Noname manuscript No.
(will be inserted by the editor)

Understanding Semistructured Merge Conflict Characteristics in Open-Source Java Projects

Paola Accioly · Paulo Borba · Guilherme Cavalcanti

Received: date / Accepted: date

Abstract Empirical studies show that merge conflicts frequently occur, impairing developers' productivity, since merging conflicting contributions might be a demanding and tedious task. However, the structure of changes that lead to conflicts has not been studied yet. Understanding the underlying structure of conflicts, and the involved syntactic language elements might shed light on how to better avoid merge conflicts. To this end, in this paper we derive a catalog of conflict patterns expressed in terms of the structure of code changes that lead to merge conflicts. We focus on conflicts reported by a semistructured merge tool that exploits knowledge about the underlying syntax of the artifacts. This way, we avoid analyzing a large number of spurious conflicts often reported by typical line based merge tools. To assess the occurrence of such patterns in different systems, we conduct an empirical study reproducing 70,047 merges from 123 GitHub Java projects. Our results show that most semistructured merge conflicts in our sample happen because developers independently edit the same or consecutive lines of the same method. However, the probability of creating a merge conflict is approximately the same when editing methods, class fields, and modifier lists. Furthermore, we noticed that most part of conflicting merge scenarios, and merge conflicts, involve more than two developers. Also, that copying and pasting pieces of code, or even entire files, across different repositories is a common practice and cause of conflicts. Finally, we discuss how our results reveal the need for new research studies and suggest potential improvements to tools supporting collaborative software development.

Keywords Collaborative software development · Merge conflicts · Empirical Software Engineering · Awareness tools

P. Accioly
Informatics Center, Federal University of Pernambuco, Brazil
E-mail: prga@cin.ufpe.br

P. Borba
Informatics Center, Federal University of Pernambuco, Brazil
E-mail: phmb@cin.ufpe.br

G. Cavalcanti
Informatics Center, Federal University of Pernambuco, Brazil
E-mail: gjcc@cin.ufpe.br

1 Introduction

In a collaborative development environment, tasks are commonly assigned to developers working independent from each other. As a result, when trying to integrate these contributions, one might have to deal with conflicting changes. Such conflicts might be detected when merging contributions (*merge* conflicts), when building the system (*build* conflicts), or when running tests (*semantic* conflicts). Regarding such conflicts, previous studies (Brun et al., 2013; Kasi and Sarma, 2013; Zimmermann, 2007; Bird and Zimmermann, 2012) show that they occur frequently, and impair developers' productivity, as understanding and solving them is a demanding and tedious task that might introduce defects.

However, despite the existing evidence in the literature, the structure of changes that lead to conflicts has not been studied yet. Understanding the underlying structure of conflicts, and the involved syntactic language elements, might shed light on how to better avoid them. For example, awareness tools that inform users about ongoing parallel changes such as Syde (Hattori and Lanza, 2010) and Palantír (Sarma et al., 2012) can benefit from knowing the most common conflict patterns to become more efficient.

With that aim, in this paper we focus on understanding the underlying structure of *merge* conflicts. At first one might think that merge conflicts do not have a direct impact on software productivity and quality, as the state of the practice merge tools identify merge conflicts, and developers solve them before resuming implementation activities. However, previous studies (Sarma et al., 2012; Bird and Zimmermann, 2012) suggest the contrary by reporting, based on experimental observations, that resolving merge conflicts is not so trivial. It might take considerable time, and is an error-prone activity.

To better understand merge conflict characteristics, here we derive a catalog of conflict patterns expressed in terms of the structure of code changes that lead to conflicts. In particular, we focus on conflicts reported by FSTMerge (Apel et al., 2011), a *semistructured* merge tool that is able to automatically resolve a large number of spurious conflicts often reported by typical unstructured, line based merge tools (Apel et al., 2011; Cavalcanti et al., 2015, 2017). For example, FSTMerge automatically resolves conflicts due to changes involving commutative and associative declarations, such as two methods inserted in the same text area— the so-called *ordering* conflicts. Moreover, FSTMerge is able to detect conflicting situations that line-based tools cannot. Namely, when developers add methods with the same signature— but with different behaviors— to the same file. Such situation likely leads to a build conflict. However, unless developers add both methods to the same text area, line-based tools will not be able to detect such a conflict. From now on in this paper, when we mention merge conflicts, we refer to the conflicts that FSTMerge reports.

To derive our conflict pattern catalog, we analysed FSTMerge's implementation and systematically derived conflict patterns by abstracting *all* kinds of conflicts that can be detected by this tool. Each pattern captures the language syntax elements involved in a conflict, besides the interaction between two revisions that should be integrated, and their common base revision— the so-called *merge scenario*. In particular, we are interested in the structure of individual changes performed along two different development branches or repository clones, and how they lead to a conflict. For example, the pattern *"Different edits to the same class*

field declaration" captures the situation when two developers, working independently, edit the same class field declaration.

To assess the occurrence of such conflict patterns in different systems, we conducted an empirical study that reproduces 70,047 merges from 123 GitHub Java projects. This sample has 10 times more projects, and more than 15 times merge scenarios than related previous studies (Brun et al., 2013; Kasi and Sarma, 2013). Like such studies, we limit our analysis to Git observable merges, since a number of merges might have been removed by the use of Git commands such as *rebase*.

Our results show that 84.57% of semistructured merge conflicts in our sample happen because developers independently edit the same or consecutive lines of the same method. This result might seem obvious at first, as most code in Java files appear inside methods. However this is only the case because we used a more sophisticated merge tool that is able to solve ordering conflicts. Constrasting, if we had used a line-based merge tool, such as GNU *diff3* (Free Software Foundation, 2016), a significant part of the reported conflicts would likely be ordering conflicts, which appear outside methods, or even crossing different methods boundaries, as indicated by previous studies (Apel et al., 2011; Cavalcanti et al., 2015, 2017). Moreover, even if one expects most conflicts inside methods, it would be hard to guess the frequency proportions among different conflict patterns considering edited language syntax elements.

By normalizing the number of conflicts considering the number of changes made to the different language syntax elements, we found out that edits to method lines, class fields, and modifier lists show similar probabilities of leading to merge conflicts. With the obtained evidence, we might say that a reasonable strategy to avoid merge conflicts is to monitor ongoing development activities, and alert developers editing the same method lines, class field or modifier lists so that they can communicate and solve potential conflicts early instead of having to resolve a merge conflict hours or even days after implementing such changes.

Additionally, our results show that merge conflicts happened in 9.38% of the analyzed merges from our sample, with a median of 6.64%, and an IQR (Interquartile Range) of 8.81%. Moreover, we noticed that part of these semistructured merge conflicts are spurious conflicts simply caused by changes to code indentation or consecutive line edits. This motivated us to implement an improved version of FSTmerge that automatically resolves such conflicts. Using this adapted tool dropped the total conflicting merge scenario rate to 8.39% (median of 6%, and IQR of 7.21%). This result reinforces the existing evidence (Apel et al., 2011; Cavalcanti et al., 2015) that semistructured merge indeed reduces the number of reported conflicts. And there is still room for improvements.

Our data also indicates that developers often do not take full advantage of proper code version, but rather copy and paste code around different branches, editing them, and then merging them back together, creating the risk of conflicts. This problem evidences the need for tools that enable partial merges where developers, instead of merging entire sequences of commits, can break commits into smaller parts/pieces of code and then choose what commits they want to merge.

Finally, our data indicates that merge scenarios often involve more than two developers' contributions, suggesting that merging branches is not likely to be a simple task, since one needs to understand and merge contributions made by different developers probably working on different assignments. This evidence reinforces

the need for tools such as TIPMerge (Costa et al., 2016) which recommends expert developers for integrating changes across branches.

In summary, in this work we make the following contributions:

- Derive a conflict pattern catalog with 9 patterns, and collect evidence on how frequently each pattern occurs using different metrics;
- Implement an improved version of the FSTMerge tool that further eliminates spurious conflicts, and provide evidence that such improvement effectively decrease the number of reported conflicts;
- Report new evidence on merge conflict frequency, by measuring how frequently merges end up with conflicting changes, which allows us to compare our results to previous studies. Moreover, we use a much larger sample, and adopt a more advanced merge tool then previous studies did;
- Reveal the need for new research studies, and suggest potential improvements to tools that support collaborative software development.

The material used to execute our study, including sample description, tools, and results can be found in our online appendix (Accioly et al., 2017).

2 Understanding Merge Conflicts Characteristics

Considering the context described in the previous section, our goal in this paper is to understand characteristics— such as structural patterns, causes, and frequency— of merge conflicts reported when reproducing real merge scenarios from the development history of different software projects. To achieve this goal, we investigate the following research questions.

2.1 Research Question 1 (RQ1): What are the structural conflict patterns?

To answer **RQ1** we need to derive a conflict pattern catalog, highlighting conflict structures in terms of the program elements independently changed in each of the revisions that lead to the conflict. We derive such catalog by abstracting the kinds of conflicts that can be detected by merge tools. Because we focus on merge conflicts, in this paper we do not use tools and strategies such as *Semantic Diff* (Jackson and Ladd, 1994), which compute semantic differences between two versions of the same method. This leaves us with the following strategies for merge tools: unstructured, semistructured, and structured (Apel et al., 2011).

As mentioned before, unstructured, line-based merge tools might report too many ordering conflicts. Besides that, it would be hard to systematically derive a catalog of conflict patterns based on edited language syntax elements, as unstructured tools analyse text lines, and have no knowledge about the underlying artifact syntax. So, mainly to avoid biasing our sample with a large number of spurious merge conflicts, we decided not to use unstructured tools to derive the conflict patterns. In contrast, structured merge tools such as JDime (Apel et al., 2012) operate on abstract syntax trees (ASTs), and incorporate full information on the underlying language syntax. However, a drawback of this strategy is that it might introduce defects in the merged version of the code. Consider, for example, when one developer edits the initialization statement of a *for* declaration while the other

developer edits the condition statement. In such context, merging these contributions might introduce build or semantic conflicts. Nonetheless, because they edit different statements, the structured strategy would be able to successfully merge them. In addition, there is a considerable performance overhead to use structured tools because they have to build the full artifacts' ASTs for every merge scenario we wish to analyze.

Finally, FSTMerge tool inherits part of the strengths from both unstructured and structured strategies by partially representing software artifacts as trees; tree leaves represent as text code elements such as method bodies. It also provides information (through an annotated grammar) about how nodes of certain types (methods, classes, etc.), and their subtrees can be merged. This way, FSTMerge is able to resolve a number of conflicts— the ordering conflicts— based on the information that the order of certain elements (classes, methods, fields, and so on) does not matter. The code elements represented as text are merged using a conventional line-based merge tool. Moreover, as we mention in Section 1, besides resolving ordering conflicts, FSTMerge is capable of detecting some types of conflicts that line-based merge cannot. For example, if two developers add to the same class, but in different parts of the text, methods with the same signature, but with different bodies, FSTMerge reports a conflict. Such strategy prevents subsequent build problems while trying to build files with duplicate methods.

Moreover, the list of conflicts detected by FSTMerge is not exhaustive. This tool has false positives and false negatives, as any other merge tool that we could have chosen to derive the conflict catalog would have. In fact, in our threats to validity section (Section 6) we present a list of FSTMerge false positives and false negatives, and analyse the impact that they might have on our results.

In order to derive the conflict patterns, our starting point was FSTMerge's annotated Java grammar. The file describing the Java grammar contains annotations on nodes declarations describing how FSTMerge should handle conflicts in each type of node. For example, the method declaration node has an annotation saying that conflicts involving this type of node should be handled by calling the line-based merge approach.[1]

So, to derive the conflict pattern catalog, the first author of this study checked all node annotations present in this file and derived the conflict patterns based on the syntactic elements involved. Then, while discussing this catalog with the other authors, we noticed that two nodes (the class extends declaration, and the enumeration constant declaration) did not have annotations. So we changed FSTMerge's annotated grammar to add annotations to these nodes as well. Finally, if our conflict analyzer tool could not match the conflict with any of the defined patterns, it would classify the conflict in a pattern called "No Pattern". However, none of the conflicts from our sample were classified in this category.

Table 1 describes the resultant catalog containing 9 conflict patterns for Java programs, expressed in terms of the performed kinds of changes to the involved syntactic language structures:

We add the word "different" in our edit related conflict patterns to remind that, if developers make equal edits— such as adding the same *get* method— there is no conflict, since their contributions do not interfere with each other. In

[1] `https://goo.gl/9BXCmn`

Table 1 Conflict patterns' description

Abbreviation	Description
EditSameMC	Different edits to the same or consecutive lines of the same method or constructor, including lines with the list of modifiers and exceptions
EditSameFd	Different edits to the same class field declaration
SameSignatureMC	Methods or constructors added with the same signature and different bodies
AddSameFd	Class fields declarations added with the same identifier and different types or modifiers
ModifiersList	Different edits to the modifier list of the same type declaration (class, interface,annotation or enum types)
ImplementsList	Different edits to the same implements declaration
ExtendsList	Different edits to the same extends declaration
EditSameEnumConst	Different edits to the same Enum constant declaration
DefaultValueA	Different edits to the same annotation method default value declaration

this case, a straightforward solution would be to simply merge the contributions choosing the first developer's version.

To better illustrate our conflict patterns, Figure 1 shows an example of the EditSameMC pattern. We found this conflict while analyzing the OpenTripPlanner project, an open source trip planner application.[2] Note that, in this example, both developers edited the declaration of the *optionsBefore* variable. Hence, the merge tool reported this conflict so that it could be manually solved.

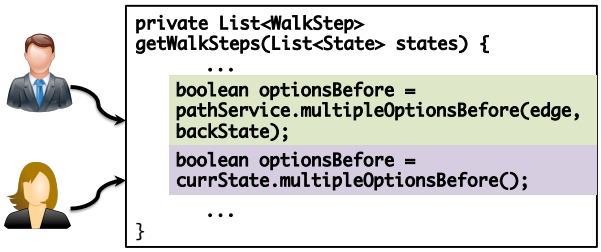

Fig. 1 Example of the EditSameMC pattern occurence.

Finally, although FSTMerge currently supports code written in Java, C#, and Python, for simplicity, in this paper we only analyze Java projects. Considering other languages would require different conflict pattern catalogs and associated analysis. Some of our general patterns, such as EditSameMC and EditSameFd, would also apply for C# and Python. But each language syntax particularities could add or remove patterns from our catalog. For example, C# would have a pattern for when developers edit a directive with the same alias. In contrast, a catalog for Python would not consider the ModifiersList pattern, since Python does not have explicit access modifiers. In the threats to validity section (Section 6) we discuss how such a decision affect our study.

[2] `http://www.opentripplanner.org/`

2.2 Research Question 2 (RQ2): How frequently does each merge conflict pattern occur?

After deriving the conflict pattern catalog, we are able to answer *RQ2* by reproducing real merge scenarios from the entire history of different Java projects, while collecting the absolute number of conflict occurrences for each conflict pattern from our catalog, using the following metric:

- Number of conflicts

By answering *RQ2* we will learn how frequently the different conflict patterns occur, and the frequency proportions among them.

2.3 Research Question 3 (RQ3): What kinds of code changes most likely lead to conflicts?

While the number of conflicts shows conflict patterns that occur more frequently, we complement this information by understanding the likelihood of ending up with a merge conflict when editing different language syntax elements. To this end we compute the following metric:

- Normalized number of conflicts = $\frac{Number\ of\ conflicts}{Number\ of\ changes}$

We compute the normalized number of conflicts by dividing the number of conflict occurrences from each pattern by the number of involved syntax elements changed during the entire project development history. For example, if, during the development history of a particular project, we observe 50 EditSameMC conflicts, and 500 edits to method or constructor elements, the normalized number of conflicts for the EditSameMC pattern would be 0.1, meaning that, when editing a method or constructor, there is a 10% chance of introducing EditSameMC conflicts. In contrast, if, in this same project, we observe 5 EditSameFd conflicts, and 10 edits to class field elements, the normalized number of conflicts for EditSameFd would be 0.5 or 50%. In such context, although EditSameMC conflicts are 10 times more frequent than EditSameFd conflicts, the probability of having EditSameFd conflicts when editing class fields is 5 times higher.

We compute both metrics because they complement each other. The number of conflicts shows which conflict patterns occur more frequently, whereas the normalized number of conflicts is useful to understand what kinds of code changes most likely lead to merge conflicts. One of the goals in our study is to provide recommendations for detecting conflicts more efficiently while working collaboratively. Therefore, computing both metrics is useful to improve collaborative development tools' recall by helping them to detect the most frequent conflicts while being careful about important conflict predictors.

Furthermore, because FSTMerge runs *diff3* to merge methods and constructor elements, we compute the normalized number of conflicts for the EditSameMC pattern considering not only the number of changed methods and constructors, but also the number of changed line chunks, and changed lines inside methods and constructors. This way, we have different alternatives to analyze the results.

2.4 Research Question 4 (RQ4): How frequently do merge conflicts occur?

By answering this question we would like to know how frequently developers have to deal with conflicting changes when merging different code revisions. For this purpose we use the following metric:

– Conflicting scenarios = $\frac{Merge\ scenarios\ with\ conflicts}{Merge\ scenarios}$

The conflicting scenarios metric measures the ratio of merge scenarios having at least one merge conflict by the total number of analyzed merge scenarios. Thus, it gives us the intuition of how often the merge process fails. This metric was also used in previous studies (Kasi and Sarma, 2013; Brun et al., 2013). This way we can compare our results to theirs.

2.5 Lessons Learned in the Pilot Study

With the purpose of testing our infrastructure, we ran a pilot version of this study with a subsample of 40 projects from a larger sample that we selected according to the requirements we describe in Section 3.3. A surprising result was that SameSignatureMC was the second most frequent pattern, representing approximately 13% of the conflict occurrences. It seems unlikely that developers working independently would so often add, to the same class, methods with the exact same signature and different bodies. To better understand the situation, one of the authors manually analyzed a few examples of SameSignatureMC conflicts, and, together with the remaining authors, discussed their underlying causes.

During this analysis, we noticed that some of those duplicate methods were simple methods such as getters and setters, which seems to be reasonable. However, some of those duplications seemed odd as they were large methods (more than 100 lines, for example), and only small parts of them differed. For example, Figure 2 illustrates a conflict extracted from project Jitsi, an instant messenger application. Note that the method called *sendFile* is complex— it has more than 100 lines— and only the highlighted part differs. When we checked Jitsi development history we noticed that in a certain commit one developer added the *sendFile* method. Then, on a different branch, another developer copied this method and made a few changes. Finally, when merging the changes, the conflict occurred. We saw other examples like these. In fact, we even found examples where, instead of copying one method, the developer copied the entire file from one repository to the other.

```
FileTransfer sendFile(Contact toContact,
                      File file,
                      String gw)
        throws  IllegalStateException,
                IllegalArgumentException,
                OperationNotSupportedException
    {
        OutgoingFileTransferJabberImpl outgoingTransfer = null;

        try
        {
            assertConnected();

            if(file.length() > getMaximumFileLength())
                throw new IllegalArgumentException(
                    "File length exceeds the allowed one for this protocol");

            String fullJid = null;
            // Find the jid of the contact which support file transfer
            // and is with highest priority if more than one found
            // if we have equals priorities
            // choose the one that is more available
            OperationSetMultiUserChat mucOpSet = jabberProvider
                .getOperationSet(OperationSetMultiUserChat.class);
            if(mucOpSet != null
                && mucOpSet.isPrivateMessagingContact(toContact.getAddress()))
            {
                fullJid = toContact.getAddress();
            }
```

```
FileTransfer sendFile(Contact toContact,
                      File file,
                      String gw)
        throws  IllegalStateException,
                IllegalArgumentException,
                OperationNotSupportedException
    {
        OutgoingFileTransferJabberImpl outgoingTransfer = null;

        try
        {
            assertConnected();

            if(file.length() > getMaximumFileLength())
                throw new IllegalArgumentException(
                    "File length exceeds the allowed one for this protocol");

            String fullJid = null;
            // Find the jid of the contact which support file transfer
            // and is with highest priority if more than one found
            // if we have equals priorities
            // choose the one that is more available
            if(jabberProvider.getOperationSet(OperationSetMultiUserChat.class)
                .isPrivateMessagingContact(toContact.getAddress()))
            {
                fullJid = toContact.getAddress();
            }
            else
            {
```

Fig. 2 SameSignatureMC example from Graylog2-server project.

Alternatively, in other examples, the duplicated method existed previously in the base revision and was equally renamed in two different repositories. For example, on project Async-http-client, an asynchronous Http Client for Java programs, there was a method called *onHttpError* in the base revision. Then, on subsequent commits from different branches this method was equally renamed to *onHttpHeaderError*. By analysing the project development history we found out that this method is overridden from the Grizzly project API.[3] This method was renamed in the new API version, and when the dependency was updated in the Async-http-client project, the system build no longer worked, causing developers to rename this method across different repositories.

After this analysis, we decided to further detail our coding and automatically analyze SameSignatureMC occurrences matching them with their underlying causes, as further explained in Section 3. Understanding such causes is useful to derive new requirements for tools supporting collaborative development. For this reason, we added an extra research question that we describe as follows.

2.6 Research Question 5 (RQ5): How frequent are the underlying causes of the SameSignatureMC pattern?

We consider the following causes for SameSignatureMC occurrences and measure their frequency:

- Copied files;
- Copied methods;
- Small methods (getters or setters);
- Renamed methods;
- Others.

We consider that copying and pasting across repositories does not necessarily imply code cloning because, after the copy, the developer might merge the branches. In this scenario, after resolving the merge conflicts, the code is no longer duplicated (cloned) in the repository. If the branches were never meant to be merged (branches for different products, for example) then we would have a code clone across software systems, like the ones detected in Svajlenko et al (Svajlenko et al., 2014).

3 Study setup

In this section we describe the setup of the designed study, including how we implemented the tools and scripts to measure the metrics defined in the previous section, and also how we selected the sample projects to conduct the study.

3.1 Conflict Analysis

The infrastructure that we built to run our study can be divided in two steps: mining and merging, which we explain in detail in the folowing subsections.

[3] `https://grizzly.java.net/`

3.1.1 Mining Step

In the mining step we have a script that clones the project locally and runs the command `git log --merges` which provides a list containing information about all merge commits of that project— commits that were the result of a *git merge* command. Subsequently we parse the result of this command to retrieve a list of all merge commit ids and their parent ids. Each merge commit has two parents that we call from now on as left and right revisions.

3.1.2 Merging Step

After retrieving the list of merge commits, we use the *JGit* API (Eclipse, 2015) to checkout and copy the three revisions involved in the merge scenario— the common base revision, and the left and right revisions derived from the base revision and later merged into the merge commit. Then we perform three-way merges using the *git merge* command— which uses *diff3* as default merge tool, and therefore can be applied to any text file— to merge non Java files, and an adapted version of FSTMerge to merge Java files. This way we compute the conflicting scenario rate considering all files in the revisions, and not only Java files so that we can compute the Conflicting Scenario Rate considering all files.

Our adapted version of FTSMerge contains the following new features:

1. An observer that intercepts FSTMerge main mechanism and collects all reported conflicts;
2. We changed FSTMerge's annotated grammar to report the ExtendsList and EditSameEnumConst patterns, which were missing in the original version;
3. We had to discard Java files that could not be parsed by the FSTMerge; it cannot parse the constructs related to lambda expressions and type annotations. However, these Java 8 new features are not the single cause for parser errors. We observed that some parsed files had syntax issues as well. This corresponds to 0.16% of the total number of Java files in our sample;
4. We changed FSTMerge's default line-based merge tool from Revision Control System's (RCS)[4] to Unix's *diff3*; RCS is no longer maintained for *mac os* and we wanted to run the same tool across different operating systems. This change does not impact the results, as RCS uses *diff3* in the background to merge files

We also implement a component called Conflict Analyzer which takes as input FSTMerge's reported conflicts to compute the metrics described in Section 2. When our adapted version of FSTMerge calls *diff3* to merge tree leaves, it first executes the command *diff3 –merge -E*. The parameter *E* makes *diff3* ignore the distinction between tabs and spaces on input. Then, if the output contains conflict markers, we call *diff3* again, but this second time without parameter *E*. We do that because the parameter *E* removes the base code version from the conflict body, and we need the base version in order to run further analysis on conflicts, as explained later.

[4] http://www.gnu.org/software/rcs/

3.1.3 Identifying Different Spacing, and Consecutive Line Edit Conflicts

Whenever our adapted version of FSTMerge reports a conflict, the Conflict Analyzer collects the conflict body to match the conflict with its respective pattern from the catalog. If it cannot match the conflict with any of the defined patterns, it classifies the conflict in the "No pattern" category. However, none of the conflicts from our sample were classified in this category. Subsequently, the Conflict Analyzer evaluates the conflict body content (base, left, and right versions) to identify two situations that likely represent spurious conflicts: different spacing, and consecutive line edit conflicts. Figure 3 illustrates both cases. The left side of the figure illustrates a different spacing conflict example. On both revisions, the same line was edited, but only the right revision made significant changes to the code: added a parameter to the method declaration. The left revision simply altered code formatting. Our tool is able to identify this type of false positive by comparing left and right revisions to the base revision, ignoring tabs, spaces and line breaks. If at least one of the revisions (left or right) is equal to the base, we classify this conflict as a different spacing conflict. We can then factor out this kind of conflict in our results, focusing on the analysis of conflicts that are likely more relevant.

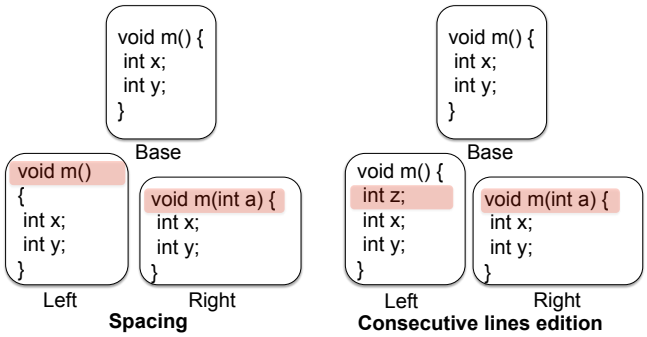

Fig. 3 Types of conflicts that diff3 cannot merge.

On the right side of Figure 3, both revisions apply significant changes, but *diff3* algorithm is not able to merge them because consecutive lines were edited (Khanna et al., 2007). While we analyze spacing conflicts for all conflict patterns in our catalog, we only look for the consecutive line edit conflicts on the EditSameMC occurrences, because it is the only case when FSTMerge calls the *diff3* algorithm. We identify such conflicts using string comparison to check if only consecutive lines were edited. While spacing conflicts always represent false positives, there is a risk that consecutive line edit conflicts might lead to semantic conflicts. For example, if the definition of a *string* variable takes more than one code line, and each developer edits one of the lines. So the analysis factoring out this kind of conflict should be considered with care. For this reason we present separate results to consider the incidence of conflicts with and without consecutive line edit conflicts. We identify such cases to compare the overall numbers with narrowed down numbers that try to focus on the more interesting merge conflicts, that is, conflicts that have a

greater chance of representing an interference between development contributions. Besides that, both conflict types are simple to resolve automatically. A straightforward solution to resolve spacing conflicts is to replace the conflict body with the significant changed version of the code. Moreover, allowing *diff3* to merge code when consecutive lines are edited could solve the second example from Figure 3.

3.1.4 "Identifying the underlying causes of SameSignatureMC conflicts

After identifying spacing and consecutive line edit conflicts, there is one extra step applied for the SameSignatureMC conflicts to understand and quantify its underlying causes. To automate this analysis, we first check if the file containing the conflicting method or constructor exists in the base revision. In case it does not, we classify this occurrence in the "Copied file" category. Such situation happens when one developer adds one file, and, after a while, another developer copies (instead of pulling and merging) this file to his/her local workspace, and then alters the method body. Conversely, if the file containing the method exists in the base revision, we check the method size and its name. If the method name contains the words *get* or *set*, or it is a small method— with no more than 3 lines of code— we consider the conflict to be in the "Small method" category. This situation we believe is more reasonable to expect: two developers working independently felt the need to add getters, setters or other kind of simple methods to the same class.

If the conflict was not classified in the "Copied file" and the "Small method" categories, there are still the two following categories: "Copied method" and "Renamed method", whose classification is more elaborated. Because the SameSignatureMC conflicts' body has no base version, we first compare both methods versions using the Levenshtein distance algorithm (Levenshtein, 1966) to check for string similarity. We used the original algorithm version considering insertion, deletion, and substitution of characters. The extended version considers also the transposition of two adjacent characters. This extension would be useful to measure the distance between smaller strings such as words, when, for example, two adjacent characters are displaced in a typo. In our work, we compare larger strings (entire method declarations with more than 3 lines of code), so this feature would be less useful. It would capture, for example, situations like when a local variable has its name slightly changed, but we believe that our threshold— as discussed next— is able to consider such cases.

We consider methods to be the same if the similarity value is greater than or equal to 70%. To choose this value, we executed our analyses, considering 68 randomly selected projects from our original sample, using 3 different thresholds (>=60%, >=70%, and >=80%). We found out that, for the Renamed Method category, a total of 78.6% of the renamed methods in the sub sample fall in the >=80% category, and an additional >=11% is considered if we use the >=70% category. For the Copied Method category, 84.4% fall in the >=80% category, and an additional 8% is considered using the >=70% category. Hence, we considered 70% to be an acceptable threshold value, since we get most part of the renamed and copied methods (more than 80% similar), and we are still able to get some of the renamed, and copied methods having similarities between 70% and 80%.

If the methods similarity values is less than 70% we classify them in the "Others" category. Otherwise, if the methods strings are more than 70% similar, we look for a similar method in that file from the base revision. If we find a similar

method on the base revision— considering the same 70% threshold of similarity— we classify this conflict in the "Renamed method" category. Finally, if we do not find a similar method in the base revision, we classify the conflict in the "Copied method" category.

3.2 Normalized number of conflicts analysis

To measure the normalized number of conflicts we need to compute the number of conflicts for each conflict pattern, and divide this number by the sum of all changes made to the involved language syntax elements during the entire project history. Figure 4 illustrates how we compute both metrics. The top part of the figure shows a graph representing the development history of a project hosted on Git. In this graph, the vertices represent the commits, and each commit has an edge pointing to its parent (or parents, in the case of merge commits such as E and G).

As previously explained, we compute the number of conflicts from the merge commits in the merging step. To compute the total changes for a given kind of syntactic element in the project history, we run the *git log* command that provides information about all commits in that project, and parse the output to retrieve the list of all commits ids and their parents ids. Then we use our adapted version of FSTMerge to compute the kinds and numbers of nodes changed between each commit and its parents. Note that in the *CHANGES* formula we sum changes only from regular commits, and exclude changes from merge commits. Otherwise we would be summing up most of the changes twice, because changes made on regular commits are replicated on merge commits. Moreover, since FSTMerge calls *diff3* to merge changes inside methods and constructors, besides computing the number of changed method and constructor nodes, we also compute the number of line chunks, and number of lines changed inside methods and constructors, so that we can have different metrics to compare.

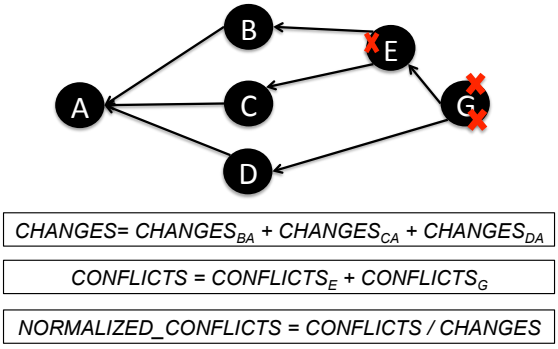

Fig. 4 Computing the number of conflicts, and the probability of ending up with conflicts while editing different language syntax elements.

3.3 Sample

To select our subjects sample, we used GitHub's advanced search page,[5] and configured the search to filter Java projects with more than 500 stars ordered by projects' recent activity. By filtering GitHub search by the number of stars we likely select more meaningful and popular projects, avoiding toy projects. From this search results, we randomly selected 123 projects out of 1,963.

Although we have *not* systematically targeted representativeness or even diversity (Nagappan et al., 2013), by inspecting our sample we observe some degree of diversity with respect to the following dimensions: size, domain, and number of collaborators. Our sample contains projects from different domains such as databases, search engines, games, and frameworks. They also have varying sizes. For example, SimianArmy, a cloud computing tool suite from Netflix, has only 4 KLOCs, while Osmand, a navigation application, has approximately 640 KLOCs. Moreover, Exhibitor has 20 collaborators, while Cassandra has 112 collaborators. Besides that, we also selected projects that are widely used by software developers, and that were analyzed in previous studies, including Junit, Jenkins, Cassandra, Gradle, and Voldemort (Kasi and Sarma, 2013; Brun et al., 2013; Cavalcanti et al., 2015). For further information on our sample, we provide a complete subject list on our online appendix (Accioly et al., 2017).

4 Results

In our empirical study we analyze 70,047 merge scenarios considering the entire version history of 123 projects hosted on GitHub. In this section, we present descriptive statistics of the results structured according to the research questions.

4.1 RQ2: How frequently does each merge conflict pattern occur?

To answer *RQ2*, we collected a total of 28,883 conflicts reported by our adapted version of FSTMerge, from the total of 4,141 merge scenarios with conflicts on Java files. Figure 5 describes the conflict pattern distribution. We found out that EditSameMC was, by far, the most frequent conflict pattern, representing 84.57% of the collected conflicts. The second most frequent pattern was EditSameFd, followed by SameSignatureMC, AddSameFd, ModifierList, ExtendsList, and ImplementList. Moreover, we did not collect any conflicts from the DefaultValueA pattern, which happens when two revisions edit the same default value of an annotation method declaration.

The lower part of Figure 5 shows the bar chart after removing the potential false positive conflicts, that is, conflicts due to different spacing, and consecutive line edit. A percentage of 28.97% of the collected conflicts were classified in one of these categories. More specifically, 48% of the false positives were due to different spacing, 37.69% due to consecutive line edit, and the remaining 14.31% were due to both reasons. EditSameMC was the pattern that had most occurrences of those conflict types (32.21%). However, after removing spacing and consecutive line edit

[5] `https://github.com/search/advanced`

conflicts, EditSameMC is still the most frequent pattern, representing a total of 80.71% of the collected conflicts, without changing the big picture of our results.

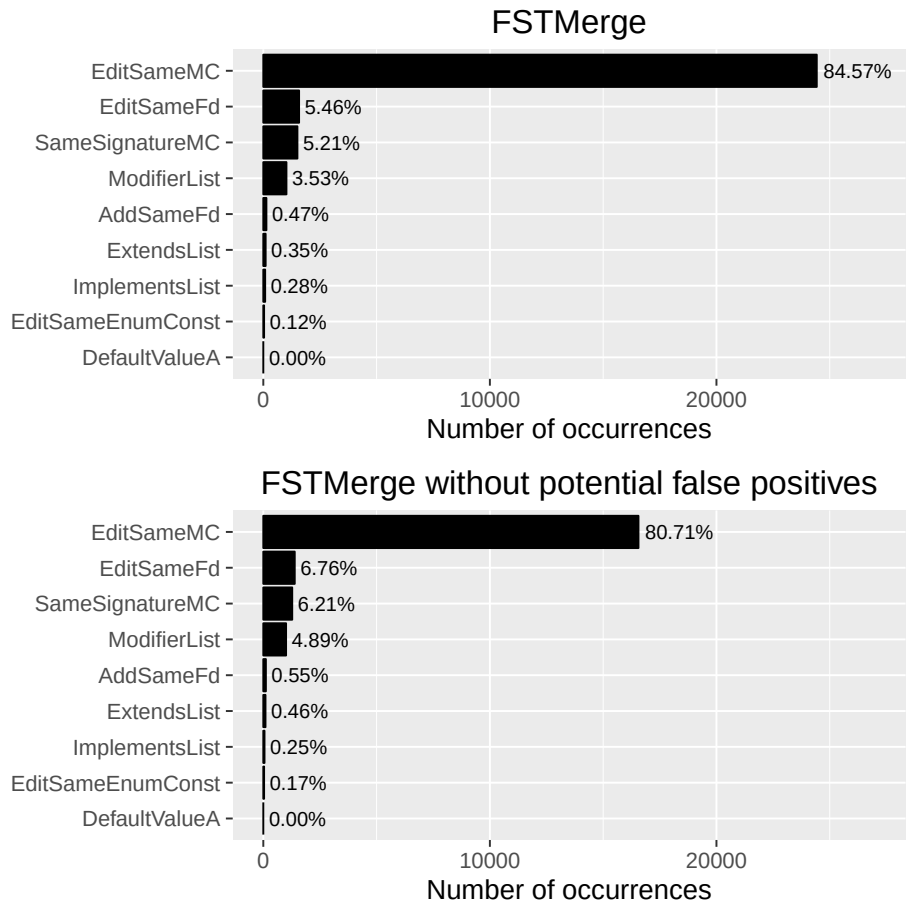

Fig. 5 Bar charts showing the conflicts pattern distribution with and without potential false positive conflicts.

Since those aggregated conflict percentages could be biased by outliers with high occurrence of EditSameMC conflicts, we checked whether conflict pattern occurrences follow a similar tendency across projects. The boxplots in Figure 6 show the conflict pattern percentage distributions considering all projects in our sample after removing the spacing and consecutive line edit conflicts. By analyzing those boxplots, for most, but not all projects, we confirm the same tendency of the aggregated sample of conflicts, since the pattern with higher percentages is indeed EditSameMC. As a matter of fact, we observe that, for more than 75% of the projects from our sample, 69.03% of the collected conflicts were from the EditSameMC pattern. Moreover, the EditSameMC boxplot is slightly skewed to the right, whereas the ones for EditSameFD and SameSignatureMC

boxplots are heavily skewed to the left. This happens because EditSameMC has 22 100%-observations, and only 13 0%-observations, while EditSameFd has no 100%-observation and 54 0%-observations. Finally, SameSignatureMC has 2 100%-observations and 49 0%-observations.

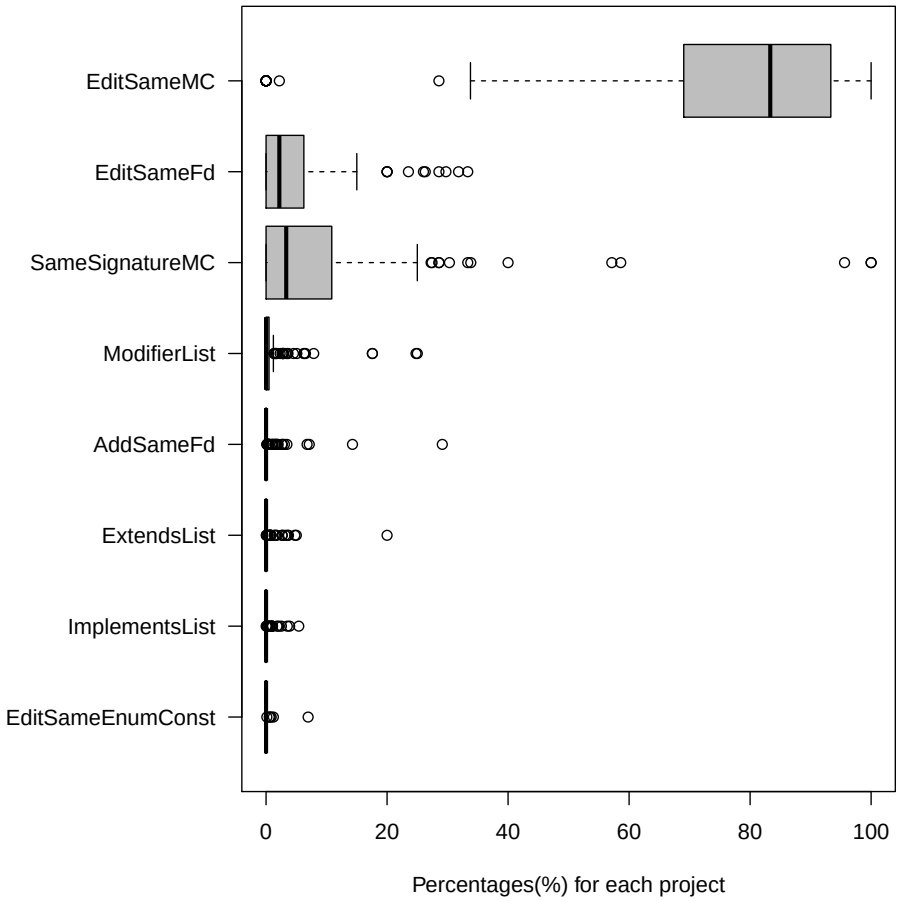

Fig. 6 Boxplots showing the dispersion of the conflict patterns percentages across projects.

4.2 RQ3: What kinds of code changes most likely lead to conflicts?

We answer *RQ3* by computing the ratio between the number of conflicts and the number of nodes changed during the project history (for EditSameMC conflicts we also compute the number of line chunks, and lines changed inside methods), Table 2 summarizes the aggregated results. While EditSameMC Nodes and EditSameMC Chunks are by far the ones more likely of leading to merge conflicts, the others have probabilities lower than 0.1%. To further compare them we used

the Wilcoxon signed rank test (Wilcoxon and Wilcox, 1964) combined with the Bonferroni correction method for multiple comparisons (Bonferroni, 1936). The results show a statistically significant difference when we compare EditSameMC Chunks to EditSameFd, SameSignatureMC, and ModifiersList (p-values < 0.01). However, there is no statistically significant difference when we compare EditSameMC Lines (each line edit inside a method counts 2 changes) to EditSameFd, SameSignatureMC, and ModifiersList.

Table 2 Probability of having merge conflicts while editing different language syntax elements.

Pattern	Probability
EditSameMC Nodes	0.30%
EditSameMC Chunks	0.26%
EditSameMC Lines	0.03%
SameSignatureMC	0.03%
EditSameFd	0.06%
AddSameFd	0.01%
EditSameEnumConst	0.07%
ExtendsList	0.04%
ModifiersList	0.06%
ImplementsList	Approximately 0.00%

Figure 7 top part depicts the boxplots containing normalized number of conflicts per project, computing EditSameMC normalization by the number of changed lines. The lower part of Figure 7 shows the boxplots describing the absolute number of conflicts per project. We can see that in the lower figure boxplots, EditSameMC is by far the most frequent conflict pattern, followed by SameSignatureMC, and EditSameFd. However, in the top graph of Figure 7 although there is not a statistically significant difference between the observations, EditSameFd has indeed higher values than EditSameMC, and SameSignatureMC.

4.3 RQ4: How frequently do merge conflicts occur?

We answer *RQ4* by reproducing 70,047 merge scenarios and computing the conflicting scenario rate, which measures the percentage of merge scenarios with at least one merge conflict (See Section 2). We also compute this metric without considering spacing and consecutive line edit conflicts. Table 3 describes the conflicting scenario rates for a few of the projects from our sample. The complete table is online (Accioly et al., 2017).

Table 4 describes conflicting scenario rate values with and without different spacing and consecutive line edit conflicts. Because our data is not normally distributed we used the Wilcoxon signed rank test (Wilcoxon and Wilcox, 1964) combined with the Bonferroni correction method for multiple comparisons (Bonferroni, 1936) to test some hypotheses. We also measure the effect size for each test as defined by Rosenthal (Rosenthal, 1994), where an effect size of 0.1 means a small effect, 0.3 a medium effect, and 0.5 a large effect. First, we compared the conflicting scenario rate with and without spacing conflicts (p-value < 0.01, effect size $= 0.51$). Then, we compared the conflicting scenario rate with and without consecutive line edit conflicts (p-value < 0.01, effect size $= 0.53$). Considering the

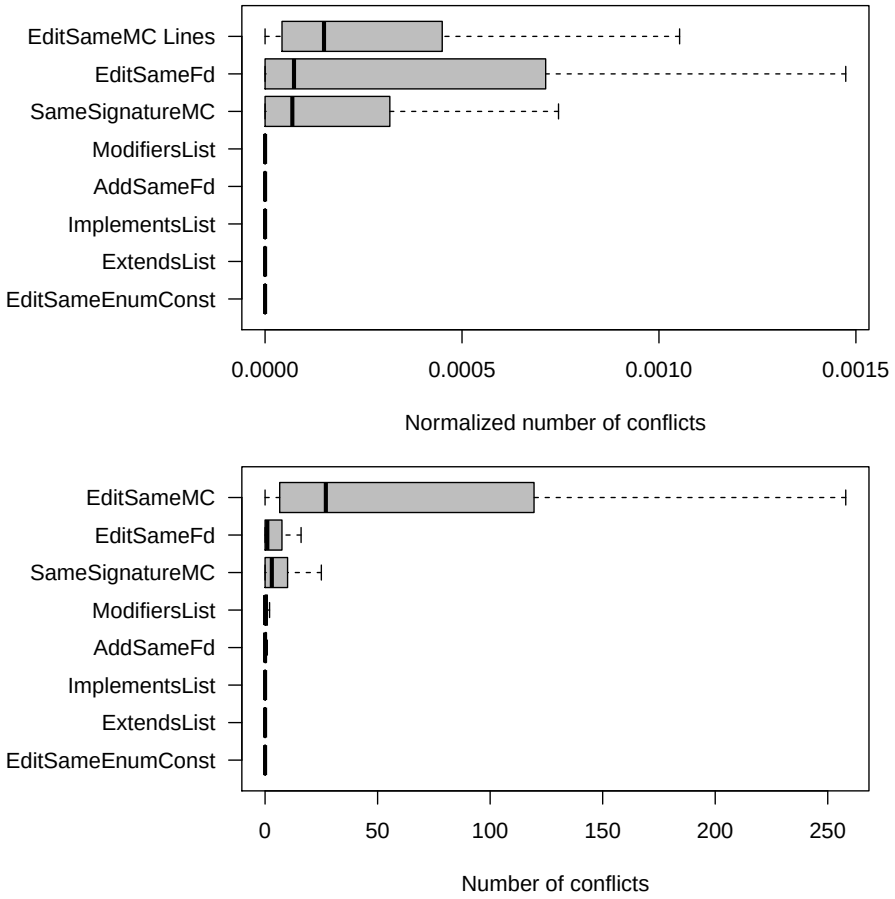

Fig. 7 The top of the image shows the normalized number of conflicts per project boxplots, computing EditSameMC changes by the number of changed lines. Conversely, the lower part of the image shows the absolute number of conflicts per project boxplots.

threshold of 0.01, we reject the null hypothesis on both tests, meaning that removing such conflicts represents a statistically significant decrease on the conflicting scenario rate.

We further analyzed the conflicting scenario rate to check how many merge conflicts occur over the total number of commits. From the 70,047 analyzed merge scenarios, 4,141 (total of 5.91%, with a median of 4.43%, and an IQR of 5.54%) contain conflicts in Java Files. In these scenarios, 28,883 conflicts were detected.

4.4 RQ5: How frequent are the underlying causes of the SameSignatureMC pattern?

Moving on with the analysis, our aim with *RQ5* is to understand why the SameSignatureMC pattern was the third most frequent pattern from our catalog. Figure 8

Project	Size (KLOC)	Merges	CR	CR WFP
Antlr4	11.4	663	8.60%	7.99%
Javaee7-samples	65.5	207	0.97%	0.97%
AndEngine	40.7	115	6.96%	6.96%
Clojure	67	40	12.5%	10%
Elasticsearch	953	2,736	5.77%	5.26%
FBReaderJ	387	1,310	14.43%	13.28%
Graylog2-server	124	1,072	12.31%	12.13%
HoloEverywhere	48.6	82	8.54%	8.54%
OpenTripPlanner	15.9	734	12.67%	10.76%
cgeo	53	2,128	8.46%	7.71%
SimianArmy	4	218	9.17%	6.42%
Titan	251	488	16.19%	12.5%
Orientdb	319	1,752	9.13%	7.25%
Hector	27.6	404	12.62%	9.41%
Nutz	33.1	448	3.57%	2.46%
Hystrix	14.7	543	1.66%	1.47%
Hive	1,003	244	42.21%	39.34%
Netty	182	169	6.51%	6.51%
Dropwizard	17.7	825	2.18%	2.18%
Kotlin	412	588	9.01%	8.67%
ListViewAnimations	8.1	117	8.55%	5.98%
Jsoup	22.2	75	5.33%	4%
K-9	103	566	6.01%	4.95%
Droidparts	10.3	105	2.86%	2.86%
BroadleafCommerce	219	1,061	22.34%	20.74%
JeroMQ	23.4	161	0%	0%
ShowcaseView	2.1	96	9.38%	8.33%
Jmxtrans	17.2	275	2.55%	2.18%
StickyListHeaders	2.8	97	3.09%	3.09%
Retrofit	8.9	526	0.57%	0.38%
Storm	139	1,689	12.31%	11.78%
Eureka	32.7	391	6.65%	6.14%
Eclipse-themes	11.9	57	1.75%	1.75%
Spout	70.5	854	4.8%	3.98%
Druid	160.3	2,061	5.34%	4.85%
Scribe-Java	5.8	91	2.2%	2.2%
Conversations	39.2	514	5.25%	5.06%
Generator-jhipster	19.3	1781	4.72%	4.72%
Mct	131	198	7.58%	5.56%
Commons	67.5	208	1.92%	1.92%
Mongo-hadoop	15.2	92	15.22%	11.96%
Rstudio	494	1,840	5.82%	5.49%
HikariCP	8.5	189	12.7%	12.17%
Jitsi	381	94	9.57%	9.57%
Gradle	550	975	28.72%	28.51%
Bukkit	32.6	19	15.79%	15.79%
Cucumber-jvm	39.9	560	16.61%	14.82%
Cxf	868	130	3.85%	3.08%
Aws-sdk-java	1,044	146	4.11%	2.05%
Groovy-core		674	9.64%	9.2%

Table 3 Examples of projects from our sample. CR means conflicting scenario rate considering all files in the revisions, and WFP means without false positives, that is, spacing and consecutive line edit conflicts.

shows the distribution of the underlying causes for this pattern. We notice that more than half of the occurrences happened because files existing in both left and right revisions, did not exist in the base revision. In such cases, the entire file was

Table 4 Conflicting Scenario Rate Description. DS means different spacing conflicts, CL means consecutive line edit conflicts, and IQR means interquartile range.

	Total	**Median**	**IQR**
CR	9.38%	6.64%	8.81%
CR Without DS Conflicts	9.04%	6.50%	8.72%
CR Without CL Conflicts	8.64%	6.39%	7.76%
CR Without DS and CL Conflicts Conflicts	8.39%	6.00%	7.21%

copied from one repository or branch to another. Thus, for each method that was edited by at least one of the subsequent commits, there was a conflict. The second most frequent causes for method duplication were small methods, such as getters and setters, followed by copied methods, and renamed methods. Finally, a total of 6.4% of the SameSignatureMC conflicts were not classified in any of the previous categories.

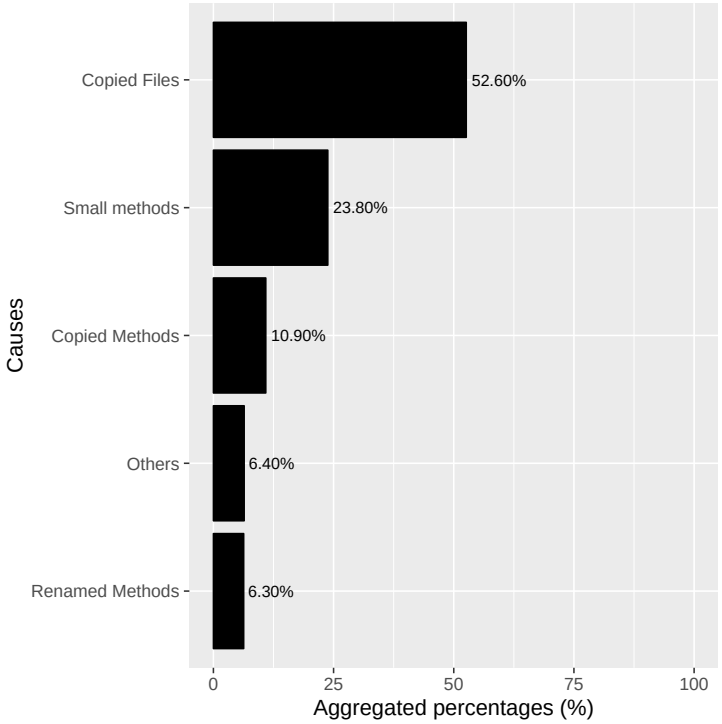

Fig. 8 SameSignatureMC different causes frequency

Another interesting aspect that we have noticed during the manual analysis conducted to understand the underlying causes of the SameSignatureMC pattern, is that developers made conflicting commits between her own branches. In fact, there are legitimate reasons for this workflow, as there are legitimate reasons for conflicting changes between different developers branches. For this reason we de-

cided to run a complementary analysis to learn to which extent do merge conflicts occur involving a single developer, two developers, and more than two developers.

We consider that conflicting merge scenarios, and merge conflicts, involving contributions from a single developer are likely less problematic to resolve than the ones involving more than two developers. In addition, more than 50% of the SameSignatureMC conflicts happened because entire files were copied from one workspace to another, changed, and then merged back together. If a single developer was involved in such a operation, it is more likely that she copied files across her own branches, which could be considered less problematic. But if two or more developers were involved, no matter if the copies were made across branches or repositories, this could be more problematic. Because of that we also check the number of developers involved in merge scenarios with SameSignatureMC conflicts caused by copy of files.

To accomplish such task, we collected, for each analyzed merge scenario, the number of different developers who authored commits between the merge commit and its base commit. We then classified the merge scenarios into three categories: single developer scenario, two developers scenario, and more than two developers scenario. With this data we answer the following questions:

1. How many developers are involved in merge scenarios, conflicting merge scenarios, and merge conflicts?
2. How many developers are involved in SameSignatureMC conflicts caused by copied files?

Table 5 describes data percentages resulting from the analysis to answer question 1. We describe the data using the total percentage, the observed median and the interquartile range. Furthermore, we used the Wilcoxon signed rank test combined with the Bonferroni correction method for multiples comparisons to test hypotheses comparing the number of developers and considering merge scenarios, conflicting merge scenarios, and merge conflicts. Table 6 describes the adjusted p-values, and effect sizes for each test.

Table 5 Description of the percentages in our data considering the number of developers (one, two, and more than two). IQR means interquartile range.

	Merge Scenarios			**Conflicting Merge Scenarios**			**Merge Conflicts**		
	Total	Median	IQR	Total	Median	IQR	Total	Median	IQR
One Dev	6.53	5.19	8.92	6.39	1.23	9.21	2.56	0.00	4.22
Two Devs	27.84	34.44	20.92	12.77	16.66	25.78	6.20	7.69	25.00
> Two Devs	65.63	55.36	28.42	80.84	75.00	36.89	91.24	87.50	46.78

We also wanted to analyze the number of developers involved in merge scenarios containing SameSignatureMC caused by copied files. We believe that if only one developer was involved, more likely she copied files across her own branches, which could be considered less problematic. But if two or more developers were involved, even if the copies were made across branches or repositories, this could

	One Developer vs Two Developers	One Developer vs More than Two Developers	Two developers vs More than Two Developers
N of Developers Involved in Merges	p-value<6.60e-16 eff. size=0.57	p-value<6.600e-16 eff. size=0.56	p-value=1.60e-06 eff size=0.32
N of Developers Involved in Conflicting Merges	p-value=2.44e-08 eff. size=0.36	p-value<6.60e-16 eff. size=0.56	p-value<6.60e-16 eff. size=0.57
N of Developers Involved in Merge Conflicts	p-value=8.38e-07 eff. size = 0.32	p-value<6.60e-16 eff. size=0.56	p-value=1.64e-14 eff. size=0.47

Table 6 Description of the adjusted p-values and their corresponding effect sizes according to the comparison being made, and the research question

be more problematic. In our data, 121 merge scenarios from 56 different projects had conflicts that happened because files were copied. From those merge scenarios, 20.66% involved a single developer, 14.87% involved two developers, and the remaining 64.47% involved more than two developers (the median was 4 developers, and the IQR was 7 developers).

5 Discussion

In this section, we discuss the consequences of our results, and actions they support.

Most merge conflicts happen when developers edit the same lines of the same method. However, perhaps awareness tools should be more careful with class field, and modifier list edits as well.

RQ2 results point that most merge conflicts— 84.57% of the collected conflicts, and 80.71% after removing spacing and consecutive line edit conflicts— happen because developers edit the same or consecutive lines of the same method or constructor. At first this result might seem obvious due to the intuition that most part of the Java code is inside methods. Thus the probability of conflicts occurring inside methods or constructors would be higher. However, we achieved such results because we used a more sophisticated merge tool. If we had used a line-based merge tool like the previous studies that measure the conflicting scenario rate (Brun et al., 2013; Kasi and Sarma, 2013), a significant part of the collected conflicts would likely be ordering conflicts (Apel et al., 2011; Cavalcanti et al., 2015), contradicting the initial reasoning. Also, FSTMerge captures the SameSignatureMC pattern that line-based merge tools do not, which increases the recall of our numbers. So far, there has been no evidence about the frequency for this type of conflict. Finally, we are not aware of previous studies providing empirical evidence about the distribution among different conflict patterns considering the granularity of edited language syntax elements.

Such results can be useful to help awareness tools becoming more efficient. For example, although we have not implemented and validated awareness tools

considering different conflict predictors, we hypothesize that a tool monitoring developers working on different repositories, identifying when they edit the same method, and alerting them, would likely have a reasonable recall since it would detect most merge conflicts (approximately 85%).

However, our biggest concern about driving conclusions based only on *RQ2* results is that there is no baseline about the proportion of changes made to the repository and the number of conflicts. Most conflicts reported by FSTMerge involve method declarations, but that could happen not because method changes are more problematic but just because most changes occur in method declarations. Perhaps, when comparing the frequency of conflicts against this baseline, our results could change. For this reason we add *RQ3* in this study. We believe that *RQ3* complements *RQ2* results because while the unnormalised data is more useful for driving awareness tools towards preventing a larger part of the conflicts, normalising the data is useful to understand the precision of each kind of code change as a conflict predictor.

In fact, after executing *RQ3* analysis we found that not only editing the same method, but editing the same class field and modifier list is an important predictor to consider when trying to prevent conflicts. As a result, the hypothetical tool we describe could also warn about the possibility of EditSameFd, and ModifiersList conflicts with low risk of being wrong. This way, developers could communicate early and avoid the occurrence of such conflicts.

In practice, *RQ2* and *RQ3* combined results might be helpful to improve existing awareness tools as well. For example, Palantír (Sarma et al., 2012), a workspace awareness tool, informs different developers of ongoing activities in the same repository. It proactively detects merge conflicts by informing when developers edit the same files using a metric based on the number of lines changed. As developers edit more lines of the same file, the higher is the risk of ending up in merge conflicts. However, such metric could potentially report false positives. For instance, when developers implement independent methods in the same class. As an improvement, one could add different types of alarms to alert developers in the presence of EditSameMC, EditSameFd, and ModifiersList patterns. This way, Palantír is still able to report most part of the conflicts, but it avoids false alarms.

Besides Palantír, Crystal (Brun et al., 2013) proactively integrates commits from developer repositories with the purpose of warning them if their changes conflict. To produce conflict information sooner, Crystal has to run its analysis often. This can be expensive because it might involve complex build and testing activities. To mitigate this problem, Crystal could use our conflict patterns as predictors for conflicts. Then it could process code contributions before performing the integration routine to check if they contain the most frequent patterns, and, in case they do not, Crystal could delay the integration until the subsequent time period. Although this suggestion likely reduces the cost of running Crystal, further studies are needed to verify if it does not compromise Crystal's accuracy regarding build and test conflicts, which are beyond our scope here.

Finally, Syde (Hattori and Lanza, 2010) is a tool that provides team awareness by capturing developers' editions as atomic AST changes and warns developers if they change the same nodes. Syde basically uses an approach that is very close to the hypothetical tool we propose in this study to detect conflicts in real time. However, as described by Syde authors, information overload could be a problem. Perhaps in an industrial environment with very large teams and many simultane-

ous changes Syde could overload developers with potential conflicts information which could impair their productivity. One possible solution to mitigate information overload in those contexts would be to capture changes concerning only methods, class fields, and modifiers as our results indicate that such changes are the most likely to lead to merge conflicts.

Sophisticated merge tools reduce conflicts and might improve productivity and quality

Compared to previous studies, our results show lower conflicting scenarios rate values. Kasi and Sarma (Kasi and Sarma, 2013), and Brun et al. (Brun et al., 2013), respectively show average conflicting scenarios rates of 14.38%, and 17%, while the median of our conflicting scenarios rate was 6.64%. Moreover, by using a slightly improved merge algorithm to remove spacing and consecutive line edit conflicts, the median drops to 6%. Such difference is likely due to the adoption of FSTMerge to merge Java files, naturally reducing the number of reported conflicts compared to line-based merge tools. This result reinforces the evidence provided by previous studies that investigates the benefits of adopting semistructured merge tools (Apel et al., 2011; Cavalcanti et al., 2015).

Thus we believe the adoption of our adapted version of FSTMerge could help to further increase not only development productivity, since developers could spend less time dealing with spurious conflicts (Bird and Zimmermann, 2012), but also product quality, given that a frequent cause of integration errors are merge conflicts that are not resolved correctly.

Finally, our results show that 90.68% of the merge scenarios has less than 10 merge conflicts which could be considered less problematic from a quantitative perspective. However, the conflict resolution effort depends on the nature of the conflicts, as fewer conflicts do not necessarily mean less effort resolving them. For example, our results show that most merge conflicts involve more than 2 developers' contributions, which suggests that resolving merge conflicts might not be simple. However, Menezes (Menezes, 2016) achieved similar numbers when he analyzed the distribution of conflict chunks, using a traditional line-based merge tool. He reports that most failed merges involved just 4 or fewer conflicting chunks, and more than half involved 1 or 2 conflicting chunks.

Depending on the project development practices, we might have only been "scratching the surface" on the number of conflicts

We also analyzed some of our sample outliers— projects with a conflicting scenario rate significantly higher or lower than the median— to understand factors that might have influenced such disparity. During this analysis, we noticed that 14 projects, including Cassandra and Hive, had higher conflicting scenario rates, comparable to those of the previous studies (higher than 16%). If we had not used a more advanced merge tool, those rates might have been even higher. By manually analyzing 4 of those projects— namely, Cassandra, Hive, Roboguice, and BroadleafCommerce— we observe that these higher rates are accompanied by a greater number of collaborators working independently at the same period of time,

and pushing their commits directly to the main repository instead of performing pull requests. Such practices resemble the development environment of centralized version control systems such as *SVN* and *CVS* (Gousios et al., 2014). Particularly, Cassandra has a patch-based contribution process, with no specific strategy to avoid conflicts.[6]

Alternatively, projects such as JeroMQ and Dagger have no conflicts on Java files. In fact, after removing spacing and consecutive line edit conflicts, a total of 10 projects from our sample turned out to have no conflicts on Java files. We suspect, but have no hard evidence, that projects with no different spacing conflicts might use tools that fix code indentation before commits. However, by analyzing 4 of those projects— Generator-jhipster, Exhibitor, JeroMQ, and OkHttp— we observed that this happened mainly for two reasons. First, projects such as Generator-jhipster only merge contributions via pull requests and after *rebasing*— a Git operation that effectively integrates code without creating a merge commit or leaving any trace about a merge being performed. This practice is explicitly mentioned in their contribution guide.[7]. Rebasing is a frequent practice in a development model known as pull-based software development, commonly used in version control systems such as Git. In this development model, instead of pushing changes to a central repository, developers work locally and register pull-requests to the master repository (Gousios et al., 2014). Then, the repository administrator reviews the pull-request and merges it to the master repository.

The second reason for having a low conflicting scenario rate is that, for some of the projects, in spite of their popularity and large number of registered contributors in the project's Github page, only one or two contributors were significantly active at the same period. This is the case of Exhibitor, which had 20 registered contributors, but only one of them was responsible for 72% of the commits. In contrast, OkHttp, which has a very low conflict rate (0.25%) had more than one active contributor but they contributed on different periods of time, so their work never really interfered with each other.

In summary, the development model used (pull-based together with rebase vs. push to shared repository) may affect the number of merge commits in history. The pull-based model, together with the systematic use of Git commands that rewrites commits history, such as *rebase*,*squash* and *cherry-pick*, decreases the number of merge commits. Nevertheless, conflicts are still being solved locally, which means that our empirical results represent a lower bound for the actual number of merge conflicts. Based on Zimmermanns analysis (Zimmermann, 2007) of systems in CVS where 23% to 46% of file integration leads to merge conflicts, we believe that running our study on a centralized version control system would show an increased number of conflicts and conflicting merge scenarios.

Developers do not take full advantage of proper code version and end up creating conflicts

In order to answer *RQ5* we analyzed the occurrences of SameSignatureMC conflicts in our sample to understand their underlying causes. In fact we did not expect

[6] http://wiki.apache.org/cassandra/HowToContribute

[7] `https://goo.gl/XQyygC`

that it would be so common for developers working on different assignments to add methods with the same signature. Our automated analysis done with a total of 1,505 conflicts of the SameSignatureMC pattern, shows that 63.5% of these conflicts happened because developers copied methods, or entire files, from one repository or branch to the other. We even observed curious cases where the same developer, working on different repositories, copied methods across them. As explained before, this is not the case of code cloning, since the developer copied that same piece of code from other branch to her branch on the same class that it was before. We believe the idea is to reuse pieces of code from branches that were not meant to be merged— different products' branches, for example— or they were not ready to be merged yet— the feature was not fully implemented and tested. Furthermore, the developer might have simply postponed the entire merge process to avoid having to deal with conflicting changes at that moment.

Either way, our results show that copy and paste across different branches or repositories is a common practice. This evidence suggests that developers do not take full advantage of proper code version, but rather copy and paste code around creating the risk of conflicts. Such finding supports the need for tools that enable partial merges, where developers, instead of merging entire sequences of commits, can break commits into smaller parts/pieces of code and then choose what commits they want to merge. Breaking commits into smaller changes is not a new idea. In fact, tools that "untangle" commits, often containing a bundle of unrelated changes, into smaller commits containing few logical units of changes, together with a more descriptive message, have been proposed (Barik et al., 2015; Dias et al., 2015). For example, the goal of Commit Bubbles, and EpiceaUntangler is to help developers to build systematic commit histories that adhere to version control best practices. Moreover, Codebase Manipulation (Muslu et al., 2015), is a tool that automatically records a fine-grained history and manages its granularity by applying granularity transformations. In addition to such tools, we suggest a partial merge tool where developers that already know which code parts (methods or files) they need at that moment, are able to isolate them in a different commit, and merge just those selected commits to their local repository/branch.

Conversely, besides copying pieces of code, an additional 6.3% of SameSignatureMC occurrences happened when a method from the base revision was equally renamed on both derived revisions. At first this seemed like an odd coincidence, but through a manual analysis we found that this was often due to a renaming in an API method, and, as a result, when the dependency is updated, it breaks the build across different repositories. Consequently, developers have to fix both the method's name, and its calls, on their local branch to successfully compile the code. For those renaming cases, or other refactoring related changes, a mechanism that allows "broadcasting" refactoring related changes across repositories could help. Of course, changes would be applied to a repository only when the developer accepts the patch. This way, developers would not need to reproduce the same code changes in different repositories. This is then extra evidence for the need of better supporting refactorings in APIs evolution (Dig and Johnson, 2005). For example, Catch up! (Henkel and Diwan, 2005), a tool that uses descriptions of refactorings to help application developers migrate their applications to a new version of a component, could be extended to support the cases we have observed.

Alternatively, a total of 23.8% of the occurrences were simple methods such as getters, setters, or methods with less than 3 lines of code. This situation we believe

is more reasonable to expect. Two developers might independently feel the need for adding a *get* or an *equals* method to the same class. However, through a manual analysis we saw that some of them were copied or equally renamed methods as well, but because they had few lines of code, we did not run the analysis of copied and renamed methods on them.

Finally, the remaining 6.4% of the SameSignatureMC conflicts did not fit in any of the previously defined categories. Through manual analysis, we observed that in some cases the methods were copied or renamed as well. However, because they were significantly changed, our string similarity algorithm returned a score smaller than our threshold (70%). Nevertheless, most of the manually analyzed cases really reflected the name of the pattern— developers indeed added complex methods with the same signature and different behavior. Furthermore, we noticed that those methods' names often contained common words from developers' vocabulary such as *initialize*, *execute*, *run*, and *load*. In the example we described back in Section 2.5 illustrates a duplicated method called "sendFile" which could be a recurrent name for methods from an instant messenger application. For such cases, we could improve awareness tools to alert when developers add methods with the same signature, so that they can communicate and solve this conflict earlier.

Merge scenarios, conflicting merge scenarios, and merge conflicts usually involve more than two developers

The bottom line of the analysis collecting the number of developers involved in merge conflicts is that those conflicting scenarios involving a single developer that we found while manually investigating underlying causes for SameSignatureMC conflicts are not so common after all. In fact, our data indicates that merge scenarios, conflicting merge scenarios, and merge conflicts often involve more than two developers. We also observe this tendency when analyzing merge scenarios containing SameSignatureMC conflicts caused by copied files.

Although the number of developers involved in merge conflicts does not measure directly the effort to resolve them, we believe that solving conflicts involving a single developer is probably easier than solving conflicts involving different developers. Moreover, Costa et al (Costa et al., 2016) reported that developers usually have a hard time while branching merges because it might hold numerous contributions from different developers and they need to understand changes in order to integrate them. Based on this problem they propose a tool called TIPMerge, which recommends expert developers for integrating changes across branches. Our work reinforces their findings.

6 Threats to Validity

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

6.1 Construct Validity

A possible threat to the construct validity of our study is our choice of metrics. We tried to mitigate this threat by using a suite of metrics that gives us alternative views. For example, to learn about the most frequent conflict pattern, besides computing the number of conflicts, we also compute the normalized number of conflicts to complement our results.

Moreover, to learn about the frequency of conflicting merges in *RQ4*, we measure the proportion between conflicting merge commits and merge commits. This metric was used in well established studies in the area (Kasi and Sarma, 2013; Brun et al., 2013). This way, we were also able to compare our results with theirs.

A different alternative to learn about conflicts' frequency would be to measure the ratio between conflicting merge commits and commits in general. However, we believe that such metric is not appropriate. For example, consider that one developer committed 8 times while performing task A, and a second developer committed 1 time while performing task B. Then, someone merged task A and task B contributions into a merge commit which resulted in a conflict. In this case, the conflict frequency would be 10% (1 conflicting merge commit out of 10 commits). However, if the second developer had the habit of making smaller and more frequent commits, the conflict frequency would decrease, but, in the end, his contributions would have conflicted with the first developer contributions regardless of the number of commits.

In summary, depending on developers habits, they might commit too often or too rarely and this metric would vary according to that. Meanwhile, by analyzing the proportion between conflicting merge commits and merge commits we have a better notion of how often developers contributions conflict with each other.

6.2 Internal Validity

In this work we analyzed 123 Java projects from Git. Three projects from our sample (JeroMQ, Dagger, and Closure-compiler) had no merge conflicts, which could affect the results for our conflicting merge scenario metric. However, the sum of their merge scenarios represents only 1.07% from our sample. By removing these projects from the analysis, the total conflicting merge scenario rate drops from 9.38% to 9.37%, without changing the values of the median (6.64%), and the IQR (8.81%).

Differently from previous studies, which used line-based merge tools, we used FSTMerge which is a semistructured merge tool with some knowledge about the underlying syntax of the artifacts. Thus FSTMerge is able to automatically solve ordering conflicts. In addition, by using FSTMerge we were able to systematically generate our conflict pattern catalog. Nevertheless, the decision of using our adapted version of FSTMerge also brings drawbacks to our analysis. Although it removes a large number of false positives (Apel et al., 2011), it might add small numbers of false negatives and other kinds of false positives, as we discuss next.

The added false negatives might happen when two developers independently add *import* declarations involving different *packages* and the same member name. For instance, if developer A adds *java.util.List*, and developer B adds *java.awt.List*. When using FSTMerge to integrate those contributions, it will treat this case as an

ordering conflict. FSTMerge will order the import list declarations, likely leading to a build conflict (type ambiguity error). Alternatively, if we used a line-based tool and the described contributions were added in the same line (or in consecutive lines), the conflict would be reported and the developer responsible for the integration could resolve it before it becomes a build problem. Moreover, a different type of false negative might happen when one developer adds a method that calls a second method that was edited by another developer, which could lead to a semantic conflict. Likewise, if those developers edit the same or consecutive lines of the same text area, the line-based merge tool would report this conflict, while the FSTMerge would not. To measure if two types of false negatives would have occurred frequently in our data, we further analyzed all merge scenarios of 50 Java projects from our sample, and observed that, from all the merge scenarios, only 1.66% had changes matching those patterns, and are, therefore, false negatives. Thus, such conflicts do not happen very often. Nevertheless, FSTMerge could be slightly improved to detect such cases.

Regarding the possibly added false positives, FSTMerge fails to identify renaming changes. If a program element such as a method is renamed in one revision, the FSTMerge algorithm is not aware of this fact and cannot map the renamed method to its previous version, and it considers that the method was removed. If the method that was renamed in one revision, is edited by the other revision, FSTMerge will report a conflict. Conversely, a line-based tool would report a conflict only if the same or consecutive lines were edited. This means that a percentage of the EditSameMC conflict occurrences that we collected might fall into this category and, therefore, be false positives, possibly affecting some of our more detailed findings, such as the normalized results. It is hard to guess whether a version of FSTMerge that properly handles renaming would improve our findings. In fact, avoiding renaming conflicts could lead to new kinds of false negatives. So fixing FSTMerge to properly handle renaming would demand careful evaluation of the renaming detection strategy. In this paper, we decided to compute a conservative (overestimated) number of renamings to check if, even considering more renamings than expected, our main results would remain the same.

We collected a total of 24,427 EditSameMC occurrences. From this total, 9,206 might be false positives due to the renaming issue. This is an overestimation because FSTMerge cannot discern a deletion of an element from a renaming. To have evidence that the FSTMerge renaming issue did not affect our main conclusions, we made a preliminary analysis using a subsample of 60 projects from our original sample to check for references on the renamed or deleted method. We observed that 30.21% of renamed methods occurrences seems to be false positives, but this is also an overestimation. So, considering that 30.21% of those 9206 occurrences are false positives, the percentage of EditSameMC conflict drops from 84.57% to 82.92%. Consequently, even with this overestimated amount of false positives, EditSameMC conflicts would still be the most frequent conflict pattern by far from our sample, without compromising our general results.

Like previous work such as Kasi and Sarma, and Brun et al., we analyzed *Git* projects, which support commands such as *rebase*, *squash*, and *cherry-pick*, that rewrite project development history. Consequently, depending on the development practices of each project, we may have lost merge scenarios where developers had to deal with merge conflicts, but that do not appear on *Git* history as merge commits (Bird et al., 2009). When those commands are used in a systematic way

they might dramatically decrease the number of merge commits. Consequently, to analyze all merge scenarios, we would need to have access to private repositories. So, our results are actually a lower bound for the real conflicting scenarios rates. In fact, our assumption is that if we use a centralized version control system such as SVN, the number of conflicts and conflicting merge scenarios would increase. However, studying SVN history is challenging in our context because there is no systematic way to precisely select merge scenarios; SVN has no standard log entry type for merges. Previous studies such as Apel et al (Apel et al., 2011) look for commit messages that suggest a commit is the result of merging, but that might be imprecise. So, unless one carefully filters the merge scenarios, the analysis could be biased.

Besides that, we could have used different merge tools to extract our pattern catalog. For example, JDime (Apel et al., 2012) is a merge tool that tunes the merge process on-line by switching between unstructured and structured merge, depending on the presence of conflicts. However, JDime has the same disadvantages of FSTMerge (renaming and import declaration problems). Moreover we managed to remove some of the false positives of FSTMerge that JDime can solve (spacing and consecutive line conflicts). Furthermore, JDime inserts new false negatives with respect to FSTMerge. For instance, if both revisions edit different parts the same field declaration— the type definition, and the initialization— JDime will solve this conflict, most likely leading to a build or test conflict. As a final point, we would not be able to use JDime's autotuning strategy because we would risk missing the false negatives of the line based merge. For example, we would miss the occurrences of SameSignatureMC conflicts when the duplicated methods are added in different areas of the file.

Finally, regarding our consecutive line edit conflict analysis, for most of the cases, the edited lines can be merged safely. Nevertheless, there are cases were this merge might lead to a build or test conflict. For example, if a string variable is initialized in two consecutive lines (by string concatenation), and those lines were edited, the revisions would be editing the value of the same variable. If such lines as merged, this could lead to a semantic conflict. Thus, further studies are needed to analyze how frequent this situation happens. Perhaps, with structured merge tools, we could assess if the edits made to consecutive lines inside methods belonged to the same statement or variable initialization. In case they did not, we could perform the merge successfully. Nevertheless we showed that, by removing just the spacing conflicts there is a statistically significant difference in the conflicting scenarios rate.

6.3 External Validity

Our sample contains only open source Java projects hosted on GitHub. We only used Java projects for simplicity. Furthermore, by choosing only popular projects— projects with more than 500 stars on GitHub, we might miss diversity in our sample. To analyze projects in different languages, we would have to derive different catalogs as well. Thus, generalization to other languages and other version control systems is limited, and further studies would be needed to confirm our findings, as discussed in Section 8. However it is not hard to see that some of our main patterns, like editing the same method or the same class field, could also be present

in other object-oriented languages similar to Java. It would be harder to automatically solve spacing conflicts for languages such as Python, given that indentation affects semantics. Also, the implications for future research discussed in this paper, such as the concept of the tool that monitors developers, the concept of partial merges, and the refactoring broadcast mechanism could be useful for a number of different languages using different types of version control systems. Furthermore, Kalliamvakou et al. (Kalliamvakou et al., 2015) showed in their survey that GitHub, with its common development practices (pull-based development), is being increasingly adopted in commercial projects as well.

7 Related Work

In this section we describe some of the previous studies that we use as base evidence for our study, and related work divided by their different topics.

Empirical Studies

A number of empirical studies provided evidence about collaborative development issues. In previous sections we have mentioned some of them. Khasi and Sarma, and Brun et al. (Kasi and Sarma, 2013; Brun et al., 2013) who reproduced merge scenarios from different GitHub systems with the purpose of measuring the frequency of merge scenarios that resulted in conflicts. In addition, they also studied the frequency of other types of conflicts, such as build and semantic conflicts. Moreover, Zimmermann did a similar analysis, but with a different metric, as the author reproduced files integration from CVS projects (Zimmermann, 2007). He found that files integration on CVS usually conflicts in a range of 23% to 47%. Regarding software merging techniques, Mens (Mens, 2002) provided a comprehensive overview of the field, and suggested directions for future research. Among them, he claimed for the need of a detailed but language independent taxonomy of the kinds of changes, and corresponding conflicts, that can be made to software. Finally, Perry et al (Perry et al., 2001) made an observational case study to analyze the effect of parallel changes on a large-scale industrial software system. They reported that, although 90% of the files could be merged without problems, the degree of parallel changes is high— merge conflicts involved between 2 to up to 16 parallel changes — we also found similar results, since most part of merge conflicts (91.24%) involved more than two developers.

In our work we also assessed the frequency of merge conflict occurrences in different projects, but, contrasting with previous studies, we use a much larger sample, and a semistructured merge tool that avoids a large number of spurious conflicts often reported by typical line-based tools that are still used in practice. In addition we also derive a conflict pattern catalog and measure how frequently those patterns occur, and the probability of having a merge conflict while editing different language syntax elements. Moreover, we bring evidence about other problems that developers often face while working collaboratively. For example, the conclusion that developers often need to copy pieces of code or rename methods across different repositories.

Concerning the cost of resolving conflicts, previous studies have tried to estimate it. For example, Kasi and Sarma (Kasi and Sarma, 2013) estimates conflict resolution effort as the time interval between when a conflict first occurred and until when it was resolved. In other words, the number of days the conflict persisted in the master repository. They report that resolving merge conflicts took substantial effort, typically spanning multiple days. However, this metric assumes that the computed time intervals reflect the efforts of developers working exclusively to resolve the conflict. Which is not always the case. This means that this metric is an over-approximation.

Moreover, a main challenge for estimating conflict resolution effort is that different conflicts might demand different resolution effort. In this sense, Cavalcanti et al (Cavalcanti et al., 2017), while comparing different merge approaches (unstructured and semistructured), estimate the effort to resolve different types of conflicts by evaluating the strategy used by developers while resolving them. They assume that resolutions including only changes from the merged contributions (without new code, nor combination of contributed code) demand less effort. While this estimation is a fair approximation of the time needed to fix the code which is part of the total integration effort it does not consider the time needed to understand the changes, reason about the conflict and then decide how to fix it.

In contrast, other studies do not quantitatively measure the cost of resolving conflicts, but they report, based on experimental observations, that resolving merge conflicts is not so trivial. It might take considerable time, and is an error-prone activity. For example Sarma et al (Sarma et al., 2012) report that developers commonly rush to commit their tasks before others so they would not have to deal with conflicts while pushing their changes to the shared repository. They claim that developers behave like that because they indeed spend significant time dealing with merge conflicts. In addition Bird and Zimmermann (Bird and Zimmermann, 2012) report that a frequent cause for integration errors are merge conflicts that were not resolved correctly.

Studies that estimate the cost of resolving conflicts are complementary to our work, since we investigate merge conflicts frequency. Moreover, we found that merge conflicts usually involve contributions from more than two developers. Thus, although this analysis does not measure directly the effort to resolve them, we believe that solving conflicts involving a single developer is probably easier than solving conflicts involving different developers.

Another empirical study performed by Cataldo and Herbsleb (Cataldo and Herbsleb, 2011) tried to understand aspects leading to conflicts. They presented an empirical analysis of a large-scale project where they examined the impact that software architecture characteristics, and organizational factors have on software integration failures. They concluded that architecture related factors such as the nature and the quantity of component dependencies, as well as organizational factors such as the geographic dispersion of development teams, can lead to higher integration failure rates. Furthermore, Shihab et al. (Shihab et al., 2012) presented an empirical study that evaluated and quantified the relationship between software quality and various aspects of the branch structure used in software projects. They reported that, indeed, the branching strategy does have an effect on software quality and that misalignment of branching structure and organizational structure is associated with higher post-release failure rates. Finally, Estler et al. (Estler et al., 2014), investigated the impact of awareness information in the context of globally

distributed software development. Among their findings, they concluded that insufficient awareness information affects more negatively developers' performance than actual merge conflicts.

Our work complements these works because we also examine factors that relate to integration failures on collaborative development environments. However, we analyse different factors. While Cataldo and Herbsleb analyzed architecture level and organizational factors that lead to integration failures, and Shihab et al. analyzed branching strategies that have an impact on software quality, we analyze which code changes often lead to merge conflicts. Conversely, like Estler et al., our results reinforce the importance of using and improving awareness tools.

Tools for Conflict Detection and Resolution

Tools and strategies to support collaborative development environments use different strategies to both decrease integration effort, and improve correctness during task integration. Throughout this paper we have mentioned two of them. Palantír (Sarma et al., 2012), which informs developers of ongoing parallel changes, and Crystal (Brun et al., 2013), which proactively integrates commits from developer repositories with the purpose of warning them if their changes conflict. Alternatively, when the performed tasks are ready for being merged, TIPMerge (Costa et al., 2016) has an algorithm that recommends developers who are best suited to perform merges considering different metrics such as developers' past experience in the project, their changes in the involved branches, and dependencies among modified files. Finally, given that it is not always possible to detect conflicts before code integration, tools like FSTMerge (Apel et al., 2011), and JDime (Apel et al., 2012) offer solutions to reduce integration effort by automating the resolution of some types of conflict. In contrast, other awareness tools, such as Syde (Hattori and Lanza, 2010), build code artifact ASTs to make the analysis of changes more precise. WeCode continuously merges uncommitted and committed changes to detect conflicts on behalf of developers before they check-in their changes (Guimarães and Silva, 2012). Moreover, Bellevue, is an IDE extension to make committed changes always visible, and code history accessible inside developers' workspaces (Guzzi et al., 2015). Finally, Cassandra (Kasi and Sarma, 2013) is a tool that analyzes task constraints to recommend an optimum order of tasks execution.

Our work brings evidence that reinforces the need of using such tools, besides providing new insights to their improvement, or even to come up with new strategies. In addition, we also provide small improvements to FSTMerge algorithm, together with empirical evidence that they indeed improve FSTMerge's results.

8 Conclusions and Future Work

When working in a collaborative development environment, developers implement different tasks in an independent way. Consequently, during the integration, one might have to deal with conflicting changes. Previous studies indicated that conflicts occur frequently, and impair developers' productivity. In this paper, to understand the structure of the changes that lead to conflicts, we derived a conflict

catalog with 9 patterns expressed in terms of the performed kinds of changes considering involved syntactic language structures. To assess the occurrence of such patterns in open-source systems, we conducted an empirical study reproducing 70,047 merge scenarios from 123 GitHub Java projects. Furthermore, we focused on conflicts reported by a semistructured merge tool, avoiding a large number of spurious conflicts often reported by typical line-based merge tools.

Our results show that 84.57% of merge conflicts happen because developers edit the same lines, or consecutive lines of the same method. However, editing methods, class fields, or modifier lists have similar probabilities of leading to merge conflicts. This means that, if we improve awareness tools to alert developers in those cases, we might avoid most merge conflicts. In addition, merge conflicts occur in a total of 9.38% of the analyzed merge scenarios. Moreover, by slightly improving the merge algorithm to better handle spacing and consecutive line edit conflicts, we got statistically significant lower numbers. Compared to previous studies, our results show that using more advanced merge tools reduces the number of conflicting merge scenarios. We also found that developers often copy methods, or even entire files across repositories, which is evidence of the need for tools that enable partial merges. Finally, as a complementary result, our data indicates that merge scenarios, conflicting merge scenarios, and merge conflicts usually involve more than two developers. This result suggests that integrating different branches is not often an easy task since one needs to understand and merge contributions made by different developers.

This work was a first exploration into semistructured merge conflicts' structure. There are several possible directions for enhancements. For example, although we analyzed a large number of merge commits, our results could benefit from replications analyzing other projects, including projects in centralized version control system such as SVN or CVS. Likewise, it would be interesting to replicate our study by deriving a new conflict pattern catalog for a different language, or even for a different merge tool. Moreover, one could answer additional questions with our data. For example, what are the conflict patterns inside method bodies? What percentage of those conflicts involve method signatures or just statements inside the method bodies? To answer this question, one would have to use a diff tool such as GumTree (Falleri et al., 2014), which builds the full AST. This would replicate Menezes (Menezes, 2016) work, which reports that most conflicts involve method invocations, method declarations, variable declarations, commentaries, and if statements.

Moreover, in this study we analyzed merge conflicts' frequency. Another important aspect to analyze is the cost associated to solving merge conflicts. We noticed that all the previous studies that try to estimate conflict resolution effort has either used experimental observations, proxies, or over-approximations (Kasi and Sarma, 2013; Cavalcanti et al., 2015; Sarma et al., 2012; Bird and Zimmermann, 2012). We believe that a solid way to estimate the effort to resolve different types of conflicts would be to conduct controlled experiments where developers have to resolve conflicts while time and other metrics are being measured.

Alternatively one could make a study to analyze our results on a per-project basis, understanding, for example, why some projects have more false positives than others, why some projects have more SameSignatureMC conflicts than others, etc. Other interesting research questions were left outside of scope of this paper,

mainly the ones involving other technical and organizational factors that might influence the presence of conflicts.

Finally, an important process-related question is who is responsible for integrating the merges. Such a decision is likely to influence the merge conflict resolution process. For some of the projects we analyzed, such as Generator-jhipster, the integrator information is not easily available at the project description pages and files. One could maybe try to infer that by making a historical analysis of merge commit authors. This would likely require a rigorous manual analysis to derive heuristics that could be used to answer this question. Another option would be to interview developers.

Acknowledgements We would like to thank the FACEPE (grants APQ 0388-1.03/14 and IBPG 0716-1.03/12), CNPq (grant 309741/2013-0), and CAPES funding agencies for partially supporting this work. We also thank our Software Productivity Group colleagues, and the anonymous reviewers who greatly contributed to improve this work.

References

Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. Online appendix. `http://twiki.cin.ufpe.br/twiki/bin/view/SPG/ConflictPatterns`, 2017. Accessed: 2017-11-01.

S. Apel, Olaf Lessenich, and Christian Lengauer. Structured merge with auto-tuning: Balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012. ACM, 2012.

Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. Semistructured merge: Rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11. ACM, 2011.

Titus Barik, Kevin Lubick, and Emerson Murphy-Hill. Commit Bubbles. In *Proceedings of the International Conference on Software Engineering, New Ideas and Emerging Results Track*, ICSE 2015. ACM, 2015.

Christian Bird and Thomas Zimmermann. 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, 2012.

Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09. IEEE Computer Society, 2009.

C. E. Bonferroni. *Teoria statistica delle classi e calcolo delle probabilità*. Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commerciali di Firenze, 1936.

Y. Brun, R. Holmes, M.D. Ernst, and D. Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, 2013.

Marcelo Cataldo and James D. Herbsleb. Factors leading to integration failures in global feature-oriented development: An empirical analysis. In *Proceedings*

of the 33rd International Conference on Software Engineering, ICSE '11. ACM, 2011.

Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. Assessing semistructured merge in version control systems: A replicated experiment. In *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement*, ESEM'15. ACM, 2015.

Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, 2017.

Catarina Costa, Jair Figueiredo, Leonardo Murta, and Anita Sarma. Tipmerge: Recommending experts for integrating changes across branches. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016. ACM, 2016.

Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stephane Ducasse. Untangling fine-grained code changes. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER 2015. IEEE Computer Society, 2015.

Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05. IEEE Computer Society, 2005.

Eclipse. Jgit user guide. `http://wiki.eclipse.org/JGit/User_Guide`, 2015. Accessed: 2017-06-16.

H Christian Estler, Martin Nordio, Carlo Furia, Bertrand Meyer, et al. Awareness and merge conflicts in distributed software development. In *Proceedings of the IEEE 9th International Conference on Global Software Engineering*, ICGSE'14. IEEE Computer Society, 2014.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering*, ASE'14, 2014.

Free Software Foundation. Diff utils user's manual. `https://www.gnu.org/software/diffutils/manual/diffutils.html`, 2016. Accessed: 2017-06-16.

Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014. ACM, 2014.

Mário Luís Guimarães and António Rito Silva. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12. IEEE Press, 2012.

Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. Supporting developers' coordination in the ide. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '15. ACM, 2015.

Lile Hattori and Michele Lanza. Syde: A tool for collaborative software development. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10. ACM, 2010.

Johannes Henkel and Amer Diwan. Catchup!: Capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05. ACM, 2005.

Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference*

on Software Maintenance, ICSM '94, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-8186-6330-8. URL `http://dl.acm.org/citation.cfm?id=645543.655704`.

Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, Leif Singer, and Daniel M. German. Open source-style collaborative development practices in commercial projects using github. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15. ACM, 2015.

Bakhtiar Khan Kasi and Anita Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13. IEEE Press, 2013.

Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A formal investigation of diff3. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'07. Springer-Verlag, 2007.

Vladimir I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Technical report, Soviet Physics Doklady, 1966.

Gleiph Menezes. *On the Nature of Software Merge Conflicts*. PhD thesis, Federal Fluminense University, 2016. Accessed: 2017-06-16.

Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 2002.

Kivanç Muslu, Luke Swart, Yuriy Brun, and Michael D. Ernst. Development history granularity transformations (N). In *30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15. IEEE Computer Society, 2015.

Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013. ACM, 2013.

Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel changes in large-scale software development: An observational case study. *ACM Transactions on Software Engineering and Methodology*, 2001.

Robert Rosenthal. *Parametric measures of effect size.* Russell Sage Foundation., 1994.

A. Sarma, D. F. Redmiles, and A. van der Hoek. Palantír: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, 2012.

Emad Shihab, Christian Bird, and Thomas Zimmermann. The effect of branching strategies on software quality. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12. ACM, 2012.

Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14. IEEE Computer Society, 2014.

Frank Wilcoxon and Roberta A Wilcox. *Some rapid approximate statistical procedures.* Lederle Laboratories, 1964.

Thomas Zimmermann. Mining workspace updates in CVS. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07. IEEE Computer Society, 2007.