmermann. 2007. Mining Workspace Updates in CVS. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR'07)*. IEEE.