Analyzing Conflict Predictors in Open-Source Java Projects

Paola Accioly, Paulo Borba, Léuson Silva, and Guilherme Cavalcanti
Federal University of Pernambuco
Brazil
prga,phmb,lmps2,gjcc@cin.ufpe.br

ABSTRACT

In collaborative development environments integration conflicts occur frequently. To alleviate this problem, different awareness tools have been proposed to alert developers about potential conflicts before they become too complex. However, there is not much empirical evidence supporting the strategies used by these tools. Learning about what types of changes most likely lead to conflicts might help to derive more appropriate requirements for early conflict detection, and suggest improvements to existing conflict detection tools. To bring such evidence, in this paper we analyze the effectiveness of two types of code changes as conflict predictors. Namely, editions to the same method, and editions to directly dependent methods. We conduct an empirical study analyzing part of the development history of 45 Java projects from GitHub and Travis CI, including 5,647 merge scenarios, to compute the precision and recall for the conflict predictors aforementioned. Our results indicate that the predictors combined have a precision of 57.99% and a recall of 82.67%. Moreover, we conduct a manual analysis which provides insights about strategies that could further increase the precision and the recall.

CCS CONCEPTS

• **Software and its engineering** → *Software configuration management and version control systems*;

KEYWORDS

Collaborative development, awareness tools, conflict predictors, precision and recall

ACM Reference Format:
Paola Accioly, Paulo Borba, Léuson Silva, and Guilherme Cavalcanti. 2018. Analyzing Conflict Predictors in Open-Source Java Projects. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196398.3196437

1 INTRODUCTION

In a collaborative development environment, tasks are commonly assigned to developers that work separately using individual copies of project files. As a result, while trying to integrate different contributions, one might have to deal with conflicting changes. These conflicts might be detected during merging, building, and testing, impairing development productivity, since understanding and resolving conflicts often is a demanding and error-prone task [6, 8, 20, 26, 32].

To alleviate that problem, different awareness tools have been proposed to alert developers about potential conflicts before they become too complex [8, 16, 18, 20, 26]. However, there is not much empirical evidence supporting the strategies used by these tools. Learning about what types of changes most likely lead to conflicts might help to derive more appropriate requirements for early conflict detection, and suggest improvements to existing awareness tools.

A previous work [2] reported that most merge conflicts happen when developers edit the same lines, or consecutive lines of the same method or constructor declaration.[1] They also claim that editing methods is one of the change types that most likely leads to merge conflicts. Conversely, a different study [22] suggested that editions to directly dependent methods are also a frequent cause for conflicts. That is, when one developer edits a method that calls a second method edited by another developer. Although this situation does not lead to merge conflicts, it is reasonable to consider that it might increase the chance of having other types of conflicts, such as build an test conflicts.

Build conflicts happen when the system building process fails after the merge. This happens, for example, when developers independently introduce the same local variable declaration inside the same method body. In contrast, a test conflict happens when merged contributions interact with each other causing the system to have different observable outputs than the system tests expect. One example of a test conflict would be when both developers change the same method, and then, after the merge, one of the test cases started to fail due to conflicting contributions.

It then sounds that a possible strategy to avoid conflicts would be to alert developers when they edit the same method and when they edit directly dependent methods. However, it is possible that developers edit different textual areas of the same method, without causing merge conflicts. Likewise, developers might edit unrelated concerns from directly dependent methods, avoiding build and test conflicts. If such situations happen frequently, the alerting strategy we discuss might raise too many false alarms.

Thus, we need further studies to investigate if editions to the same method— which we refer to as EditSameMC changes from now on—, and editions to directly dependent methods— or EditDepMC changes— are good conflict predictors. In particular, we are interested in investigating these conflict predictors precision, that is, how frequently the conflict predictor presence is associated with

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

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196437

a conflict. In addition, we also want to measure their recall, that is, what percentage of conflicts we avoid by using such predictors.

For establishing build and test conflicts ground truth, we rely on the status of building and testing processes executed by the Travis CI [29] service. Whereas this provides quite precise guarantees for build conflicts, the guarantees for test conflicts are as good as the project test suites. So, even for projects with strong test suites, unexpected interferences between merged contributions might be missed by the existing tests

To this end, in this paper we conduct an empirical study that analyzes 5,647 merge scenarios from 45 Java-maven-travis projects from GitHub to collect instances of merge, build, and test conflicts together with conflict predictors. Then, we compute how frequently a predictor occurrence is associated with a conflict occurrence, and the percentage of conflicts that can be captured by detecting predictor instances.

Additionally, we conduct a manual analysis to understand what other types of changes cause conflicts, and what changes were associated with predictor instances that did not cause conflicts. Based on the collected evidence we derive more appropriate requirements for detecting conflicts early, and suggest improvements to existing conflict awareness tools.

Our results indicate that, considering both conflict predictors together, we achieve a precision of 57.99%. In particular, EditSameMC individual precision is 56.71%, and EditDepMC precision is 8.85%. Moreover, we achieve a recall of 82.67% if we consider both predictors together, while EditSameMC individual recall is 80.85% , and EditDepMC recall is 13.15%.

The manual analysis points out that part of the predictor occurrences in our sample not associated with conflicts are actual missed contributions interferences. For example, in one EditSameMC predictor from project Web Magic[2] while one developer changes an *if* statement condition, the other developer removes code located inside this same *if* statement. This is expected given the limitations of how we establish test conflicts ground truth. Consequently, the precision results we report represent lower bounds of actual contributions interferences, whereas the recall results are upper bounds because other missed interferences not caused by the predictors might have occurred as well.

Such evidence is useful to guide different conflict awareness strategies. For instance, a more conservative strategy would be to alert developers about a large part of potential conflicts at the cost of dealing with some false positives. In this case, warning developers about all predictor occurrences is a reasonable strategy. In contrast, a strategy that aims at precision, even at the cost of loosing conflicts, would be alerting developers about EditSameMC instances only when developers edit the same lines of the same method. Finally, based on our false positives and false negatives analysis, we discuss different strategies that could further increase the predictors' precision and recall.

2 ANALYZING CONFLICT PREDICTORS

The goal of this work is to analyze EditSameMC and EditDepMC effectiveness as conflict predictors. Specifically, we want to measure the conflict predictors' precision and recall. Besides that, we want to understand what happens when one conflict predictor occurrence is not associated with a merge, build or test conflict occurrences. Finally, we analyze what other change patterns, besides the defined predictors, could also be considered important conflict predictors. To achieve such a goal, we analyze merge scenarios from the development history of different software projects while answering the following research questions:

2.1 Research Question 1 (RQ1): How precise are EditSameMC and EditDepMC predictors?

To answer this question we measure the conflict predictors' precision. When we reproduce project merge scenarios, we collect occurrences of merge, build and test conflicts, together with occurrences of conflict predictors. This way, when a merge scenario has a conflict predictor and a conflict occurrence, we classify it as a true positive instance. In practice, if an awareness tool had alerted developers about the occurrence of such predictor, it would have indeed, detected a conflict. However, if a merge scenario has predictors but no conflicts, we classify it as a false positive instance. This means that the awareness tool might have raised a false alarm. In contrast, if the merge scenario has conflicts but no predictors, we classify it as a false negative instance because the awareness tool would not have triggered an alarm, and the conflict would only be detected during the integration process. Finally, if the merge scenario has no predictors nor conflicts, we consider it to be a true negative instance. Therefore, we can compute the precision considering both predictors together, and for each predictor individually, using the following formula:

- Precision = $\frac{True\ positives}{True\ positives+False\ positives}$

2.2 Research Question 2 (RQ2): What percentage of conflicts can we avoid by detecting EditSameMC and EditDepMC predictors?

To answer this question we need to measure the conflict predictors' recall. We compute this metric considering the predictors together, and individually, using the following formula:

- Recall = $\frac{True\ positives}{True\ positives+False\ negatives}$

2.3 Research Question 3 (RQ3): How are EditSameMC and EditDiffMC possible without causing conflicts?

To answer this question, we conduct a manual analysis considering a sub sample of the detected false positives. With this analysis, we aim to understand how developers managed to edit the same method or directly dependent methods without causing merge, build or test conflicts. This way, we can improve the precision of the awareness strategy. Conversely, we also want to check if contributions interferences are being missed by projects test suites. As mentioned before, this is expected given the limitations of how we establish test conflicts ground truth.

While analyzing false positive instances we try to understand if the contributions clearly do not interfere with each other or if there is a possibility of interference. To this end, we rely on a broader

[2]https://github.com/code4craft/webmagic

notion of interference defined by Horwitz et al. [19] and used in our previous work [11]. This definition states that "two contributions (changes) to a base program interfere when the specifications they are individually supposed to satisfy are not jointly satisfied by the program that integrates them; this often happens when there is, in the integrated program, data or control flow between the contributions. We then say that two contributions to a base program are conflicting when there is no valid program that integrates them and has no unplanned interference". The challenge associated with such a more comprehensive comparison criteria is that it is not computable in this context [5, 19].

Therefore, we manually analyze the contributions to check if there is a possibility of interference, for example, if one contribution edits a variable assignment which is later used by a command that the other contribution adds or edits. We consider that it would be reasonable to advise developers in such cases. However, if the contributions edit unrelated variables, or if one of them does not change the program behaviour, we consider that they do not interfere with each other and, therefore, could be merged together without further problems.

2.4 Research Question 4 (RQ4): What other change patterns are associated with conflicts?

We answer this question by manually analyzing the false negative instances. Our aim is to learn about other change patterns associated with conflicts in our sample. This way, new conflict predictors could arise, increasing the recall of our results.

3 EMPIRICAL STUDY SETUP

To explain how we answer our research questions, here we present our study setup and describe the selection, mining and analysis of our data. All the scripts and data used in this study are available in our online appendix [1].

In this paper we focus our analysis on Java projects hosted on GitHub. In addition, we select projects using Travis CI services and Maven as build manager. We select projects using Travis CI because, besides being the most used CI service [31], it provides all build information associated with a commit.[3] We need such information to compute our metrics as we further detail. In addition, we focus on projects using Maven because we use its log report information for filtering conflicts without human effort.

Figure 1 illustrates the study design, which is divided in the three following phases: in the first phase, we select Java projects from GitHub using Travis CI and Maven and filter those projects containing at least one build or test conflict. Section 3.1 describes this phase in more detail. Then, in the second phase, we use the Conflict Analyzer tool used in our previous work [2] to reproduce merge scenarios from each selected project. We enhance this tool to collect conflict predictor instances as well. In this phase we compute the metrics used to answer **RQ1** and **RQ2**. We explain how we do so in Section 3.2. Finally, in the third phase, we perform a manual analysis on a sub sample of reported false positives and false negatives so that we can answer **RQ3** and **RQ4**. Section 3.3 explains this analysis in further detail.

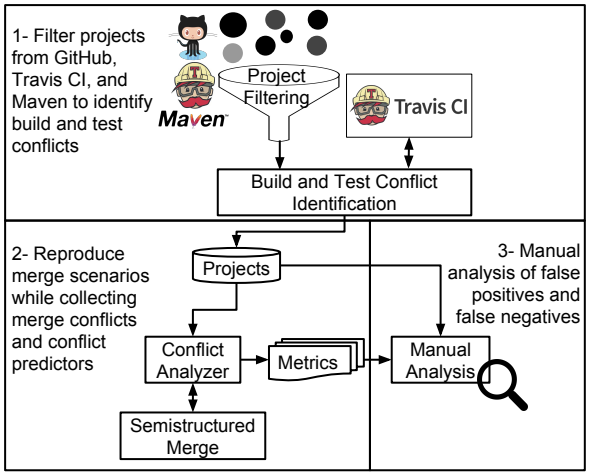

Figure 1: Study design.

3.1 Phase 1: Filtering Projects Containing Build and Test Conflicts

We start selecting our sample on GitHub by filtering Java projects containing at least 40 stars and 50 forks. We choose a minimum number of stars and forks to avoid selecting toy or personal projects. With the list of selected projects, for each project we check if the repository contains both Travis CI and Maven configuration files— *.travis.yml* and *pom.xml*. We also check the project current status on Travis (active or not). This way we ensure we select only projects using Maven as build manager, and having data available on Travis CI.

For projects meeting those requirements, we execute a script that clones each project locally and retrieves their merge commit list. However, as most projects adopted Travis CI later in its life cycle, we filter project merge commits dated after the first finished build on Travis. Then, for each selected merge commit, we use its build status on Travis CI, together with its Maven build log report, and its parent commits build status, to identify build and test conflicts. However, Travis CI builds only the latest commit in the push command or pull request to run the analysis, so not all commits in a project have an associated build status on Travis CI. Because of that, we use a script that forces the commit build creation when there is no build yet. Basically, we create a project fork, activate it on Travis CI, and clone it locally. Then, every push to the remote fork creates a new build on Travis CI. So, for each merge commit, or merge commit parent, without an associated build on Travis CI, we reset the fork repository head to this commit and push it to the remote fork.

If the merge commit build status on Travis CI is *passed* it means that there is no build error, and none of the tests fails. For these merge commits, we consider that there are no build or test conflicts. In contrast, if the merge commit build status is *errored*— when the build is broken— or *failed*— the build is ok, but at least one

[3] https://docs.travis-ci.com/api

of the tests failed—, we consider it to be a build or a test conflict **candidate**, respectively. However, there are some conditions that must be satisfied first.

It is possible that a build breaks or a test fails due to external configuration problems such as trying to download a dependency that is no longer available, or exceeding the time to execute tests. To eliminate these cases, we analyze for each build its Maven log report seeking for specific message errors [28]. Basically, there are two external causes responsible for interrupting a build process. First, when the build fails because Travis or another external service required by a build process was temporarily unavailable. We discard these scenarios because they do not reflect issues caused by conflicting contributions. Second, when the build process fails due to unsolvable or wrong project dependencies. We only discard such scenarios when no changes were made to configuration files. This restriction ensures an external problem is responsible for the build failure.

After discarding those cases, we check the merge commit parents' status to eliminate cases where the build was already broken or with failing tests before the merge. If this is the case, we consider the merge commit broken build or failed test was carried over from its broken parents, instead of being caused by conflicting contributions. Therefore, we consider that a merge commit with an *errored* build status has a build conflict if its parents have a *passed* or *failed* build status. Likewise, we consider that a merge commit with a *failed* build status has a test conflict if its parents have a *passed* build status. Finally, by the end of this phase, we filter projects containing at least one build or test conflict to proceed to the second phase of the study where we collect merge, and conflict predictors occurrences.

3.2 Phase 2: Collecting Merge Conflicts and Predictors

In this phase we use the Conflict Analyzer tool together with the FSTMerge tool [2] to reproduce all merge scenarios from the filtered projects dated after Travic CI first finished build from the projects selected in Phase 1 while collecting information about merge conflicts. FSTMerge consists of 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 [4, 10, 11]. 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. Whenever FSTMerge reports a conflict, the Conflict Analyzer tool gets notified and classifies it into different conflict patterns. Moreover, we had to implement some changes to this infrastructure in order to collect EditSameMC and EditDepMC instances. We also analyze EditSameMC predictors to check if they are different spacing false positives. This happens when one of the contributions only made changes related to code spacing, which is irrelevant for Java code syntax. This way we can compute and compare our metrics in both ways, considering all EditSameMC predictors, and filtering the different spacing ones.

3.2.1 Collecting EditSameMC predictors. Collecting EditSameMC instances is straightforward since we had to slightly alter the merge algorithm to notify the Conflict Analyzer tool when both merge commit parents add changes to the same method. Because FSTMerge uses *diff3* algorithm to merge the content inside methods, if the output text contains conflict markers, we collect an EditSameMC instance associated with a merge conflict. Otherwise, if there are no conflict markers, we compare the parents version to the base version and, if both parents differ from the base version, we collect an EditSameMC instance without merge conflicts.

3.2.2 Collecting EditDepMC predictors. To detect EditDepMC predictors, while reproducing the merge, we collect all method instances with non-spacing changes made by at least one parent—we discard changes related to different spacing. We also keep the information about which parent was responsible for editing each method. By the end of the merge process, we have a list of all methods changed by one of the parents.

Then, for each method changed by parent 1, we check if any other method changed by parent 2 has a method call to it. Similarly, we do the same inverting parent 1 and parent 2 in the description above. Figure 2 illustrates this approach. Suppose that, by the end of the merge process, we have three methods in our list. Methods *m* and *n* from class *A*, edited by parent 1, and method *o* from class *B*, edited by parent 2. In this example, we need to check if *m* calls *o*, if *n* calls *o*, if *o* calls *m*, and if *o* calls *n*. There is no need to check if *m* calls *n*, or if *n* calls *m* since these methods were edited by the same parent.

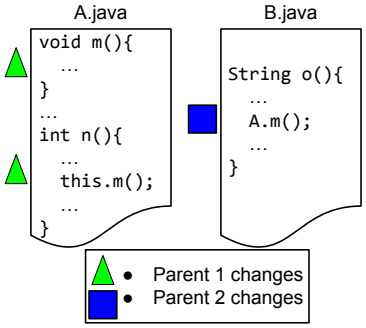

Figure 2: Looking for EditDepMC predictor instances.

To check if one method calls another one, for performance reasons, we use a two step approach. For example, consider that we need to check if method *o* calls method *m*. First, we perform a simple textual search to see if the name of the method *m* is inside method *o* body declaration. In case it does, we use Eclipse JDT library[4] to parse class *B* and build its AST. Then, we visit *B* AST nodes until we get to method *o* declaration. There, we list all the method invocations, and check if any of them matches with method *m* from class *A*.

In Figure 2 example, after performing all necessary method reference checks, we note that there is one EditDepMC predictor instance involving methods *o* and *m*. As we do for EditSameMC instances not associated with merge conflicts, we check if the EditDepMC instances are associated with a build or a test conflict collected in Phase 1.

[4] https://www.eclipse.org/jdt/

3.3 Phase 3: false positives and false negatives analysis

After computing the conflict predictors precision and recall, we conduct an analysis to understand the causes of the false positives, and false negatives from our sample. We start by randomly selecting a sub sample of the false positive instances to manually analyze them. Because we reproduce the merge scenarios locally, we keep a copy of the files containing EditSameMC and EditDepMC instances that are not associated with conflicts.

During these files' analysis, we check the changes made by each developer trying to understand if there is a possibility of interference between their contributions. If one of the contributions performs changes that does not change program semantics, such as renaming a local variable, or removing an extra pair of parenthesis, or if the developers edit variables that are not related, we consider that there is no interference. Conversely, if both developers change the program semantics, and they edit related variables, for example, when one developer changes an *if* statement condition, while the other edit commands inside that same *if* statement, then we consider that there is an interference.

Furthermore, we have two different strategies to analyze the false negatives from our sample. First, we use the Conflict Analyzer tool to automatically collect the merge false negative conflicts causes. Second, we manually analyze the Maven log reports associated with the *errored* and *failed* builds in order to understand what caused build and test false negative conflicts.

4 RESULTS

In the first phase of this study we analyze a total of 64,445 merge scenarios from 422 Java projects from GitHub using Travis CI and Maven. From this total, 646 merge scenarios have an *errored* build status. From this total, 551 merge scenarios meet our build conflict criteria —the merge commit build status is *errored* while the parents' status are either *passed* or *failed*. However, after performing the Travis log report analysis, we eliminate 467 of those merge scenarios because their builds fail dues to external reasons not related to the contributions being merged. In such cases, we cannot be sure that there is a build conflict due to conflicting contributions. Therefore, we only consider the remaining 84 merge scenarios to have build conflicts. Alternatively, we only found 5 merge scenarios meeting the test conflict criteria —merge commit build fails while parents' builds passes. This time, the log report analysis did not eliminate any of the test conflict instances. In summary, by the end of this study first phase, we select a total 45 projects containing at least one build or test conflict.

While selecting that sample, we did not target representativeness or diversity in a systematic fashion as Nagappan et al. [24] propose. Nonetheless, we consider that our sample contains substantial and active software systems with some degree of diversity with respect to dimensions such as size, domain, and number of collaborators. For example, Cloudify, a cloud infrastructure platform, has 408 KLOCs, and 23 active collaborators, Java Jwt, a library for creating and verifying JSON Web Tokens on the JVM has only 8 KLOCs and 8 collaborators, and OkHttp, an HTTP client for Java and Android applications has 57 KLOCs and 128 collaborators. For further information on our sample, we provide a complete subject list on our online appendix [1].

4.1 Conflict predictors' precision and recall

In the second phase of this study, we take as input the list of 45 projects and use FSTMerge to reproduce a total of 5,647 merge scenarios dated after each project first finished build on Travis CI. In this sample, a total of 290 merge scenarios have merge conflicts, and 508 have conflict predictors. If we remove different spacing instances, the total number of merge scenarios containing merge conflicts drops to 251, while the number of merge scenarios containing predictors drops to 469.

Moving on with the analysis, we cross information about merge scenarios containing predictors associated with merge, build, and test conflicts. In total, there are 286 merge scenarios containing at least one predictor occurrence associated with a conflict occurrence. If we remove the different spacing predictors and conflicts, this number drops to 272. Moreover, there are 282 merge scenarios containing EditSameMC instances associated with conflicts. By removing the different spacing occurrences, this number drops to 266. Finally, there are 45 merge scenarios containing EditDepMC instances associated with conflicts. As explained in Section 3.2, we do not collect EditDepMC different spacing instances. We use these numbers to compute precision and recall considering the conflict predictors combined, and individually. Moreover, we also measure these metrics considering all predictors and conflicts instances, and filtering the different spacing instances. Table 1 summarizes **RQ1** and **RQ2** answers.

5 FALSE POSITIVES MANUAL ANALYSIS

In the third phase of this study, we answer **RQ3** by conducting a manual analysis of the false positives from our sample. In total, our sample has 226 merge scenarios containing predictors not associated with conflicts. If we remove the different spacing instances, this number drops to 203. From this sample, we randomly select 10 EditSameMC, and 10 EditDepMC instances to conduct the manual analysis. Table 2 and Table 3 summarize EditSameMC and EditDepMC false positives analysis, respectively. In summary, we consider that 8 predictor instances have the possibility of interference, while 12 do not. All the false positives and false negatives manually analyzed are available in the online appendix [1].

6 FALSE NEGATIVES MANUAL ANALYSIS

To answer **RQ4**, we analyze false negatives conflict causes to learn what types of changes— besides the defined predictors— are associated with conflicts. In our sample there are 56 conflicting merge scenarios where no conflict predictor was involved. From this total, 20 merge scenarios have merge conflicts (35.71%), 33 have build conflicts (35.71%), and 3 have test conflicts (5.35%).

Because we use our previous work infrastructure to reproduce merge scenarios, we automatically collect merge conflict causes using their conflict pattern catalog [2]. Among the 20 merge scenarios containing merge conflicts, 11 scenarios (55%) had conflicts caused by developers editing the same class field, 7 scenarios (35%) had conflicts because two developers independently added methods with

Table 1: Precision and recall results according to the predictors considered. WDS means without different spacing.

	Both Predictors	Both Predictors WDS	EditSameMC	EditSameMC WDS	EditDepMC
Precision	56.29%	57.99%	55.51%	56.71%	8.85%
Recall	83.62%	82.67%	82.45%	80.85%	13.15%

Table 2: EditSameMC false positive analysis.

Project	Editions Summary	Interference
JavaPoet	Parents change unrelated variables	No
OpenGrok	One parent changes a variable assignment passed as an argument in a method call edited by the other parent	Yes
Jackson Databind	One parent changes a variable assignment used by the other developer to change an if statement condition	Yes
CorfuDB	One parent changes a variable assignment, while the other parent changes this same variable method call	Yes
Swagger Core	Parents change unrelated variables	No
Wire	One parent changes a variable assignment used in a for statement condition changed by the other parent	Yes
Jackson Databind	Parents edit unrelated variables	No
OkHttp	One parent refactors	No
Restheart	One parent refactors	No
Web Magic	One parent removes commands inside an if body declaration while the other parent changes the if statement condition	Yes

the same signature and different declarations, in 1 scenario (2.5%) both developers edited the same list of access modifiers, and in 1 scenario (2.5%) both developers edited the same class implements declaration.

Furthermore, in our sample, 33 merge scenarios have build conflicts not associated with the predictors. In this sample, the most frequent situation in build conflicts— a total of 20 conflicts (60.61%)— happens when one developer adds a new reference to a program element— such as a class, a method, or a variable— while the other developer deletes or renames that element. For example, in one of project Blueprints merge scenarios, one developer adds a new method calling another method that was removed by the other developer. Consequently, after the merge the compiler could not build the file containing the reference to the removed method.

In contrast, the second most frequent cause for build conflicts in the false negatives sample are syntactic malformed programs after the merge. More specifically, 10 merge scenarios (30.31%) from projects Java Driver, Cloud Slang, and Hdiv have broken builds because some of the files did not have the expected license header, causing a compilation error on Travis CI.

The remaining three merge scenarios in the false negatives sample have different conflict causes. In one merge scenario from project ScribeJava, while one contribution adds a new class implementing an existing interface, the other developer adds a new method to this interface. After the merge, there is a compilation error because the newly added class does not implement all interface methods.

One build conflict from project Blueprints has an occurrence of the SameSignatureMC pattern where both contributions copy and paste the same method across different repositories and one of them edits the method indentation. Because of that, the line-based merge tool reported a conflict and the developer responsible for the integration tried to fix the conflict by copying and pasting the two versions of the same method to the resulting file. As a result, there is a compilation error due to duplicate method declarations. Conversely, if one had used a semistructured merge tool in this merge scenario, there would not be a merge nor a build conflict because FSTMerge only reports SameSignatureMC conflicts when one of the contributions edits the code content, ignoring spacing changes.

Table 3: EditDepMC false positive analysis.

Project	Editions Summary	Interference
OkHttp	One parent changes one method while the other parent changes this method call inside the other method	Yes
Jackson Databind	One parent refactors	No
Cloudify	One parent changes an if statement condition inside of which there is a call to the method edited by the other parent. The other parent adds a new return command to the second method	Yes
Jackson Databind	One parent edits comments	No
Wire	Both parents change the same variable assignment which is passed as an argument from one method to the other	Yes
Truth	One parent refactors	No
Moshi	One parent refactors	No
JavaParser	One parent refactors	No
Retrofit	One parent refactors	No
Singularity	One parent refactors	No

The last merge scenario containing a build conflict in the false negatives sample comes from the Jackson-core project. Figure 3 depicts the differences between the merge commit parents. The left side parent removed the line containing the local declaration of variable *f* and added it as a parameter to the method. Meanwhile, the right side parent edited *INPUT* variable content.

Because the contributions edited consecutive lines of the same method —variables *f* and *INPUT* declarations— the line-based tool reports a conflict involving these lines. The developer tried to resolve this conflict by copying and pasting both local variable declarations to the resulting file. However, he did not notice the new parameter added by one of the parents. Consequently, variable *f* has two local declarations, causing a compilation error.

This last build conflict is actually an EditSameMC instance missed by FSTMerge. This happens because, as explained in Chapter **??**, FSTMerge cannot match methods when their signature is changed. Therefore, this merge scenario has an EditSameMC instance associated with a conflict, which makes it a true positive in our sample that the actual infrastructure is not able to detect.

As for the false negative test conflicts, our sample has three instances coming from projects Jedis and Wire. Two of these conflicts happened because not directly dependent methods were edited, and the third one happened because one developer updated a test case executing a method that was edited by the other developer.

7 HOW EFFECTIVE ARE THE CONFLICT PREDICTORS?

The precision and recall metrics gives a notion of how effective an awareness tool considering EditSameMC and EditDepMC predictor would be if it was used during the development of the 45 projects from our sample. Such evidence can guide better decisions regarding a awareness tool conflict awareness strategy.

The precision indicates that for over half— 57.99%, after removing the different spacing cases— of the merge scenarios where the tool triggers an alarm, it is indeed alerting developers about changes associated with merge, build and test conflicts. Meanwhile, the recall indicates that we capture 82.67% of the merge scenarios containing merge, build and test conflicts by using an awareness tool considering both predictors.

However, to establish the ground truth for build and test conflicts we rely on the status of building and testing processes executed by Travis CI. Whereas this provides quite precise guarantees for build conflicts, the guarantees for test conflicts are as good as the project test suites. So, even for projects with strong test suites, actual interferences might be missed by the existing tests. As a matter of fact, during the false positive analysis we find that 8 out of 20 (40%) false positive instances are missed interferences. This evidence suggests that if we had better test cases our precision would increase. In contrast, better test cases would find more interferences not caused by the predictors as well. Consequently, the precision results we report represent lower bounds of actual interferences, whereas the recall results are upper bounds of interferences.

Furthermore, we note a significant difference between the two predictors' precision and recall when we analyze them individually. While EditSameMC precision is 56.71%, EditDepMC precision is only 8.85%. Likewise, while EditSameMC recall is 80.85%, EditDepMC recall is 13.15%. Because there is not much difference between the measured precision and recall considering the predictors together and EditSameMC individually, there is significant evidence that EditSameMC instances dominate our measurements. This is due to the fact that EditDepMC instances are not associated

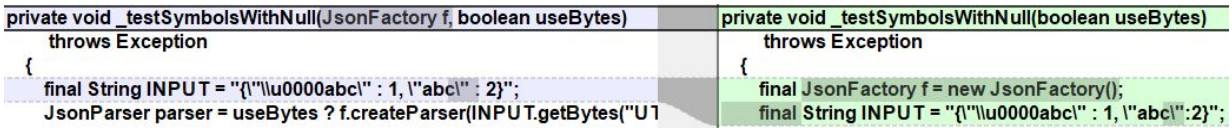

Figure 3: Merge scenario from Jackson-core project.

with merge conflicts, which are the most numerous in our sample. Nonetheless, we consider that a solution containing both predictors would still be advisable.

Our precision and recall results provide evidence to guide different conflict detection strategies depending on each team preferences. For instance, if one particular team prefers to be conservative and alert developers about a large part of the conflicts at the cost of dealing with some false positives, than detecting EditSameMC and EditDepMC as we do in this study is a reasonable strategy. In contrast, if a team aims at precision, even at the cost of loosing some conflicts, than it could alert developers about EditSameMC instances only when the contributions edit the same or consecutive lines of the same method, which necessarily leads to merge conflicts. This approach is similar to Crystal [8], a tool that proactively integrates commits from different developer repositories with the purpose of warning developers of merge, build and test conflicts. Nevertheless, checking if the same lines of the same method were edited would cost less than performing Crystal's integration routine. Finally, there is also the possibility of using other methods to further increase the predictors' precision without compromising their recall as we further detail in the next section.

8 STRATEGIES TO IMPROVE THE PRECISION AND RECALL OF THE CONFLICT PREDICTORS

The false positives and false negatives analysis provides insights of opportunities to further increase an awareness tool precision and recall. In this section we discuss such results and the actions they support.

During the false positives analysis, we find 8 cases— 5 EditSameMC, and 3 EditDepMC— where, even though there are no conflicts associated, it would still be advisable to alert developers about such changes because there are interferences. For example, in one EditSameMC instance from project Web Magic, while one developer removes code inside an *if* statement, the other developer changes the *if* statement condition.

Conversely, we also find 9 false positive cases— 5 EditSameMC and 7 EditDepMC— where there is clearly no interference because one of the contributions does not alter program semantics. For instance, in one EditDepMC instance from project Jackson Databind, one of the contributions renames a local variable, while in an EditSameMC case from project RESTHeart one of the contributions simply removes an extra semicolon. By observing the types of changes that were made, we learn about strategies that might improve the conflict predictors precision. We can divide them in the following two categories:

(1) Implement strategies to identify and ignore cases where there is no interference;
(2) Implement strategies to identify possible interferences.

Regarding the first category, we already use in this study a strategy to detect cases where one of the contributions changes only code spacing. Considering our sample, this strategy improves the overall precision in 2.93%. Another simple strategy would be detecting when one of the contributions edits only comments, as it happens in one case from project Jackson Databind

Perhaps, a possible strategy to detect and ignore cases with clearly no interference would be to run refactoring detection tools, such as Tsantalis et al. [30] propose, to detect and ignore predictor cases where one of the contributions performs solely refactoring editions without changing program semantics. However, some refactorings, such as renamings, might cause build conflicts. For example, when inside the same method one developer renames a local variable while the other developer adds code using this same variable. Perhaps, a safer way to use this strategy would be in the context of a tool such as Crystal. While running its integration routine Crystal would be able to provide more comprehensive alerts by adding a refactoring detection algorithm to its integration routine. For example, if no conflicts are detected, but both contributions change the program semantics it could alert developers to be more cautious about this integration scenario. Conversely, if no conflicts are detected and one of the contributions performs only refactorings, then Crystal could report that there is no interference in this scenario. We suggest this analysis as a future work in Section 12. Nonetheless, in an environment with many developers committing often, Crystal's speculative analysis might become to expensive.

For the second category of strategies, in Section 2, we mention that interferences often happens when there is data or control flow between the contributions. Therefore, one possible mechanism to identify possible interferences would be to check the existence of information control flow between the contributions as an approximation for computing interferences. This is exactly what Filho [15] investigates. He analyzed a total of 157 merge scenarios from 52 Java projects containing EditSameMC predictors. He finds information control flow in 64% of the merge scenarios. Then, after a manual analysis, he reports that there was indeed interference in 42.86% of the merge scenarios with information flow between contributions. He also describes improvements to increase the precision of his technique.

Alternatively, one could identify interference by using an approach similar to Böhme et al. [7], which proposes to generate regression tests that expose change interaction errors. They do that by generating a graph called Change Dependence Graph (CDG) to summarize the control flow and dependencies across changes. The

CDG is then used as a guide during program path exploration via symbolic execution— thereby producing test cases which witness change interaction errors. An extension of this strategy, generating test cases exercising commands changed by both developers, might be able identify more interferences.

Regarding the false negatives analysis, we consider that it would not be hard to detect most part of them. For example, all merge conflicts from the false negatives sample would be detected by the conflict pattern described in our previous work [2]. Syde [18], a tool that provides team awareness by capturing developers' editions as atomic AST changes and warns developers if they change the same nodes, would be able to detect such cases as well. Furthermore, except for the build false negatives caused by the missing license headers from specific project rules, the other build conflicts related to one contribution adding a reference to a program element which was renamed, moved, or deleted by the other contribution are already detected by existing awareness tools such as Palantír, Syde, and Crystal [8, 18, 26]. Such evidence suggests that practices used by these tools are feasible as well.

9 ARE BUILD AND TEST CONFLICTS NOT THAT FREQUENT AFTER ALL?

Although this study does not aim to measure build and test conflicts frequency, we could not help to notice that despite analyzing a considerable amount of merge scenarios and projects in the first phase of the study, we did not find many build and test conflicts. This becomes more evident when we compare our results to previous studies assessing the frequency of build and test conflicts. Kasi and Sarma [20] report build conflicts occurring in ranges between 2-15%, while Brun et al. [8] describe both kinds of conflicts ranging around 33%. In this section we discuss some of the reasons why our numbers are so different compared to other studies.

We believe there are mainly two reasons for such contrasting numbers and they are both related to differences between how we collect test and build conflicts. First, the previous studies rely only on the merge commit build status. They do not consider parents commit build status. This way, false positives might have been introduced. For example, the build might have been already broken or with failing tests before the merge. In such cases, we consider the merge commit broken build or failed test was carried over from its broken parents, instead of being caused by conflicting contributions. Second, because previous studies perform build and tests locally, some part of *errored* and *failed* builds might have been caused by external or configuration problems, for example, due to unsolved project dependencies. In our study we mitigate both threats since we analyze Travis CI log report to filter builds with errors caused by external problems, and we also check the merge commit parents status. This way we increase the confidence that the merge commit build problems are caused by conflicting contributions.

The decision of analyzing merge commits that occurred after the project has adopted Travis CI might have impacted the conflicts frequency. According to previous studies [31], the adoption of CI practices help to maintain the code quality. This is so because, when a project adopts CI practices it uses automated scripts to run build and testing. Thus, the developer responsible for the integration might be detecting and resolving most part of the conflicts locally, before pushing changes to the shared repository. Such perception seems to be aligned with previous empirical evidence [23] that broken builds occur more frequently in regular commits than in merge commits.

We believe that build and test conflicts occur more frequently than what we report here. As we narrow our numbers while trying to increase the soundness of our results, we might be loosing build and test true positives as well. However, we need further studies to understand the impact of our methods in our results. A better way for evaluating conflicts— not only build and test, but also merge conflicts— would be by having access to developers private workspaces instantaneously evaluating the cases without any external influences.

10 THREATS TO VALIDITY

Our empirical analyses and evaluation naturally leave open a set of potential threats to validity. We discuss some of these threats in this section. One of the threats concerning our metric for detecting test conflicts is that we rely on the projects existing test suites to detect them. This means that part of the interferences might escape. As mentioned before, better test cases would probably increase the precision, and decrease the recall reported in this study. Nonetheless, because the notion of interference we use is not computable, it is impossible to detect all semantic conflicts.

With regard to the internal validity, as we reuse previous infrastructure to reproduce merge scenarios, we inherit part of its threats as well [2]. In particular, because FSTmerge fails to identify renaming changes, we miss EditSameMC instances where one of the contributions changes the method signature. In fact, one of the build false negatives we analyze is actually a true positive that FSTMerge misses because of method renaming. In addition, on Travis, a build can be composed of a set of jobs; each job varies itself in some way. For example, different jobs can be used to simulate the same project with different environment configurations. Therefore, it is possible to declare which jobs should not be considered for the final build status. Thus, if a build conflict happens on a non-valid job, we do not detect it. For future work, we could edit *travis.yml* file aiming to consider all jobs for the final build status. However, non-valid jobs are used only to verify how the project behaves on a specific configuration. Therefore, problems in these scenarios possibly would not lead developers to spend time with them.

Regarding external validity, in this study we focus on open-source Java projects hosted on GitHub, using Travis CI and Maven. Thus, results generalizability to other platforms and programming languages is limited. Such requirements were necessary to reduce the influence of confounds, increasing internal validity. We need subsequent studies to further understand the precision and recall of the predictors for other programming languages. Nevertheless, we are confident that we have analyzed active and substantial systems from various domains. Finally, the samples we manually analyze are small, which makes it hard to estimate the impact of implementing the proposed improvements. For future work, we could increase the sample size.

11 RELATED WORK

A number of empirical studies provide evidence about collaborative development issues. In previous sections we have already mentioned some of them. For example, Kasi and Sarma [20] and Brun et al. [8] studies reproduced merge scenarios from different GitHub systems with the purpose of measuring the frequency of merge scenarios that resulted in merge, build and test conflicts. In this paper, although we do not aim to measure the conflicts frequency, we discuss the reasons why our results are different from those studies. Moreover, 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.

A empirical study performed by Cataldo and Herbsleb [9] tried to understand different 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 the number of conflicts. 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.

Likewise, Leßenich et al. [21] performed a survey to understand what factors are perceived by developers as being critical for increasing the risk of merge conflicts. From this survey, they selected 7 potential conflict predictors such as the size of changes, the number of files changed, and the location of changes. However, none of these predictors correlates with the number of merge conflicts.

Furthermore, Shihab et al. [27] 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. [14], 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 related to integration failures on collaborative development environments. However, we analyze different factors. While Cataldo and Herbsleb [9] and Leßenich et al. [21] analyzed architecture level and organizational factors, and Shihab et al. [27] analyzed branching strategies, we analyze the effectiveness of code changes as conflict predictors.

Tools and strategies to support collaborative development environments use different strategies to both decrease integration effort, and improve correctness during task integration. Throughout this work we have mentioned most of them. Cassandra [20], for example, is a tool that analyzes task constraints to recommend an optimum order of tasks execution so that conflicts can be avoided. While the tasks are being developed, Palantír [26], informs developers of ongoing parallel changes, and Crystal [8], proactively integrates commits from developer repositories with the purpose of warning them if their changes conflict.

In contrast, other awareness tools, such as Syde [18], build code artifact ASTs to make the analysis of changes more precise. WeCode [16] continuously merges uncommitted and committed changes to detect conflicts on behalf of developers before they check-in their changes. Moreover, Bellevue [17], is an IDE extension to make committed changes always visible, and code history accessible inside developers' workspaces.

Alternatively, when the performed tasks are ready for being merged, TIPMerge [12] 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 [4], and JDime [3] offer solutions to reduce integration effort by automating the resolution of some types of conflict, such as the ordering conflicts.

Finally, MergeHelper [25] captures code changes as sequences of fine-grained atomic operations. This way, developers can replay all changes involved in a conflict, which can help in resolving them. Similarly, MolhadoRef [13], is also an operation-based approach that records refactoring operations used to produce one version and replays them when merging different contributions. Our work brings evidence and insights to improve such tools.

12 CONCLUSIONS

In this paper we conducted an empirical study to measure the precision and recall for EditSameMC and EditDepMC predictors. Our results indicate that, considering both conflict predictors together, we achieve a precision of 57.99% and a recall of 82.67%. Such evidence is useful to guide different conflict awareness strategies. Moreover, we also suggest strategies that could further improve the precision and recall of our predictors.

This study presents different directions for future works. Perhaps, the most important one would be to implement the many improvements we suggested here to provide empirical evidence of their practical value. Moreover, we believe that replicating this study monitoring developers private repositories would be an interesting contribution. This study would provide a better notion about the real frequency of conflicts.

ACKNOWLEDGMENTS

We thank our research colleagues, and the anonymous reviewers for contributing to improve this work. For partially supporting this work, we also thank INES (National Software Engineering Institute) and the Brazilian research funding agencies CNPq (grant 309741/2013-0), FACEPE (grants IBPG 0716-1.03/12 and APQ/0388-1.03/14), and CAPES.

REFERENCES

[1] 2018. Appendix. https://conflictpredictor.github.io/onlineAppendix/. (2018). Accessed: 2018-01-23.

[2] Paola Accioly, Paulo Borba, and Guilherme Cavalcanti. 2017. Understanding semi-structured merge conflict characteristics in open-source Java projects. *Empirical Software Engineering* (2017).

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

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

[5] Valdis Berzins. 1986. On merging software extensions. *Acta Informatica* 23, 6 (1986).

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

[7] Marcel Böhme, Bruno Oliveira, and Abhik Roychoudhury. 2013. Regression Tests to Expose Change Interaction Errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA.

[8] Y. Brun, R. Holmes, M.D. Ernst, and D. Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *Software Engineering, IEEE Transactions on* (2013).

[9] Marcelo Cataldo and James D. Herbsleb. 2011. 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.

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

[11] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. *Proceedings of the ACM on Programming Languages* (2017).

[12] Catarina Costa, Jair Figueiredo, Leonardo Murta, and Anita Sarma. 2016. TIPMerge: Recommending Experts for Integrating Changes Across Branches. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM.

[13] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. 2008. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Trans. Softw. Eng.* 34, 3 (2008), 321–335.

[14] H Christian Estler, Martin Nordio, Carlo Furia, Bertrand Meyer, et al. 2014. 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.

[15] Roberto Filho. 2017. *Using Information Flow to Estimate Interference Between Developers Same-Method Contributions.* Ph.D. Dissertation. Universidade Federal de Pernambuco. Accessed: 2018-01-22.

[16] Mário Luís Guimarães and António Rito Silva. 2012. Improving Early Detection of Software Merge Conflicts. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press.

[17] Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. 2015. Supporting Developers' Coordination in the IDE. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '15)*. ACM.

[18] Lile Hattori and Michele Lanza. 2010. Syde: A Tool for Collaborative Software Development. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM.

[19] Susan Horwitz, Jan Prins, and Thomas Reps. 1989. Integrating Noninterfering Versions of Programs. *ACM Transactions on Programming Languages and Systems* 11, 3 (1989).

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

[21] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian Kästner, and Claus Hunsen. 2017. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering* (2017).

[22] Gleydson Lima. 2014. *Uma Abordagem para EvoluÃğÃčo e ReconciliaÃğÃčo de Linhas de Produtos de Software Clonadas.* Ph.D. Dissertation. Universidade Federal do Rio Grande do Norte. Accessed: 2018-01-12.

[23] Ward Muylaert and Coen De Roover. 2017. Prevalence of Botched Code Integrations. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press.

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

[25] Yuichi Nishimura and Katsuhisa Maruyama. 2016. Supporting Merge Conflict Resolution by Using Fine-Grained Code Change History. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2016).

[26] A. Sarma, D. F. Redmiles, and A. van der Hoek. 2012. Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes. *IEEE Transactions on Software Engineering* (2012).

[27] Emad Shihab, Christian Bird, and Thomas Zimmermann. 2012. 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.

[28] Leuson M. P. Silva. 2018. *Build and Test Conflicts in the Wild.* Master's thesis. Universidade Federal de Pernambuco. Accessed: 2018-01-22.

[29] Travis CI community. 2018. Travis CI. https://travis-ci.org/. (2018). Accessed: 2018-01-25.

[30] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 2018 International Conference on Software Engineering*. IEEE Press.

[31] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The Impact of Continuous Integration on Other Software Development Practices: A Large-scale Empirical Study. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA.

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