Evaluating and Improving Semistructured Merge

GUILHERME CAVALCANTI, Federal University of Pernambuco, Brazil
PAULO BORBA, Federal University of Pernambuco, Brazil
PAOLA ACCIOLY, Federal University of Pernambuco, Brazil

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 that, 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 conflicts incorrectly reported by one approach but not by the other (false positives), and conflicts correctly reported by one approach but missed by the other (false negatives). Our results and complementary analysis indicate that, in the studied sample, the number of false positives is significantly reduced when using semistructured merge. We also find evidence that its false positives are easier to analyze and resolve than those reported by unstructured merge. However, we find no evidence that semistructured merge leads to fewer false negatives, and we argue that they are harder to detect and resolve than unstructured merge false negatives. Driven by these findings, we implement 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 in our sample, reduces the number of reported conflicts by half, has no additional false positives, has at least 8% fewer 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

ACM Reference Format:
Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (October 2017), 27 pages. https://doi.org/10.1145/3133883

1 INTRODUCTION

In a collaborative software development environment, developers often independently perform tasks, 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

Authors' addresses: Guilherme Cavalcanti, Informatics Center, Federal University of Pernambuco, Av. Jorn. Aníbal Fernandes, s/n, Recife, Pernambuco, 50740-560, Brazil, gjcc@cin.ufpe.br; Paulo Borba, Informatics Center, Federal University of Pernambuco, Av. Jorn. Aníbal Fernandes, s/n, Recife, Pernambuco, 50740-560, Brazil, phmb@cin.ufpe.br; Paola Accioly, Informatics Center, Federal University of Pernambuco, Av. Jorn. Aníbal Fernandes, s/n, Recife, Pernambuco, 50740-560, Brazil, prga@cin.ufpe.br.

2475-1421/2017/10-ART59
https://doi.org/10.1145/3133883

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 various approaches to decrease integration effort and improve integration correctness. For example, unstructured merge tools [Khanna et al. 2007], which are widely used in industry, rely on purely textual analysis to detect and resolve conflicts. Alternatively, semistructured merge tools [Apel et al. 2011] go further by partially exploiting the syntactic structure and semantics of the involved artifacts. For program elements whose structure is not exploited, such as method bodies,semistructured merge tools simply apply the usual textual resolution adopted by unstructured merge.

Previous studies [Apel et al. 2011; Cavalcanti et al. 2015] compare these two merge approaches with respect to the number of reported conflicts, showing, for most but not all projects and merge situations, reduction in favor of semistructured merge. This reduction is mainly due to the automatic resolution of obvious unstructured merge false positives that are reported when, for example, developers add different and independent 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, we find a larger average reduction of 62%, again in favor of semistructured merge [Cavalcanti et al. 2015].

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 reduction of reported conflicts actually leads to integration effort reduction (productivity) without negative impact on the correctness of the merging process (quality). Although one might expect only accuracy benefits from the extra structure exploited by semistructured merge, we have no guarantees that this is the case. So the observed reduction could have been obtained at the expense of missing actual conflicts between developers changes (false negatives). In that case, semistructured merge users would be simply postponing conflict detection to other integration phases such as building and testing, or even letting more 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 each merge approach.

So, to better compare these two merge approaches and understand how merge tools could be improved, in this paper we reproduce 34,030 merges from 50 GitHub Java projects. We collect evidence about conflicts incorrectly reported by one approach but not by the other (false positives), and conflicts correctly reported by one approach but missed by the other (false negatives). 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 false positives?*
- **RQ2** *When compared to unstructured merge, does semistructured merge compromise integration correctness by having more false negatives?*

In summary, in our sample, our results show that the number of false positives is significantly reduced when using semistructured merge, and we find evidence that its false positives are easier to analyze and resolve than those reported by unstructured merge. On the other hand, we find no evidence that semistructured merge leads to fewer 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 reported conflict, this last finding, and the associated lack of evidence in support of semistructured merge, 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 implement 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 in our sample, reduces the number of reported conflicts by half, has no additional false positives, has at least 8% fewer 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** based on the *diff3* algorithm [Mens 2002]. 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, such merge tools check whether the three revisions have a common text area that separates the chunks contents. If such separator is not found, the tool reports a conflict [Khanna et al. 2007].

In contrast, **semistructured merge** [Apel et al. 2011] exploits 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. Representing all language elements as nodes, as in structured merge [Apel et al. 2012], can considerably 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 program that integrates both contributions. Under this criteria, semistructured merge would be superior because it often reports fewer conflicts, as shown 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 do not 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. Consequently, we adopt a broader notion of conflict. We rely on the notion of interference defined by Horwitz et al. [1989]— two contributions (changes) to a base program interfere when the specifications they are individually supposed to satisfy are not jointly satisfied by the program that integrates them; this often happens when there is, in the integrated program, data or control flow

[1]From now on, we use method declarations to refer both to method and constructor declarations.

between the contributions. We then say that two contributions to a base program are conflicting when there is no valid program that integrates them and has no unplanned interference.

The challenge associated to such a more comprehensive comparison criteria is establishing ground truth for integration conflicts (and therefore false positives and false negatives) between development tasks, as this is not computable in this context [Berzins 1986; Horwitz et al. 1989]. Semantic approximations through static analysis or testing are imprecise and often too expensive, especially 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 misjudgement. 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 occurrence of false positives and false negatives of one approach *in addition* to the ones of the other. We do that by simply analyzing when the merge approaches report different results for the same merge scenario—each scenario comprehends the three revisions involved in a three-way merge. We identify conflicts reported by one approach but not by the other (false positives), and conflicts reported by one approach but missed by the other (false negatives).

To better understand this notion of *additional* false positives and false negatives, consider the diagram in Figure 1. We illustrate the set of conflicts (positives) reported by unstructured ($P(UN)$) and semistructured ($P(SS)$) merge. Unstructured merge's set (in yellow) includes its false positives, represented in the bottom part of the yellow circle. This is the union of the false positives reported by both approaches ($FP(UN|SS)$) with the false positives reported only by unstructured merge ($aFP(UN)$). So we say $aFP(UN)$ is the set of additional false positives of unstructured merge. Semistructured merge's set (in green) includes conflicts detected by this approach but missed by unstructured merge ($aFN(UN)$); these are the additional false negatives of unstructured merge. Similarly, semistructured merge's set includes its additional false positives ($aFP(SS)$), and unstructured merge's set includes conflicts detected by this approach but missed by semistructured merge ($aFN(SS)$). Finally, the sets also include true positives (actual conflicts) common to both approaches ($TP(UN|SS)$). For simplicity, we do not name the sets of true and false negatives common to both approaches. As our comparison is relative, the common cases do not interest us.

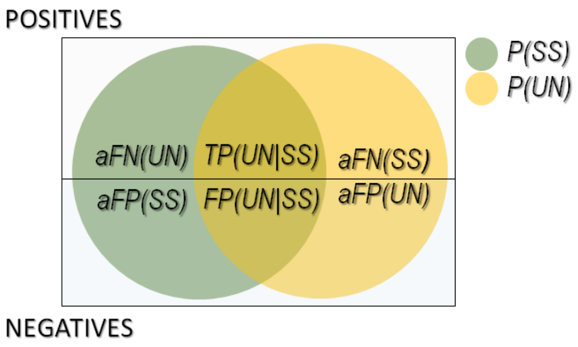
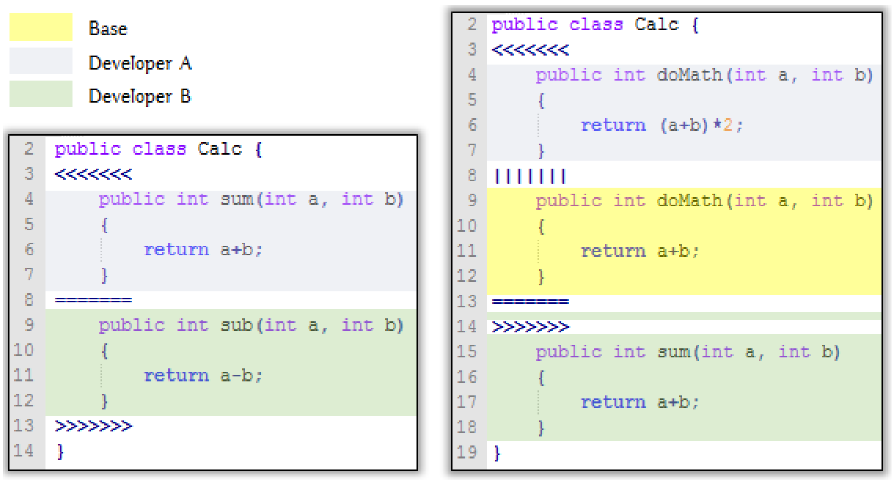


Fig. 1. Sets of conflicts reported by unstructured and semistructured merge. Notation explained in the text.

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 supporting Java 5, 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



```
public class Calc {
<<<<<<<
    public int sum(int a, int b)
    {
        return a+b;
    }
=======
    public int sub(int a, int b)
    {
        return a-b;
    }
>>>>>>>
}
```

(a) Ordering conflict
(unstructured merge)

```
public class Calc {
<<<<<<<
    public int doMath(int a, int b)
    {
        return (a+b)*2;
    }
|||||||
    public int doMath(int a, int b)
    {
        return a+b;
    }
=======
>>>>>>>
    public int sum(int a, int b)
    {
        return a+b;
    }
}
```

(b) Renaming conflict
(semistructured merge)

Fig. 2. Unstructured and semistructured merge additional false positives. Conflict markers in blue.

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 when they behave differently, and how this might lead to additional false positives and false negatives. In particular, for each conflict reported by unstructured merge, we checked whether semistructured merge also reported that 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 additional 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 Additional False Positives of Unstructured Merge

One of the main weaknesses of unstructured merge is its inability to detect re-arrangeable 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 2(a) illustrates this situation: a reported conflict caused by different developers adding 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. By exploiting knowledge about Java syntax and static semantics, semistructured merge identifies commutative and associative declarations, and understands that the changes to be merged can be integrated because they are related to different nodes. As discussed later, not all ordering conflicts are (additional) false positives of unstructured merge. For example, import declarations are often, but not always, re-arrangeable.

2.2 Additional False Positives of Semistructured Merge

Renamings challenge semistructured merge, as illustrated by Figure 2(b), which shows a false positive *renaming conflict*: one of the developers renamed the `doMath` method to `sum`, whereas the

[2]http://kdiff3.sourceforge.net/

other developer kept the original signature but edited the method body. Semistructured merge reports this as a conflict because its algorithm interprets method renaming as method deletion, consequently assuming that a developer deleted the method (doMath in this case) 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 its formal parameters. So, when an element is renamed, the merge algorithm is not able to map the base element to the newly named element 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 of both contributions; there is no conflict 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 unstructured merge does. 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. If this character was in line 9, unstructured merge would also report a conflict, and this would be a common false positive. This helps to explain why a renaming conflict involving a field declaration is an additional false positive only when the declaration is split into multiples lines of code.

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

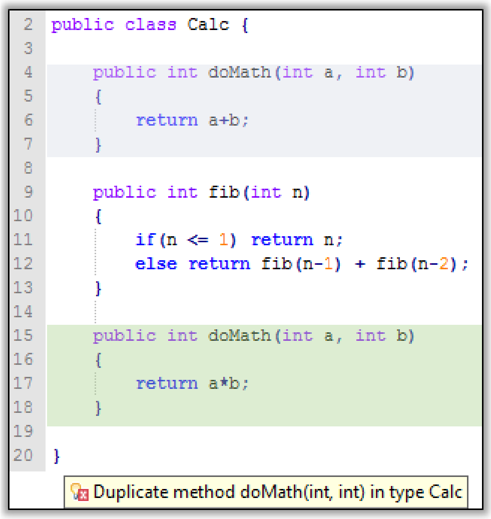
2.3 Additional False Negatives of Unstructured Merge

The additional false negatives of unstructured merge 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*. Figure 3(a) illustrates this situation: both developers added methods with the same signature but with different bodies, in clearly separated areas, not leading to an unstructured merge conflict. As semistructured merge matches elements to be merged by their type and identifier, it would detect the problem and correctly report a conflict.

Besides that, renaming conflicts might also be additional false negatives of unstructured merge. As explained at the end of 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 an example similar to the one in Figure 2(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 developers changes, unstructured merge would erroneously not report a conflict.

2.4 Additional False Negatives of Semistructured Merge

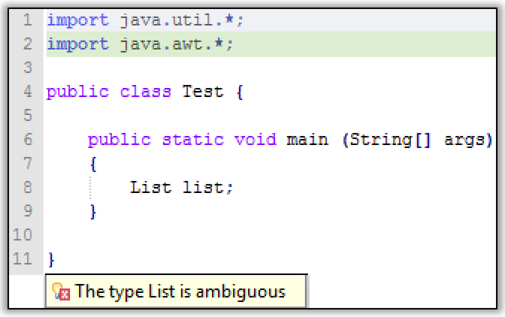
As mentioned in Section 2.1, ordering conflicts involving import declarations are often, but not always, false positives. In fact, merging developers contributions that add import declarations to



```java
public class Calc {

    public int doMath(int a, int b)
    {
        return a+b;
    }

    public int fib(int n)
    {
        if(n <= 1) return n;
        else return fib(n-1) + fib(n-2);
    }

    public int doMath(int a, int b)
    {
        return a*b;
    }

}
```

Duplicate method doMath(int, int) in type Calc

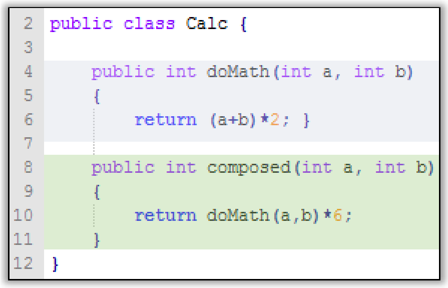
(a) Duplicated Declaration Error (unstructured merge)



```java
import java.util.*;
import java.awt.*;

public class Test {

    public static void main (String[] args)
    {
        List list;
    }

}
```

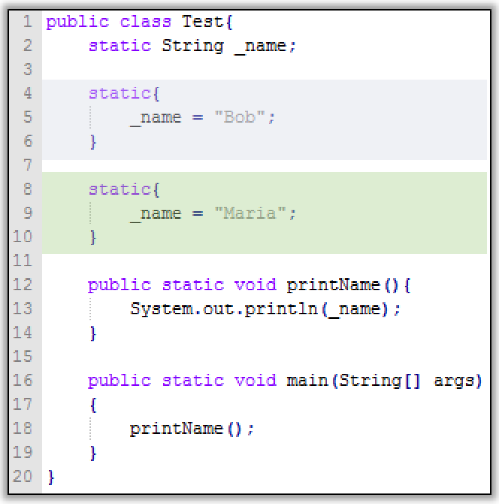
The type List is ambiguous

(b) Type Ambiguity Error (semistructured merge)



```java
public class Calc {

    public int doMath(int a, int b)
    {
        return (a+b)*2; }

    public int composed(int a, int b)
    {
        return doMath(a,b)*6;
    }
}
```

(c) New element referencing edited one (semistructured merge)



```java
public class Test{
    static String _name;

    static{
        _name = "Bob";
    }

    static{
        _name = "Maria";
    }

    public static void printName(){
        System.out.println(_name);
    }

    public static void main(String[] args)
    {
        printName();
    }
}
```

(d) Initialization Blocks (semistructured merge)

Fig. 3. Unstructured and semistructured merge additional false negatives.

the same text area might lead to additional false negatives of semistructured merge, as it assumes that such declarations are always re-arrangeable. For example, this might lead to a *type ambiguity error* (see Figure 3(b)) when the import declarations involve members with the same name but from 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, unstructured merge correctly reports a conflict, whereas semistructured merge lets 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 behaviors, the presence of both declarations 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 the issue with import statements, semistructured merge has additional false negatives due to the way it handles *initialization blocks*. Since it uses elements types and identifiers to match nodes, the algorithm is unable to match nodes without identifiers. This leads to problems like the one illustrated in Figure 3(d); as the two independently added initialization elements have no identifier, semistructured merge cannot match them, and therefore keeps both. A conflict should have been reported so that developers could negotiate how the class field should be initialized. In case the initialization blocks are added to the same text area, unstructured merge reports a conflict.

Semistructured merge has also other kinds of additional false negatives. Although they do not conform to a small set of recurring syntactic patterns, 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 declarations. In particular, we observed that when one developer added a new declaration that references an existing one edited by the other developer. In such cases, the developer who added the new declaration might not be expecting the changes made to the referenced one, possibly leading to negative impact on the merged program behavior. This is illustrated in Figure 3(c), where the new method `composed` references the `doMath` method, which was changed by the other developer. Although one might assume that methods provide the lowest level of information hiding and modularity (with 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 this situation as a semistructured merge additional false negative. In fact, as the changes correspond to different elements (technically, different nodes in the semistructured merge tree), semistructured merge reports no conflict. In contrast, unstructured merge might accidentally detect such conflicts when the changes are 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 relatively— not to establish how accurate they are in relation to a general notion of conflict— 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 body, 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 harmful 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 do not 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 additional 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 false positives?*
- **RQ2** *When compared to unstructured merge, does semistructured merge compromise integration correctness by having more false negatives?*

To answer **RQ1**, we compute the *overestimated number of additional false positives of semistructured merge— aFP(SS)* (false positives reported by semistructured merge and not reported by unstructured merge) metric. We also compute the *underestimated number of additional false positives of unstructured merge— aFP(UN)* (false positives 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, our plan is to evaluate the merge approaches by computing the number of diverging false positives and false negatives of each approach. However, precisely computing that is hard, as discussed later in this section. Nevertheless, if we find that an upper bound is inferior to a lower bound, 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 reported 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 required to resolve different kinds of false positives, 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 additional false negatives of semistructured merge— aFN(SS)* (conflicts missed by semistructured merge and correctly reported by unstructured merge), and the *underestimated number of additional false negatives of unstructured merge— aFN(UN)* (conflicts missed by unstructured merge and correctly reported by semistructured merge) metrics. We also discuss the integration effort 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 is composed by 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 scenarios and to find potential additional false positives and false negatives. In the *analysis* step, we confirm the occurrence of additional false positives and negatives. Finally, we use R scripts to analyze the results. We now detail these steps, jointly explaining the execution and analysis steps for simplicity.

3.1 Mining Step

To select 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 a certain 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 used tools to mine their GitHub repositories and collect merge scenarios from their full histories. In particular, we used 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

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

database and retrieve a list of the identifiers of all merge commits— commits having a true value for their `isMerge` attribute— 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. As the compared tools could be easily adapted to behave identically for non-Java files, this is not a major problem. Moreover, the number of non-Java files not merged corresponds to 1.73% of the total number of files in the sample (we discuss this threat later in Section 5).

Although we have not systematically targeted representativeness or even diversity [Nagappan et al. 2013], 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, a 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 includes projects such as cassandra, 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 [Cavalcanti et al. 2017].

3.2 Execution and Analysis Steps

After collecting the sample projects and merge scenarios, we use the `KDiff3` unstructured tool, and the `FSTMerge` semistructured tool (see Section 2) to merge the selected scenarios. We then identify and compare the occurrence of additional false positives and false negatives, as 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 [Cavalcanti et al. 2017], where we also point to the version of `FSTMerge` that contains the interceptors we implemented for our study.

3.2.1 Computing the Overestimated Number of Additional False Positives of Semistructured Merge—aFP(SS). The additional false positives of semistructured merge, as explained in Section 2.2, are due to renamings. To identify these false positives, we first intercept conflicts detected by `FSTMerge`. We then check whether the involved triple of base, left and right elements contains a non-empty base, and an empty left or right element. Not having the left or right version, represented by the red empty string in Figure 4, indicates that the semistructured merge algorithm could not map the element (method in this case) to its previous version. This happens when 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. For simplicity, we also conservatively assume that unstructured merge did not report the same conflict, in case there were no separators between the changes.

3.2.2 Computing the Underestimated Number of Additional False Negatives of Unstructured Merge—aFN(UN). The main pattern of unstructured merge additional false negatives occurs, as explained

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

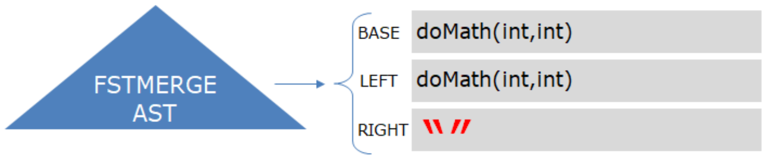


Fig. 4. Intercepting `FSTMerge` tool to find *renaming* false positives (*aFP(SS)*) candidates.

in Section 2.3, when developers independently 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 non-empty left and right elements. This indicates the addition of two new elements with the same signature. To confirm that the elements were added to different areas, we merge the files with the unstructured tool and verify that there is no reported conflict that contains the duplicated declaration. In case there is such a conflict, the additional false negative candidate is discarded. If there is no such a conflict, we compile the resulting file and search the compiler output for duplicated declarations errors related to the identified elements.

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

3.2.3 Computing the Overestimated Number of Additional False Negatives of Semistructured Merge— aFN(SS). As explained in Section 2.4, the additional false negatives of 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. We then try to merge the corresponding files with unstructured merge, and check if it reports a conflict involving the pair of imports statements. If it does not report, we have a common true or false negative of both approaches; so no further action is needed. If it does report such a conflict, we further check if the import declarations lead to type ambiguity errors. We do that by compiling the resulting file merged by semistructured merge and searching for type ambiguity compilation errors. We also check if the import statements might lead to behavioral issues: we search for the name of the package member imported by one developer in the changes introduced by the other, and vice versa. This is a grep based analysis, having as search scope the file containing the import statements. With a positive result in one of the checks, we conservatively consider the pair of import statements as a conflict missed by semistructured merge (additional false negative).

To identify the second kind of false negative, we use `FSTMerge` to identify all nodes representing initialization blocks in the different tree versions. Using textual similarity, based on the *Levenshtein distance algorithm* [Levenshtein 1966] with 80% of degree of similarity, we group triples of similar initialization nodes. These triples are then merged with unstructured merge. If they conflict, we conservatively compute the conflicts as additional false negatives. As the same block might have been substantially changed by both developers, they might not satisfy our similarity threshold. This way we could miss false negatives. To make sure this is not the case, we carry on a manual analysis. We discuss this threat in Section 5. Choosing a lower threshold could be too conservative, and not substantially reduce the number of cases to be manually analyzed. An alternative metric could consider the number of edited initialization blocks as the number of additional false negatives of this kind. However, this might not be conservative as changes to initialization blocks might lead to more than one conflict.

All other conflicts reported by unstructured merge but missed by semistructured merge are conservatively classified as additional false negatives, except in the following two cases. In both cases, we are able to parse the text of the unstructured merge conflict and then classify the reported conflict. First, when the reported conflict contains only field declarations that do not reference each other. Such reported conflicts are unstructured false positives instead because there is no dependence among the field declarations. Second, when the conflict resolution keeps all changes from both left and right revisions, and adds no new code. We assume that the developer correctly analyzed the conflict and decided there was no problem (we discuss this threat later in Section 5); so that would be an unstructured false positive, not a semistructured false negative. We check that by parsing and inspecting the original merge commit in the project repository.

More precise analyses, such as those based on 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 Additional False Positives of Unstructured Merge—aFP(UN). The additional false positives of unstructured merge are due to its inability to perceive that some declarations are commutative and associative, and therefore avoid 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. As explained earlier, Figure 1 illustrates the set of conflicts reported by unstructured and semistructured merge. For instance, unstructured merge set includes its additional false positives ($aFP(UN)$). The sets also include true and false positives common to both approaches, denoted by $TP(UN|SS)$ and $FP(UN|SS)$. 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 unstructured merge additional false positives 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. This represents a semistructured merge reduction of approximately 24% in the total number of reported conflicts. As each merge scenario might have conflicts reported by both tools, only one, or none, we observed that the 19,238 conflicts reported by unstructured merge occurred in 8.8% of the sample merge scenarios. Moreover, the 14,544 conflicts reported by semistructured merge occurred in 7.1% of the sample merge scenarios. We also observed that in 54.6% of the sample merge scenarios having at least one conflict, regardless of the tool, semistructured merge reported fewer conflicts than unstructured merge. This is similar to previous studies, differing at most by 5% [Apel et al. 2011; Cavalcanti et al. 2015]. In these scenarios, the observed reduction in the total number of conflicts was of 71%

[6]We formally derive the formula in the online appendix.

± 30% (average ± standard deviation), compared to 62% ± 24% in our previous study [Cavalcanti et al. 2015], and 34% ± 21% in the study of Apel et al. [2011]. This evidence of conflict numbers, however, is not enough for justifying the adoption of a merge tool because of the risk of missing actual conflicts (false negatives), or introducing new kinds of false positives. Considering the merge scenarios having conflicts, we found that in 27.1% of them one approach detected at least one conflict, and the other none. In 17.1% of them the approaches reported the same conflicts, and in 25.9% the approaches detected only different conflicts. So, when they report conflicts, 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 [Cavalcanti et al. 2017].

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

To answer RQ1, we compare the number of additional false positives of 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 additional false positive (*aFP(UN)*). Moreover, 43.47% ± 19.01% of the reported conflicts are additional false positives according to our metric (*aFP(UN)*). This is bigger than the percentage of semistructured merge additional false positives (*aFP(SS)*): 30.21% ± 20.68%. In addition, only 3.12% ± 3.55% of the merge scenarios have at least one additional false positive (*aFP(SS)*). So, considering the aggregated scenarios of all projects, we conclude that semistructured merge has fewer additional false positives and fewer scenarios with additional 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. In contrast, in only 2% of the projects semistructured merge had more merge scenarios with more additional false positives

The large error bounds, and the lack of uniformity of the results across projects, are caused by variations in the analyzed projects. For example, project histories containing renaming of directories might significantly increase the number of renaming conflicts reported by semistructured merge. We observed that in projects such as OG-Platform and Equivalent-Exchange-3, which have a greater number of scenarios containing directory renamings, and consequently show semistructured merge false positive rates substantially above the average. In such cases, unstructured merge reports, for each file of the renamed directory, a single large conflict; it cannot map the files of the renamed directory to the corresponding files of the other revision. Conversely, due to semistructured merge finer granularity, the conflicts are reported per element (method, constructor, etc.) in the files of the renamed directory, substantially increasing the number of reported conflicts. The error bounds are also explained by projects such as AndEngine, mct, and Voldemort, in which most conflicts occur inside method bodies. In these situations, the tools behave identically and report a large number of common conflicts, decreasing the percentage of additional false positives.

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, based on the Pearson Correlation Coefficient). There is also significant difference between the percentages of additional 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.

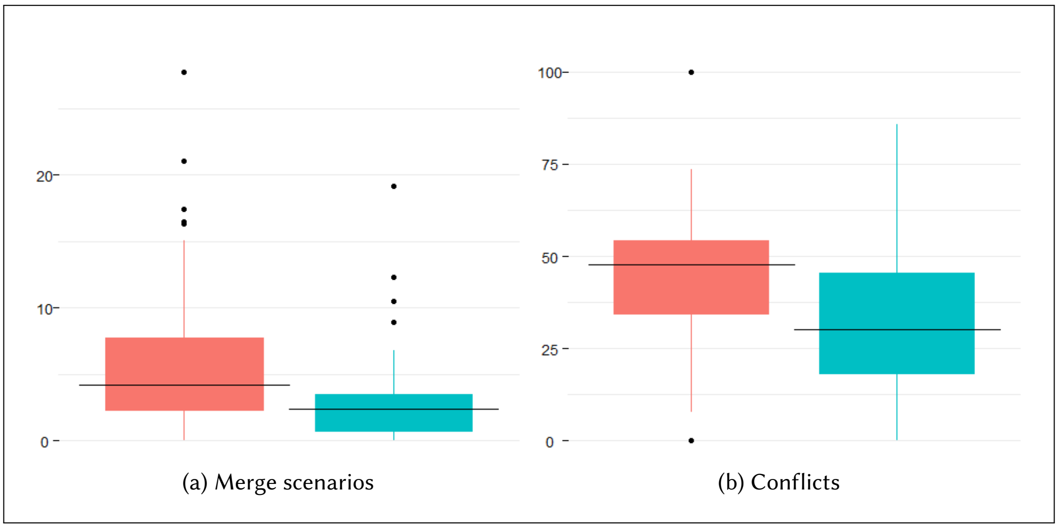


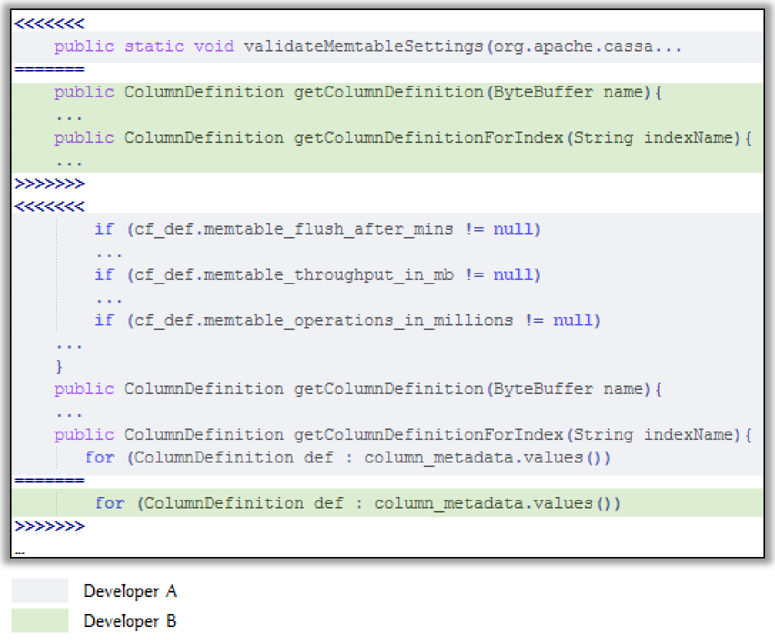
Fig. 5. Box plots describing the percentage, per project, of additional false positives in terms of merge scenarios and conflicts. Unstructured merge in red, semistructured in blue.

4.1.1 *Additional False Positives of Semistructured Merge are Easier to Analyze and Resolve.*

Semistructured merge reduction in the number of additional false positives for most projects and scenarios suggests that it might reduce integration effort. However, 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 manually analyzed 100 randomly selected merge scenarios: 50 containing semistructured merge renaming conflicts (likely false positives), and 50 with unstructured merge ordering conflicts (likely false positives). These are appropriate sample sizes for estimating proportion [Eng 2003] considering margins of error of 10% and 15%, respectively, for renaming and ordering conflicts (we further discuss this in Section 5). For each scenario, we observed the reported conflicts and attempted to resolve them 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 resolutions including only changes from the merged contributions (without new code, nor combination of contributed code) demand less effort. We believe this is a fair approximation of the time needed to fix the code— which is part of the total integration effort— but not of the time needed to reason about the conflict and then decide how to fix it. The manual analysis considers this last part.

The 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 2(b), this kind of reported 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.[7] 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.

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 reported conflicts caused by the introduction of declarations (methods or fields) in the same text area can often be analyzed and resolved with little effort. 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— so we name it a *crosscutting conflict*. Such reported conflicts involve incomplete parts of different language structural elements (methods, fields, etc.). We illustrate this 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 which method contains the `for` and `if` statements. As such kind of conflict involves different syntactic elements (and then different nodes in a parse tree), semistructured merge automatically resolves them.


```
<<<<<<<
    public static void validateMemtableSettings(org.apache.cassa...
=======
    public ColumnDefinition getColumnDefinition(ByteBuffer name){
    ...
    public ColumnDefinition getColumnDefinitionForIndex(String indexName){
    ...
>>>>>>>
<<<<<<<
        if (cf_def.memtable_flush_after_mins != null)
        ...
        if (cf_def.memtable_throughput_in_mb != null)
        ...
        if (cf_def.memtable_operations_in_millions != null)
    ...
    }
    public ColumnDefinition getColumnDefinition(ByteBuffer name){
    ...
    public ColumnDefinition getColumnDefinitionForIndex(String indexName){
       for (ColumnDefinition def : column_metadata.values())
=======
        for (ColumnDefinition def : column_metadata.values())
>>>>>>>
...
```

⬜ Developer A
🟩 Developer B

Fig. 6. Crosscutting ordering conflicts.

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 additional 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

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

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 it has fewer additional false positives than unstructured merge. Furthermore, we argue that semistructured merge additional false positives are easier to understand and resolve.

4.2 When compared to unstructured merge, does semistructured merge compromise integration correctness by having more false negatives?

To answer RQ2, we compare the number of additional false negatives of each merge approach. Our results show that the number of semistructured merge additional false negatives (*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 semistructured merge additional false negative (*aFN(SS)*), compared with 0.88% ± 1.08% of unstructured merge (*aFN(UN)*). So, considering the aggregated merge scenarios of all projects, semistructured merge has more additional false negatives and more scenarios with additional false negatives. However, in practice, we should expect a lower, or even no advantage in favor of unstructured merge since *aFN(SS)* is overestimated and *aFN(UN)* is underestimated. So, in this aspect, our study brings no conclusive evidence to answer RQ2. Besides, as for RQ1, this does not uniformly hold across projects: *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 more additional false negatives (*aFN(UN)*).

As in the previous section, the observed error bounds are partly explained by some projects having high rates of common conflicts. But here, they are additionally influenced by some projects having high rates of accidental detection of conflicts by unstructured merge. In fact, projects such as AntennaPod and mockito presented considerably above the average percentage of additional false negatives (*aFN(SS)*) due to accidental detection of conflicts by unstructured merge. Most of these conflicts were conservatively classified as semistructured additional false negatives, but turned into false positives (*aFP(UN)*) after further analysis. Conversely, analyzing unstructured merged additional false negatives (*aFN(UN)*), we found projects with a higher incidence of duplicate simple methods declarations such as getters and setters, or methods containing common words from developers' vocabulary such as initialize, execute, run, or load. We also found that copy and paste across repositories was a common practice in some projects. For example, in a certain commit one developer added a method. Then, on a divergent branch, another developer copied this method and made a few changes. When merging these changes, the conflict occurred with semistructured merge, but not with unstructured merge. We even found examples where, instead of copying one method, the developer copied entire files from one repository to the other. All these situations led to above average percentage of additional false negatives (*aFN(UN)*) in projects such as atmosphere, cloudify and gradle.

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 additional 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.

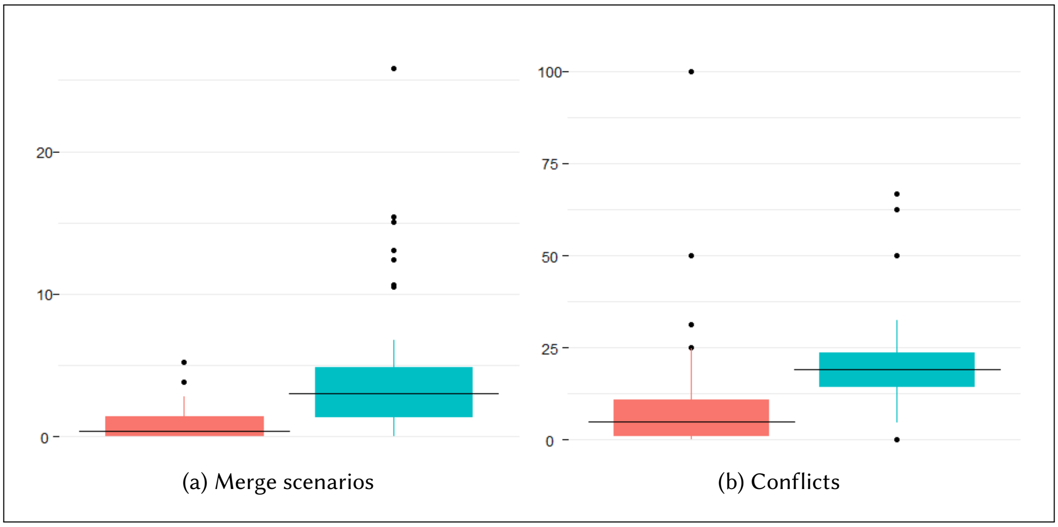
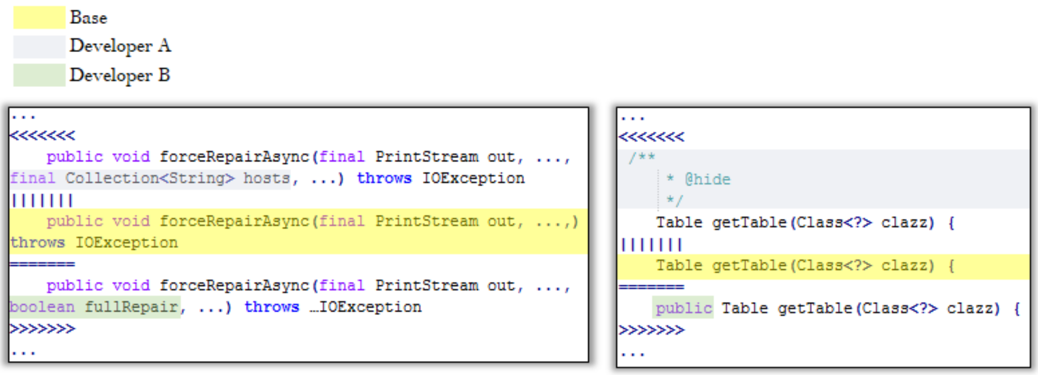


Fig. 7. Box plots describing the percentage, per project, of the additional false negatives in terms of merge scenarios and conflicts. Unstructured merge in red, semistructured in blue.

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 additional 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 due to unstructured merge accidental conflict detection. So, to understand how high our upper bound could be, we manually analyzed 50 randomly selected possible additional false negatives of semistructured merge. We checked if they indeed represent missed conflicts. From the 50 analyzed cases, only 6 were confirmed false negatives.[8] 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 situation incorrectly classified as additional false negative by our metric. Developer A added a comment to the `getTable` declaration, while Developer B added an access modifier. These changes clearly do not conflict. Accordingly, as they correspond to different parts of the node representing the `getTable` declaration, semistructured merge does not report conflict. Unstructured merge does report a false positive because the changes occurred in the same text area.

4.2.1 Additional False Negatives of Semistructured Merge are Harder To Detect and Resolve. When comparing false negatives, at first thought, the reasoning seems straightforward because the greater

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



```
...
<<<<<<<
    public void forceRepairAsync(final PrintStream out, ...,
final Collection<String> hosts, ...) throws IOException
|||||||
    public void forceRepairAsync(final PrintStream out, ...,)
throws IOException
=======
    public void forceRepairAsync(final PrintStream out, ...,
boolean fullRepair, ...) throws …IOException
>>>>>>>
...
```

```
...
<<<<<<<
 /**
   * @hide
   */
   Table getTable(Class<?> clazz) {
|||||||
   Table getTable(Class<?> clazz) {
=======
   public Table getTable(Class<?> clazz) {
>>>>>>>
...
```

(a) False Negative (b) False Positive

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

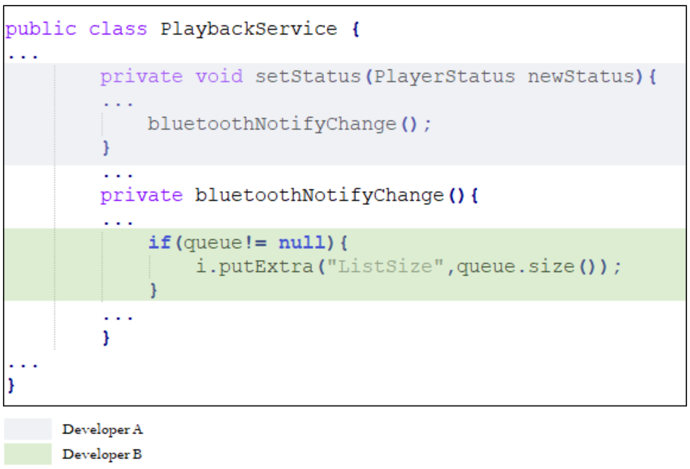
the number of false negatives, the greater the number of post-merge build and behavioral errors. Therefore, the weaker the correctness guarantees of the merging process. In that sense, the achieved results suggest that unstructured merge beats semistructured merge for most, but not all, scenarios and 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 to detect. In particular, unstructured merge additional false negatives cause compilation errors, guiding the developers toward the location and cause of the problem. Conversely, semistructured merge additional false negatives 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. This situation is illustrated in Figure 9, showing code from the project AntennaPod. The developer who added the `setStatus` method might not be expecting the extra notification added by the other developer on the `bluetoothNotifyChange` method (referenced by the first developer). We believe that in such cases the detection and resolution of the issue is likely more difficult and demands more effort.

In our sample, though not uniformly across projects, semistructured merge has more additional false negatives than unstructured merge. Moreover, we argue that semistructured merge additional 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 explained 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 crosscutting conflicts, and how contributions were integrated in the project repositories. Further analysis involving integrators would be important to reinforce our conclusions. Also, a more rigorous analysis based on conflict detection and resolution timing data, in the spirit of Berry [2017], could differently weight false positives and false negatives and better assess integration effort reduction. We also conducted a manual analysis to estimate the impact of false positives on integration effort. As explained in



```
public class PlaybackService {
...
        private void setStatus(PlayerStatus newStatus){
        ...
            bluetoothNotifyChange();
        }
        ...
        private bluetoothNotifyChange(){
        ...
            if(queue!= null){
                i.putExtra("ListSize",queue.size());
            }
        ...
        }
...
}
```

Fig. 9. Observed new element referencing edited one.

Section 4, the number of analyzed cases (50) is appropriate considering margins of error of 10% and 15%, respectively, for renaming and ordering conflicts. Reducing the margins of error would increase the accuracy of our conclusions, but would substantially increase the number of cases to manually analyze. Note, however, that the adopted margins are safe for the reached conclusions, which show large differences.

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. The main issue is related to the semistructured merge additional false negatives metric; its upper bound is too high. We confirmed that by manually inspecting 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 additional false positives and false negatives, but only as sufficient evidence to support our conclusions.

Still regarding our approximations, we check conflict resolution in projects repository to classify semistructured merge additional false negatives (see Section 2.4). We assume that the developer correctly analyzed the conflict and decided that there was no problem, so that would be an unstructured false positive, not a semistructured false negative. The problem is that the changes might still lead to semantic problems not perceived by the integrator, missing actual conflicts.

Finally, as described in Section 3.2.3, when textually similar initialization blocks conflict with unstructured merge, we consider the resulting number of conflicts as the number of semistructured merge additional false negatives related to initialization blocks. However, this might not be safe because the triple of initialization blocks that should be matched might involve elements with less than the adopted degree of similarity (80%). To check this was not a problem in our sample, we manually analyzed all files merged by unstructured merge having at least one reported conflict and edited initialization blocks. These necessary conditions for false negatives of this kind were satisfied only by 40 files in our sample. In this analysis, we checked if the reported conflicts involve the same initialization blocks matched by our similarity threshold. If the conflicts involve different initialization blocks, our metric could be missing actual conflicts. This only happened in a single file of project cassandra (`Cassandra.java`, an atypical file with more than 40K lines of codes and more than 50 conflicts). Nevertheless, further inspection of this file showed us that the involved initialization block was not edited, and that the corresponding conflict was actually a crosscutting conflict— the conflict was an unstructured merge additional false positive. In all other files, unstructured merge did not report conflicts involving edited initialization blocks other than those accused by our metric.

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 that might have suffered the effect of commands such as `rebase` and `cherry-pick`, which rewrite project history [Bird et al. 2009]. 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, they dramatically decrease 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 in our sample. We believe this has insignificant impact on our results. We also discarded non-Java files from the analyzed projects. This corresponds to 1.73% of the total number of files in the sample. Per project, we discarded on average 15.33% ± 19.46% files. Projects substantially differ in the overall number of files and non-Java files. For instance, Junit has 539 files, with only 0.90% of non-java files, whereas clojure has 308 files, with 49.5% of non-java files. So the results for projects such as clojure could be different if one considers all kinds of files. Nevertheless, for both non-Java and invalid Java files, semistructured merge could be easily adapted to invoke unstructured merge, similarly as it does to method bodies. As a consequence, the approaches would behave identically and, therefore, present the same numbers for these files.

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 similar to 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 reported 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 false positives renaming conflicts if they opt for semistructured merge. Similarly, if project changes often occur in the same text area, conflicts that would otherwise be detected my unstructured merge 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.

6.1 Improvements

Our improved merge tool implements the algorithms underlying the scripts we used to detect false positives and false negatives (see Section 3.2) in our empirical analysis.[9] On top of a more efficient version of the FSTMerge tool [Apel et al. 2011] we implemented, we added a module (or *handler*) for semistructured merge additional false positives and three kinds of additional 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 [Levenshtein 1966] to map renamed elements. Afterwards, the handler verifies if unstructured merge has reported a conflict with the original element's signature. If that is the case, the improved tool reports the renaming conflict, now filled with information of the renamed version gathered in the first step. As explained in Section 2.2, a renaming conflict is characterized by the presence of the original element's signature in the conflict text. So, if unstructured merge does not report conflict with such signature, there is no guarantee that the renaming conflict is not an additional false positive; thus the improved tool does not report the 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 accurate. For example, eliminating such a false positive could result in a false negative, but a common one to unstructured merge.

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 textually reference edited ones. If that is the case, and if unstructured merge also reports a conflict involving the added elements, our tool also reports a conflict. This way we eliminate a possible false negative, making sure we are not adding an additional false positive in relation to unstructured merge. As, in this context, false negatives might be more disruptive than false positives [Berry 2017], increasing the chances of detecting more false negatives at the expense of possible new false positives could be an option for the design of the new tool. However, aiming at reducing trade-offs and facilitating industrial adoption, we opted for a design that tries to ensure that the new tool is not worse than unstructured merge with regard to false positives. This is why this handler only reports conflicts if unstructured merge reports a similar one.

The *initialization blocks handler* uses the Levenshtein distance algorithm to match elements of this type, using 80% of degree of similarity. The handler then 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. If more than one initialization block is matched, the handler takes the first one with the highest similarity. Note that using unstructured merge as oracle for this handler was not possible. In particular, checking if unstructured merge conflicts involve initialization blocks is challenging, as initialization blocks have no identifiers. Thus, there is no precise way to determine if unstructured merge conflicts solely identify an initialization block. This way, trying to reduce false negatives, the improved tool could potentially create new false positives. We opted for this design because of the small risk and reduced impact involved.

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. Regarding Java support, the tool uses an updated annotated grammar to support

[9] Publicly available at https://github.com/guilhermejccavalcanti/jFSTMerge.

Java 8. Annotating other languages grammars, and implementing specific false positive and false negative handlers for these languages would make semistructured merge benefits more widely applicable. Our tool can be configured to resolve false positives due to code indentation: it compares elements content ignoring spacings. Finally, regarding usability, after installation, the tool is entirely integrated with git version control system (integration with others VCS is likely not hard too); whenever 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.

6.2 Gains

Based on the sample and results of our empirical evaluation, Table 1 summarizes how conflict numbers of the improved tool compare to unstructured merge and the original semistructured 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.

Table 1. Comparing conflict numbers of unstructured, semistructured, and improved tools.

	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%)

With respect to false positives and negatives, Table 2(a) summarizes the main result discussed in Section 4. Contrasting, Table 2(b) shows how the improved tool compares to unstructured merge. In our sample, the improved tool completely eliminates semistructured merge additional false positives. Although our *initialization blocks handler* might lead to false positives, as discussed in Section 5, we had no such case. Moreover, as the *new element referencing edited one handler* of the improved tool uses unstructured merge as oracle to reduce false negatives, it might actually have new false positives in common with unstructured merge. In our sample, this corresponds to at most 535 conflicts, in case all conflicts detected by that handler are inaccurate. So the reported numbers of additional false positives of unstructured merge in relation to the original and improved tool (see both tables) differ in approximately 7%.

Table 2(b) also shows that the improved tool eliminates a few kinds of false negatives, leading to a reduction of approximately 23% in the number of additional false negatives in relation to the original semistructured tool. This way, the improved tool is superior to unstructured merge with respect to the overall number of additional false negatives: it misses at least[10] 8% fewer false negatives than unstructured merge. Due to the quite conservative nature of our metric, and the results of our manual analysis (which suggests that the semistructured merge false negatives numbers might be around 12% of the reported numbers), we expect the advantage in favor of the improved tool to be much higher. Those advantages of the improved tool do not uniformly hold across projects, but most projects follow this pattern.

[10]Remember that our metrics are approximations.

Table 2. Comparing false positives and false negatives numbers of the improved and original tools with unstructured merge. Arrows indicate whether the number is underestimated (↑, meaning the numbers should be bigger in practice) or overestimated (↓).

(a)

	Unstructured tool	**Semistructured tool**
Additional False Positives	7,958 ↑	5,201 ↓
Additional False Negatives	2,714 ↑	3,260 ↓

(b)

	Unstructured tool	**Improved tool**
Additional False Positives	7,423 ↑	0
Additional False Negatives	2,714 ↑	2,489 ↓

Notice that only part of the false negatives is eliminated with the improved tool. The false negatives that were not eliminated are hard to detect because they do not follow a syntactic pattern. Our oracle for detecting them in the study relies on analyzing the merged code, which is appropriate for our retrospective analysis, but not for using the tool in practice. So a handler could not rely on that. For instance, as explained in Section 3.2.3, when the resolution of an unstructured merge conflict does not keep all developers' changes, or adds new code, we consider the conflict as a semistructured merge false negative. However, this information is only available after the merge result has been committed. Finally, as the *renaming handler* uses unstructured merge as oracle to reduce false positives, it might lead to new common false negatives with unstructured merge. However, as our metric of unstructured merge false negatives does not include renaming conflicts, the corresponding numbers are the same in both tables.

6.3 Performance Evaluation

Although performance was not a priority for our design, we evaluated the improved tool performance on a random subsample of 1731 merge scenarios from 25 projects. This is a subset of the full sample described in Section 3.1, considering only scenarios that reported at least one conflict, regardless of the merge approach. For each merge scenario, we invoked the original, improved and unstructured tools, 5 times each, measuring execution time. We conducted the evaluation on a desktop machine with Intel Core i5, 4 cores @4.0 GHz, 16 GB RAM and 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 that an optimized tool would still be much slower than unstructured merge. It is though, much faster than the original semistructured merge tool, which took 123 minutes to merge the entire sample. We observed that the original tool has severe performance issues due to its prototype nature. In particular, the original tool creates a single representation (tree) in memory for all files in the merge scenario— including unchanged files, and files differing only by spacing. This leads to complex trees with expensive node matching, and slow tree merges. We address this issue by creating a tree per changed merged file, ignoring files that only had spacing related changes. We kept the tree merging algorithm, but the fewer and simpler trees substantially improve performance. These optimizations more than compensate the extra complexity associated to the false positives and false negatives detection handlers we implemented.

We also observed that, in practice, the performance difference between the improved and unstructured tool 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 2% of the scenarios, with a maximum of 67 seconds in a lucene-solr scenario that merges 303 files, resulting in 618 conflicts and 17,567 LOC of conflicting code. For the same scenario, unstructured merge took 6 seconds, resulting in 866 conflicts and 27,397 LOC of conflicting code. In our sample both the improved and unstructured tool 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, but 4,27 ± 14,75 seconds with the original tool).

7 RELATED WORK

A number of studies propose development tools and approaches 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 as a basis 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 additional 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; Malpohl et al. 2000] 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`, an 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 an additional 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, the 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.

Finally, other empirical studies provide evidence about the frequency and impact of conflicts, and their associated causes. For example, Brun et al. [2011] and Kasi and Sarma [2013] 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. Adams and McIntosh [2016] and Henderson [2017] even report that companies have migrated to single-branched repositories to avoid merge problems. Our work complements these studies by bringing evidence about the frequency of integrations that had certain types of false positives and false negatives. This shows how often integrations demand unnecessary integration effort. It also show how often conflicts were undetected by unstructured and semistructured merge tools, leading to merge problems. In addition, Brun et al. [2011] and Kasi and Sarma [2013] 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 merge situations, 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 problems. Besides that, false negatives represent build or behavioral errors, negatively impacting software quality and correctness of the merging process. For most projects and merge situations, we observed that semistructured merge not only reduces the number of reported conflicts, but it also has fewer additional false positives when compared to unstructured merge. Furthermore, we find evidence that semistructured merge additional false positives are easier to analyze and resolve than those reported by unstructured merge. However, we found no evidence that semistructured merge has fewer additional false negatives than unstructured merge. We also argue that semistructured merge false negatives 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 in our sample, reduces the number of reported conflicts by half, has no additional false positives, has at least 8% fewer false negatives, is not prohibitively slower, and presents no extra usability barriers in relation to state of the practice of merge tools. In practice, we expect the reduction in the number of false negatives to be much higher, given the conservative nature of the metrics we use here. Although the improved tool has fewer false negatives, they might be harder to detect and resolve than unstructured merge false negatives. As we have no conflict detection and resolution timing data, we cannot, in a precise way, differently weight different kinds of false negatives and better assess the benefits of our tool. This would be needed to, in the spirit of Berry [2017], compare the tools in a more rigorous way. Similarly, we cannot precisely weight false positives and false negatives. This, however, is less critical in our case because, in the analyzed sample, the improved tool has no additional false positives in relation to unstructured merge.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, Daniel Berry, and members of the Software Productivity Group for the quite pertinent suggestions that contributed to improve this work. We also thank Daniel Maida for helping with the integration of our tool with git. Rodrigo Bonifácio, and his team at University of Brasília, helped using and providing useful feedback about our tool. Finally, for partially supporting this work, we would like to thank INES (National Software Engineering Institute) and the Brazilian research funding agencies CNPq (grant 309741/2013-0), FACEPE (grants IBPG-0546-1.03/15 and APQ/0388-1.03/14), and CAPES.

REFERENCES

B. Adams and S. McIntosh. 2016. Modern Release Engineering in a Nutshell – Why Researchers Should Care. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*. IEEE.

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).

Daniel M. Berry. 2017. Evaluation of Tools for Hairy Requirements Engineering and Software Engineering Tasks. (2017). https://cs.uwaterloo.ca/~dberry/FTP_SITE/tech.reports/EvalPaper.pdf

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

Valdis Berzins. 1994. Software Merge: Semantics of Combining Changes to Programs. *ACM Transactions on Programming Languages and Systems* (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, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. 2009. The Promises and Perils of Mining Git. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR'09)*. IEEE.

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.

Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. 2017. Online Appendix of the paper Evaluating and Improving Semistructured Merge. Hosted on https://spgroup.github.io/s3m. (2017).

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).

John Eng. 2003. Sample size estimation: how many individuals should be studied? *Radiology* (2003).

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

Fergus Henderson. 2017. Software Engineering at Google. *CoRR* (2017).

Susan Horwitz, Jan Prins, and Thomas Reps. 1989. Integrating Noninterfering Versions of Programs. *ACM Transactions on Programming Languages and Systems* (1989).

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.

Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*.

Guido Malpohl, James J. Hunt, and Walter F. Tichy. 2000. Renaming detection. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*. IEEE.

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

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 4th International Workshop on Mining Software Repositories (MSR'07)*. IEEE.