Empirical Software Engineering (2023) 28:27
https://doi.org/10.1007/s10664-022-10266-8

Using acceptance tests to predict merge conflict risk

Thaís Rocha[1] · **Paulo Borba**[2]

Accepted: 30 November 2022

Abstract

Merge conflict resolution might be time-consuming and lead to defects, compromising development productivity and system quality. Developers might reduce such adverse impacts by avoiding concurrent programming tasks that are more likely to change the same files and cause merge conflicts. As manually predicting such risk is hard, we propose the TAITIr tool, which approximates the set of files changed by a task (*task interface*) and reports conflict risk whenever there is an intersection between task interfaces. TAITIr uses as input the acceptance tests related to the tasks for predicting file changes, deriving test-based task interfaces. To assess TAITIr's conflict risk predictions, we measure precision and recall of 6,360 task pairs from 19 Rails projects on GitHub. Our results confirm that the intersection among task interfaces is associated with a higher probability of merge conflict risk. A minimal intersection predicts conflict risk with 0.59 precision and 0.98 recall. We observe that the higher the intersection size, the higher the number of files changed by both tasks. This way, developers might use the intersection size between interfaces as a degree of conflict risk between tasks, choosing a task to work on depending on it. We also find that TAITIr's predictions outperform predictions based on changed files by similar past tasks. Our analysis derives several other results, considering variations of our notion of an interface in two dimensions: parts of the test code considered for computing interfaces, kinds of files abstracted by the interfaces.

Keywords Collaborative development · Task prioritization · Behaviour-driven development · Prediction of conflict risk

Communicated by: Bram Adams

✉ Thaís Rocha
thais.burity@ufape.edu.br

Paulo Borba
phmb@cin.ufpe.br

[1] Federal University of Agreste of Pernambuco, Garanhuns, Brazil

[2] Informatics Center, Federal University of Pernambuco, Recife, Brazil

Published online: 09 January 2023

1 Introduction

Software development is often a collaborative activity where developers share artifacts through a repository allowing them to work on programming tasks independently. As a consequence, without proper coordination, developers might often make code changes that are inconsistent with other changes, leading to textual or syntactic merge conflicts during code integration (Brun et al. 2013; Kasi and Sarma 2013), even when adopting advanced merge tools (Apel et al. 2011, 2012; Cavalcanti et al. 2017; Accioly et al. 2017) that avoid spurious conflicts reported by the state of the practice tools.

Conflict resolution might be time-consuming and lead to defects, compromising development productivity and system quality, as such defects often end up reaching end users. To reduce conflicts occurrence, developers adopt risky practices such as rushing to finish changes first (Grinter 1997; Sarma et al. 2012), and partial check-ins (de Souza et al. 2003). Similarly, partially motivated by the need to reduce conflicts, or at least avoid large ones, development teams adopt techniques such as trunk-based development (Adams and McIntosh 2016; Potvin and Levenberg 2016; Henderson 2017) and feature toggles (Bass and Weber 2016; Adams and McIntosh 2016; Fowler 2010; Hodgson 2017a), which support Continuous Integration (Fowler 2020) but might lead to extra code complexity (Hodgson 2017b).

An alternative strategy to reduce textual merge conflict[1] occurrence is to support developers with task context information to safely choose a task to work on by avoiding the concurrent development of tasks that are likely to change files in common. As developers are often unable to predict the files that a programming task will change accurately, we have developed the TAITI (*Test Analyzer for Inferring Task Interface*)[2] tool to automatically predict that in the specific context of BDD (Behaviour-Driven Development) (Smart 2014; Zampetti et al. 2020) projects. In such a context, developers write automated acceptance tests before feature implementation. As TAITI's prediction is based on the analysis of the tests associated with a task, we say that the tool computes a *test-based task interface*, which tries to approximate the actual task interface or context (the set of files changed by a programming task).

A previous evaluation study (Rocha et al. 2019) brought evidence that test-based task interfaces (*TestI*) generated by TAITI is a promising predictor of file changes. In sum, it is 45.6% – 70.9%[3] more precise than a random predictor (a predictor that indiscriminately points to file changes), and it performs better when there is broader test coverage per task. Also, although *TestI* is less precise than a predictor based on task interfaces obtained by observing changed files by similar past tasks (*TextI*), it covers much more file changes, getting a 12.77% – 20.14% higher mean recall rate. The predictions favor the recall measure, which is more critical for our context as false negatives could lead to unexpected conflicts (i.e., it is impossible to prevent concurrent changes on a file with no planned changes). False positives are undesirable as well. However, they discourage but do not prevent parallel tasks that would not conflict, as developers have decision-making power.

[1] The paper focuses on textual merge conflicts caused by parallel changes in a file hunk. This way, in the rest of the text, references to *merge conflicts* mean *textual merge conflicts*.

[2] https://github.com/thaisabr/TestInterfaceEvaluation

[3] The range refers to the results related to the prediction of file changes of two samples (the smallest and the largest, respectively) when using *TestI* with no filtering strategy.

Although *TestI* shows promising results for predicting files to be changed, we have no evidence that it might predict conflict risk and support developers to avoid conflict occurrence. Specifically, when dealing with predictions of file changes, we focus on individual tasks. In case of conflict risk, we focus on task pairs (at least), and a requirement is the predictions of file changes should apply for both tasks. As such a condition is not always satisfied, conflict risk predictions might derive unexpected results, motivating an in-depth investigation.

So in this paper, we go further and, by using TAITIr (TAITI *risk*)[4] tool, an extension of TAITI, we assess whether *TestI* can predict the risk of merge conflicts between two tasks based on the intersection between their interfaces. Knowing that textual merge tools like diff3 point out a conflict in case of parallel changes in a file hunk, we assume there is a merge conflict risk when two tasks change common files. We predict that there is no risk of a merge conflict if the intersection between two *TestI* is empty. Otherwise, we predict there is a risk. Note that *TestI* does not predict changes in program elements such as fields and methods, impairing us refining our definition of conflict risk.

To assess *TestI* potential, we conduct a retrospective study by collecting a sample of 990 tasks[5] from 19 Ruby on Rails[6] projects that use Cucumber[7] for specifying acceptance tests. Then we simulate the integration of possible concurrent pairs of tasks per project, computing the intersection between the set of files changed by both tasks. As a result, we have a set of 6,360 task pairs, for which we evaluate precision and recall measures of conflict predictions based on *TestI*. Our study derives several other results, considering variations of our notion of an interface in two dimensions: parts of the test code considered for computing interfaces, kinds of files abstracted by the interfaces.

In this sense, the novelty of this paper is the evaluation study concerning predicting conflict risk between tasks pairs based on *TestI*. TAITI computes *TestI*. TAITIr extends TAITI to evaluate conflict risk based on the intersection between two *TestI* interfaces. We developed both tools as scripts and used them in the context of an evaluation study. Because we conduct two studies, we opt to separate these tools conceptually. But considering that *TestI* is useless without predicting conflict risk, TAITIr is the tool developers will use.

Our results reveal that, in our sample, a minimal intersection between *TestI* (1 file) predicts conflict risk with 0.59 precision and 0.98 recall. In other words, the intersection between test-based task interfaces denotes higher chances of tasks changing files in common. Again, the predictions favor the recall measure, which is more critical for our context, as explained. We also find evidence that the larger the intersection size between two interfaces, the higher the number of files changed in common by the tasks associated with these interfaces. This way, developers might use the intersection size between *TestI* as a degree of conflict risk between tasks. Thus, they might prioritize parallel execution of tasks that are likely to change fewer files in common whenever other factors (time and resource constraints, stakeholders priority, task complexity, developer skills, etc.) allow.

[4] https://github.com/thaisabr/TAITIr

[5] A programming task is an activity performed by a developer that results in code creation or edition, such as developing a new feature, bug fix, or refactoring. Considering the usage of a repository to integrate code contributions, we extracted tasks from merge scenarios in this study. From a merge scenario, we determine a triple formed by left and right commits to be merged, and a base commit that is a common ancestor to left and right. This way, a task is a set of all reachable commits between the left/right commit and the base commit.

[6] https://rubyonrails.org/

[7] https://cucumber.io/

Although the intersection between *TestI* might predict potentially conflicting task pairs, we also observe that it cannot predict the potentially conflicting files in many cases. In such cases, the intersection between *TestI* might reflect a degree of dependence between the parts of the code changed by both tasks, eventually leading to semantic conflicts, even in files not directly reached by the interfaces.

In addition, we also observe that even though *TestI* only contains Ruby and HTML files, risk predictions are better when we compare them with the risk result by considering all kinds of files changed by the tasks. Such a phenomenon reinforces the previous observation. Alternatively, we find that we can improve predictions by discarding test preconditions when computing *TestI* interfaces as a strategy to reduce false positives. However, as the test coding style varies between projects, such a strategy does not universally benefit all of them.

Finally, to better assess the performance of our predictor based on *TestI*, we compared it with a predictor based on *TextI*. Similar to when dealing with file changes prediction, we find that a *TextI*-based predictor is more precise than a *TestI*-based one. Still, its low recall rate discourages its usage. So we conclude that, for our context, *TestI* has overall better performance.

The rest of the paper is organized as follows. Section 2 illustrates how developers could use *TestI* to evaluate the risk of a merge conflict between programming tasks and briefly presents TAITI, the tool for inferring *TestI*, and TAITIr, the tool for assessing conflict risk between tasks based on their *TestI*. Section 3 presents our research questions and Section 4 provides information about our task sample. Section 5 presents the obtained results. Section 6 discusses the implications of our results. Section 7 points out the threats to the validity. Section 8 presents related work. Finally, Section 9 brings final considerations. All data, tools, and complementary scripts used in the empirical study are available in our online Appendix (Rocha and Borba 2019).

2 Motivating Example

To illustrate how test-based task interfaces might help developers to avoid conflicts, let us see a simplified example of a merge conflict from project *allourideas/ allourideas.org*.[8] The project is a web system for collecting and prioritizing ideas through a kind of collaborative survey. Andrew concludes task T_{175}, which consists of a set of refactorings to improve code quality (commits `e3cc71`, `58195`, `6847c`, `18ff5`, `f1b0d`, and `f8e9f`). One day later, Becca concludes task T_{176}, which fixes a set of bugs and enhances the GUI layout (commits `5a6e4`, `ae40e`, `6f349`, and `2a33a`). When Becca integrates her contributions, the conflicts illustrated in Figs. 1 and 2 are reported. The first conflict (Fig. 1) occurs because task T_{175} removes a set of code comments whereas task T_{176} includes a method declaration just above the code comments, both affecting the same area of the file. The second conflict (Fig. 2) occurs because both tasks change, in the same area of another file, the body of the `results` method.

Andrew and Becca could avoid these conflicts by opting not to perform these tasks in parallel if they could predict the files they would have to change when working on their

[8]The project is part of our sample, but task T_{175} is not because it does not satisfy the selection criteria explained in Section 4 related to *TextI*. In sum, there are no older tasks than T_{175} in the sample from project *allourideas/allourideas.org*, resulting in an empty *TextI*. We present the example by using fictitious developers. We found the conflict occurrence by merging the tasks, given they were extracted from the same merge scenario — the same base and merge commits.

Fig. 1 Merge conflict in a model file caused by the integration of tasks T_{175} and T_{176}. Due to space constraints, we omitted parts of the code

tasks, whenever other factors allow. In the case of refactorings like Andrew's task, when a specific set of files has been perceived as problematic and is expected to be refactored, file change predictions could be more realistic. However, as in Becca's task, file change predictions related to bug fixes are less realistic and would demand strong developer expertise and experience with the system and the features. When the developer is responsible for fixing a recent bug introduced by himself, he might quickly point out the set of files related

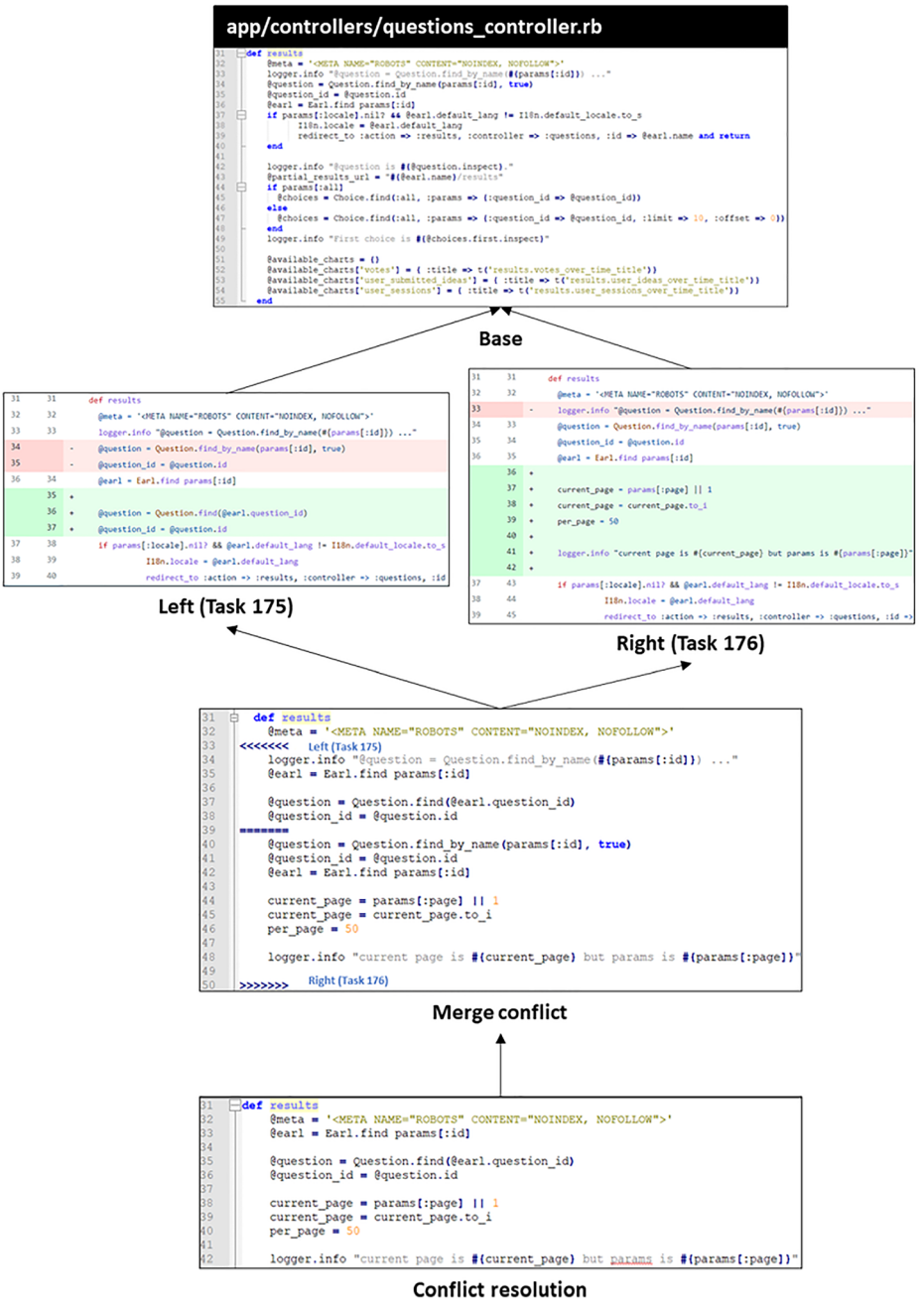

Fig. 2 Merge conflict in a controller file caused by the integration of tasks T_{175} and T_{176}. Due to space constraints, we omitted parts of the code

to the bug, making file change prediction a simple task. But in the case of an old bug or a bug introduced by a teammate, a bug fix might be challenging and require not obvious code changes. The same applies to predictions related to the development of new system features.

```
1  Scenario: Lots of ideas
2  Given an idea marketplace exists with url 'test'
3  And idea marketplace 'test' has 100 ideas
4  When I go to the View Results page for 'test'
5  Then I should see "Idea #2"
6  And I should not see "Idea #50"
...
```

(a) Scenario. Source: paginate_all_results.feature, lines 6-11.

```
1  When /^idea marketplace '(.*)' has (\d*) ideas$/ do |url, num_ideas|
2              ....
3              #Pairwise sorts by last created, but this makes testing pagination annoying, let's create these going down
4              (1..num_ideas.to_i).to_a.reverse.each do |n|
5                the_params = {'auto' => 'test choices', :data => "Idea ##{n}", :question_id => @question.id}
6                    Choice.post(:create_from_abroad, :question_id => @question.id, :params => the_params)
7              end
8
9              unless prev_auto_activate
10                   @question.put(:set_autoactivate_ideas_from_abroad, :question => { :it_should_autoactivate_ideas => false})
11             end
12 end
```

(b) Step definition that automates the second step. Source: idea_marketplace_steps.rb, lines 56-76.

Fig. 3 Example of a Cucumber test related to task T_{176}

Instead of relying on such developer prediction capacity, we consider automatic test-based prediction in the particular context of BDD, where automated acceptance tests are written before implementing features, and each feature is associated with test scenarios. The idea is to use the TAITI tool to systematically analyze the code of acceptance tests associated with a task and infer the application code that could be exercised by the tests, approximating the files that the task would change.

By inspecting the Cucumber tests related to each discussed task, TAITI recursively searches for references to programming elements (such as fields and methods) and links to web pages, yielding the file sets (test-based task interfaces) that developers might use as predictions of file changes. To better explain, let us consider the Cucumber test related to task T_{176} in Fig. 3.[9] TAITI receives as input the first part of the test (Fig. 3a), which is a high-level usage scenario written in Gherkin with test setup steps (`Given`), test actions (`When`), and expected results (`Then`). There are other keywords, such as `And`, which makes the scenario read more fluidly by avoiding a repetitive sequence of steps. From such a scenario, relying on regular expressions, TAITI reaches the second part of the test (Fig. 3b), which is Ruby code that automates the scenario steps. As a matter of brevity, we only partially illustrate the step definition that automates the second step of the scenario. Next, by inspecting such a step definition, TAITI finds the usage of the class `Choice` and the instance variable `@question` (both marked in red), among other programming elements, mapping them to the files *app/models/choice.rb* and *app/models/question.rb*, respectively.

[9] As a matter of clarity, we slightly simplify the Cucumber test, and we omit some parts of it that are not relevant to our explanation.

Fig. 4 Test-based task interface of T_{175}. The files in red are in the intersection between interfaces $TestI(T_{175})$ and $TestI(T_{176})$, and the underlined files are the conflicting ones

The inspection of all Cucumber tests related to tasks T_{175} and T_{176} yields the file sets in Figs. 4 and 5. We can then observe that $TestI(T_{176})$ is a subset of $TestI(T_{175})$. The seven files marked in red are at the intersection of the two interfaces and are more vulnerable to conflicts when developers integrate their contributions. Indeed, the two conflicting files previously presented in Figs. 1 and 2 are in such intersection.

Thus, assuming that Andrew and Becca have developed the Cucumber tests before the application code, according to BDD's dynamics, they could avoid the conflicts by using TAITIr, an extended version of TAITI that evaluates conflict risk between a task pair based on *TestI*. By observing the warning of conflict risk revealed by the non-empty intersection between $TestI(T_{175})$ and $TestI(T_{176})$, Becca could choose another task to perform. For example, she could select a task with a *TestI* that does not intersect with $TestI(T_{175})$, or at least

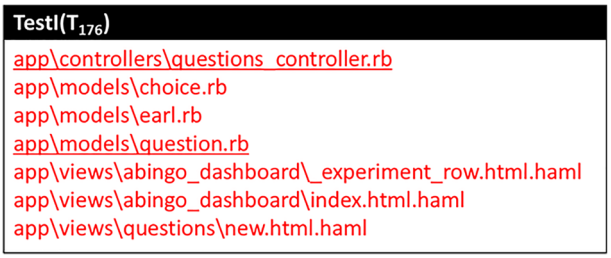

Fig. 5 Test-based task interface of T_{176}. The files in red are in the intersection between interfaces $TestI(T_{175})$ and $TestI(T_{176})$, and the underlined files are the conflicting ones

one having a smaller intersection size, suggesting a low conflict risk. Intuitively, if *TestI* predicts the files a task will change, the larger the intersection between two *TestI*, the higher the conflict risk between the tasks related to these *TestI*. The reason is developers will change more files in common, increasing the chance that parallel changes affect the same file hunk.

By relying on static analysis, TAITI cannot accurately identify the files executed by the tests, even though its predictions of file changes seem promising, as detailed by Rocha et al. (2019). Also, files in *TestI* might not require changes because developers might have implemented part of the functionality before, as expected for refactoring and bug fix. For example, $TestI(T_{175})$ has 21 files, and only 4 were actually changed by T_{175} (commits `58195d`, `18ff59`, `f8e9f6`, and `f1b0da`). Consequently, the tasks might not change all files in the intersection between *TestI*. In the presented case, the intersection of their *TestI* has seven files, but only two have changed (by commits `58195d`, `18ff59`, `f8e9f6`, and `f1b0da`, which are related to T_{175}, and by commit `6f349c` related to task T_{176}), and both files have a conflict. Finally, the tests might not sufficiently cover the tasks, preventing relevant files into *TestI*. For instance, both tasks changed the file *app/controllers/choices_controller.rb* without conflict, and such a file is not part of the intersection between *TestI*. All these possibilities motivate us to go deeper and investigate the viability to use *TestI* as a predictor of conflict risk between programming tasks and better understand how we can improve our whole strategy and TAITI.

Note that our simplified example abstracts other criteria developers usually evaluate when prioritizing programming tasks. Most of the time, the business value defined by stakeholders is the main criteria. Other influence factors are time and resource constraints, task complexity, and developer skills. Therefore, in a realistic context, merge conflict risk might support developers in case of an impasse among tasks. In other circumstances, the upfront knowledge about conflict risk might not change task prioritization. Still, it might guide the team to improve test planning and communication strategy to reduce possible side effects caused by conflicts. In addition, our example demonstrates a successful prediction of conflict risk even though task T_{175} is a refactoring. Considering that in the case of refactorings, the task's changeset might not include all relevant tests for it, the input for TAITIr might be incomplete, compromising predictions of conflict risk.

3 Research Questions

We conducted a retrospective study to answer a few research questions to evaluate whether *TestI* helps predict the risk of merge conflicts when integrating the code produced by two programming tasks.

First, given that *TestI* approximates the set of files changed by a programming task, we investigate whether the intersection between *TestI* interfaces could predict conflict risk. So, we ask the following question.

3.1 Research Question 1 (RQ1): Are Tasks with Non-Disjoint *TestI* Interfaces Associated with Higher Merge Conflict Risk?

To answer this question, we collect pairs of tasks that could be integrated (merged). Given that the state-of-the-practice often preconizes short sprints (between 1 and 4 weeks duration), we select tasks concluded with no more than 30 days of difference. We extract tasks from merge scenarios. For each merge commit, we determine the triple formed by left and right commits to be merged and a base commit, which is a common ancestor to left and right.

This way, a task is a set of all reachable commits between the left/right commit and the base commit. Using a logistic regression model, we assess the association between two binary variables related to task pairs: conflict risk and non-disjoint *TestI*. Conflict risk is the dependent variable that worthes true whenever the intersection between the set of changed files by the tasks is not empty. The non-disjoint *TestI* is the independent variable that worthes false whenever the *TestI* for tasks are disjoint. We use TAITIr to compute *TestI*.

Although the answer to *RQ1* allows us to observe whether the intersection between *TestI* relates to merge conflict risk, it does not allow us to assess how often the predictions of conflict risk apply. So, we ask the following question to evaluate the accuracy of *TestI* as a predictor of conflict risk between programming tasks.

3.2 Research Question 2 (RQ2): How Often Does *TestI* Predict Conflict Risk Between Two Tasks?

For answering this question, we evaluate precision and recall measures for predictions based on *TestI*. In this context, precision is the proportion of predictions that genuinely applies. In other words, it is the ratio between the number of task pairs in which both the tasks' changed files sets and the tasks' test interfaces intersect (true positives), and the number of task pairs in which their *TestI* intersect (true positives plus false positives). The recall is the proportion of conflict risk that *TestI* predicts. That is, the ratio between the true positives and the number of pairs whose tasks' changed files sets intersect (true positives plus false negatives).

In the context of conflict risk, recall might be more relevant than precision. A lower recall implies unexpected conflicts might often be observed, and the development team might not be prepared to deal with them. In contrast, a lower precision discourages (but not prevents) parallel execution of tasks that would not conflict, meaning the team would unnecessarily delay work on a task. Although it is undesirable, from the developer's perspective, a false alert of conflict risk does not require extra effort, contrasting with other tools based on static analysis. This way, developers might ignore conflict alerts when necessary. As discussed in Section 2, the teams usually evaluate other criteria when prioritizing programming tasks, such as the business value defined by stakeholders. Complementarily, we evaluate F_2 (Berry 2017), a variation of F-measure that weights recall higher than precision.

Next, if there is conflict risk between many remaining tasks in a backlog and the tasks under development, it would be helpful to provide developers with criteria for comparing conflict risks so that they could choose a less risky new task to work on. So, we ask the following question.

3.3 Research Question 3 (RQ3): Is the Intersection Size Between Two *TestI* Interfaces Proportional to the Number of Files Changed in Common by the Corresponding Tasks?

To answer *RQ3*, we investigate whether there is a correlation between the number of files simultaneously changed by both tasks and the size of the intersection between their *TestI* interfaces. A positive answer suggests that it is possible to reduce conflict risk by choosing or allocating tasks that minimize *TestI* intersection.

The questions so far assess the potential of *TestI* for predicting conflict risk between tasks. But to better assess *TestI* potential, we compare it with *TextI*, a predictor based on task interfaces obtained by observing files changed by past tasks with similar descriptions (a textual explanation about the task meaning). We design *TextI* based on Hipikat (Cubranic

et al. 2005), a tool that investigates project history to predict artifacts related to a task. A previous evaluation study (Rocha et al. 2019) shows that *TextI* is more precise than *TestI* when predicting the set of files changed by a given task, even though it presents a significantly lower recall rate. Thus, according to the following question, we further investigate such interfaces by comparing their ability to predict merge conflict risk.

3.4 Research Question 4 (RQ4): Is *TestI* a More Correct and Complete Predictor of Conflict Risk than *TextI*?

We answer that by checking whether *TestI* has better precision, recall, and F_2 measures than *TextI*. But we first assess whether *TextI* intersection relates to merge conflict risk, using a logistic regression model as *RQ1*. As previously explained, the dependent variable is conflict risk, and it worthes true whenever the intersection between the set of changed files by the tasks in a pair is not empty. But this time, the independent variable is non-disjoint *TextI*, which worthes false whenever the *TextI* for tasks are disjoint.

The *TextI* of a task t is the intersection between the three sets of files changed by the three past tasks with the most similar textual description to t. We assume the textual description of a task is the text of its Cucumber scenarios written in Gherkin with no automation code. Given a task relates to a commit set, the conclusion date of a task is the date of its last commit, and a past task of t is a task whose conclusion date is earlier than t's conclusion date. We compute the similarity between Cucumber scenarios as the cosine similarity between vectors of TF-IDF values (Salton and McGill 1986), as it is a simple, solid, and extensively used approach. In addition, we found that a relevant similar study (Thompson and Murphy 2014) used it. Such a study investigates a strategy to recommend a start point for a programming task that relies on the similarity between task descriptions and the overlap of considered and changed resources associated with the similar tasks as the basis for recommendations. As part of the process of computing similarity, with the aim of clean task descriptions, we preprocess the Cucumber scenarios by tokenizing the text based on spaces and punctuation, stemming it, and eliminating English and Gherkin keywords, such as `Given`, `When` and `Then`.

Besides studying the relation of *TestI* intersection with conflict risk, we explored whether *TestI* similarity relates to higher conflict risk, wondering if it might be an alternative predictor. So, we ask the following question.

3.5 Research Question 5 (RQ5): Does a Predictor Based on *TestI* Intersection Outperform a Predictor Based on *TestI* Similarity?

Given that *TestI* is a set of files, to answer *RQ5*, we compute *TestI* similarity using the cosine similarity between vectors of TF-IDF values (Salton and McGill 1986), reusing the approach explained in *RQ4* in a more simple context, as *TestI* is a set of filenames. The similarity is in the scale [0,1], zero meaning no similarity and one, maximum similarity. Then, we assess and compare precision, recall, and F_2 measures for predictions based on *TestI* intersection and *TestI* similarity.

Finally, according to Rocha et al. (2019), when dealing with an MVC-like application (e.g., web applications developed in Rails), *TestI* performs better when predicting changes in controller files. In addition, when a controller appears in *TestI*, the task quite often changes at least one file from the associated slice. As controllers uniquely identify an MVC slice (related model, view, controller, and auxiliary files), one could also use *TestI* to predict slices changes. Therefore, it would be interesting to investigate whether controllers in the

intersection of two *TestI* point out a higher chance of merge conflict risk in the related slices. So, we ask the last question.

3.6 Research Question 6 (RQ6): Are Tasks with Non-Disjoint *TestI* Interfaces Associated with Higher Merge Conflict Risk in MVC Slices?

For each task pair in our sample, we use TAITI to compute *TestI* for both tasks and check whether the interfaces have some controller file in common. In case it contains, we check if the tasks' changed files set intersects with any file from the MVC slice identified by the common controller. We developed a script for classifying files into MVC slices based on the folder structure of a Rails project, which organizes system files into a folder named `app` organized into subfolders such as `models`, `views`, and `controllers`. As when answering *RQ1*, we use a logistic regression model for assessing the association between two binary variables. The dependent variable tells whether there is any controller file in the intersection between *TestI*. The independent variable informs whether both tasks change the slice identified by the intersected controllers in *TestI*.

4 Study Setup

For answering the presented research questions, we analyze several task pairs. In this section, we describe how we construct our task pair sample and collect data.

4.1 Initial Project Selection

Given that TAITIr is specific for Rails projects that use Cucumber for specifying acceptance tests, we use a script[10] to mine GitHub repositories looking for projects that satisfy these requirements. The script queries Ruby projects using the GitHub API through a Java library (Eclipse EGit GitHub API Core). Next, it downloads the latest version of each project and checks whether the project uses libraries related to Rails and Cucumber, based on the gemfile (a Ruby file that lists all project dependencies) content. We restrict the search by avoiding projects created earlier than 2010,[11] as Cucumber and BDD were less popular before that, and TAITIr might not be compatible with older versions of Ruby and Rails.

Hoping to find a significant number of relevant projects, we perform the search in two independent steps, changing the strategy for sorting results, as the GitHub API provides up to 1,000 values per search. First, aiming to find the most active projects and avoid toy projects, we sort results by the date of the last commit on any branch in the repositories (the definition of "last update" by GitHub API), finding 877 Ruby projects. Second, to find popular projects, we limit the project's number of stars, starting with 15,000 and going down to 50, and sorted results by descending order of stars numbers. This time, we find 6,404 Ruby projects. In addition, aiming to find projects with a significant test dataset, we

[10] Available in our online Appendix (Rocha and Borba 2019).

[11] When searching for GitHub projects, we avoided projects created earlier than 2010. But we also selected projects referenced by the Cucumber's site, which includes projects created before 2010. Also, the creation date refers to the date a repository was created on GitHub, which does not necessarily reflect the date of the first commit. The project might have been first created in another code hosting platform and moved to GitHub.

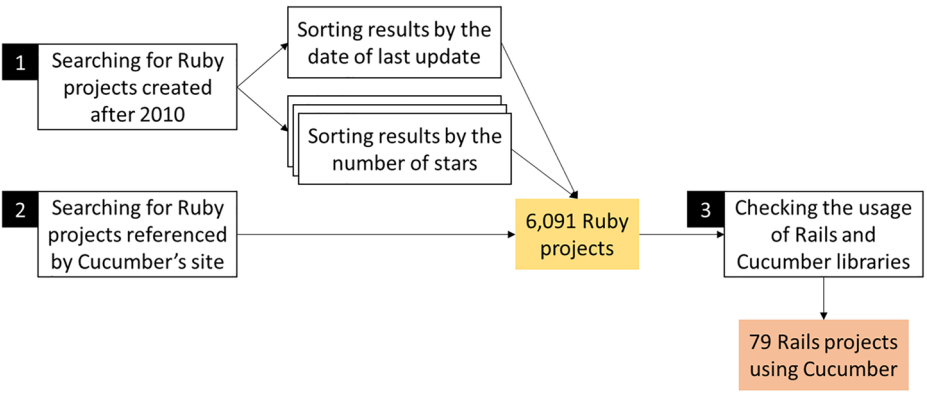

Fig. 6 Searching Rails projects that use Cucumber

investigate the Ruby projects referenced by Cucumber's site,[12] finding 26 projects. Figure 6 summarizes the whole searching process. In the end, we find 6,091 Ruby projects (excluding intersections among partial results). From these, we find 1,164 Rails projects, but only 80 projects use Cucumber. Finally, we discard one project because it is a fork of another project in the sample, resulting in 79 projects.

4.2 Task Extraction and Further Project Selection

From the 79 selected projects, we try to extract tasks with associated Cucumber tests. We consider a task consists of the commit set between a merge commit and the common ancestor[13] with the other commit the merge integrates. This way, we clone each project and search for merge commits performed until June 2019, excluding fast-forwarding merges, as it cannot cause conflicts. From each merge commit, we try to extract two tasks (which we call *merge tasks*), one derived from each parent of the merge commit. To illustrate, let us consider the merge scenario Fig. 7 presents. We can extract two tasks from base to merge, passing throw left (commit C_3) and right (commit C_5). The first task is the commits set $\{C_1, C_2, C_3\}$ and the second task is the commits set $\{C_4, C_5\}$.

During such a process, it is possible to find tasks that contain intermediate merges. When it happens, for constructing a more independent sample, we only collect the commits between the merge and the intermediate merge. Intermediate merges derived extensive tasks (large sets of commits and changed files) that lack cohesion and do not align with BDD tasks. Consequently, *TestI* intersects with many others, increasing false positives in the results concerning the prediction of conflict risk.

For instance, imagine that C_2 in Fig. 7 is a merge commit, as Fig. 8 illustrates. For simplicity, Fig. 8 omits detailed information related to the intermediate merge, such as the base commit and the commit set between left/right and base. This time, Task 1 is the unitary set $\{C_3\}$. We show a straightforward merge scenario, but many projects have a complex history that contains nested merges. In this sense, we do not always extract two tasks from

[12] https://cucumber.io/docs/community/projects-using-cucumber

[13] For simplicity, we assume a single most recent common ancestor. With so-called criss-cross merge situations in Git, there could be more than one.

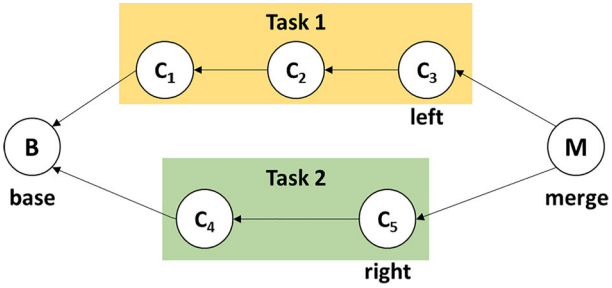

Fig. 7 Task extraction from a merge scenario

a merge commit. Also, we discard merges from which we cannot extract any task, which happens when there are successive merge commits.

Next, we discard tasks that do not contribute with (the associated commits do not change) both application code and Cucumber tests, as we cannot compute *TestI* and *TextI* for them. This way, we assume that the created or updated tests during a task execution relate to the task. We call the resultant task set as *candidate tasks*. Then, we discard redundant tasks that accumulate the contributions of previously concluded tasks. Specifically, tasks whose commit set is a subset of other tasks. Now we get a task set we call as *independent tasks*.

Finally, given we need task pairs to answer the research questions, we filter out projects with less than two tasks. After task extraction, we remain with 40 projects that have at least two tasks that satisfy our requirements and a set of 4,222 tasks. Among the 39 discarded projects, 11 projects do not have tasks extracted from merge commits: 5 projects only have fast-forwarding merges, and 6 projects have no merge commits. All projects with no merge commits have at least one not merged branch, two projects do not use pull requests, and the other 4 projects have open or closed pull requests. So, possibly there are no integrations in these projects, even by Git rebase. Also, 24 projects do not have tasks that contribute with application code and Cucumber tests (there are no tests related to the task, or they were added to the project by different merge commits). Finally, four projects have less than two eligible tasks.

4.3 Collecting Task Data and Further Project Selection

Finally, we collect the set of changed files, the *TestI*, and the *TextI* of each task. The set of files changed by a task is the union of the files modified by its commits, and that exists

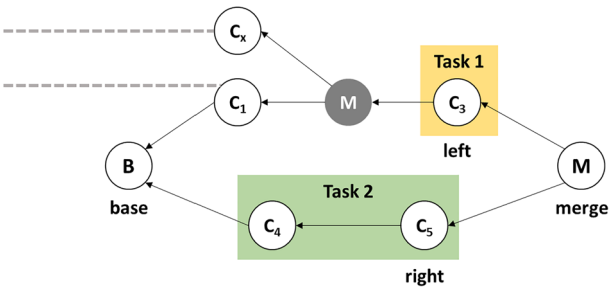

Fig. 8 Task extraction from a merge scenario containing an intermediate merge commit

when the task was concluded (date of its most recent commit). Verifying files' existence is necessary to avoid inconsistencies, such as a task that removes a file that another task concurrently modifies back to the project. We do not evaluate the occurrence of commits that revert changes of other commits. In such cases, we simply add the changed file into the set of files modified by a task. We use TAITI for computing *TestI* (as described in Section 2). As part of this process, TAITI collects the tests associated with each task by further analyzing the task commits using a syntactic differencing strategy for Cucumber tests. TAITI compares each commit with its parent, identifying changed Cucumber scenarios and step definitions; each changed scenario or step is considered to be a test associated with the task. For avoiding inconsistencies, such as when a commit deletes a Cucumber test created by a previous commit, TAITI consolidates the result by selecting the last version of the Cucumber test (i.e., the version from the most recent commit).

As we need to compute *TestI* and these rely on valid test description and code, we discard the following: tasks with no associated Cucumber test; tasks with partially implemented tests (some unimplemented step definitions); and tasks with step definitions that TAITI cannot parse. We also discard tasks whose *TestI* is empty, as this is often associated with TAITI limitations or limited test coverage of the tests related to the task, obtaining a set of *relevant tasks*.

As we need task pairs to answer the research questions, we filter out projects with less than two relevant tasks. As a result, we get a sample of 1,762 tasks from 27 projects, which represents 41.7% of the tasks extracted from the initial set of 4,222 tasks. Among the exclusion criteria we adopted, the last one, the ability to successfully compute a non-empty *TestI*, is the most restrictive.

Additionally, for answering *RQ4*, we try to compute a non-empty *TextI* for the 1,762 relevant tasks. Given *TextI* depends on project history, we discard 36 tasks for which we cannot find three similar past tasks[14] (as explained in Section 3.4). We also discard 665 tasks with empty *TextI* interfaces. An empty *TextI* means there is no intersection between the changed files sets of their three most similar past tasks. We also filter out projects with less than two tasks with non-empty *TextI*, getting a preliminary sample of 1,057 tasks from 22 projects after computing *TextI*. In summary, we exclude tasks with empty *TestI* or empty *TextI* because we focus on understanding the cases where both interfaces seem to work, hoping to conduct a fair evaluation.

At last, we simulate the integration (merge) of likely concurrent tasks, computing the intersection between the set of files changed by both tasks. We compute all task pairs per project by grouping tasks concluded with no more than 30 days of difference. This way, we group tasks extracted from different merge commits. We adopt such an approximation rather than merging tasks to check conflict occurrence to increase our sample, given we could only faithfully reproduce a few merges because most task pairs are not extracted from the same base and merge commit. In addition, we collected a few effectively concurrent tasks (based on commits' date) that satisfy our inclusion and exclusion criteria. We filter out projects with no task pair (3 projects). In the end, we get a final sample of 990 tasks from 19 Rails projects and a set of 6,360 task pairs. Figure 9 summarizes the whole process for constructing our dataset explained so far.

[14]Note that every project has at least the oldest task, for which there are no similar past tasks.

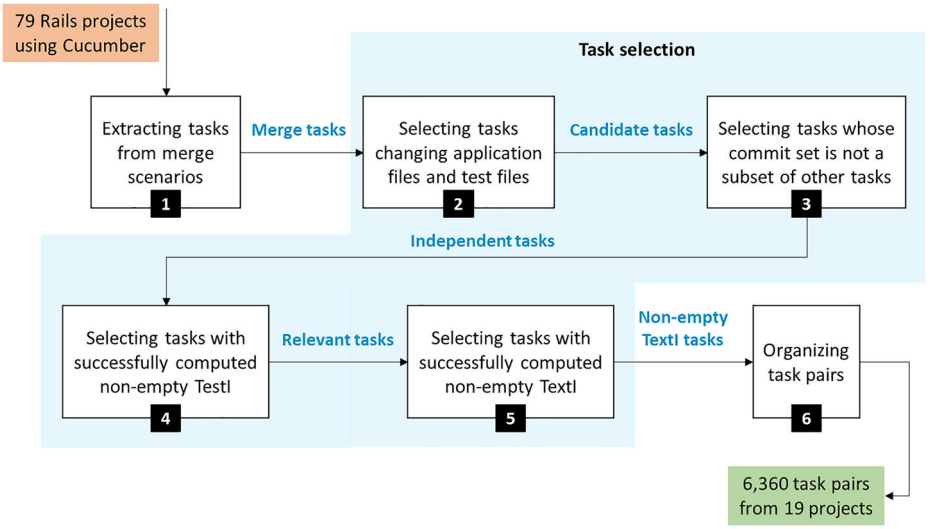

Fig. 9 Process for constructing our dataset

4.4 Task pair sample

Table 1 summarizes the steps for constructing our task sample and the corresponding task numbers for the final project set. Note that it organizes tasks in named groups to distinguish them according to the selection step. Still, the primary definition of a task as a set of commits between the base and left/right commits of a merge scenario does not change. As explained in the previous sections, we collect tasks extracted from merge commits that do not contain intermediate merges (*Merge tasks*). Next, we select tasks that contribute to both application code and Cucumber tests (*Candidate tasks*) and tasks that are not a subtask of other tasks, i.e., tasks whose commit set is not a subset of other tasks (*Independent tasks*). We compute *TestI* for all independent tasks and select tasks for which we can successfully compute a non-empty *TestI* (*Relevant tasks*). Similarly, we compute *TextI* for all relevant tasks and select the ones that have a non-empty *TextI*. At last, we select tasks that have at least one other task that is less than 30 days apart, which we classify as possible concurrent tasks (*Concurrent tasks*). Finally, we derive a number of task pairs or integrations (*Pairs*) per project. This way, the number of pairs varies among projects. For example, project *diaspora/diaspora* has 92 concurrent tasks and 296 pairs, whereas project *rapidftr/RapidFTR* has 112 concurrent tasks and 592 pairs. Such a difference relates to the time interval between tasks. The second project has more likely concurrent tasks than the first.

Although the overall process for constructing the task sample is similar to the one from our previous evaluation study, their selection and exclusion criteria are different, resulting in distinct task sets that have only one task in common.

While constructing our task sample, we do not systematically target representativeness and diversity (Nagappan et al. 2013). Even so, we observe some variety concerning the attributes in Table 2. Note that we collected these attributes from the project perspective

Table 1 Construction of our task pair sample

Repository Name	#Tasks						
	Merge	Candidate	Independent	Relevant	TextI	Concurrent	Pairs
allourideas.org	188	30	30	26	21	13	41
e-petitions	446	136	136	99	71	64	1,140
whitehall	4,525	368	365	291	161	157	781
bsmi	254	52	51	9	3	3	3
enroll	6,079	202	188	29	20	20	82
diaspora	4,347	284	274	161	100	92	296
action-center-platform	247	32	32	25	18	17	38
gitlabhq	29,147	504	500	6	5	5	10
wontomedia	63	10	8	7	2	2	1
jekyll	2,403	135	134	88	64	55	88
Claim-for-Crown-Court-Defence	2,164	301	298	214	161	160	2,294
one-click-orgs	336	39	32	31	28	20	46
opengovernment	632	3	3	3	2	2	1
openproject	9,565	418	409	25	10	9	12
otwarchive	2,625	496	483	365	141	138	495
RapidFTR	1,280	203	198	149	112	112	592
quantified	233	19	18	9	5	4	2
sequencescape	2,221	436	411	13	6	6	7
sharetribe	3,034	330	322	183	121	111	431
TOTAL	69,789	3,998	3,892	1,733	1,051	990	6,360

in October 2019,[15] but these might not apply during the time the developers concluded the tasks of our sample. The project *gleneivey/wontomedia* has three collaborators. Still, it does not seem like a toy system, i.e., a system with no practical usage that someone creates for study purposes. Also, the most recent version of two projects has no Cucumber tests, implying they gave up using Cucumber. This fact does not compromise our results because the projects used Cucumber when the extracted tasks were concluded. All data related to our sample is available in the online Appendix (Rocha and Borba 2019).

5 Results

In this section, we present the results of our retrospective study based on the development history of a set of completed tasks, following the structure defined by our research questions. Our Appendix (Rocha and Borba 2019) provides detailed information.

[15]We first search for merge commits performed until June 2019. Only when we consolidate our task pair sample, we collect detailed information about the selected projects.

Table 2 Diversity of projects in our task pair sample

Repository Name	Description	#Stars	#LOC	#Tests	#Commits	#Authors	#Forks
allourideas.org	A tool for groups to collect and priorize information.	134	56,884	126	2,372	20	44
e-petitions	The UK Government's petitions service.	251	93,447	257	2,692	47	63
whitehall	A content management app for the UK Government.	549	217,235	263	23,435	336	180
bsmi	The website of an educational program of Berkeley University.	3	245,862	49	944	11	0
enroll	An enrollment system for health benefit exchanges.	20	2,292,391	231	49,693	196	33
diaspora	A privacy-aware, distributed, open source social network.	12,296	195,654	277	19,915	585	2,911
action-center-platform	A tool for creating targeted campaigns where users sign petitions, contact legislators, and engage on social media.	168	31,509	48	2,205	29	41
gitlabhq	A DevOps platform.	22,020	2,346,933	0	149,603	3,282	5,598
wontomedia	A web app for community creation of an information classification scheme.	5	76,106	89	694	3	0
jekyll	A simple, blog-aware, static site generator.	38,807	59,733	259	11,093	1,063	8,476
Defence	A web app of the UK Ministry of Justice.	8	211,081	46	8,333	59	1
one-click-orgs	A platform for creating a legal structure and get a system for group decisions.	42	46,689	206	2,496	25	12
opengovernment	An app for presenting US open government data.	213	134,313	11	2,239	21	157
openproject	A project management system.	3,259	742,225	0	53,437	264	1,009
otwarchive	An app for hosting archives of fanworks.	447	294,635	1,235	14,361	164	229
RapidFTR	An app for sharing info about children in emergencies.	287	98,820	274	4,939	252	332
quantified	A time tracking system.	47	61,532	75	1,090	7	8
sequencescape	A cloud based and extensible LIMS system.	48	40,005	461	10,515	48	24
sharetribe	A platform to create peer-to-peer marketplaces.	1,813	415,784	279	20,655	107	1,101

5.1 RQ1: Are Tasks with Non-Disjoint *TestI* Interfaces Associated with Higher Merge Conflict Risk?

5.1.1 Tasks with Non-Disjoint *TestI* Interfaces More Likely Modify Files in Common

Tasks with non-disjoint *TestI* interfaces are 2.07 times (the odds ratio of the logistic regression, as explained in Section 3.1-RQ1) more likely to change at least one file in

common. This finding corroborates the idea of using *TestI* to assess conflict risk between programming tasks in the context of BDD projects, as explained in Section 2.

5.1.2 Tasks with Non-Disjoint *TestI* Interfaces are More Strongly Associated with Concurrent Modifications to Ruby and HTML Files

TestI only contains a few specific application files, i.e., Ruby files, .html files, and its variants, such as .haml and .erb files, which are the main files used or accessed by Cucumber tests. For such reason, we also investigate whether *TestI* performs better when predicting merge conflict risk exclusively in *files reachable from TestI*. So, we repeat the same kind of analysis as before. Still, we filter the tasks' changeset by excluding configuration files, test files, and files in other programming languages such as JavaScript.[16] This time, we find that when the *TestI* interfaces of two tasks intersect, the tasks are 2.95 times more likely to change a common file reachable by *TestI*. As expected, we obtain a higher odds ratio by reducing false negatives because the restricted set of changed files more likely relates to the task purpose, which is better predicted by *TestI*. For example, we have disjoint *TestI* pairs when two tasks only make parallel changes in a configuration file, which plays a more general-purpose in the project, given configuration files are not part of *TestI*.

In summary, we observe that tasks with non-disjoint *TestI* interfaces are associated with higher merge conflict risk, especially conflict risk in Ruby and HTML files.[17]

5.2 RQ2: How Often Does *TestI* Predict Conflict Risk Between Two Tasks?

5.2.1 A Minimal Intersection Between *TestI* Interfaces is the Best Predictor of Conflict Risk Between Tasks

For answering *RQ2*, we evaluate precision, recall, and F_2 measures for predictions based on *TestI* intersection for 6,360 task pairs. To explore how *TestI* intersection size affects prediction accuracy, we consider five predictors, varying the minimum intersection size from 1 to 5. That is, we predict conflict risk between a given task pair when the size of the intersection between their *TestI* is at least n, ranging from 1 to 5.

As Table 3 summarizes, we find that a minimal *TestI* intersection (1 file) is the best predictor of conflict risk, as it has the best values of recall and F_2. We can also observe that there is a subtle variation[18] in precision when we change the value of the minimal intersection size. When we increase the value of the minimal intersection, the false-negative numbers also increase, whereas both the true positives (TPs) and the false positives (FPs) might decrease. The final balance impacts more recall than precision because the first depends on TP and FN (false negative, which inversely vary according to different rates), while the second depends on TP and FP (which decrease). For example, when the minimum intersection size is 4, there are 3,106 TPs, 1,912 FPs, and 606 FNs. When the minimum intersection size is 5, there are 2,855 TPs, 1,665 FPs, and 857 FNs.

We observe that recall outperforms precision in the overall result with a significant advantage. By considering a reduced minimal intersection size as one file, we find the intersection between *TestI* points out conflict risk for 96.7% task pairs of our sample (the

[16] We restrict the set of changed files to Ruby and .html files (and common variations) into `app` or `lib` folders.

[17] As a matter of brevity, we omit variants of HTML files, such as .haml and .erb files.

[18] For simplicity, we round the values, but they are not identical in the third decimal.

Table 3 Precision, recall, and F_2 measures of the *TestI* intersection predictor

Intersection lower bound	Predicted positive condition rate (%)	Precision	Recall	F_2
1	96.70	0.59	0.98	0.86
2	92.41	0.60	0.95	0.85
3	88.07	0.60	0.91	0.83
4	78.90	0.62	0.84	0.78
5	71.07	0.63	0.77	0.74

percentage of flagged integrations presented by the second column at Table 3). As Table 4 summarizes, only 58.4% of them are risky (the tasks did change at least one file in common). We observe there is much FP and not much FN, which benefits recall and compromises precision. This way, by comparing the predicted positive condition rate, someone might consider the predictor with a minimum intersection size of 4 files as a better option because it suggests conflict risk for 79% task pairs and presents acceptable values of precision, recall, and F_2. In all cases, as better explained in Section 1, a false positive in our context does not imply extra wasted developer effort; instead, it discourages the parallel execution of tasks. This way, developers might ignore low-risk rates.

To better understand results, we inspect them from a project perspective. As an example, Table 5 summarizes the results per project of the predictor with the minimum intersection size. Note that the dataset is unbalanced: Projects with one integration and others with more than 1,000 integrations. Most projects (18) have a recall value in range]0.9, 1.0]. The exception is project *action-center-platform*, whose recall value is 0.78. Regarding precision, we observe a more diverse distribution: 5 projects whose precision is in range]0.9, 1.0], 12 projects whose precision is in range]0.5, 0.9], and two projects whose precision value is under 0.5 (projects *otwarchive* and *whitehall*). This way, we investigate the three mentioned projects in detail: *action-center-platform*, *otwarchive*, and *whitehall*.

The prediction results of *action-center-platform* have no FPs and 8 FNs. Contrasting, the results of *otwarchive* and *whitehall* have few FNs (3 and 4, respectivelly) and much FPs (251 and 412, respectivelly). In all cases, prediction quality relates to *TestI* ability to predict file changes: *TestI* fails to predict the files changed by the tasks, deriving failing risk predictions. In the case of project *action-center-platform*, *TestI* has few files (if we compare its size with the size of the set of changed files), meaning TAITI ends test analysis prematurely. For example, the tests reference unknown views, which might happen when the tests are outdated, or TAITI cannot map the referenced view into a project file. In addition, many file changes are unpredictable by *TestI* because they affect other files than Ruby and HTML files, or there are implicit relationships among files defined by Rails that are not reachable by the tests. As previously discussed by Rocha et al. (2019), projects *otwarchive* and *whitehall* consistently have lower precision values related to file change prediction. The

Table 4 Characterization of the intersection between changed file sets in our task pair sample

Property	Pairs	% of Pairs
The intersection has at least one file (risky pairs)	3,712	58.4% of all pairs
The intersection has at least one Ruby or HTML file	2,316	36.4% of all pairs
The intersection has no Ruby or HTML files	1,396	37.6% of all risky pairs (21.9% of all pairs)

Table 5 Precision, recall, and F_2 measures of the *TestI* intersection predictor with a minimum intersection size of one file from a project perspective

Repository Name	Integrations	Precision	Recall	F_2
allourideas.org	41	0.85	1.00	0.97
e-petitions	1,140	0.64	0.94	0.86
whitehall	781	0.46	0.99	0.80
bsmi	3	0.67	1.00	0.91
enroll	82	0.67	1.00	0.91
diaspora	296	0.74	0.99	0.93
action-center-platform	38	1.00	0.78	0.81
gitlabhq	10	0.90	1.00	0.98
wontomedia	1	1.00	1.00	1.00
jekyll	88	0.53	1.00	0.85
Claim-for-Crown-Court-Defence	2,294	0.57	0.99	0.86
one-click-orgs	46	0.72	1.00	0.93
opengovernment	1	1.00	1.00	1.00
openproject	12	0.92	1.00	0.98
otwarchive	495	0.49	0.99	0.82
RapidFTR	592	0.59	0.95	0.85
quantified	2	1.00	1.00	1.00
sequencescape	7	0.86	1.00	0.97
sharetribe	431	0.72	1.00	0.93

explanation is the limitations of the conservative strategy of code analysis supported by TAITI and the test coding style adopted by the projects, causing much coincidence of identifiers among the project and library methods, and confusion when dealing with overridden methods.

To conclude, we clarify that we experimentally delimit a small value range for the minimal intersection size (our predictor), according to the size of the intersection between *TestI* when there is a conflict risk and to the quality of prediction results. First, we observe that 58.4% of the integrations (see Table 4) have conflict risk. Also, almost half of the risky integrations (49%) have up to 10 files in the intersection between *TestI*. So, we first computed precision and recall using an intersection limit ranging from 1 to 10. Still, we observed a substantial reduction in results quality (mainly in the recall measure) when such a limit is larger than 5, as Fig. 10 illustrates.

5.2.2 The Prediction of Conflict Risk is Better When Considering All Files Concurrently Modified by the Tasks

As in Section 5.1, we also investigate the alternative result that restricts a task's changed files set by excluding files not reachable by *TestI* (Table 6), expecting that such a strategy would show better results by reducing FNs. We observed a reduction of FNs. But we further found that 37.6% of all risky integrations (see Table 4) only affect files that are not

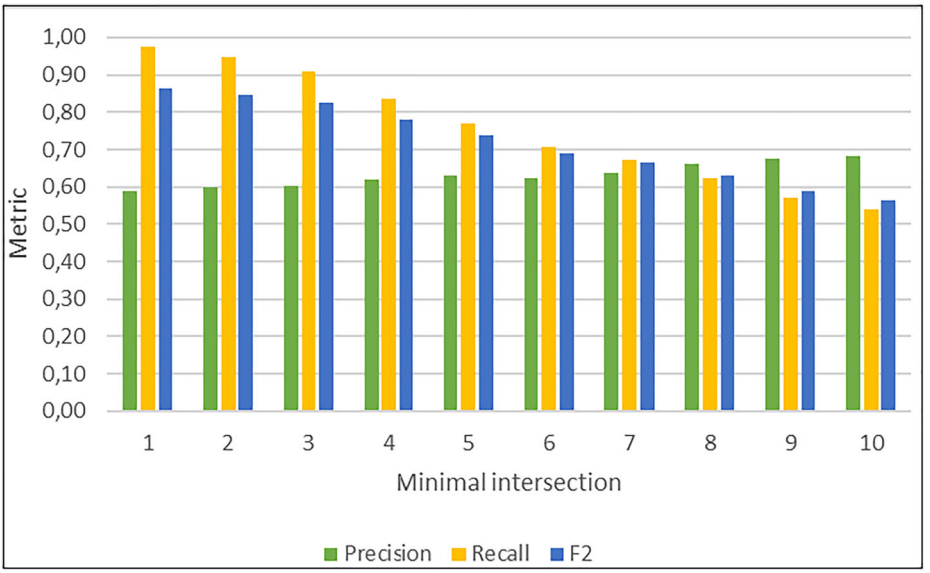

Fig. 10 Predictions quality according to the minimal intersection size between two *TestI* interfaces. The presented metrics are the mean value for our sample

reachable by *TestI*, decreasing TP, and increasing FP. Such a phenomenon severely decreases precision and explains the slight increase in recall (a reduction of FN benefits recall whereas a decrease of TP impairs recall), whether we compare results to those applied to all files (Table 3).

5.2.3 For some Projects, Discarding Test Preconditions when Computing *TestI* Interfaces Might Benefit the Prediction of Conflict Risk

Given that many tests require similar setup steps, such as those for specifying user authentication actions, the interface files derived from these steps could not be relevant to most tasks. Therefore, test interfaces might have many unchanged files by the corresponding task. To investigate whether *TestI* intersections could consist mainly of such files, we

Table 6 Precision, recall, and F_2 measures of the *TestI* intersection predictor when restricting a task's changed file set to contain only Ruby and HTML files

Intersection lower bound	Precision	Recall	F_2
1	0.37	0.98	0.74
2	0.38	0.95	0.73
3	0.38	0.92	0.71
4	0.39	0.85	0.69
5	0.40	0.78	0.66

Table 7 Precision, recall, and F_2 measures of the alternative *TestI* intersection predictor, which discards application files accessed by Cucumber `Given` steps

Intersection lower bound	Precision	Precision (%)	Recall	Recall (%)	F_2	F_2 (%)
1	0.64	+8.34	0.88	−9.43	0.82	−4.86
2	0.65	+8.97	0.78	−17.39	0.75	−11.30
3	0.66	+9.58	0.70	−22.77	0.69	−15.97
4	0.67	+7.64	0.65	−21.94	0.66	−16.12
5	0.67	+6.22	0.61	−20.39	0.62	−15.44

The percentual values mean the proportion of increment or decrement related to the original predictor

additionally used TAITI to compute a variety of *TestI* that discards the files inferred from test preconditions (`Given` steps of Cucumber scenarios, as Fig. 3a illustrates).

We find that such an alternative predictor slightly increases precision but decreases recall, as summarized in Table 7. For instance, by adopting a minimal *TestI* intersection, the precision increases by 8.34%, and recall decreases by 9.43%, resulting in a reduction of F_2 by 4.86%, discouraging the usage of the *TestI* variation as a predictor. However, from a project perspective (see Table 8), we found the alternative predictor performs better for 3

Table 8 Precision, recall, and F_2 measures of the alternative *TestI* intersection predictor from a project perspective

Repository Name	Precision	Precision (%)	Recall	Recall (%)	F_2	F_2 (%)
allourideas.org	0.83	−2.94	0.83	−17.14	0.83	−14.30
e-petitions	0.69	+6.96	0.87	−6.78	0.83	−3.47
whitehall	0.48	+4.60	0.96	−2.81	0.80	−0.35
bsmi	1.00	+50.00	1.00	0.00	1.00	+10.00
enroll	0.67	0.00	1.00	0.00	0.91	0.00
diaspora	0.75	+1.04	0.96	−2.75	0.91	−1.83
action-center-platform	0.95	−4.55	0.54	−30.77	0.59	−27.53
gitlabhq	0.88	−2.78	0.78	−22.22	0.80	−18.69
wontomedia	1.00	0.00	1.00	0.00	1.00	0.00
jekyll	0.64	+20.11	1.00	0.00	0.90	+5.64
Claim-for-Crown-Court-Defence	0.67	+18.02	0.84	−15.26	0.80	−7.30
one-click-orgs	0.79	+10.80	1.00	0.00	0.95	+2.58
opengovernment	1.00	0.00	1.00	0.00	1.00	0.00
openproject	0.86	−6.49	1.00	0.00	0.97	−1.47
otwarchive	0.49	0.00	0.99	0.00	0.82	0.00
RapidFTR	0.61	+3.17	0.93	−2.44	0.84	−0.89
quantified	1.00	0.00	1.00	0.00	1.00	0.00
sequencescape	0.86	0.00	1.00	0.00	0.97	0.00
sharetribe	0.80	+11.76	0.81	−19.23	0.81	−13.01

The percentual values mean the proportion of increment or decrement related to the original predictor

projects from our sample (*bsmi*, *jekyll*, and *one-click-orgs*): the values of precision and F_2 increase, with no changes in the recall, implying a reduction of FP numbers. Also, we found no changes in the results of 6 projects, meaning there are no application files exclusively accessed by `Given` steps. The test coding style varies between projects as some of them reproduce user interactions faithfully and others omit details to improve performance. In this sense, these results suggest that our alternative *TestI* predictor depends on the particularities of each project and should not be universally applied for all projects.

5.2.4 The Intersection Between *TestI* Interfaces Might Predict Conflict Risk, Even When it Does Not Predict the Potential Conflicting Files

So far, we discuss the prediction of conflict risk among tasks by abstracting the files in which the conflict might occur. To better evaluate our predictions, we investigate how often *TestI* guesses the risky files, that is, the files that are more vulnerable to conflicts because tasks concurrently changed them. We observe that 58.4% of the integrations in our sample have the risk of a merge conflict, i.e., the tasks change at least one file in common. For 35.8% of the risky integrations (see Table 9), the intersection between *TestI* predicts at least a potential conflicting file. Whether we consider files genuinely reachable by *TestI*, 36.4% of the integrations are risky (see Table 4). And as expected (the number of correct predictions does not change), the accuracy is higher in such a case: the intersection between *TestI* predicts some potential conflicting file in 57.4% of the risky integrations.

Sometimes limitations and imprecisions of TAITI tool, which Rocha et al. (2019) explains, prevent the inclusion of a potential conflicting file in *TestI*. For example, because TAITI does not analyze other application files beyond views, *TestI* reaches only the surface of the code that the tests could exercise. Consequently, conflicting files are not covered. In other cases, *TestI* cannot guess a potential conflicting file because of limitations related to the tasks. For instance, by manually inspecting some task pairs, we observe no clear relationship between tasks and their tests, suggesting they are not cohesive and probably have poor test coverage.

By relating all observations so far, we conclude that even when developers do not change the files included in *TestI*, the intersection between *TestI* might help. Such an intersection might reflect a degree of proximity between the parts of the code changed by both tasks, eventually leading to conflicts, even in files not directly reached by the interfaces, as Fig. 11 illustrates. In such a figure, *File 2* points out that there is conflict risk between tasks T_1 and T_2. The conflict affects *File 1*, which is not part of the intersection of $TestI(T_1)$ and $TestI(T_2)$, but *File 2* depends on *File 1*. Therefore, even when *TestI* does not predict the potential conflicting files, it might indicate the risk of conflicts between programming tasks.

Table 9 Characterization of the intersection between *TestI* interfaces in our task pair sample

Property	#Pairs	#Pairs (%)
The intersection predicts at least one potential conflicting file	1,330	35.8% of all risky pairs (20.9% of all pairs)
The intersection has some controller file	4,644	73% of all pairs
The intersection has at least one file in common from some related slice	1,205	26% of integrations that have some controller file in the *TestI* intersection (19% of all integrations)

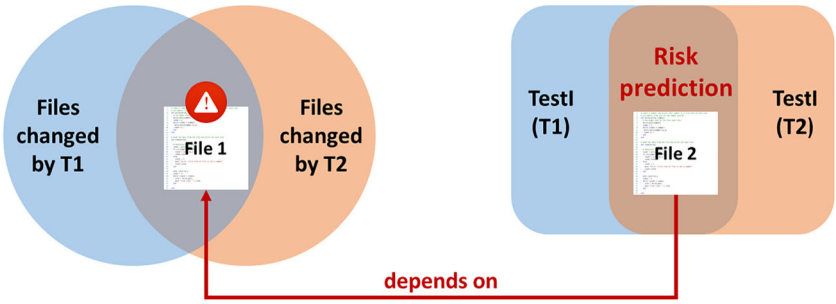

Fig. 11 The intersection between *TestI* interfaces might predict conflict risk between programming tasks, even when it does not predict the conflicting file

5.3 RQ3: Is the Intersection Size Between Two *TestI* Interfaces Proportional to the Number of Files Changed in Common by the Corresponding Tasks?

5.3.1 The Larger the Intersection Between *TestI*, the Higher the Risk of Merge Conflict Between Tasks

Given that our data deviates from normality, we answer *RQ3* by computing the Spearman's rank correlation coefficient with $\alpha = 0.05$ and the Cohen's assignment of effect size's relative strength ($0.10 <=$ small < 0.3, $0.3 <=$ medium < 0.50, and large $>= 0.50$).

We find that the larger the intersection size between *TestI*, the larger the number of files changed by both tasks, considering a small correlation ($p < 0.001$ and $\rho = 0.21$). The same applies when dealing only with Ruby and HTML files (i.e., files reachable from *TestI*, as explained in Section 5.1), with $\rho = 0.15$. We cannot expect a significant correlation because developers might not change all files in *TestI*.

In practice, this result suggests that developers can reduce the likelihood of merge conflicts by prioritizing tasks based on the size of the intersection among their test-based task interfaces. It is preferable to concurrently develop tasks whose *TestI* interfaces have smaller intersection sizes, as they likely change fewer files in common. In such a context, we consider a dynamic schedule that depends on selecting the next task to be handled by each developer. As illustrated in Section 2, Andrew and Becca could prevent conflicts by selecting a task whose *TestI* has a minor intersection with the *TestI* of tasks T_{175} or T_{176}.

In summary, we observe that the intersection size between two *TestI* interfaces correlates with the number of files changed in common by the corresponding tasks.

5.4 RQ4: Is *TestI* a More Correct and Complete Predictor of Conflict Risk than *TextI*?

5.4.1 Tasks with Non-Disjoint *TextI* Interfaces More Likely Change a Common File

When the *TextI* interfaces of two tasks intersect, the tasks are 3.20 times (the odds ratio of the logistic regression, as explained in Section 3.4) more likely to change a file in common. Such a result seems better than the *TestI* result (odds ratio of 2.07) because the odds ratio is higher and the deviance[19] is lower. In the case of *files reachable from TestI* (i.e., Ruby and HTML files, as explained in Section 5.1), the likelihood is 2.25, a smaller value, contrasting

[19] As a matter of brevity, we present detailed results in our online Appendix.

with the *TestI* result. This result implies many potential conflicting files according to *TextI* cannot be part of *TestI*, as false negatives increase.

In any case, these findings corroborate with the idea that someone might use *TextI* as a predictor of conflict risk. Still, a complete comparison between *TestI* and *TextI* as predictors depends on the evaluation of precision, recall, and F_2.

5.4.2 *TextI* is a More Precise Predictor of Conflict Risk than *TestI*, but it Predicts Fewer Risks

Table 10 summarizes the evaluation results of *TextI*. We show only minimum intersection sizes 1 and 2 because the recall is insignificant for larger values. We can observe that the improvement in precision is expressive for *TextI*, but the recall is severely affected, resulting in significantly worse F_2 values for *TextI*. In practice, if a file relates to the most similar past tasks (all of them modified the file), it has more chance to connect to the new task. Then, whether most files in *TextI* genuinely relate to the tasks, non-disjoint *TextI* interfaces have more chance of a TP of conflict risk. This way, the number of FPs decreases. However, given that recall is more relevant than precision in the context of conflict prevention (see Section 3.2), such a result discourages the usage of *TextI* as a predictor of conflict risk between programming tasks.

To better understand results, we inspect some task pairs for which predictions are contradictory: The *TestI*-based predictor points to conflict risk and the *TextI*-based predictor does not, and vice-versa. For example, let us see tasks T_{215} and T_{256} from project *e-petitions*. These tasks make parallel changes in two files, the *TestI*-based predictor points to conflict risk (although it cannot guess the conflicting files), but the *TextI* interfaces are disjoint. First, we observed limited test coverage for both tasks. The tests of T_{215} focused on archived petitions, but the code changes include refactorings and generic layout improvements. The tests of T_{256} focused on record notes, but the code changes include refactorings, layout improvements, and requirements changes, such as changes on user permissions. We rely on textual similarity among Cucumber scenarios for computing *TextI* and test code for computing *TestI*, so tests impact both predictors. Even so, the intersection among *TestI* interfaces reflects some degree of functional proximity between the tasks. Contrasting, *TextI* interfaces do not capture such proximity in this case. By inspecting the entry similar past tasks of T_{215} and T_{256}, we observed that they seem not cohesive tasks, resulting in a diverse set of changed files.

In case of tasks T_{332} and T_{345} from project *e-petitions*, we found the oposity result. These tasks make parallel changes in four files, the *TextI*-based predictor points to conflict

Table 10 Precision, recall, and F_2 measures of *TextI* intersection predictor

Intersection lower bound	Predicted positive condition rate (%)	Precision	Recall	F_2
1	35.82	0.75 (+27.84%)	0.46 (−52.65%)	0.50 (−41.94%)
2	14.72	0.81 (+34.80%)	0.20 (−78.53%)	0.24 (−71.81%)

The percentual values mean the proportion of increment or decrement related to the *TestI* intersection predictor

risk, but the *TestI* interfaces are disjoint. This time, we observed that both tasks relate to the searching mechanism for petitions provided by the application. The intersection among *TextI* interfaces shows that similar past tasks changed some test files in common. However, TAITI computes a limited *TestI* interface for task T_{345}, i.e., the tool prematurely concludes the test code analysis. This way, based on the presented examples and many others we carefully analyzed, we reinforce the influence of the test coverage, the task cohesion, and the limitations of TAITI on results.

As in Section 5.2, we also investigate the alternative result that restricts a task's changed files set by excluding files not reachable by *TestI*. But the effect over predictions of conflict risk based on *TextI* is similar to the previously reported.

5.4.3 A Predictor of Conflict Risk Based on Both *TestI* and *TextI* Underperforms a Predictor Based on *TestI*

Finally, as a previous study (Rocha et al. 2019) suggests that the files in the intersection between *TestI* and *TextI* are more likely changed by the corresponding task, we investigate here whether they are more strongly associated with conflict risk. Thus we also evaluate a hybrid task interface based on the intersection between *TestI* and *TextI*. The hybrid task interface further improves precision (0.86) and decreases recall (0.14) and F_2 (0.17). The explanation is similar to the *TextI* results. The intersection between *TestI* and *TextI* is less probable and probably filters out files wrongly included in *TestI* by TAITI. So, FPs further decrease, benefiting precision. However, most potential conflicting files are not covered, increasing FNs and compromising recall.

In summary, we observe that *TestI* is a better predictor of conflict risk than *TextI*, and a hybrid task interface based on both *TestI* and *TextI* is not a viable alternative predictor.

5.5 RQ5: A Predictor Based on *TestI* Intersection is Better than a Predictor Based on *TestI* Similarity

First, given that *TestI* is a set of files, we confirm a relation between *TestI* similarity and conflict risk using a logistic regression model, similar to *RQ1*. This time, the independent variable is the similarity between two *TestI*, which worthes false whenever there is no similarity between interfaces. Next, we observe that the precision and recall results from the predictor based on *TestI* similarity are very similar to those from the predictor based on *TestI* intersection, but the second is slightly better.

By further investigating the results, we realize that the proportion between equalities and differences, which impacts the similarity measure, might disturb the conclusion about conflict risk. It does not matter how different two *TestI* interfaces are; if they have common files, there is conflict risk. For example, the similarity between *TestI* of two task pairs from project *alphagov/whitehall* is 0.86 and 0.95, respectively. The task pair with lower similarity changed 34 files in common, and the other task pair, two files, meaning the conflict risk is higher for the first task pair, contradicting the conclusion of the similarity rate. However, confirming the conflict risk, the intersection between *TestI* of the task pair with lower similarity has 58 files, whereas the other task pair has 27 files.

In summary, although a predictor based on *TestI* similarity performs similar to the one based on *TestI* intersection, the second is the best option because it better reflects the notion of conflict risk and is cheaper to compute.

5.6 RQ6: *TestI* Cannot Predict Conflict Risk in MVC Slices

By investigating our sample, we observe that 73% of the task pairs have some controller file in the intersection between the *TestI* of the integrated tasks (see Table 9). From this subset, the tasks of 26% integrations (see Table 9) change at least one file in common from some slice identified by controllers in the intersection between two *TestI*. But the logistic regression model does not confirm that controllers in *TestI* are predictors of conflict risk in their associated MVC slices. Thus, we conclude that *TestI* cannot predict conflict risk in MVC slices, which means that recommending developers avoid concurrent execution of tasks that focus on the controller of the same slice might be overkill.

6 Implications

Our results suggest that test-based task interfaces, whenever possible, could be used by developers as an additional factor to consider when dynamically scheduling programming tasks to prevent merge conflicts. The prerequisite is the adoption of BDD, where the development team designs acceptance tests before application code. In addition, extensive test coverage is crucial. Even when dealing with well-designed tests, predictions will be fragile whenever the tests do not cover the task appropriately, given tests are the start point for computing *TestI*. If *TestI* approximates the changed files, the intersection between *TestI* better approximates the concurrently changed files, which are more vulnerable to merge conflicts.

A BDD context is often associated with self-managed agile teams where developers choose programming tasks to perform. But the usage of *TestI* applies even if a project manager allocates tasks to developers. The idea is that a developer might only execute a task with associated acceptance tests, and task prioritization should consider conflict risk, whereas other factors are under control. Project restrictions concerned with time and resources, stakeholders' priority, task complexity, and developer skills are examples of priority factors.

This way, given the sprint planning, a developer or project manager should quickly identify the candidate tasks to perform organized in a ranking related to conflict risk with the other ongoing tasks. Given that the intersection size between two *TestI* interfaces is a degree of conflict risk between a task pair, we expect that tasks with fewer overlapping interfaces would be less likely to lead to conflicting code changes when executed concurrently with the ongoing tasks. Therefore, we can estimate the conflict risk between a candidate task and ongoing tasks as the sum of the size of its intersection values related to each ongoing task. In addition, tasks ranking would sort tasks by ascending order of risk rate.

Concerning the specificities of a predictor of conflict risk based on *TestI*, our results show that we favor precision and prejudice recall when inflating the intersection size between *TestI*. And we reach the opposite effect when we reduce the intersection size. Considering that we cannot propose a unique intersection value for all projects, we conclude that a promising solution is to provide a configurable predictor of conflict risk based on test-based task interfaces. Thus, each team might define a minimal intersection limit between *TestI* (see Table 3), enabling an optimistic (emphasis on precision, larger intersection size) or pessimist strategy (emphasis on recall, fewer intersection size). For us, recall seems more relevant than precision in the context of conflicts, as a lower recall rate implies a higher number of unexpected conflicts. But a lower precision rate indicates that predictions might disturb task scheduling, and someone might perceive it as more critical than unforeseen conflicts.

Similarly, our results show that discarding test preconditions when computing *TestI* interfaces might benefit the prediction of conflict risk for some projects. So, ignoring test

preconditions should be a second configurable property of our predictor of conflict risk. We recommend that a team opts to enable it if the team has a culture of design acceptance tests that faithfully reproduce user interactions.

7 Threats to Validity

In this section, we explain potential threats to the validity of our empirical study.

7.1 Construct Validity

We try to select relevant projects to construct our task pair sample but accurately evaluating relevance is not trivial, leading us to adopt broadly accepted proxies by other studies. For example, Dias et al. (2020) used project activity and star count as proxies for meaningful projects, and Borges et al. (2016) used the star count as a proxy of project popularity. However, relevant projects might have been concluded a while, or a developer can star a repository for creating a bookmark for later analysis, meaning our proxies might wrongly exclude or include projects.

We assume that merged contributions correspond to programming tasks defined by a BDD team, which might not always apply. We did not verify if developers wrote tests before implementing the associated functionality. We checked if tasks both contribute with application code and Cucumber tests. This way, by dealing with completed tasks in a retrospective analysis, we derived predictions based on mature tests (tests on its final version). Differences in test maturity might impact risk predictions. An alternative would be to delimit a task as a set of commits whose messages refer to the same task id or consider each pull request as a task. However, in conjunction with other restrictive criteria, such as the usage of Cucumber, task id or pull requests further restricted our task sample. Also, the limitation about the BDD context would remain.

As previously explained, we simulate the integration of possible concurrent tasks, i.e., tasks concluded with no more than 30 days of difference. But we would better restrict our sample of possible concurrent tasks based on the tasks' init date and duration. Alternatively, we would verify if possible concurrent tasks change or add files on the same date, reinforcing a realistic sample of parallel tasks. For simplicity, we adopted a more straightforward solution. Even so, we can expect that most task pairs satisfy the condition of making parallel changes.

7.2 Internal Validity

Our study evaluates whether *TestI* helps predict the risk of merge conflicts when integrating the code produced by programming tasks. However, TAITIr tool is limited and derives an approximation of *TestI*, wrongly including or excluding files into it. For example, TAITIr might wrongly exclude files in case the tests reference a view file by using code that relies on a variable, impairing the identification of such a file (Rocha et al. 2019). Also, our definition of a programming task as a set of commits is an approximation (see our criteria in Section 4.2). In addition, by ignoring that some commits might revert the changes made by other commits, we might wrongly delimit the set of changed files by tasks, impacting the evaluation of conflict predictions. Even so, given that tangled contributions (Dias et al. 2015) are frequent in practice, such a limitation might make our definition of a programming task more realistic.

Besides, we assume the Cucumber tests that are part of a contribution validate the supposed task's expected behavior. However, in the case of bug fix or refactorings, for instance, it is possible that tests previously developed also validate the contribution, which means that we might provide an incomplete input for TAITIr, ignoring relevant tests when computing *TestI*.

Also, when computing *TestI*, we considered the final version of the tests. In the case of BDD, the tests would be in an initial version that developers might refine until the conclusion of a programming task. The difference in test maturity might impact risk predictions.

When dealing with conflict risk, we check whether the tasks changed any file in common by ignoring file renaming. Consequently, we might miss some intersections between the tasks' changed files set, which impacts the quality of our predictions. Given that we do not integrate tasks by Git, we cannot reuse its mechanism for detecting file renames. Also, we do not adequately deal with file remotion. In practice, a merge conflict happens when a task removes a file changed by another task. Thus, in such a situation, we registry there is a conflict risk. However, *TestI* does not predict file remotion, which might inflate the false negatives, reducing the quality of our predictions. In turn, hoping to conduct a fair evaluation, we discarded tasks with empty *TestI*, given an empty *TestI* does not intersect with others, which might artificially improve predictions.

Finally, we might have missed integration scenarios during the construction of our task sample, as the projects might use Git mechanisms such as rebase, squash, stash apply, and cherry-pick, which rewrites project history. As a consequence, we might further restrict our sample. The impact of an increased sample on our results is unpredictable. Even so, we expect that we have lost a small number of tasks, given that the good practice is to rebase locally only, and we extracted tasks from the master branch.

7.3 External Validity

Our sample contains only GitHub Rails projects that use Cucumber because TAITIr requires it. Such a limitation prevents us from generalizing the results. Even so, we imagine that it is possible to get more accurate test-based interfaces when dealing with statically typed languages and more straightforward frameworks. As a consequence, predictions related to the risk of merge conflicts might be more accurate as well. This way, we can see our results as a pessimistic approximation of the true potential of *TestI*.

8 Related Work

A previous study (Rocha et al. 2019) proposes a strategy and a proof of concept tool (TAITI) for *predicting the files that a programming task will change*. The prediction is based on the acceptance tests that validate the behavior of the functionality underlying a task, and applies in the context of BDD projects. In this paper, we use an extended version of the same tool (TAITIr), but to investigate whether the test-based interfaces it generates has the potential of *predicting merge conflict risk among programming tasks*. This way, we conduct an empirical study by collecting a sample of 990 tasks extracted from merge scenarios from 19 Rails projects that use Cucumber for specifying acceptance tests. Then we simulate the integration of possible concurrent tasks per project, computing the intersection between the set of files changed by both tasks in a pair. As a result, we have a set of 6,360 task pairs, for which we evaluate precision and recall measures of conflict predictions based on *TestI*.

Kasi and Sarma (2013) developed the Cassandra tool for recommending an optimum order of task execution per developer aiming to prevent conflicts (not only merge conflicts but also build and test failures). The recommendation considers the files each task is supposed to edit, which the developers should inform; the files that depend on the files that will change, which are identified by call-graph analysis; and developers' preference about the order of task execution. This way, Cassandra assumes tasks that change the same files are more vulnerable to cause merge conflicts, whereas tasks changing dependent files are more likely to cause a build or test failure. Contrasting, we envision a task prioritizing strategy for preventing merge conflicts that support the developers when selecting a new task to perform. Developers dynamically decide the order of task execution based on information about merge conflict likelihood. Also, rather than asking developers to guess the set of files to be changed by a task, which is challenging and error-prone, we propose using the TAITI tool for predicting it, deriving test-based task interfaces. Despite the strategy differences, one could integrate TAITI to Cassandra for predicting the files that tasks will change, provided we have language compatible versions of these tools (currently, Cassandra analyzes Java systems, whereas TAITI analyzes Rails systems).

Other studies (Zimmermann et al. 2004; Ying et al. 2004; Denninger 2012; Giger et al. 2012; Bailey et al. 2012) try to predict code changes for anticipating software faults and bugs. They rely on code dependencies, assuming that they can propagate code changes, similar to Cassandra tool. However, the developers still need to provide an initial file set for the predictive analysis. In our context, we also can use dependencies to predict the files a task will change and evaluate the risk of merge conflict between tasks. In this sense, we understand that we can complement test-based task interfaces by verifying the structural and logical dependencies of its files.

Some studies investigate merge conflicts to understand their cause and support development teams to avoid them by recommending development practices. For instance, Leßenich et al. (2018) analyze the predictive power of several indicators over the number of merge conflicts, such as the number, size, or scattering degree of commits in each branch. Surprisingly, they did not find evidence that the indicators apply for the whole sample, but only on a per-project basis. Dias et al. (2020) reproduce and analyze several merge scenarios of Rails and Django projects to understand how technical and organizational factors affect the occurrence of conflicts. They found evidence that the likelihood of merge conflict increases significantly when merged contributions involve files from the same MVC slice. Also, more extensive contributions involving more developers, commits, and changed files are more likely associated with merge conflicts and contributions developed over extended periods.

Ahmed et al. (2017) investigate the effect of code smells on merge conflicts and found that entities that are smelly are three times more likely to be involved in merge conflicts. From a complementary perspective, our study aims to predict and prevent merge conflicts by predicting the files a programming task will change based on the code of the automated acceptance tests that validate its behavior. This way, teams might combine our strategy with development practices for detecting code smells. In the first case, developers integrate their contributions frequently to verify them by automatically running build and test scripts. In turn, continuous delivery extends continuous integration to enable developers to release software to production at any time. For such purpose, developers frequently deploy the application into production-like environments to ensure the software will work in production. In both cases, the main objective is to detect conflicts and defects as quickly as possible. Although the early detection of conflicts might avoid increasing conflict complexity, developers still will have to spend time solving conflicts.

Besides, the practice of code review (Bacchelli and Bird 2013), which recommends reviewers to search for issues before integrating code into the central repository, might also help to prevent conflicts. However, its emphasis is on code quality rather than conflicts, and it is an expensive activity, even with tool support. Some agile practices, such as daily stand-up meetings, might prevent conflicts by promoting communication and clarifying ongoing tasks (Stray et al. 2016). Even so, communication might be more imprecise than software artifacts. That's why we prioritize an automatic solution for predicting conflict risk, aiming to promote developers' productivity and effectiveness.

Similar to development practices, workspace awareness tools (Biehl et al. 2007; Dewan and Hegde 2007; Sarma et al. 2012; Brun et al. 2013; Guimarães and Silva 2012) also support early conflict detection. They monitor the developers' workspace and emit notifications when they detect potential conflicts by analyzing the code changes in progress. Early detecting conflicts is a strategy to reduce the effort to solve them, as previously argued. Even so, whether conflict occurs, developers spend time to understand and solve it. Likewise, specific-language merge tools (Apel. et al. 2011, 2012) solve and avoid some spurious conflicts reported by the state of the practice tools. Thus, these merge tools focus on conflict resolution, whereas our focus is conflict avoidance. As developers might change files not aligned with their programming tasks and predictions of conflict risk might fail, there is no solution to extinguish merge conflicts. Thus, we understand our solution and specific-language merge tools complement each other, and a development team might benefit from adopting both.

Mylyn (Kersten and Murphy 2006) is an Eclipse plugin that monitors developers' workspace to track relevant resources (e.g., selected or edited files) and updates the IDE accordingly. Its main objective is to improve developer productivity, focusing their attention on what matters to complete a task. Mylyn calls the set of relevant resources for a task as task context. Using a prioritizing policy for resources based on user interaction, Mylyn delineates a task context during the development of the application code. Instead, we predict the task context before developing application code related to the task. Therefore, someone can use Mylyn to identify ongoing tasks that might cause conflicts and adopt a coordination strategy to alleviate or even prevent conflicts, as explored by the tool ProxiScientia (Borici et al. 2012). Nevertheless, Mylyn is not able to predict task interfaces nor predict that planned tasks might cause conflicts.

9 Conclusions

Given merge conflicts occur frequently and demand extra effort to be solved, the ability to predict conflict risk might promote development productivity. Aware of such a risk, a developer might wisely choose a task to work on, reducing the chances of conflict occurrence. Even when choosing a risky task due to other project restrictions or priorities, the upfront knowledge about conflict risk might be helpful. Teams might coordinate efforts like improving test coverage planning to check the risky code better and detect stronger communication needs among members.

Our retrospective study results show that tasks with non-disjoint *TestI* interfaces are more likely to change a common file, which means they are more likely to cause a merge conflict. Also, the larger the intersection between *TestI*, the higher the risk of a merge conflict between tasks. Thus, when choosing the next task to perform, a developer should prioritize

the one that has the smaller intersection with the *TestI* of other tasks. Although the intersection between *TestI* might predict potentially conflicting task pairs, *TestI* does not always indicate the potential conflicting files. In such cases, *TestI* might suggest a degree of proximity between the parts of the code changed by both tasks, eventually leading to semantic conflicts. Complementarily, we find our predictions are better when we compare them with the risk result by considering all kinds of files changed by the tasks, though *TestI* only contains Ruby and HTML files. In addition, we verify that we can improve predictions for some projects by discarding test preconditions when computing *TestI* interfaces. Finally, compared to a *TextI*-based predictor, our predictor is less precise but covers most conflicts (i.e., it has a high recall rate). We conclude that *TestI* has overall better performance, given that recall is more important than precision in the context of conflicts.

Note that we focus on understanding the cases where *TestI* and *TextI* interfaces seem to work. A complementary analysis would investigate cases where both interfaces do not work, or some work, aiming to propose alternative solutions. We plan to do it as future work.

Concerning the prediction of merge conflict occurrence, we conclude that our predictor based on the intersection between *TestI* interfaces has potential. It detects conflict risk when tasks are likely to change files in common, which is a precondition of conflict occurrence. The conflict only occurs when the tasks change the same hunk of a file, indeed. Thus, we expect conflicts to occur in a subset of risk predictions, whether the tests genuinely relate to the tasks.

These findings motivate the conduction of a case study to evaluate further the potential of *TestI* for reducing conflicts in the field. For instance, a requirement for computing *TestI* is writing automated acceptance tests before implementing features, a BDD practice, but BDD is not widespread in the context of software development. If projects do not adopt BDD, they might feel uninterested in our approach to avoid conflicts. On the other hand, we believe that our strategy promotes BDD practices. In this sense, someone might feel motivated to adopt BDD once he understands that acceptance tests might help to reduce merge conflicts besides other software quality benefits. Therefore, we need to conduct a case study to estimate the influence of our strategy on the dynamics of development teams and other human and social effects that might compromise its performance and acceptance.

Also, in our retrospective study, we considered the final version of the tests. In the case of BDD, the tests might evolve in conjunction with the task development, impacting risk predictions. A case study might enable us to understand better and evaluate this phenomenon.

Our future study requires the conclusion of the development of TAITIr for supporting developers to choose a task to perform based on the conflict risk with other tasks (instead of task pairs), integrating it with the development environment and a task management tool.

Acknowledgements For partially supporting this work, we would like to thank INES (National Software Engineering Institute) and the Brazilian research funding agencies CNPq (grant 309741/2013-0), FACEPE (grants IBPG-0546-1.03/15 and APQ/0388-1.03/14), and CAPES.

Author Contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by Thaís Rocha. The first draft of the manuscript was written by Thaís Rocha and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding Partial financial support was received from INES (National Software Engineering Institute) and the Brazilian research funding agencies CNPq (grant 309741/2013-0), FACEPE (grants IBPG-0546-1.03/15 and APQ/0388-1.03/14), and CAPES.

Data Availability The datasets generated during and/or analyzed during the current study are available in this website: https://thaisabr.github.io/conflict-risk-prediction-study-site/.

Declarations

Conflict of Interests The authors have no competing interests to declare that are relevant to the content of this article. The authors have no relevant financial or non-financial interests to disclose. All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript. The authors have no financial or proprietary interests in any material discussed in this article.

References

Accioly P, Borba P, Cavalcanti G (2017) Understanding semi-structured merge conflict characteristics in open-source java projects. Empirical Software Engineering https://doi.org/10.1007/s10664-017-9586-1

Adams B, McIntosh S (2016) Modern release engineering in a nutshell – why researchers should care. In: 2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER), vol 5. pp 78-90

Ahmed I, Brindescu C, Mannan UA, Jensen C, Sarma A (2017) An empirical examination of the relationship between code smells and merge conflicts. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 58–67

Apel S, Liebig J, Brandl B, Lengauer C, Kästner C (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. ACM, New York, pp 190–200. ESEC/FSE '11, https://doi.org/10.1145/2025113.2025141

Apel S, Leβenich O, Lengauer C (2012) Structured merge with auto-tuning: Balancing precision and performance. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, pp 120–129. ASE 2012, https://doi.org/10.1145/2351676.2351694

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, pp 712–721

Bailey M, Lin KI, Sherrell L (2012) Clustering source code files to predict change propagation during software maintenance. In: Proceedings of the 50th Annual Southeast Regional Conference. ACM, New York, pp 106–111. ACM-SE '12, https://doi.org/10.1145/2184512.2184538

Bass L, Weber I (2016) Zhu l, A Software Architect's Perspective. Addison-Wesley Professional, DevOps

Berry DM (2017) Evaluation of tools for hairy requirements engineering and software engineering tasks. Tech. rep., University of Waterloo, https://cs.uwaterloo.ca/~dberry/FTP_SITE/tech.reports/EvalPaper.pdf. Accessed: Jan 2021

Biehl JT, Czerwinski M, Smith G, Robertson GG (2007) Fastdash: A visual dashboard for fostering awareness in software teams. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, New York, pp 1313–1322. CHI '07, https://doi.org/10.1145/1240624.1240823

Borges H, Hora A, Valente MT (2016) Understanding the factors that impact the popularity of github repositories. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 334–344, https://doi.org/10.1109/ICSME.2016.31

Borici A, Blincoe K, Schröter A, Valetto G, Damian D (2012) Proxiscientia: Toward real-time visualization of task and developer dependencies in collaborating software development teams. In: 2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE), pp 5–11, https://doi.org/10.1109/CHASE.2012.6223024

Brun Y, Holmes R, Ernst MD, Notkin D (2013) Early detection of collaboration conflicts and risks. IEEE Trans Softw Eng 39(10):1358–1375. https://doi.org/10.1109/TSE.2013.28

Cavalcanti G, Borba P, Accioly P (2017) Evaluating and improving semistructured merge. Proc ACM Program Lang 1(OOPSLA):59:1–59:27. https://doi.org/10.1145/3133883

Cubranic D, Murphy GC, Singer J, Booth KS (2005) Hipikat: A project memory for software development. IEEE Trans Softw Eng 31(6):446–465. https://doi.org/10.1109/TSE.2005.71

Denninger O (2012) Recommending relevant code artifacts for change requests using multiple predictors. In: Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering. IEEE Press, Piscataway, pp 78–79. RSSE '12, http://dl.acm.org/citation.cfm?id=2666719.2666737

Dewan P, Hegde R (2007) Semi-synchronous conflict detection and resolution in asynchronous software development. In: ECSCW 2007, Springer, pp 159–178
Dias K, Borba P, Barreto M (2020) Understanding predictive factors for merge conflicts. Inf Softw Technol 121:106256
Dias M, Bacchelli A, Gousios G, Cassou D, Ducasse S (2015) Untangling fine-grained code changes. In: 2015 IEEE 22Nd international conference on software analysis, evolution, and reengineering, SANER, IEEE, pp 341–350
Fowler M (2010) Feature toggle. https://martinfowler.com/bliki/FeatureToggle.html. Accessed: Jan 2021
Fowler M (2020) Feature Branch. https://martinfowler.com/bliki/FeatureBranch.html. Accessed: Jan 2021
Giger E, Pinzger M, Gall HC (2012) Can we predict types of code changes? an empirical analysis. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on, pp 217–226
Grinter RE (1997) Supporting articulation work using software configuration management systems. Comput Supported Coop Work 5(4):447–465
Guimarães ML, Silva AR (2012) Improving early detection of software merge conflicts. In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, Piscataway, pp 342–352. ICSE '12, http://dl.acm.org/citation.cfm?id=2337223.2337264
Henderson F (2017) Software engineering at Google. https://arxiv.org/abs/1702.01715, Accessed: Jan 2021
Hodgson P (2017a) Feature branching vs. feature flags: What's the right tool for the job? Tech. rep., DevOps Blog. https://devops.com/feature-branching-vs-feature-flags-whats-right-tool-job/, Accessed: Jan 2021
Hodgson P (2017b) Feature toggles (aka Feature Flags). https://martinfowler.com/articles/feature-toggles.html. Accessed: Jan 2021
Kasi BK, Sarma A (2013) Cassandra: Proactive conflict minimization through optimized task scheduling. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, ICSE '13, pp 732–741
Kersten M, Murphy GC (2006) Using task context to improve programmer productivity. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, pp 1–11, https://doi.org/10.1145/1181775.1181777
Leßenich O, Siegmund J, Apel S, Kästner C, Hunsen C (2018) Indicators for merge conflicts in the wild: survey and empirical study. Autom Softw Eng 25(2):279–313
Nagappan M, Zimmermann T, Bird C (2013) Diversity in software engineering research. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, ESEC/FSE 2013, pp 466–476. http://doi.acm.org/10.1145/2491411.2491415
Potvin R, Levenberg J (2016) Why google stores billions of lines of code in a single repository. Commun ACM 59:78–87. http://dl.acm.org/citation.cfm?id=2854146
Rocha T, Borba P (2019) Online Appendix. https://thaisabr.github.io/conflict-risk-prediction-study-site/, Accessed: Jan 2021
Rocha T, Borba P, Santos JP (2019) Using acceptance tests to predict files changed by programming tasks. J Syst Softw 154:176–195
Salton G, McGill MJ (1986) Introduction to Modern Information Retrieval. McGraw-Hill, Inc., New York
Sarma A, Redmiles D, van der Hoek A (2012) Palantír: Early detection of development conflicts arising from parallel code changes. IEEE Trans Softw Eng 38(4):889–908
Smart J (2014) BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle. Manning Publications Company, https://booktitles.google.com.br/booktitles?id=2BGxngEACAAJ
de Souza CRB, Redmiles D, Dourish P (2003) "breaking the code", moving between private and public work in collaborative software development. In: Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work, ACM, GROUP '03, pp 105–114
Stray V, Sjøberg DI, Dybå T (2016) The daily stand-up meeting: A grounded theory study. J Syst Softw 114:101–124. https://doi.org/10.1016/j.jss.2016.01.004. https://www.sciencedirect.com/science/article/pii/S0164121216000066
Thompson C, Murphy G (2014) Recommending a starting point for a programming task: An initial investigation. 4th International Workshop on Recommendation Systems for Software Engineering, RSSE 2014 - Proceedings, https://doi.org/10.1145/2593822.2593824
Ying ATT, Murphy GC, Ng R, Chu-Carroll MC (2004) Predicting source code changes by mining change history. IEEE Trans Softw Eng 30(9):574–586
Zampetti F, Di Sorbo A, Visaggio CA, Canfora G, Di Penta M (2020) Demystifying the adoption of behavior-driven development in open source projects. Inf Softw Technol 123:106311–0. https://doi.org/10.1016/j.infsof.2020.106311. https://www.sciencedirect.com/science/article/pii/S095058492030063X

Zimmermann T, Weisgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, Washington, pp 563–572. ICSE '04, http://dl.acm.org/citation.cfm?id=998675.999460