Pós-Graduação em Ciência da Computação

Thaís Alves Burity Rocha

Avoiding Merge Conflicts by Test-Based Task Prioritization

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2020

Thaís Alves Burity Rocha

Avoiding Merge Conflicts by Test-Based Task Prioritization

Tese de Doutorado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Doutor em Ciência da Computação.

Área de Concentração: Engenharia de Software
Orientador: Paulo Henrique Monteiro Borba

Recife
2020

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

R672a Rocha, Thaís Alves Burity
Avoiding merge conflicts by test-based task prioritization / Thaís Alves Burity Rocha. – 2020.
103 f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências.

1. Engenharia de software. 2. Desenvolvimento colaborativo. I. Borba, Paulo Henrique Monteiro (orientador). II. Título.

005.1 CDD (23. ed.) UFPE - CCEN 2020 - 70

Thaís Alves Burity Rocha

"**Avoiding Merge Conflicts by Test-Based Task Prioritization**"

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Aprovado em: 17/02/2020.

Orientador: Paulo Henrique Monteiro Borba

BANCA EXAMINADORA

Prof. Dr. Hermano Perrelli de Moura
Centro de Informática / UFPE

Prof. Dr. Fernando José Castor de Lima Filho
Centro de Informática / UFPE

Prof. Dr. Breno Alexandro Ferreira de Miranda
Centro de Informática / UFPE

Prof. Dr. Alessandro Fabricio Garcia
Departamento de Informática / PUC/RJ

Prof. Dr. Alfredo Goldman Vel Lejbman
Instituto de Matemática e Estatística / USP

I dedicate this work to my family.

ACKNOWLEDGEMENTS

Eu agradeço à Deus pelas inúmeras oportunidades que me foram concedidas até hoje e pelas pessoas fantásticas que Ele colocou/coloca na minha vida. Sem elas, esse trabalho não seria possível. Uma família que sempre me incentivou a ter curiosidade, a ter sede de conhecimento e, principalmente, a não desistir diante das dificuldades. Em especial, meu pai Fernando pelo seu exemplo de disciplina, minha mãe Silvana pelo entusiasmo em viver e pela grande capacidade expressiva, minha irmã Beta pelo bom humor e coragem de recomeçar. Meu esposo Betinho, que me aturou durante toda essa jornada, nos altos e baixos, com tanto amor e paciência, acreditando em mim quando nem eu mesma acreditava. Não foram poucas as vezes que abrimos mão de momentos de lazer, descanso, diálogo, para que eu me dedicasse a esse trabalho. Sem contar toda a ajuda prestada, resolvendo coisas do dia-a-dia por mim, para o tempo render. Um orientador zeloso, o professor Paulo Borba, que pacientemente soube me mostrar um mundo de possibilidades quando eu me via sem saída, lidando com todo o meu pessimismo com leveza e sabedoria. O grupo de pesquisa Labes/SPG que tive a felicidade de conviver por um bom período, compartilhando momentos de discussão técnica e construção de conhecimento, mas também momentos de descontração e desabafos pertinentes ao mundo acadêmico. Em especial, minhas irmãs de pesquisa Klissio, Paola e Gabi. A amiga Renata, que fez parte do início da minha vida de pesquisadora ainda na graduação, e prestou consultorias estatísticas com toda boa vontade. As amigas de infância Gisa e Tatá, por dividirem tantos momentos de alegria e desabafos, apesar de toda a distância que nos separa, tornando as preocupações e obrigações mais leves. As amigas da UFRPE/UAG Kádna, Priscilla e Maria Aparecida, por compartilharem suas experiências de professoras e doutorandas, me motivando nos momentos de desânimo. A minha Lulu, minha filha Luísa Helena, que com seu sorriso semi-banguela, seus passinhos apressados, e sua fala insistente por "mamam", me ensina tanto e me motiva a superar meus limites. As pessoas que me ajudam a cuidar da rotina com Lulu e assim ter condições de me dedicar à pesquisa, em especial Kelly, Elisângela e vovó Ieda.

ABSTRACT

In a collaborative development context, merge conflicts might compromise software quality and developers' productivity. To reduce conflicts, one could avoid the parallel execution of programming tasks that are likely to change the same files. Although hopeful, this strategy is challenging because it relies on the prediction of the required file changes to complete a task. As predicting file changes is hard, we investigate its feasibility for BDD (Behaviour-Driven Development) projects, which write automated acceptance tests before implementing features. We develop a tool that, for a given task, statically analyzes Cucumber tests and conservatively infers test-based interfaces or *TestI* (files that could be executed by the tests), approximating files that would be changed by the task. To assess the accuracy of this approximation, we measure precision and recall of test-based interfaces of 513 tasks from 18 Rails projects on GitHub. We also compare such interfaces with randomly defined interfaces, interfaces obtained by the textual similarity of test specifications with past tasks, and interfaces computed by executing tests. Our results give evidence that, in the specific context of BDD, Cucumber tests might help to predict files changed by tasks. We find that the better the test coverage, the better the predictive power. Next, we evaluate whether it is possible to predict the risk of a merge conflict when integrating the code produced by two programming tasks based on the intersection among their *TestI*. To assess the predictions of conflict risk, we measure precision and recall of 6,360 task pairs from 19 Rails projects on GitHub. Our results confirm that Cucumber tests might help to predict the risk of merge conflicts, given the intersection among interfaces denotes a higher probability that the tasks change some file in common. A minimal intersection predicts conflict risk with 0.59 of precision and 0.98 of recall. Also, the higher the intersection size, the higher the number of files changed by both tasks. This way, developers might use the intersection size between *TestI* as a degree of conflict risk between tasks, prioritizing the selection of a task to work on whose *TestI* has the lowest intersection with others. Finally, a predictor of conflict risk based on *TestI* outperforms a predictor based on similar past tasks.

Keywords: Collaborative development. Task prioritization. Behaviour-driven development. File change prediction. Prediction of conflict risk.

RESUMO

No contexto de desenvolvimento colaborativo, conflitos de integração podem comprometer a qualidade do software e a produtividade dos desenvolvedores. Para reduzir conflitos, uma possibilidade seria evitar a execução paralela de tarefas de programação que irão alterar os mesmos arquivos. Embora esperançosa, essa estratégia é desafiadora porque depende da predição dos arquivos que precisam ser alterados para concluir uma tarefa. Como é difícil prever os arquivos alterados, investigamos sua viabilidade para projetos BDD (Behavior-Driven Development), que escrevem testes de aceitação automatizados antes de implementar funcionalidades. Desenvolvemos uma ferramenta que, para uma determinada tarefa, analisa estaticamente os testes do Cucumber e, de maneira conservativa, infere as interfaces baseadas em testes ou *TestI* (arquivos que podem ser executados pelos testes), aproximando os arquivos que seriam alterados pela tarefa. Para avaliar a confiabilidade dessa aproximação, medimos a precisão e a revocação de interfaces baseadas em teste de 513 tarefas de 18 projetos Rails no GitHub. Também comparamos interfaces baseadas em testes com interfaces definidas aleatoriamente, interfaces obtidas pela similaridade textual das especificações de teste com tarefas anteriores e interfaces calculadas pela execução de testes. Nossos resultados evidenciam que, no contexto específico de BDD, os testes do Cucumber podem ajudar a prever arquivos alterados pelas tarefas. Em seguida, avaliamos se é possível prever o risco de conflitos de integração com base na interseção entre interfaces de tarefas baseadas em teste. Para avaliar as previsões de risco de conflito, medimos a precisão e a revocação de 6.360 pares de tarefa de 19 projetos Rails no GitHub. Dentre outras descobertas, nossos resultados confirmam que os testes do Cucumber podem ajudar a prever o risco de conflitos de integração, uma vez que a interseção entre interfaces de tarefas indica uma maior probabilidade das tarefas alterem algum arquivo em comum. Uma interseção mínima prediz risco de conflito com 0,59 de precisão e 0,98 de revocação. Além disso, quanto maior o tamanho da interseção, maior o número de arquivos alterados por ambas as tarefas. Dessa forma, os desenvolvedores podem usar o tamanho da interseção entre *TestI* como grau de risco de conflito entre tarefas, priorizando a seleção de uma tarefa para executar cujo *TestI* possui a menor interseção com as demais. Finalmente, um preditor de risco de conflito baseado em interfaces de tarefa tem melhor performance que um preditor baseado em tarefas passadas similares.

Palavras-chaves: Desenvolvimento colaborativo. Priorização de tarefas. Desenvolvimento Dirigido a Comportamento. Predição de mudança de arquivo. Predição de risco de conflito.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

When collaborating, developers create and change software artifacts often without full awareness of changes being made by other team members. While such independence is essential for medium and large teams and promotes development productivity, it might also result in conflicts when integrating developers changes. In fact, high degrees of parallel changes and integration conflicts have been observed in a number of industrial and open-source projects that use different kinds of version control systems (PERRY; SIY; VOTTA, 2001; ZIMMERMANN, 2007; BRUN et al., 2013; KASI; SARMA, 2013). This has been observed even when using advanced merge tools (APEL et al., 2011; APEL; LESSENICH; LENGAUER, 2012; CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY; BORBA; CAVALCANTI, 2017) that avoid common spurious conflicts identified by state of the practice tools.

Resolving such integration conflicts might be time consuming and is an error-prone activity (SARMA; REDMILES; HOEK, 2012; BIRD; ZIMMERMANN, 2012; MCKEE et al., 2017), negatively impacting development productivity. Quality might be impacted too, since developers maybe miss semantic conflicts, leading to defects that escape to end users. So, to avoid dealing with conflicts, developers have been adopting risky practices such as rushing to finish changes first (SARMA; REDMILES; HOEK, 2012; GRINTER, 1996), and partial check-ins (SOUZA; REDMILES; DOURISH, 2003). Similarly, partially motivated by the need to reduce conflicts, or at least avoid large conflicts, development teams have been adopting techniques such as trunk-based development (ADAMS; MCINTOSH, 2016; HENDERSON, 2017; POTVIN; LEVENBERG, 2016) and feature toggles (BASS; WEBER; ZHU, 2015; ADAMS; MCINTOSH, 2016; FOWLER, 2016; HODGSON, 2017a), which are important to support actual Continuous Integration (FOWLER, 2009), but might lead to extra code complexity (HODGSON, 2017b).

By wisely choosing which tasks to work on in parallel, a development team could likely reduce conflict occurrence. In particular, we should expect lower integration conflict risk from parallel tasks that focus on unrelated features and affect disjoint and independent file sets. However, developers might not be able to accurately predict which files will be changed by a given task. Automatically predicting such code changes, in general, is also hard. It is worth, though, to investigate whether this kind of automatic prediction is feasible for specific contexts (BRIAND et al., 2017). In this thesis, by conducting two empirical studies, we investigate the usage of acceptance tests in BDD (SMART, 2014) projects to predict the files the developers will change when working in parallel and, consequently, the risk of integration conflicts between their programming tasks.

In particular, in a BDD context, automated acceptance tests are written before implementing features. Moreover, each feature implementation task can be often associated to a number of usage scenarios that are directly linked to the tests. So, by statically

analyzing the code that automates the tests associated to a task, following references we could conservatively infer parts of the application code that might be exercised by the tests, and these could perhaps approximate the files that would be changed by the task.

To assess the accuracy of this approximation, we first build the TAITI tool to compute, for a given task, its *test-based task interface* (*TestI*): the set of application files that might be exercised by the tests associated to the given task. We then compare a *TestI* with the corresponding *task interface*: the set of files actually changed by the task.[1] We compute precision and recall measures for *TestI* considering 513 tasks[2] from 18 Ruby on Rails[3] projects that use Cucumber[4] for specifying acceptance tests. We compare the predictive capacity of different variations of *TestI*. We also compare *TestI* with randomly defined task interfaces (*RandomI*), task interfaces obtained by observing textual similarity of test specifications with past tasks (*TextI*), and task interfaces computed by executing tests (*DTestI*)[5].

Our results bring evidence that, in the specific context of BDD, Cucumber tests associated to a task might help to predict application files changed by developers responsible for the task. We find that the better the test coverage of a task, the better the *TestI* predictive power. On average, *TestI* presents similar recall but better precision than *RandomI*. Contrasting, *TestI* presents better recall but inferior precision than *TextI*. As the adverse impact of false positives in this context is basically to discourage parallel execution of tasks that would not conflict, or encourage slightly more unneeded coordination, the false negatives are more important because they could lead to conflict occurrence and extra effort for resolving it. This favours the recall measure and, as a consequence, supports *TestI* instead of *TextI*. As maybe expected *DTestI* presents better recall than *TestI*, but the first can only be computed by executing the tests, which assumes that tasks have been finished and application code is ready. As such, *DTestI* results are only useful to help us understand the limits of *TestI* and how our algorithms and tool could be improved. Also, when dealing with an MVC-like application (e.g., web applications developed in Rails), *TestI* performs better when predicting changes in controller files. As controllers uniquely identify an MVC segment (related model, view, controller, and auxiliary files), one could also use *TestI* to predict changes in segments.

In particular, our results suggest that a hybrid approach for computing test-based interfaces is promising. For example, failing tests would trigger our static analysis for computing the interface, but ready tests for features that are being maintained could be

[1] Although this is a coarse and limited notion of task interface (BALDWIN, 2000) that considers only the "provides" dimension, for simplicity we take the liberty of adopting this terminology.

[2] In this thesis, a task is a commit set between a merge commit and the common ancestor with the other contribution the merge integrates.

[3] <https://rubyonrails.org/>

[4] <https://cucumber.io/>

[5] We design both *TextI* and *DTestI*. The inspiration for designing *TextI* came from studies related to code change prediction that investigate the past to predict the future.

executed to complement the interface and improve recall. Concerning conflict avoidance, our results suggest that files inferred by analyzing certain parts of the test, such as the test setup, should have less importance than files inferred from other parts.

Even so, our results about the ability of *TestI* for predicting file changes do not bring evidence that *TestI* might actually support developers to avoid conflicts. For this reason, we go further and assess whether it is possible to predict the risk of merge conflicts between two tasks based on the intersection between their *TestI*. If the intersection is empty, we assume there is no risk of a merge conflict. Otherwise, we consider there is a risk. This way, we conduct a second empirical study by collecting a sample of 990 tasks from 19 Rails projects that use Cucumber for specifying acceptance tests. Then we simulate the integration of possible concurrent tasks per project by verifying the intersection between the set of files changed by the tasks. We adopt such an artifice rather than merging tasks to see if conflicts occur to increase our sample, given we can reproduce few merges (i.e., task pairs extracted from the same base and merge commit). 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 new results reveal that, in our sample, a minimal intersection between *TestI* (1 file) predicts conflict risk with 0.59 of precision and 0.98 of recall. That is, the intersection between test-based task interfaces is associated with higher chances of tasks changing files in common. The predictions favor the recall measure, which is more critical for our context, as previously explained. We also find evidence that the larger the intersection size between two interfaces, the higher the number of files actually 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, prioritizing parallel execution of tasks that are likely to change fewer files in common, when possible. Task prioritization depends on other factors, like project restrictions concerned with time and resources, stakeholders priority, task complexity, and developer skills. So, even when it is not possible to avoid the concurrent execution of some tasks, based on our results, the upfront knowledge about conflict likelihood might support task coordination.

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, we conclude that the intersection between *TestI* 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. Finally, we compare the performance of *TestI* and *TextI* when predicting conflict risk. Similarly to when dealing with file changes prediction, we find *TextI* is more precise than *TestI*, but its low recall rate discourages its usage. So we conclude that, for our context, *TestI* has overall better performance. In the end, the results of both empirical studies corroborate that test-based task interfaces might help developers to avoid merge conflicts.

We organize the remainder of this document as follows:

- In Chapter 2, we review the essential concepts used throughout this work.
- In Chapter 3, we describe our first empirical study, investigating the accuracy of test-based task interfaces for predicting the files changed by programming tasks. This chapter is published in Rocha, Borba e Santos (2019), with the co-authoring of Paulo Borba and João Pedro Santos. Paulo Borba carefully guided and reviewed this work, and João Pedro Santos computed *DTestI* and implemented a module of TAITI that is responsible for parsing view files.
- In Chapter 4, we describe our second empirical study, assessing the ability of test-based task interfaces for predicting the risk of merge conflicts between programming tasks. This chapter will be published soon.
- In Chapter 5, we present our concluding remarks, and future and related work.

2 BACKGROUND

In this chapter, we review the essential concepts explored in this work. First, we explain the concept of programming tasks for our context in Section 2.1. Then, we review merge conflicts in collaborative software development in Section 2.2. Subsequently, in Section 2.3, we present BDD and acceptance tests. In particular, we illustrate the Cucumber tool, given TAITI (our tool) requires Cucumber tests as input to infer test-based task interfaces. Finally, we briefly present the architecture of a Ruby on Rails application in Section 2.4, given TAITI is specific for Rails projects.

2.1 TASKS IN AGILE SOFTWARE DEVELOPMENT

In this work, a programming task means a work item assigned to a developer that results in code creation or code edition in the context of agile projects. For example, developing new functionalities, maintaining functionalities (e.g., updating, correcting, or enhancing existing features), bug fixing, and code refactoring.

Agile teams perform tasks according to an iterative and incremental development process. They work in small system increments to get rapid feedback from stakeholders and continuously improve software quality. In turn, teams organize increments into iterations with a short and fixed duration, considering subsequent iterations improve the work of the previous one. So, developers frequently perform several tasks per release or even per iteration.

Regarding task management, agile teams often plan system increments as *user stories* (COHN, 2004) that they prioritize, build, test, and release. Accurately, a user story describes a piece of functionality (feature) from the perspective of a user or customer, in varying levels of detail. For example, Figure 1 illustrates a user story for a travel reservation system, specifying a user role, an action, and a goal. The development of such a story encompasses other details, such as the usage of a credit card to ensure the reservation, detailed view of available hotels, and a mechanism to search hotels based on criteria like city and price, for instance. In this sense, a task might cover only a slice of this user story. On the other hand, we might split this user story based on each detail, derivating multiple tasks, each one covering a full user story.

Figure 1 – User story example.

```
As a user
I want to reserve a hotel room
So that I can be accommodated according to my needs
```

For management purpose, teams often document tasks as tickets (or requests) in a

task management system, such as JIRA[1] or PivoltalTracker,[2] or an issue tracker system like Bugzilla[3] or Mantis.[4] Finally, in such a collaborative development context, developers share code through a repository managed by a version control system (VCS). Therefore, a task results in at least a commit that adds, edits, or removes files. Also, some teams explicitly link the tasks to the code that supports it by including the task ID into the commit message. However, this is not a universal pattern.

2.2 MERGE CONFLICTS

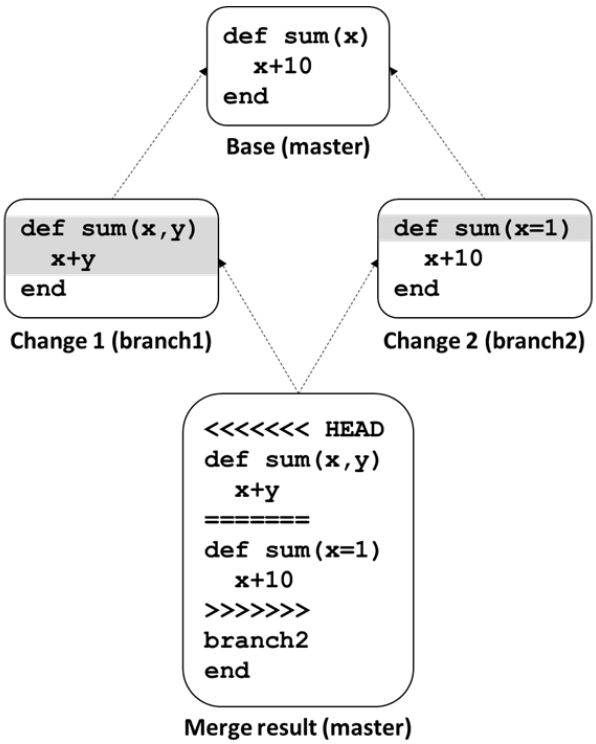
Supported by a VCS, developers implement tasks using their copy of project files. In such a process, they work independently from others and only integrate their contributions as they complete a task. Even so, developers often make code changes that are inconsistent with other changes, leading to conflicts during code integration (BRUN et al., 2013; KASI; SARMA, 2013), despite the usage of advanced merge tools (APEL et al., 2011; APEL; LESSENICH; LENGAUER, 2012; CAVALCANTI; BORBA; ACCIOLY, 2017; ACCIOLY; BORBA; CAVALCANTI, 2017) that avoid false conflicts alerted by the state of practice tools.

Conflicts might occur when developers change the same file (*direct conflict*, the focus of this work) or different files that contain obvious or subtle relationships (*indirect conflict*). During code integration, for example, a *merge conflict* emerges when it is not possible to automatically integrate development contributions affecting the same file. A typical merge conflict happens when a developer tries to update the repository by publishing changes he has made in a file, and someone already edited the same area of the same file (the same text line or textually close lines).

Figure 2 illustrates a merge conflict. The example shows a snippet of the utility file *calculator.rb* written in Ruby and considers the usage of Git[5] as VCS. A developer adds a parameter to the signature of method `sum` (**change 1**), whereas another developer attributes a default value for the original parameter (**change 2**). The first developer successfully merges his version file to the master branch. However, when the second developer tries to merge his version, the illustrated conflict emerges (**Merge result**). Despite its compatibility, Git (and most VCS) requires manual intervention to integrate these code changes, because Unix diff3— the default diff tool— does not consider language syntax. Instead, it checks concurrent changes on the same text line or the same hunk of a file (a notion of proximity between text lines).

[1] <https://www.atlassian.com/software/jira>
[2] <https://www.pivotaltracker.com/>
[3] <https://www.bugzilla.org/>
[4] <https://www.mantisbt.org/>
[5] <https://git-scm.com/>

Figure 2 – Example of merge conflict.



2.2.1 Merge conflicts in practice

Some empirical studies bring quantitative evidence that conflicts occur frequently. For example, Zimmermann (2007) analyzed four large open-source systems in CVS[6] repositories and reported that between 23% and 47% of all merges had conflicts. Brun et al. (2013) analyzed 9 of the most active open-source systems in Git repositories and reported that 16% of all merges had conflicts. Kasi e Sarma (2013) analyzed four open-source projects hosted in Github[7] and reported that each project exhibited a different distribution of merge conflicts, ranging from 4.2% to 19.3%. Furthermore, conflicts occurred irrespective of the size of the project (KLOC) or the number of developers. Cavalcanti, Borba e Accioly (2017) analyzed 34,030 merge scenarios from 50 Java projects and reported that 8.8% of them had conflicts. When using an advanced merge tool (a semistructured merge tool that exploits part of the language syntax and semantics), 7.1% of the merge scenarios had conflicts. Accioly, Borba e Cavalcanti (2017) analyzed 70,047 merge scenarios from 123 GitHub Java projects and reported that 9.38% had conflicts (8.39% when using a language-specific merge tool).

Also, these and other studies present evidence that conflicts are costly to solve, as developers dedicate time and effort to understand, effectively trace the cause, and seek a solution. As an illustration, Bird e Zimmermann (2012) investigated how developers used branches in a large industrial project and reported that integrating changes from

[6] <http://www.nongnu.org/cvs/>
[7] <https://github.com/>

multiple branches can be difficult and error-prone. As a consequence, branches incur overhead in both development effort and time. By inspecting diverse projects, Brun et al. (2013) reported conflicts persist 3.2 days average (0.7 days median). In turn, Kasi e Sarma (2013) reported that developers needed between 6 days average (2 days median) and 22.93 days average (10 days median) to solve merge conflicts. These numbers are a proxy for the effort of conflict resolution. Given version control history only provides information about the time that the conflict emerges and the time after which it no longer exists, the study assumes that if a developer faced a conflict, then she worked exclusively to resolve that conflict in subsequent merges, which is unrealistic. Even so, the results corroborate with evidence reported by other studies that resolving merge conflicts is not trivial and negatively impact software development. For example, the fear of conflicts might affect developers' behavior (BRUN et al., 2012). Some developers avoid working in parallel to ensure not running into conflicts. So, they share their code hurriedly in an attempt to avoid responsibility for conflicts (SARMA; REDMILES; HOEK, 2012; GRINTER, 1996), many times making partial check-ins (SOUZA; REDMILES; DOURISH, 2003) (the developer checks in a file back in the repository but did not finish the intended changes). Other developers postpone the incorporation of teammates' work because they fear tricky conflicts.

Intending to reduce conflicts, or at least avoid large conflicts, some teams adopted trunk-based development (ADAMS; MCINTOSH, 2016; POTVIN; LEVENBERG, 2016; HENDERSON, 2017), according to which developers directly commit to the master branch, eliminating the need of other branches. In such a context, to avoid compilation or testing failure caused by incomplete features, teams adopt feature toggles (BASS; WEBER; ZHU, 2015; ADAMS; MCINTOSH, 2016; FOWLER, 2016; HODGSON, 2017a) as an alternative solution for branches. In sum, feature toggles enable developers to share incomplete code by isolating them into conditional blocks that can be activated or deactivated when necessary. In contrast, such modern practices might introduce extra complexity to code development (HODGSON, 2017b). Finally, software quality might be impacted by merge conflicts too, since developers maybe miss semantic conflicts, leading to defects that escape to end users.

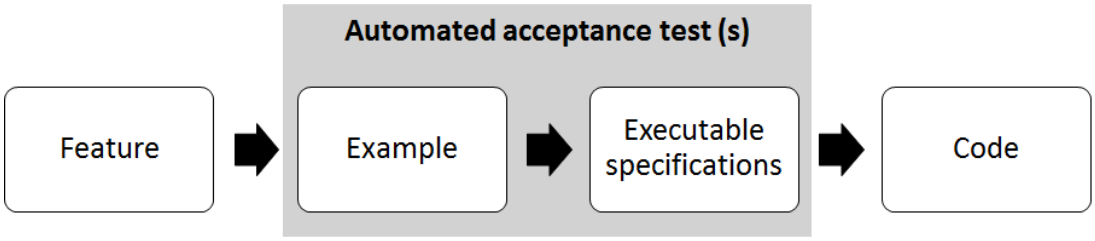
2.3 BEHAVIOUR-DRIVEN DEVELOPMENT

For supporting developers to avoid merge conflicts, we propose a strategy to predict the set of files a task will change and estimate the risk of merge conflicts between a task pair. Such a strategy targets a Behaviour-Driven Development (BDD) context, which we present as follows.

The term Behaviour-Driven Development designates several agile and lean practices concerned with the identification, understanding, and delivery of well-designed and well-implemented features that matter for business (SMART, 2014). Figure 3 illustrates the development dynamics in BDD by identifying its key concepts and how they relate to each other. Initially, an agile team (accurately, business analysts, developers, and testers)

describes a new *feature* (i.e., a system functionality). Next, the team designs at least one *acceptance test* to evaluate (part of the) acceptance criteria that describes the behavioral intent of the feature. At first, acceptance tests fail, as the system does not support the feature yet. Then, developers write just enough *code* to make tests pass.

Figure 3 – Key elements in BDD.



It is possible to define manual or automatic acceptance tests. BDD requires automatic acceptance tests to improve productivity. As Figure 3 illustrates, an automatic acceptance test has two distinct parts: *example* and *executable specifications* (SMART, 2014). The *example* illustrates the usage of a feature written in a readable language, similar to plain text. The *executable specifications* refer to code that automates the execution of an *example*. This dynamic also applies to maintenance tasks, bug fixing, or refactoring. The difference is that developers might have to deal with previous existing acceptance tests.

Although many times there is a one-to-one mapping between a user story and a BDD feature, it is not mandatory. For supporting task management, a BDD feature might represent a whole user story or just a piece of it, depending on the diversity of acceptance criteria. For instance, we can split the user story illustrated by Figure 1 into three BDD features: usage of a credit card to ensure a reservation, detailed view of available hotels, and hotel search by city and price. The whole user story is complete when all of its BDD features are complete, .i.e., passing the acceptance tests. However, developers might independently release and validate each BDD feature.

2.3.1 Cucumber

To illustrate an automatic acceptance test used in BDD, we introduce Cucumber. As said before, BDD requires the adoption of automated test tools to work effectively, the so-called BDD tools. There is a variety of such tools. In this work, we use Cucumber, which is a popular open-source tool initially written for Ruby that provides support for several languages.

Figure 4 illustrates a feature[8] from project *alphagov/whitehall*,[9] which is a Rails content management application for the UK Government. The feature relates to the management

[8] The feature is declared in file *features/edition-attachments.feature* (version of commit 9deb239c100f2f13d7f82dfe70095b4208460f5a).

[9] <https://github.com/alphagov/whitehall>

of attachments when publishing some content. As can be seen, it adopts the *role-action-objective* template, clearly expressing the value of an action to a type of user, like a user story. Developers might write features by using any format, given they are plain text, but such a template is a good practice. The only keyword is **Feature**.

Figure 4 – Feature example from project *alphagov/whitehall*.

```
Feature: Managing attachments on editions
As a writer or editor
I want to attach files and additional HTML content to
publications and consultations
In order to support the publications and consultations
with statistics and other relevant documents
```

Next, Figure 5 illustrates a BDD example[10] related to the previous feature. As can be seen, Cucumber calls BDD examples as *scenarios*. Every scenario has a title as an identifier, which is the content after the colon that succeeds keyword **Scenario**. In addition, a scenario contains a set of *steps* identified by the illustrated keywords **Given**, **When**, **Then**, and **And**, which have different meanings. A **Given** step identifies a test pre-condition, which is responsible for the system setup. A **When** step identifies the central action or functionality that the test verifies, which often transforms the system state. A **Then** step identifies the expected result like a system state or a perceived condition by the user. Finally, **And** makes the scenario read more fluidly by avoiding a repetitive sequence of steps (e.g., consecutive **When** steps). The values used to specify the publication title, and other information (usually) in quotes,[11] such as the ISBN, are parameters for the test.

Figure 5 – Scenario example from project *alphagov/whitehall*.

```
Scenario: Editing metadata on attachments
Given I am an writer
And I start drafting a new publication "Standard Beard Lengths"
When I start editing the attachments from the publication page
And I upload an html attachment with the title "Beard Length
Graphs 2012" and the isbn "9781474127783" and the web isbn
"978-1-78246-569-0" and the contact address "Address 1"
And I publish the draft edition for publication "Standard Beard
Lengths"
And I preview "Standard Beard Lengths"
Then previewing the html attachment "Beard Length Graphs 2012"
in print mode includes the contact address "Address 1" and the
isbn "9781474127783" and the web isbn "978-1-78246-569-0"
```

[10] The scenario is declared in file *features/edition-attachments.feature* (version of commit 9deb239c100f2f13d7f82dfe70095b4208460f5a).

[11] The usage of quotation marks is not mandatory.

Cucumber requires developers to write features and scenarios in Gherkin,[12] a domain-specific language that is similar to plain text, as illustrated. Gherkin provides other mechanisms to write tests, aiming readability and maintenance; we illustrate the more relevant ones.

To run a scenario, it is necessary to implement a *step definition* for each step. Note that Cucumber calls BDD executable specifications as *step definitions* and, for simplicity, this is the terminology we use throughout the text. Figure 6 illustrates the step definition[13] related to the last step depicted by Figure 5 in Ruby. Basically, a step definition is a method identified by a regular expression[14] that Cucumber uses to match steps and step definitions when running tests. Also, in Ruby, values in quotes in a regular expression are arguments to the method. The body of each step definition contains code that effectively implements the test by invoking Rails and test frameworks methods that refer to files or programming elements of the features that are supposed to be implemented. For instance, the call to `visit`, a method of Capybara,[15] accesses a view, and the call to `assert`, a method of RSpec,[16] compares the expected results to actual results (in Figure 6, whether the view contains some information). Cucumber supports integration with a variety of UI technologies, according to the system programming language. For instance, Capybara and RSpec are compatible with Rails applications.

Figure 6 – Example of step definition from project *alphagov/whitehall*.

```
Then (/^previewing the html attachment "(.*?)" in print mode includes
the contact address "(.*?)" and the isbn "(.*?)" and the web isbn
"(.*?)"$/) do |attachment_title, contact_address, isbn, web_isbn|
    html_attachment = HtmlAttachment.find_by title: attachment_title
    publication = html_attachment.attachable
    visit publication_html_attachment_path publication_id:
    publication.slug, id: html_attachment.slug, medium: "print"
    assert page.has_content? contact_address
    assert page.has_content? isbn
    assert page.has_content? web_isbn
end
```

We illustrate a Cucumber test that considers user interaction by accessing a web page and checking its content. However, developers might write Cucumber tests that verify the system state and abstracts its user interface. Given these details relate to the test code, they should be perceived only in the body of the step definitions. However, developers usually write Gherkin scenarios describing user interaction, as a click on a button, making

[12] <https://github.com/cucumber/gherkin>

[13] The step definition is declared in file *features/step_definitions/attachment_steps.rb* (version of commit 9deb239c100f2f13d7f82dfe70095b4208460f5a).

[14] The latest Cucumber version also supports Cucumber expressions, a kind of regular expression with a syntax that is more friendly for humans to read and write.

[15] <http://teamcapybara.github.io/capybara/>

[16] <https://rspec.info/>

the whole Cucumber test dependent on the user interface. Note that throughout the text, we use the term "Cucumber test" referring to the whole automated acceptance test, whereas "Gherkin scenarios" mean the high-level usage scenario only.

By default, features and its scenarios are declared in Gherkin files (extension .feature) into the "features" folder, and step definitions are placed in files written in the application programming language (Ruby files in our example) placed on subfolder "step_definitions".

2.4 RUBY ON RAILS APPLICATIONS

TAITI, our tool for inferring test-based task interfaces, supports Ruby on Rails applications. To better clarify its implementation details and limitations, we provide a brief overview of Rails. Rails is an MVC (Model-View-Controller) framework for web applications, meaning every Rails application organizes code into three distinct layers. The model layer is responsible for business logic and manages the interaction with elements in a database, including data validation. The view layer represents the user interface as HTML files (and variants) with embedded Ruby code and can provide content in different formats, such as HTML, PDF, XML, and so on. The controller layer interacts with models and views, receiving requests from views, processing data from models, and transferring data back to views.

By default, Rails define an extense directory structure for a project. Models, views, and controllers are in the "app" folder, into subfolders "models", "views", and "controllers", respectively. Extended modules are put into the "lib" folder. There are other folders, but these are the relevant ones to this work. Rails (and Ruby projects in general) have the *gemfile* in the root folder that lists all project dependencies, that is, Ruby libraries named *gems*.

Also, Rails provides a router mechanism to assist controllers in dispatching incoming requests. That is, a router receives each incoming request, parses it, and sends it to a controller class as a method call (*action*). To illustrate, suppose a user clicks on the `update profile` button on her profile page. Then, the application receives a URL request `http://my.app/profile/update/1`. The routing component translates the received request to a method call for `update` in the controller class `ProfileController` using `1` as an argument that identifies the profile ID. The method `update` finds the profile, organizes the view to show the current contents of the profile for edition, and invokes the view code. Developers configure the Rails router mechanism by the file *config/routes.rb*.

Rails became a popular framework, especially in the agile software development community, because it improves developers' productivity. Relying on naming conventions, Rails automatically configures code elements, enabling teams to fast release Web applications.

3 PREDICTING FILE CHANGES

To assess the ability of test-based task interfaces for supporting developers to avoid merge conflicts, we need to verify whether such interfaces can predict file changes. Otherwise, conflict risk predictions based on *TestI* might not help. So, in this chapter, we present the first empirical study we conducted to evaluate the potential of *TestI* for predicting file changes. In this sense, we detail *TestI* and how we compute it.

3.1 MOTIVATING EXAMPLE

To illustrate how test-based task interfaces might be useful to predict changes and avoid conflicts, let us consider that Adam and Betty are members of an agile team that is developing a Rails school management system that keeps student grades and lets teachers visualize them. In a given iteration, suppose Adam was assigned a task for developing class evaluation functionality (task T_1). He is then creating a method to compute the mean of student grades and writing code that shows this extra information in the class visualization page. Meanwhile, Betty had to choose a new task and opted for supporting teachers to quickly identify low-performance students (task T_2). She is then creating a method to return students with grades over a given limit, and writing code that highlights these students in the class visualization page. The iteration backlog includes tasks such as fixing the news feed to exhibit only recent messages (task T_3).

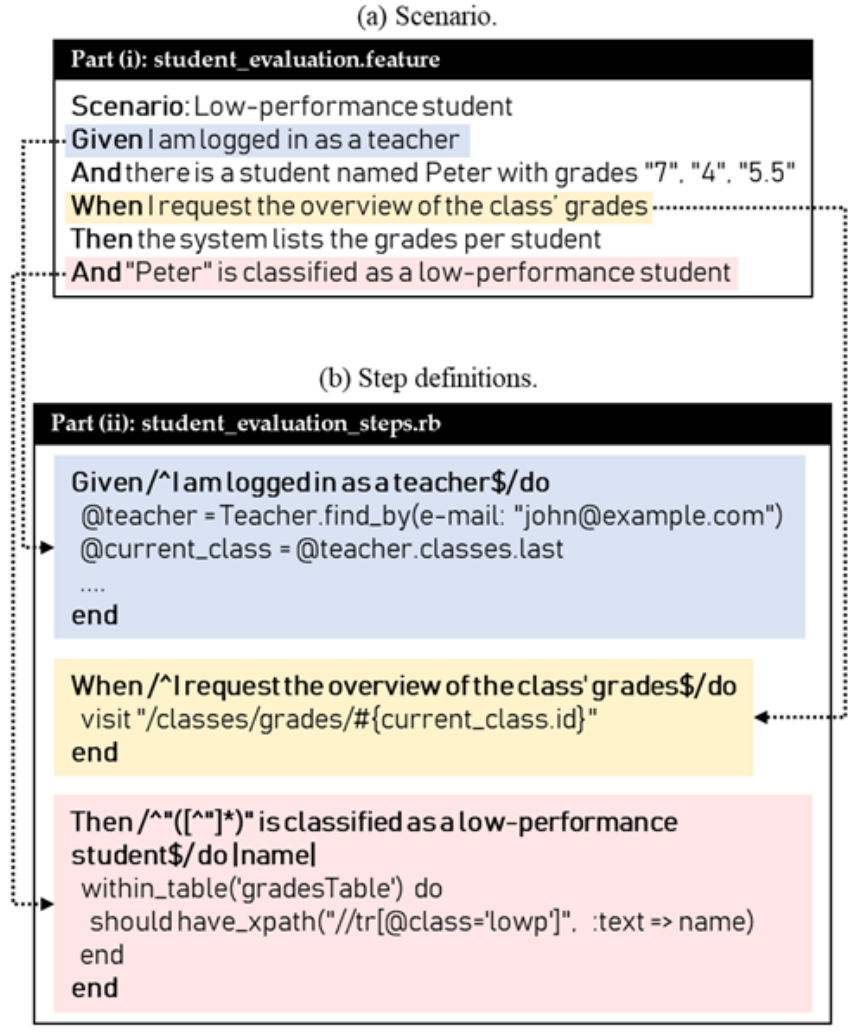
By independently working on T_1 and T_2 in their private repositories, Adam and Betty add different methods (`compute_grade_average` and `low_perf_students`) at the end of copies of the same file (app/controllers/classes_controller.rb), and change the same area of the class visualization page (app/views/classes/grades.html.erb), which later leads to merge conflicts when the tasks are finished and they integrate their contributions. The conflicts would have been avoided if Betty had opted for T_3 instead of T_2, since T_1 and T_3 are associated to unrelated features and affect disjoint and independent file sets. Whereas T_1 and T_2 require changes in the same Rails controller and views, related to the class evaluation feature, T_3 requires changes in the controller related to the notification feature.

More experienced developers could have chosen parallel tasks more wisely, but it is not always easy to infer which files will be changed by a given task. Moreover, a task not always involves changes to a single MVC segment (set of related model, view, and controller files) as in our example, further complicating matters even for experienced developers. Automatically predicting code changes associated to a development task is, in general, hard. But such prediction might be feasible for a specific context (BRIAND et al., 2017). In particular, in a BDD context, automated acceptance tests are written before implementing features. Moreover, each feature implementation task can be often

associated to a number of usage scenarios that are directly linked to the tests. So, by inspecting the code that automates the tests associated to tasks T_2 and T_3, Betty could have inferred parts of the application code that would be exercised by the tests, and these could perhaps approximate the files that would be changed by the tasks. Such inspection could then have helped Betty to safely opt for task T_3 and avoid the conflicts.

To better explore this possibility, consider in Figure 7 the Cucumber automated test related to task T_2. The test has two parts: (i) a high-level concrete usage scenario written in Gherkin, with test setup steps (**Given**), test actions (**When**), and expected results (**Then**), among other keywords, such as **And**, which makes the scenario read more fluidly by avoiding a repetitive sequence of steps (e.g., consecutive **Given** steps); and (ii) Ruby code that automates the scenario steps (so called *step definitions*). Note that in this chapter we use the term "Cucumber test" refering to the whole automated test, whereas "Gherkin scenarios" means the high-level usage scenario only.

Figure 7 – Example of an automated acceptance test related to task T_2.



Developers implement these step definitions by invoking Rails and test frameworks methods that refer to files or programming elements of the features that are supposed to be

implemented. For example, the first line of the **Given** step refers directly to the `Teacher` class. The body of the **When** step accesses a view by calling the test framework method `visit`. In this case, the accessed view is the class visualization page that both Adam and Betty changed; by default Rails routes "/classes/grades/#{current_class.id}" to the method `grades` from `ClassesController` by using `current_class.id` as parameter, and this method renders the mentioned page. So, with further analysis, one can conclude that the app/models/teacher.r and app/views/classes/grades.html.erb files, among others, will likely be exercised by the illustrated test.

Although full details about tasks and tests are omitted here for brevity, by systematically analyzing the step definitions in our complete example, we would obtain the file sets (test-based task interfaces) in Figure 8.

Figure 8 – Test-based task interfaces of tasks T_1, T_2, and T_3. The files in bold are the ones actually changed by developers.

TestI(T_1) and TestI(T_2)	TestI(T_3)
app/models/class.rb	app/models/principal.rb
app/models/student.rb	app/models/notice.rb
app/models/teacher.rb	**app/controllers/notices_controller.rb**
app/controllers/classes_controller.rb	app/views/notices/index.html.erb
app/views/classes/grades.html.erb	app/views/notices/new.html.erb
	app/views/noticies/_form.html.erb

By noting that $TestI(T_1)$ and $TestI(T_2)$ have the same content, and assuming that this safely approximates files to be changed by tasks, Betty would have avoided choosing to work on $TestI(T_2)$ knowing that Adam is still working on $TestI(T_1)$. As $TestI(T_3)$ has no intersection with $TestI(T_1)$, Betty would have concluded that the parallel execution of T_1 and T_3 has a lower conflict risk. This is indeed the case, as T_3 ends up changing only a single file (in bold in Figure 8), which is not changed by the other tasks.

Although, for simplicity, this is an idealised example, it reflects the main aspects of a number of real integration scenarios we later analyze and discuss. In particular, it shows that not all files exercised by the tests associated to a task are actually changed by the task; among other reasons, part of the feature might have been implemented before. Contrasting, there might be files that are changed by a task but not captured by its test-based task interface; among other reasons, tests might not sufficiently cover feature functionality. This situation then demands evaluation of the ideas we discuss in this section.

Finally, someone might wonder whether a developer might manually compute the task interface by reading the step definitions. Actually she can, but these often involve chains of method calls and references to a number of web pages that can be complicated to manually follow, being error-prone and demanding extra effort. The presented example is quite simple and does not match such a statement, but let us see a task example[1] of a project on GitHub.

[1] <https://github.com/AgileVentures/WebsiteOne/commit/ab1723df7878152b4a814fa5fbfe86c7076aecab>

The task was concluded by one commit that changed 15 application files (considering the files of our interest) and has 9 Cucumber tests. For computing *TestI* it is necessary to analyze 43 methods into eight step definition files as well as 15 web pages. Similarly, others might wonder whether some agile practices such as daily stand-up meetings might prevent conflicts like our motivating example. It is also true, but communication might be more imprecise (STRAY; SJøBERG; DYBå, 2016). That's why we prioritize an automatic solution aiming to promote developers productivity and effectiveness. So, although BDD principles could help code change prediction, we expect more benefits when BDD is used together with a tool that computes test-based task interfaces.

3.2 TEST-BASED TASK INTERFACES

To better explore the code prediction idea illustrated in the previous section, and assess its predictive power, we implemented TAITI, a tool that, for a given task, computes its *test-based task interface* (*TestI*): set of application files that might be exercised by the tests associated to the task. The tool works for tasks associated with Cucumber acceptance test scenarios, and approximates the set of files having code that could be executed by running the scenarios. To compute such set, we first parse scenarios and link them to the corresponding step definitions[2]. We then statically analyze the associated step definitions, collecting references to programming elements (such as fields and methods) and Rails views. From these references, we conservatively infer files that declare the programming elements, files associated with the views, and, recursively, further elements and files referenced by the views.

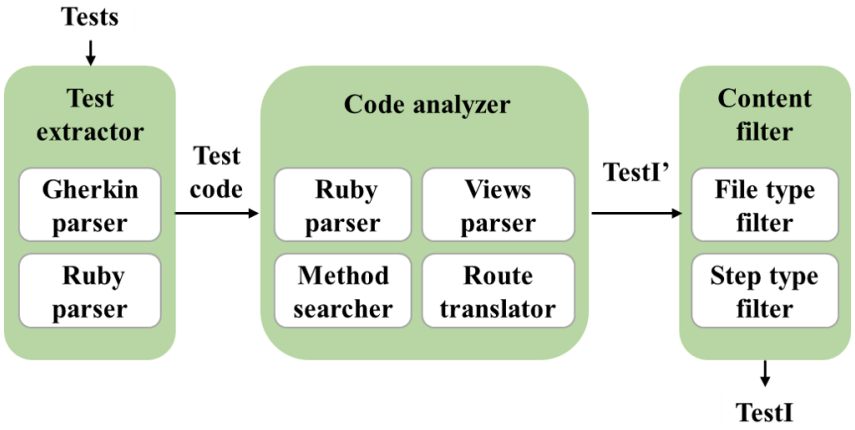
In this manner, any project can use our tool as long as the team develops acceptance tests before the application code. In our vision, such a requirement fits better into BDD teams but considering that BDD encompasses different activities beyond writing and executing acceptance tests, we do not impose the adoption of BDD, neither we suggest how to conduct any development activity; the teams are free to perform as they want. Furthermore, we adopted Cucumber tests as a reference to acceptance test tool given the impossibility to deal with the specificities of the various existent tools. In practice, another tool instead of Cucumber might be used. Cucumber is also a BDD tool, but the difference between a standard or a BDD tool for automating acceptance tests is not important in our context. In this sense, throughout the text, we refer to "acceptance tests" when possible to decouple the conceptual idea of test-based task interfaces and TAITI, the tool that computes it. We adopt the Cucumber terminology when we need to reference implementation details.

In the following subsections, we better explain the logical stages of our tool, which is structured as in Figure 9, and we provide information about its implementation. In

[2] "Step definition" is the Cucumber terminology for referencing the code that automates the tests. This way, throughout the text we always mention it.

brief, the *test extractor* module parses scenarios and finds the step definitions related to a task, and the *code analyzer* module analyzes the step definitions collecting references to invocation of application code, generating a initial version of *TestI*. Then, the *content filter* module refines *TestI* by filtering files according to different strategies. TAITI is a public and opensource prototype tool that has 13,178 LOC.

Figure 9 – TAITI architecture.



3.2.1 Finding step definitions

Our tool receives as input a Rails project and a set of Cucumber automated acceptance tests, with scenarios written in Gherkin and step definitions in Ruby. The tests are supposedly associated with an existing development task, but the tool doesn't actually rely on that. First, the *test extractor* module parses the scenarios and step definitions, and, by traversing ASTs (Abstract Syntax Tree), tries to match each scenario step to a regular expression that identifies one of the step definitions, as Cucumber does. If no step definition matches a scenario step, we simply ignore the step. If more than one step definition matches a scenario step, we consider only the first match we found. Although Cucumber raises an error in case of such ambiguity error, we tolerate it as a "repairing strategy", given the error might be caused by TAITI, that is, developers might declared unique steps definitions but our tool might wrongly extract the regex expression that identifies it. For example, suppose the given regex identifies a step definition: `^the code of (#{m}) is "([^\"]+)"$`. Our tool represents the value involved by "`#{}`" as "`(.+)`" because it is defined during runtime, and we do not run tests when computing *TestI*. Thus, such a regex might match a step and multiple step definitions, causing an error that Cucumber would alert.

As a result, the *test extractor* module yields a set of matched step definitions, excluding duplicate entries (different scenarios might refer to the same step) with the aim of improving code analysis performance in the next stage.

3.2.2 Finding references to application code

Having obtained the matched step definitions, we analyze them. Given that the task functionality is not yet available in a BDD context, dynamic analysis through test execution is not promising— tests would prematurely break. So we opt for statically analyzing the step definitions according to a straightforward strategy, first collecting references to programming elements, as they are likely executed by the tests and can help us to identify the set of files that constitute a test-based interface. The *code analyzer* module considers references through method calls, constructor calls, and access to constants; field access is represented as a method call in Ruby. Since our aim is to identify application specific files that might be exercised by the tests and potentially might cause conflicts, we ignore the usage of operators (which are Ruby methods) and other library elements (methods, constructors, and constants).

After collecting references, *code analyzer* tries to identify the files that declare the referred elements. Due to Ruby's dynamic nature, statically matching, for example, a method call to a method declaration, and consequently the file in which it appears, is hard. So we adopt a lightweight and mostly conservative solution, as detailed in Subsection 3.2.3. If a method call targets a class, we look for files matching the class name. For example, the call `Teacher.find_by(e-mail:"john@example.com")`[3] leads to the search for a file with suffix *teacher.rb*. For calls that target expressions with no type information about the target object, we search for a matching method declaration with the same method name and number of arguments. When possible, we check whether naming conventions apply. For example, the call `@contract.sign(@user)` leads to the search for a `Contract` class. If such a class exists, we check whether it contains the method `sign`, otherwise we search other files for a method `sign` that accepts one argument.

In case multiple method declarations match a call, we conservatively consider all declarations with parameters compatible[4] with the arguments in the call. Moreover, we do not take into account arguments types, similarly leading to imprecision in the matching process. For example, the call `@contract.sign(@current_user)` receives an `User` object as argument, but we do not consider that because we do not know the type of `current_user`, neither there is a class named `CurrentUser`. At this point, someone might question the reason we do not use an approach for type inference for Ruby or even Ruby on Rails. We do not find a mature solution supported by a tool for practical usage, although the research initiatives (FURR et al., 2009; AN; CHAUDHURI; FOSTER, 2009; MIRANDA; VALENTE; TERRA, 2016).

Note that we do not specially deal with polymorphic method calls, which might cause a wrong match between a method call and a method declaration. For example, suppose

[3] `find_by` is a query method provided by Rails according to the Active Record pattern.

[4] Due to *var-args* and parameters with a default value, arguments lists for a given method might have varying sizes in Ruby.

there is an overridden library method. TAITI considers that the overriding version is always used, which might not be the case.

Contrasting, our matching process might miss relevant code elements too. This might be the case when methods called by the step definitions have not been declared in the application files, and the calls are not easily associated to a class. So, they do not contribute to *TestI* content. Probably much of the called method with no matching declaration is a library method or a Rails auto-generated method, that is, they represent code that might not lead to conflicts. But they also might be code to be implemented. We assume, though, that test developers create empty declarations of the elements they refer to in the step definitions, as this is an important way of establishing a contract with feature developers.

In the process of matching references to files, we ignore files that correspond to auxiliary code used by step definitions. To distinguish auxiliary code used by the tests from actual application code, we rely on the project's directory structure and file extension. By default, test files are Gherkin files into `features` folders, and Ruby files into `features/step_definitions` folders. Application files are Ruby and HTML files (and variants, including ERB and HAML files) into `app` or `lib` folders.

3.2.3 Finding references to views

Besides collecting references to classes and their files as just explained, the *code analyzer* module also collects references to Rails views, as these are widely exercised by acceptance tests. To find references to views, we basically inspect calls to the `visit` method that is provided by Capybara,[5] a library that is often used by Cucumber tests to simulate user interaction with a GUI. This method receives as argument a path that Rails maps to a route to dispatch a controller's action, avoiding the need for hard-coding URL strings in tests, although it is also accepted too. The *When* step in Figure 7 illustrates a call to `visit`.

By analyzing the path (or URL) and the route information, we can then identify the view and controller files that will likely be executed by the `visit` call, and that should be included in the test-based task interface. However, there are alternative ways to call such method. Rather than explicitly defining the path in the step definition code as our example shows, tests often indirectly refer to the path as the value yielded by an expression (such as an argument or the result of another method call). Moreover, Rails can be configured so that routes are personalized and auxiliary auto-generated methods can be associated with routes (that we call as "Rails path methods"). Therefore, when we find a call to `visit`, we first have to identify the accessed route by extracting the arguments used in Gherkin scenarios, propagating them to the step definition code, and finally translating the route to controller and view files. Due to Ruby's dynamic nature and the limitations of our propagation and route mapping algorithms, we might miss view accesses, or we might

[5] <http://teamcapybara.github.io/capybara/>

consider view access that did not happen. For example, we cannot find a route related to a path whether there are multiples alternatives in the configuration file and the decision requires the evaluation of a condition. Similarly, we might incorrectly translate a route to files if the route value depends on a variable. Thus, we might wrongly include or exclude files into *TestI*.

Directly using Rails route interpreter would require running the system being developed, but this is not always applicable, especially in early development phases. So we implemented a simplified route interpreter, which has limitations in comparison to Rails interpreter, but is not significantly less accurate, as explained later in this chapter.

3.2.4 Finding application code referenced by views

Having identified view files referenced by the tests, the *code analyzer* parses these files and, recursively, tries to identify further programming elements and files referenced by the views. We basically search for usage of instance variables and method calls related to forms, buttons, links, and file rendering commands, as these are associated with user actions and might be exercised by the tests too. By naming convention, Rails establishes implicit relationships among views. As these are partially reachable by analyzing the step definitions, we do not capture all possible references and connections among views. We only conservatively capture all explicit connections. During early development of a feature, the referenced views might not yet exist due to the BDD practice we are assuming in our context. In that case, our code analysis reaches only the surface of the code that could be exercised by the tests.

In the ideal situation, we would apply a static reference analysis for Rails applications similar to the one proposed by Rountev e Yan (2014) for Android apps, that provide support for translating GUI interactions to method calls, improving the whole process of code analysis.

3.2.5 Design alternatives for test-based task interfaces

By analyzing step definitions as explained so far, the resulting set of files that can be exercised by the tests might contain files that won't be changed by the task associated with the tests. Moreover, some of these files might not even be related to the task. For instance, many tests begin with user authentication setup steps, but it does not mean that the task validated by these tests will require changes on code related to user authentication. By considering these steps, we add to *TestI* files that are not related to the task in the sense that they will not be changed or accessed during task execution. So, envisioning to consider the subset of more relevant files to complete a task, we explore three design alternatives for *TestI*. Basically, they adopt different policies to restrict the *TestI* content, varying both the parts (e.g., setup, expected results, etc.) of the test and the kinds of files (e.g., models, controllers, etc.) we consider when computing the interfaces.

3.2.5.1 Filtering by step type

Given different tests require similar `Given` (setup) steps, as we note by manually analyzing tasks, they could not be relevant to most tasks, as just illustrated. In contrast, `When` steps often focus on the core functionality being tested, and so one could expect that they exercise the most important files for the underlying task. So, instead of considering files possibly exercised by any of the steps in a test, a design alternative is to just consider files exercised by `When` steps, discarding `Given` and `Then` steps. However, because calls to the `visit` method are crucial for understanding the context of `When` actions, in addition to `When` steps, we partially analyze `Given` steps searching only for `visit` calls, and analyzing the associated views. For example, applying this *TestI* `When` Filter (WF) to Task T_1, instead of yielding the interface in Figure 8, our tool would yield the interface with just the two files in bold at the left side of the figure.

To support the WF filter, our tool tries to infer step types, given that steps can also call other steps, and be declared with generic keywords such as `And` and `But`, or even the `*` wildcard, which all apply to the three kinds of step types. In brief, the step type is determined by its correspondent step in Gherkin, except when it is qualified by a wildcard (in this specific case, we can only consider the keyword used in the step definition). The tool also has to keep information about step type for each found reference to application code.

Alternatively, we would consider as part of *TestI* only the application files that might be exercised by all tests associated to a task. However, we intuitively expected that `When` steps would exercise more files in common than other step types, as the tests would be related to the same core functionality. Thus, the resultant *TestI* would approximate to *TestI* filtered by step type.

3.2.5.2 Filtering by file type

As many tasks in the context we consider focus on changing a single MVC segment (that is, related model, view and controller files), in principle one could avoid conflicts by avoiding the parallel execution of tasks that focus on common segments. So, considering that controller files uniquely identify segments, interfaces could perhaps focus on these files. Besides that, the current version of our tool does not reach the full method chain underlying a system functionality, but only the initial calls. We then consider a design alternative that only includes controller files in the interface. For example, applying this *TestI* Controller Filter (CF) to Task T_1, instead of yielding the interface in Figure 8, our tool would yield the interface with just the first file in bold at the left side of the figure.

3.2.5.3 Applying multiple filters

We also combine the previous filtering strategies, as Table 1 briefly presents. Our motivation is to improve the selection of the *TestI* content by combining the underlying policy from the other filters; i.e., we intend to investigate whether we can sum the potential individual benefit of each filter.

Table 1 – Filters for task interfaces content.

Filter	Meaning
NF	We do not apply any filter
CF	We filter out non controller files
WF	We filter out files not exercised by When steps
WCF	We apply both CF and WF filters

3.3 EMPIRICAL STUDY

To investigate whether *TestI* helps to predict file changes (additions or deletions) associated to a task, we try to answer a number of research questions. First we want to compare the predictive capacity of the variations of *TestI* (described in the previous section) with the corresponding *task interface* (*TaskI*): the set of files actually changed by the task, i.e., our oracle. So we consider the following question.

Research Question 1 (RQ1): How often does TestI predict file changes associated with a task?

To answer *RQ1*, we compute precision and recall measures for *TestI*. Precision is the percentage of files in *TestI* that are also in *TaskI*, whereas recall is the percentage of files in *TaskI* that are also in *TestI*. As the adverse impact of false positives (files in *TestI* that are not actually changed by the task) in this context is basically to discourage parallel execution of tasks that would not conflict, or encourage slightly more unneeded coordination, the false negatives (files changed by the task but not included in *TestI*) are more important because they could lead to conflict occurrence and extra effort for resolving it. So, in our analysis, we favor recall over precision.

To consider *TestI* design alternatives, we derive the following questions from *RQ1*:

- RQ1.1: Does filtering out non controller files improve predictive ability?
- RQ1.2: Does filtering out files not exercised by When steps improve predictive ability?
- RQ1.3: Does filtering out non controller files, and files not exercised by When steps, improve predictive ability?

- RQ1.4: Does restricting entry tests to created tests improve predictive ability?

To answer the first three questions, we compare precision and recall by computing both *TaskI* and *TestI* under two conditions: under the presence and the absence of the investigated filter. If the filter presence improves both precision and recall, we conclude the evaluated filter refines *TestI*. Otherwise, we favor recall over precision, as previously discussed. To answer the last question, we proceed with a similar comparison approach, but varying the entry test set. This way, we consider the tests created or changed with the aim of validating the task results and only the created tests.

To understand the limits of static code analysis for computing such interfaces, we assess our algorithms and compare *TestI* with task interfaces computed by executing tests (*DTestI*) and parsing the test coverage report. In this sense, *DTestI* is a kind of oracle: We apply static analysis to predict the files exercised by tests, and we assess whether the tests truly exercise them. Note that this kind of dynamic analysis based interface is not as applicable as *TestI*, since it requires running the tests; in practice, given the BDD context we consider, the tests associated with a task often fail early because they exercise possibly inexistent, outdated, or premature functionality implementation that is only fixed after performing the task. Nevertheless, given we conducted a retrospective study with completed tasks whose tests we certified that succeed, the comparison sheds light on *TestI* strengths and drawbacks. So we ask the following question.

Research Question 2 (RQ2): Is static code analysis suitable to compute TestI?

To answer *RQ2*, we first investigate whether our simplified routing mechanism (see Section 3.2.3) significantly impacts predictive capacity, either by mapping a path to a wrong route (increasing the number of false positives) or by losing a route (affecting the number of false negatives and false positives). To this end, we compare *TestI* content (using both Jaccard index and cosine similarity), precision, and recall by computing task interfaces under two different conditions: using our routing mechanism and using the Rails routing mechanism. Besides that, we check whether *TestI* has better precision and recall measures (with respect to *TaskI* as in RQ1) than *DTestI*. We compute *DTestI* by running the tests and extracting, from test coverage reports, the file set each test exercises. The resulting interface for a given task is then the union of the obtained file sets for each test associated to the task.

To consider a baseline that would be as applicable as *TestI*, we compare *TestI* with randomly defined task interfaces (*RandomI*), answering the following question. *RandomI* is an analogy to an inexperienced developer that is responsible for manually determining the task interface; in the absence of knowledge about the task and the system, such an interface might approximate to a guess.

Research Question 3 (RQ3): Is TestI a better code change predictor than RandomI?

To answer *RQ3*, we check whether *TestI* has better precision and recall measures (with respect to *TaskI*) than *RandomI*. To compute *RandomI*, for each project file that has the potential to be in a task interface— that is, ruby files and supported view files— we randomly decide whether it should be in the task interface. To compute *RandomI* measures for a given task, we generate 10 *RandomI* for the task, compute precision and recall measures for each interface, and consider as final measure the mean of the 10 intermediate measures. Intermediate measures vary little per task.

Finally, to understand how more informative is the code that automates the tests in comparison to the test descriptions, we compare *TestI* with task interfaces obtained by observing textual similarity of test specifications, considering past tasks and the files they changed (*TextI*). Given the dependence on project history, this is not as applicable as *TestI* but the comparison sheds further light on *TestI* strengths and drawbacks. The inspiration for designing *TextI* came from studies related to code change prediction that investigate the past to predict the future, relying on the idea that similar tasks are likely to change or use the same code elements (see Hipikat (CUBRANIC et al., 2005) in Sec. 5). As we cannot adequately compare *TestI* and Hipikat because of the differences of the required inputs of each, we conceived an alternative solution that reuses the main idea of Hipikat.

Research Question 4 (RQ4): Is TestI a better code change predictor than TextI?

As before, we answer this question by checking whether *TestI* has better precision and recall measures (with respect to *TaskI*) than *TextI*. For computing *TextI* for a task *t*, we take the intersection of the sets of files changed by the three past tasks with most similar specifications to *t*. Our vision is the intersection means the most relevant files whereas the union represents the maximum set of changes. As code changes usually are not cohesive, the union might increase the number of false positives, substantially reducing precision. Concerning the limited number of similar past tasks (three), we define it based on the restricted number of tasks per project in our sample (there is a project with only two entry tasks for computing *TextI*, for instance), avoiding bias. In Section 3.4.5 we provide detailed information about our task samples.

For simplicity, we assess specification similarity by using cosine similarity between vectors of TF-IDF values (SALTON; MCGILL, 1986). In our context, the task specification is the Cucumber usage scenario written in Gherkin (as in the first part of Figure 7), including feature descriptions. Therefore, we compute the vectors of TF-IDF values by preprocessing specifications based on the standard information retrieval approach, which tokenizes the text using spaces and punctuation, stems it and eliminates English stopwords as well as Gherkin keywords (in our specific context). Our implementation uses the standard

analyzer of the Apache Lucene library[6].

3.4 STUDY SETUP

Before answering the just introduced questions, we describe our study setup, including how we collect data and the infrastructure supporting it.

3.4.1 Project selection

Given the nature of our tasks, and anecdotal knowledge about the popularity of BDD communities and tools, we first searched for GitHub Rails projects that use Cucumber[7] for implementing acceptance tests. As *RQ2* relies on collecting test coverage information, we also searched for a subset of projects that additionally use SimpleCov[8] or Coveralls[9], which are widely used test coverage tools in the Rails community. We could have considered projects with other coverage tools but, as computing *DTestI* requires parsing tool specific test coverage information, we focused on these two for simplicity.

We performed our searches with a script[10] we implemented to query GitHub's database using GitHub Java API.[11] As GitHub does not provide a mechanism to query projects according to the tools they use, we first queried and then downloaded the latest version of each resulting project to check, in the main branch, whether the project uses the tools of interest. Such investigation was performed in September 2017. Given Ruby projects have a so called *gemfile* that lists all project dependencies (so called *gems*, that is, Ruby libraries), we could easily check the use of Rails, Cucumber, and the test coverage tools we mentioned before.

For optimizing the search and improving the chances of finding projects that satisfy our requirements, we only considered projects created after 2010. Before that, Cucumber and BDD were less popular. Moreover, dealing with older versions of Ruby and Rails could be a problem for the parsers we use, and would likely be a problem for *RQ2*, which requires running tests and executing the system.

In brief, we performed three main search rounds. In the first round, we restricted the project's maximum number of stars, sorting results by descending order of stars number, hoping to select more meaningful and popular projects. In the second round, we just sorted results by the date of the last update, to analyze first active projects. Finally, we verified

[6] <https://lucene.apache.org/core/>

[7] It is possible that a project using Cucumber does not adopt BDD. In the context of our retrospective study, it is not a problem whereas we have a sample of completed programming tasks with two information: the set of files that implement the task and the set of Cucumber tests that validate the task.

[8] <https://github.com/colszowka/simplecov>

[9] <https://coveralls.io/>

[10] Available in our online Appendix (ROCHA; BORBA; SANTOS, 2018).

[11] <https://github.com/eclipse/egit-github/tree/master/org.eclipse.egit.github.core>

the Ruby projects referenced by Cucumber's site[12]. As a result of this mining phase, we find 950 Rails projects (531 in the first round, 414 in the second round, and 11 in the third round). Among these projects, we have a set of 61 Rails projects that use Cucumber, and a subset of 18 projects that additionally use the mentioned coverage tools.

3.4.2 Task extraction

After obtaining relevant projects that use the tools of interest for our study, we further filter out projects without tasks that contribute with both application code and Cucumber tests. We opt to work with this kind of task because we can, presumably, more easily identify the acceptance tests (which are used to compute *TestI*, *DTestI*, and *TextI*) and the application code that implements the task requirements (which is the basis to compute *TaskI*).

Given that not all projects use patterns (identifiers or others) in commit messages, as a strategy to prevent a substantial reduction of our sample size, we assume the following for relating commits to tasks: (i) task contributions are integrated through merge commits; (ii) the contribution of a task consists of the commits in between the merge commit and the common ancestor with the other contribution the merge integrates; (iii) code changes in a task contribution are needed to conclude the task; (iv) tests added or changed in a task contribution are needed to validate the task. Thus, for extracting tasks from a given project, we clone the project repository and search for merge commits (excluding fast-forwarding merges) by using JGit API,[13] sorting them by descending chronological order. We admit merge commits performed until September 30th, 2017. Then we extract two tasks from each merge commit, each one corresponding to one of the merged contributions. Therefore, a task consists of a set of commits. As preliminary filtering, we select tasks that change both application and Gherkin test files. Moreover, as a matter of performance while computing interfaces, we discard tasks that exceed 500 commits.

The result of this mining phase refines the set of 61 projects we had from the previous phase, discarding 30 projects that do not satisfy the extra requirement explained here (14 projects do not contain merge commits, and 16 projects do not contain tasks that both change application and Gherkin files). We end up with a set of 31 Rails projects that use Cucumber, and a subset of 15 projects that additionally use the mentioned coverage tools. From the larger set, we extract tasks for which we can compute the interfaces we study here except *DTestI*. From the smaller set, we extract tasks for which we can compute all interfaces.

[12] <https://cucumber.io/docs/community/projects-using-cucumber/>
[13] <https://www.eclipse.org/jgit/>

3.4.3 Collecting task data

To precisely identify the acceptance tests associated with each task selected in the previous phase, we further analyze the task commits. First, we search for added or modified usage scenarios in these commits. To support such process, we developed a syntactic differencing tool[14] for Gherkin files. So, by comparing each commit and its parent with this tool, we identify which scenarios changed, and then infer an initial set of the acceptance tests associated with the task. Next we consolidate this set by analyzing the versions of the changed scenario files that were merged when integrating the task contribution. This way we avoid inconsistencies by, for example, removing from the set tests that appeared in earlier commits of the contribution but are not in the merged version.

With the acceptance tests of each task, we can finally compute *TestI* (as described in Section 3.2) and the other interfaces we study here, and later compute precision and recall measures as explained in the previous section. We compute eight different variations of *TestI* per task, by applying the four filters (see Table 1) in two different situations: when considering the set of changed tests by a task, and when considering the subset of tests created by the task, as further explained in Section 3.5.1. Hereafter, we use *TestI* configuration to refer to one of the eight filter–situation combinations.

Regarding *TaskI*, our oracle, we identify the set of files changed by the task commits according to Git. Therefore, in our retrospective study a task is a set of commits, from which we can identify the acceptance tests and the application files related to the task, and then we can compute *TestI* and *TaskI*. In our context, we can say the oracle is reliable in the sense that each *TaskI* corresponds to actual changes that developers carried on when working on a task whose results were eventually integrated to the main project repository. However, developers often make non cohesive code changes, by mixing changes related to a task with unrelated changes like (some, not all) refactorings and minor improvements. We make no effort to filter out the non related changes. So, although this might bring some noise, this kind of noise is nevertheless what one should expect in a non retrospective BDD context: we assume tasks scheduled with the help of our tool will often mix related and unrelated changes as well. In this way, our predictions likely share important properties with predictions expected in practical uses of our tool.

3.4.4 General exclusion criteria

Aiming to fairly evaluate task interfaces, we need to select a sample of completed programming tasks with two pieces of information: the set of files that implement the task and the set of Cucumber tests that validate the task. For such reason, we apply the following exclusion criteria before answering the research questions. First, we discard tasks

[14] Please see our Appendix (ROCHA; BORBA; SANTOS, 2018).

that lead to empty *TestI* or *TaskI* in at least one configuration we consider. Such empty interfaces might reflect limitations of our tools (e.g., in case *TestI* is empty even when a task has implemented tests that fulfill filter requirements) or our sample (e.g., in case *TaskI* is empty because a task only changes unsupported content like JavaScript files). For example, when using the CF filter (*TestI* filter by controller files), we discard tasks that do not change controllers (i.e., *TaskI* does not contain any controller).

Likewise, we discard tasks with no implemented acceptance tests, or partially implemented ones, that is, tasks that add or change scenario steps that do not have a corresponding step definition. We also discard tasks with step definitions that cannot be parsed. Furthermore, we discard tasks associated with project versions that do not satisfy the tool requirements explained in the first step (see Section 3.4.1). Note that when we first selected projects, we just checked tool usage in the most recent project version. But project contributors might have started to use the tools only later in project history. So not all tasks necessarily use the required tools.

3.4.5 Samples

To answer *RQ2*, we needed to generate Rails routes, meaning we have to start up the application, and run tests for each task, demanding extra restrictions. Considering that each task corresponds to a different time in the project history, we needed to reproduce the environment configuration required by each of them, that is, we needed to install all gems and, sometimes, also edit some configuration files, and change the installed version of Ruby or Rails. For this reason, we end up with a smaller sample consisting of 74 tasks from 9 Rails projects (of the 15 from the previous step) that use Cucumber and coverage tools. By analyzing tasks by descending chronological order, we selected at most 10 tasks per project to reduce project bias and minimize configuration effort, but some projects have less than 10 tasks satisfying our criteria. Thus, we did not compute *TestI* for all tasks per project but just the necessary to reach our goal. For fairness, we had to apply specific exclusion criteria in addition to the ones in the previous section. So, to obtain this smaller sample, we discard tasks whose tests we could not run, generate test coverage report, or verify Rails routes. Moreover, for independence and diversity, we discard tasks with identical test sets.

Table 2 summarizes the steps for constructing the smaller sample. The column "Cucumber & coverage tasks" means the number of tasks that have the gem 'cucumber-rails' installed as well as a gem for test coverage. Note that this does not mean such tasks have implemented Cucumber tests. When the column "Problem to compute *TestI*" shows "YES" means it is not possible to compute *TestI* for any task and then, we did not proceed on the next stages (tests execution and routes generation). Such a situation happens for two projects. In the first case, there is no valid task for all variations of *TestI*. In the second one, there is a parse error that affects all tasks. As observed, the selected 74 tasks represent

0.69% from the entry tasks (10,765 tasks), but we did not investigate all of them, but only the needed to reach at least 10 tasks per project. In sum, we discarded 337 tasks with failing tests (332 tasks from the 3 projects for which we cannot select any task and 9 tasks from other 2 projects).

Table 2 – Construction of the smaller sample.

Repository	Tasks	Cucumber and coverage tasks	Problem to compute *TestI*	Tests execution	Rails routes	Selected tasks
MetPlus_PETS	231	13	YES	-	-	0
WebsiteOne	1,122	970	NO	PASSED	PASSED	10
whitehall	1,620	1202	NO	PASSED	PASSED	10
bsmi	193	193	NO	FAILED	-	0
diaspora	1,428	583	NO	PASSED	PASSED	10
CBA	297	138	NO	FAILED	-	0
wpcc	47	46	NO	PASSED	FAILED	0
one-click-orgs	123	61	NO	PASSED	PASSED	10
openproject	2,567	2,529	YES	-	-	0
otwarchive	2,602	1,374	NO	PASSED	PASSED	10
RapidFTR	372	63	NO	PASSED	PASSED	10
theodinproject	44	44	NO	PASSED	PASSED	5
tip4commit	29	29	NO	PASSED	PASSED	7
tracks	89	67	NO	PASSED	PASSED	2
re-education	1	1	NO	FAILED	-	0
Total	10,765	7,313	-	-	-	74

To answer the other research questions, we consider a larger sample. This time, we compute *TestI* for all tasks per project, initially resulting in a sample of 568 tasks from 18 (of the 31 from the previous step) Rails projects. This way, we discarded 13 projects because they had less than two valid tasks for all variations of *TestI*, disabling us of answer the research questions. Next, we compute *TextI*, discarding tasks with empty *TextI*, which happens when a project does not have a rich history or no similar past tasks. As a result, we have a final set of 463 tasks from 18 projects. Table 3 summarizes the steps for constructing the larger sample. In such table, "Entry tasks" is the number of tasks that changed both application and test files, have no more than 500 commits, and have the gem 'cucumber-rails' installed. In turn, "Candidate tasks" is the number of tasks for which we compute *TestI* (all its variations). Differently from the entry tasks, we know the candidate tasks did change some Cucumber test (rather than a Gherkin file merely) and their commits set is not a subset of the commit set of other tasks (avoiding a kind of dependence among tasks that might compromise the study). Finally, "Valid tasks" means tasks that are valid for all variations of *TestI*. The final set of selected tasks represents 3.06% of the tasks of the 31 original projects (15,129 tasks).

Table 3 – Construction of the larger sample.

Repository	*TestI*			
	Entry tasks	Candidate tasks	Valid tasks	Non-empty *TextI*
hackful	7	4	4	3
MetPlus_PETS	231	62	2	1
WebsiteOne	1,122	332	77	66
whitehall	1,620	370	215	175
bsmi	193	59	0	-
rigse	399	49	0	-
blacklight-cornell	464	102	32	27
rabel	4	1	0	-
enroll	2,903	399	0	-
diaspora	1,428	339	55	49
action-center-platform	92	27	18	13
hr-til	6	2	0	-
CBA	297	41	21	11
raidit	2	2	0	-
folioapp	22	10	6	5
makrio	434	148	0	-
wpcc	47	22	0	-
one-click-orgs	123	31	30	28
opengovernment	6	3	3	2
openproject	2,567	237	0	-
otwarchive	2,602	779	33	26
moumentei	2	2	2	1
ticketee	3	3	2	0
rails3-devise-rspec-cucumber	1	0	0	-
RapidFTR	372	67	48	43
time_stack	17	6	6	5
theodinproject	44	31	4	3
tip4commit	29	3	3	1
tracks	89	38	5	4
re-education	1	1	1	-
spectre	2	1	1	-
TOTAL	15,129	3,171	568	463

In sum, the samples constitute a set of 513 tasks from 18 Rails projects. Table 4 summarizes our task samples. Although we have not systematically targeted representativeness or even diversity (NAGAPPAN; ZIMMERMANN; BIRD, 2013), by inspecting our samples we observe some degree of diversity with respect to the dimensions in Table 5.[15] Although two projects have no stars, they are not toy systems. But the project that has just two collaborators is an educational project related to a Rails book. Also, note there are two projects with no tests. The reason is these projects do not use Cucumber anymore (considering the time of the text writing), but our results concern to historical data. Further information about our sample appears in the Appendix (ROCHA; BORBA; SANTOS, 2018), including the complete name of git repositories.

Table 4 – Tasks distribution per sample and project.

Repository	Smaller sample	Larger sample
hackful	0	3
MetPlus_PETS	0	1
WebsiteOne	10	66
whitehall	10	175
blacklight-cornell	0	27
diaspora	10	49
action-center-platform	0	13
CBA	0	11
folioapp	0	5
one-click-orgs	10	28
opengovernment	0	2
otwarchive	10	26
moumentei	0	1
RapidFTR	10	43
time_stack	0	5
theodinproject	5	3
tip4commit	7	1
tracks	2	4
Total	74	463

For better characterizing our samples, in the following we present some complementary data. On average the tasks from the smaller sample contain $42.09 \pm 93.48(9)$[16] commits, whereas the tasks from the larger sample contain $55.49 \pm 74.62(31)$ commits. Concerning task interfaces, Table 6 summarizes the average size of all interfaces from the larger sample.

[15] Information collected on January 2019.

[16] We use this notation to represent mean ± standard deviation (median).

Table 5 – Diversity of projects in our task samples.

Git Repository Name	Description	Stars	LOC	Tests	Forks	Commits	Authors
hackful	A platform for entrepreneurs to share demos, stories or ask questions.	71	4,168	9	18	155	10
MetPlus_PETS	A platform for searching and announcing job opportunities.	19	54,306	127	65	1,935	46
WebsiteOne	A platform for promoting the agile methodology by the development of solutions to IT charities and nonprofits.	115	552,582	391	222	5,556	128
whitehall	A content management application for the UK government.	503	208,763	281	170	22,535	309
blacklight-cornell	Cornell University library catalog.	4	681,072	209	3	5,918	38
diaspora	A privacy-aware, distributed, open source social network.	12,126	190,259	265	2,912	19,727	580
action-center-platform	A tool for creating targeted campaigns where users sign petitions, contact legislators, and engage on social media.	138	29,547	51	41	1,172	22
CBA	A template for developing applications with preconfigured utility services.	76	44,184	122	16	842	10
folioapp	A site for artists and writers to share work and submit to opportunities.	0	56,676	15	3	218	5
one-click-orgs	A website where groups can create a legal structure and get a system for group decisions.	42	46,689	206	12	2,496	25
opengovernment	An application for aggregating and presenting US open government data.	200	134,31	11	117	2,231	20
otwarchive	An application for hosting archives of fanworks, including fanfic, fanart, and fan vids.	368	289,490	1,193	211	14,215	158
moumentei	A chinese project with no documentation.	375	21,294	4	136	200	8
RapidFTR	An application for collecting and sharing information about children in emergency situations.	286	98,820	274	334	4,939	252
time_stack	A timesheet system.	0	29,256	33	1	763	18
theodinproject	A community and curriculum for learning web development.	708	39,316	0	570	2,843	133
tip4commit	A platform to donate bitcoins to open source projects or receive tips for code contributions.	159	15,443	68	116	550	70
tracks	A management tool based on Getting Things Done (GTD) methods.	891	96,940	0	519	4,056	115

Table 6 – Size of interfaces from the larger sample.

Interface	Size (average)
TestI-NF	$57.15 \pm 46.78(51)$
TestI-CF	$10.79 \pm 12.82(7)$
TestI-WF	$42.08 \pm 38.46(34)$
TestI-WCF	$8.06 \pm 9.40(6)$
TestI-CT-NF	$46.65 \pm 42.51(41)$
TestI-CT-CF	$8.92 \pm 11.22(6)$
TestI-CT-WF	$34.89 \pm 34.53(28)$
TestI-CT-WCF	$6.60 \pm 6.82(4)$
RandomI-NF	$238.46 \pm 184.47(185.30)$
RandomI-CF	$28.11 \pm 22.73(18.90)$
TextI-NF	$38.63 \pm 53.14(16)$
TextI-CF	$6.96 \pm 8.82(3)$
TaskI	$62.31 \pm 64.10(41)$
TaskI-CF	$9.62 \pm 11.19(6)$

Specifically related to *TextI*, the histogram of Figure 10 illustrates the similarity distribution of the three past tasks with the most similar test specifications in our larger sample, which we used for answering *RQ4*.

Finally, Table 7 sums up how the noise caused by TAITI affects our samples, according to the logical stages described in Sec. 3.2. "Ambiguous steps" is the number of steps that match with more than one step definition, which possibly inflates *TestI* content, as explained in Sec. 3.2.1. "Undeclared called methods" is the number of called methods by tests for which we did not find a compatible method declaration in the project, as explained in Sec. 3.2.2. They are auto-generated methods by Rails and library methods. Given we are interested in preventing conflicts occurrence in code produced by the development team, these methods are not relevant for computing *TestI*.

The next four problems might prevent the inclusion of relevant files into *TestI*. "Error to generate route" is the number of routes that we did not correctly generate for the project while computing *TestI* for a given task. In practice, an incorrect route only affects *TestI* if it is (direct or indirectly) used by the tests. Then, to better quantify its consequence, we evaluated *RQ2*. "Unknown called Rails path methods" is the number of Rails auxiliary auto-generated methods used by tests that we did not translate to a route. Sec. 3.2.3 explains the referenced routing mechanism and the relation with *TestI*. Complementarily, "Inexistent accessed views" is the number of views directly accessed by tests that did not exist in the project. This kind of error might represent two situations: the test is out of date, or we did not correctly extract the view path, which might happen when such information depends on dynamic data. "Error to analyze views" is the number of views we

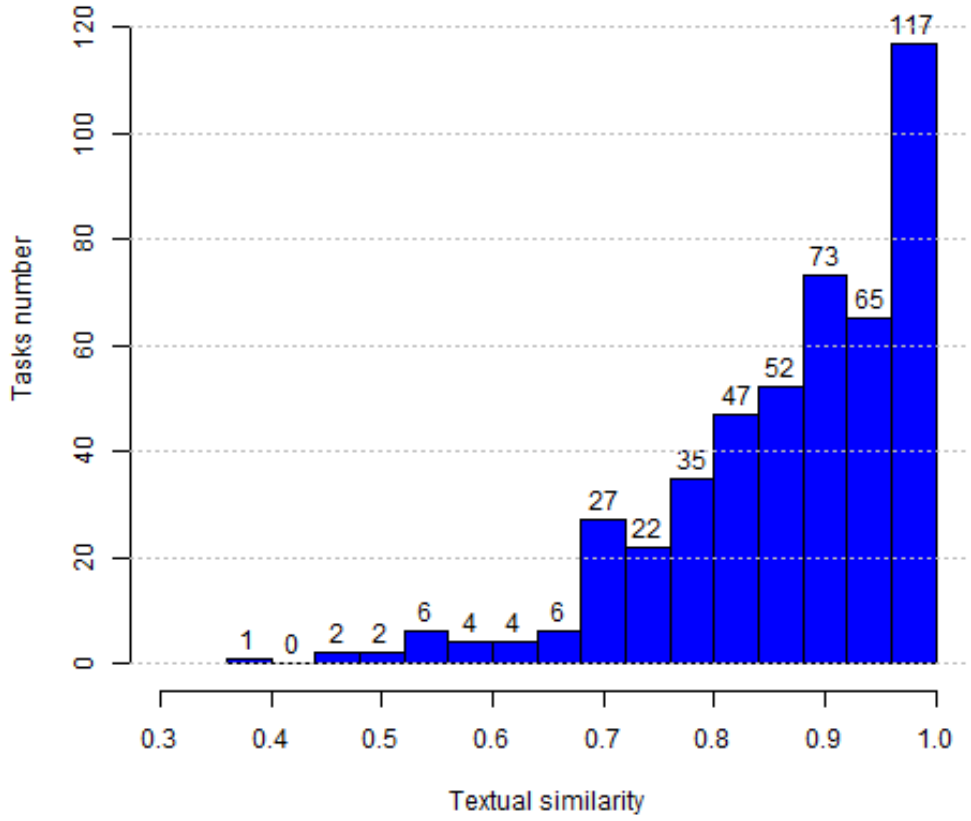


Figure 10 – Distribution of textual similarity related to the larger sample.

could not parse, as described in Sec. 3.2.4. The tests of our sample did not use a wildcard as step type, causing no noise related to WF filter (Sec. 3.2.5.1).

Table 7 – Noise caused by TAITI. Noise means the mistakes or limitations that might affect *TestI*. The percentages refer to the proportion of affected tasks per sample.

Noise	Smaller sample	Larger sample
Ambiguous steps	18 tasks (24.32%), 2 projects $6.56 \pm 4.59(6.00)$ steps	177 tasks (38.23%), 3 projects $11.01 \pm 25.88(5.00)$ steps
Undeclared called methods	74 tasks (100%) $31.16 \pm 17.65(29.50)$ methods	463 tasks (100%) $26.15 \pm 16.61(22.00)$ methods
Error to generate route	61 tasks (82.43%) $10.80 \pm 9.68(6.00)$ routes	375 tasks (81%) $12.80 \pm 10.66(10.00)$ routes
Unknown called Rails path methods	55 tasks (74.32%) $5.04 \pm 3.07(4.00)$ methods	295 tasks (63.71%) $5.18 \pm 4.58(4.00)$ methods
Inexistent accessed views	29 tasks (39.19%) $6.41 \pm 6.11(4.00)$ views	216 tasks (46.65%) $14.46 \pm 36.49(3.00)$ views
Error to analyze views	20 tasks (27%) $3.60 \pm 2.30(3.00)$ views	50 tasks (10.80%) $2.90 \pm 1.52(3.00)$ views

3.5 RESULTS AND DISCUSSION

In this section, we present and discuss the results of our empirical study and answer the research questions. Given that our data is paired and deviates from normality, we analyze differences in precision and recall measures with the paired Wilcoxon Signed-Rank test (WILCOXON; WILCOX, 1964) adopting $\alpha = 0.05$, and the Cohen's assignment of effect size's relative strength (small = 0.10, medium = 0.30, and large = 0.50). Specifically, we report the p-value obtained after run the paired Wilcoxon test as p and the size effect based on the Cohen test as r.

We answer most questions using both samples, but, for brevity, we focus here on the results of the larger sample. We exclusively use the smaller sample to answer *RQ2* and investigate secondary issues related to *RQ1*. Finally, we exclusively use the larger sample to answer *RQ4*.

As previously explained, the recall and precision measures evaluate whether *TestI* might be used to predict file changes. The reasoning is the higher the predictive power of *TestI*, the higher its potential for supporting developers to avoid conflicts. That is, if the developers have upfront knowledge about the file changes they will do to perform a programming task, they might decide to work on disjoint and independent tasks in parallel, reducing the integration effort and minimizing conflict risk. To better understand the results, we also looked for outliers, manually analyzing code changes and interfaces from some tasks.

3.5.1 RQ1: How often does *TestI* predict file changes associated with a task?

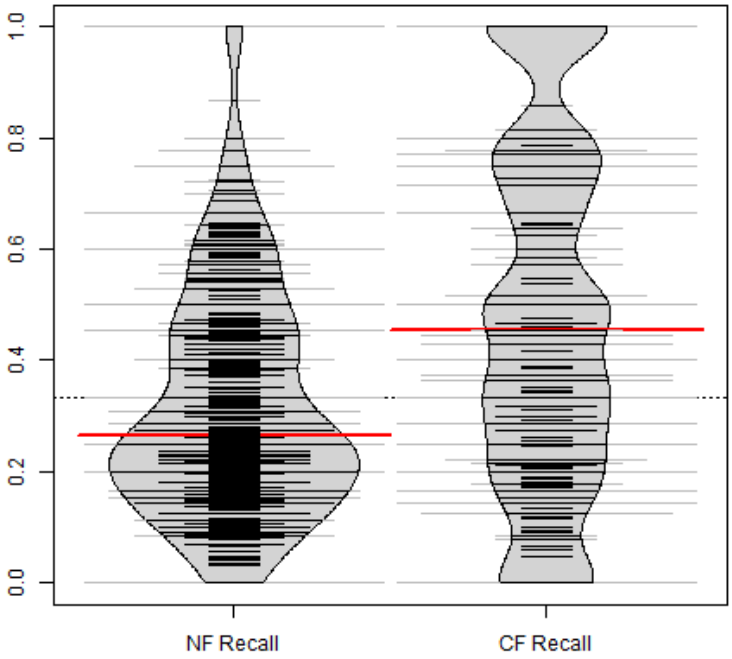
TestI helps to predict changes in controllers and MVC segments, but not for all tasks

Considering the larger sample of 463 tasks, *TestI-CF* recall is $0.48 \pm 0.32(0.45)$ (see Figure 11),[17] contrasting with $0.62 \pm 0.35(0.76)$ in the smaller sample. The results for the other analyzed *TestI* configurations are inferior. The median results show that *TestI-CF* performs well for at least half of the tasks in the analyzed samples, with better performance for the tasks in the smaller sample. But, given the large variation of the recall measures among the analyzed tasks, the results also show that *TestI-CF* performs poorly for a number of tasks. On average, *TestI-CF* can predict nearly half of the changes in controllers from the larger sample, with better prediction rate for the tasks in the smaller sample.

To better understand these results, and how they are influenced by differences in the tasks and samples, we manually inspected more than 60 tasks. We chose tasks based on their precision and recall measures, prioritizing extreme cases and tasks from the smaller sample, for which we had complementary information such as test coverage report. We consistently observed low recall measures for tasks with tests that little exercise the

[17] Beanplots appear in the online appendix (ROCHA; BORBA; SANTOS, 2018) for this and other results.

Figure 11 – Beanplots describing the recall value of *TestI-NF* and *TestI-CF* per task from the larger sample.

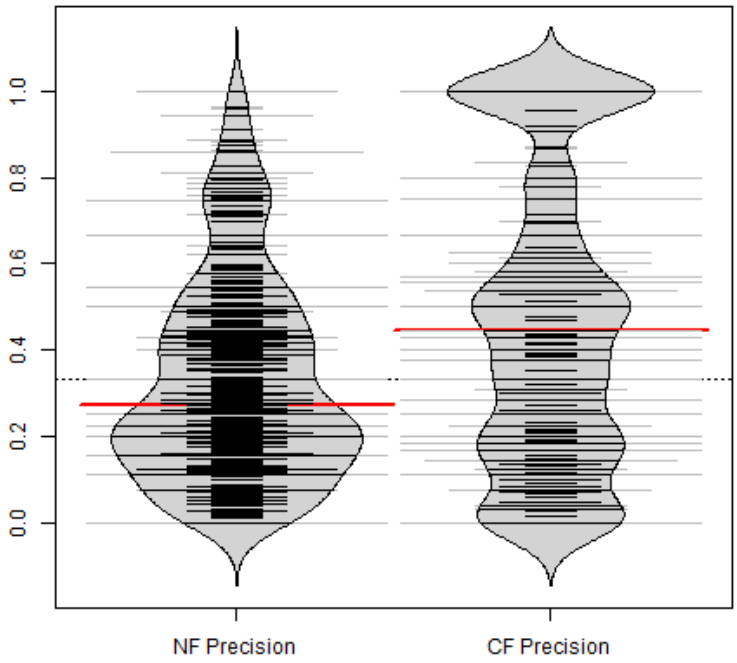


implemented functionality. This might have happened for a number of reasons: developers have not strictly adopted a BDD process and tests were not jointly integrated to the repository with the functionality they test; developers have weak testing expertise and superficially tested the task functionality; the task functionality has low correctness priority, not demanding much testing; the task is not cohesive in the sense that, besides adding functionality and the associated tests, it involves changes— such as refactorings and the introduction of gems auto-generated code at installation time— without associated tests. Besides that, we observed tasks with reduced recall measures due to current limitations of our tool, not of the test-based task interface idea. For example, TAITI does not explore structural relationships among classes, nor method call dependencies, decreasing recall. Similarly, Rails defines implicit relationships among files (especially views) that are not explored by our tool and explains some code changes that are not predicted by *TestI*.

Provided BDD and testing practices are seriously adopted by a team, these results suggest that *TestI-CF* might help to predict changes in controllers. As controllers uniquely identify an MVC segment (related model, view, controller, and auxiliary files), one could also use *TestI-CF* to predict changes in segments. This is reinforced by the observation that, for 97% of the tasks in our sample, controllers predict changes in corresponding segment files; that is, when a controller appears in *TestI-CF*, *TaskI* quite often contains at least one file from the associated segment. So, in principle, *TestI-CF* could be used to reduce conflicts by avoiding the parallel execution of tasks that focus on common segments. Nevertheless, one should anyway expect *TestI-CF* to have reduced predictive power for tasks that are either non cohesive or are superficially tested, as discussed before.

These conclusions are confirmed by the precision measures we obtain, as Figure 12 illustrates. Considering the larger sample, *TestI-CF* precision is $0.47 \pm 0.34(0.44)$. Similarly to what was observed for recall, most of the results for the other analyzed *TestI* configurations are inferior or very close to it.

Figure 12 – Beanplots describing the precision value of *TestI-NF* and *TestI-CF* per task from the larger sample.



Under the segment perspective, we observe that, on average, 68% ($0.68 \pm 0.32(0.75)$) of the segments inferred from the controllers in *TestI-CF* are actually changed by developers responsible for the corresponding task. These are promising results, especially considering the conservative nature of the static analysis we apply, and that the adverse impact of false positives in this context is often low: they simply discourage parallel execution of tasks that would not conflict, or encourage slightly more unneeded coordination. Besides that, we have observed that some of the files in *TestI-CF* that were not changed by the task were actually relevant to the task, and related to changed files, but have nevertheless helped to decrease the precision rates.

Under the project perspective, as Figure 13 illustrates, most projects have low recall for *TestI-NF*. So, we further investigate the project with the better result for such a *TestI* variation (*tip4commit/tip4commit*). We observed that, differently from others, it has only one task with 203 commits, 40 changed files, and 68 tests; i.e., an expressive test amount, given the average number of tests per task of the larger sample is $19.81 \pm 30.83(9.00)$. The static analysis of these tests reached 29 files, and the developers responsible for the task changed 28 of them.

Although *TestI-CF* recall is better than *TestI-NF*, we also try to understand the reason some projects still have low recall values for such interface variation. Thus, we investigate in

more depth the project (with more than one task) with the lower recall value for *TestI-CF* (*TracksApp/tracks*). We observe that the project has 4 tasks, 35.5 commits and 3.5 tests per task on average. Besides the limited test number, we observed that the tasks changed 17 files on average, and about 28% of them are controllers. The CF filter reduces *TestI* and *TaskI* content by 78% and 67%, respectively. Also, 23% of the *TestI* content corresponds to controllers. Even so, the CF filter slightly reduced the precision and recall values of the project, because the developers most changed view files and the files reached by our test analysis do not explicitly reference such views. A poor view analysis also affects the CF result, given the implicit relationship among views and controllers. All these findings reinforce the conclusions previously reported according to the task perspective.

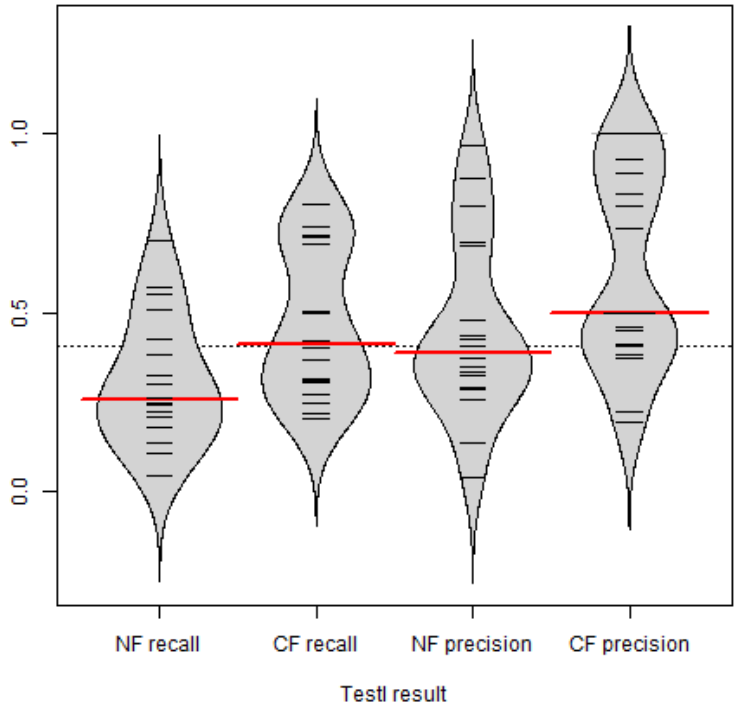
Concerning precision, two projects consistently have lower values for both *TestI-NF* and *TestI-CF*: *otwcode/otwarchive* (26 tasks) and *alphagov/whitehall* (175 tasks). We verified that although the high number of tests per task (24 on average), some tests from the project *otwcode/otwarchive* seem to be out of date, as they directly referenced invalid views. Furthermore, there is some coincidence of identifiers among the project methods and library methods. For instance, the test calls a method `index`, and all controllers declare a compatible method (it is a Rails convention). In the context of our most conservative analysis strategy, both facts contribute for increasing the inclusion of unnecessary files in *TestI*. Similar problems to the previous reported affect the project *alphagov/whitehall* as well. For instance, some tests call the Rails method `to_s`, which is overridden by many classes, leading to multiple matches among method calls and declarations. Like the project *tip4commit/tip4commit*, the less cohesive tasks (i.e., with more than 100 commits) present better precision values in case of much noise in *TestI*, because they change a significant amount of files, improving the chances of intersection with the *TestI* content.

Contrasting, the project *jpatel531/folioapp* (6 tasks) consistently has higher precision value for both *TestI* configurations. A possible explanation is its coding style that restricts the *TestI* content. For example, our code analysis precisely identified the referenced views because the tests did not rely on runtime data to specify them. Also, we did not find occurrences of the causes of noise that affect the other projects, such as method overriding and method declarations that are confusing with common library methods.

Our routing mechanism does not strongly compromise the predictive ability of *TestI*

As 32.8% of the tasks in the larger sample have at least one call to method `visit` that we cannot correctly analyze due to the static nature of our tool, considering *TestI-NF* (to avoid misinterpretation of results caused by filters influence), we decided to investigate whether our simplified, static, routing mechanism (see Section 3.2.3) could be causing that. This would indicate that the results from our retrospective study, in which we do not build and execute the analyzed project versions, should actually be improved when using TAITI in practice, when dynamically running the system and accessing the actual

Figure 13 – Beanplots describing the results of *TestI-NF* and *TestI-CF* per project from the larger sample.



routes is possible.

We observed highly similar *TestI* contents when using our routing mechanism and the Rails mechanism: $0.95 \pm 0.10(1)$ according to the Jaccard index, and $0.99 \pm 0.03(1)$ according to cosine similarity. This suggests the simplified routing mechanism has a small impact in the overall results. To confirm that, we compare mean values of precision and recall for both mechanisms. We observe a significant difference in recall values. For the smaller sample, our routing mechanism slightly decreases the average recall: with Rails routes, *TestI-CF* recall is $0.66 \pm 0.33(0.77)$, contrasting with $0.62 \pm 0.35(0.76)$, with $p = 0.014$ and $r = 0.28$, when using our simplified routing mechanism. For precision, we do not observe statistically significant differences between mean values: $p = 0.60$ for *TestI-CF*, and $p = 0.34$ for *TestI-NF*.

Although recall is more relevant than precision in our context, when balancing the observed high similarity and the low rates related to recall reduction, we conclude our routing mechanism does not strongly compromise the predictive ability of *TestI*.

TestI has higher predictive power for tasks with higher test coverage

As discussed at the beginning of the section, tasks with superficially tested functionality might have reduced predictive power. This follows from the assumption that a task with a weak test suite likely does not throughly exercise the code contributed by the task. As a consequence, for such a task, we have poor alignment between *TestI* and *TaskI*, decreasing precision and recall.

To study that, we first measure the test coverage of a given task t as the percentage of the files touched (changed or created) by t that is exercised by running the tests of t:

$$coverage(t) = \frac{|DTestI(t) \cap TaskI(t)|}{|TaskI(t)|} \tag{3.1}$$

On average, the tasks from the smaller sample— the one for which we can compute *DTestI*— have $20.86 \pm 30.42(9.5)$ tests, with the following test coverage: $0.46 \pm 0.28(0.48)$. By applying the controller filter in Definition 3.1, the test coverage results are $0.71 \pm 0.38(0.95)$.

Supporting the initial hypothesis, we find that the observed test coverages are positively correlated with the change predictions according to the Spearman's rank correlation coefficient with $\alpha = 0.05$ and $p < 0.001$. We observe a strong correlation for precision ($\rho = 0.82$ for all files, and $\rho = 0.78$ for controllers) and a moderate correlation for recall ($\rho = 0.47$ for all files, and $\rho = 0.50$ for controllers). This suggests that *TestI* is more effective to predict changes and, as a consequence, to support developers for avoiding conflicts, whenever the tests satisfactorily cover the files associated with the task.

Discarding test precondition and postcondition compromises *TestI* recall with slight improvement in precision

By focusing on `When` steps, and excluding from the interface files exercised only by test precondition (`Given` steps) and postcondition (`Then` steps), significantly reduces recall. *TestI-NF* recall is 27.90% higher than *TestI-WF* recall: the first is $0.32 \pm 0.21(0.27)$ whereas the second is $0.25 \pm 0.19(0.20)$ ($p < 0.001$, $r = 0.75$). Contrasting, the average precision increases by 7.02%, with $0.36 \pm 0.26(0.31)$ for *TestI-WF* and $0.34 \pm 0.24(0.26)$ for *TestI-NF* ($p < 0.001$, $r = 0.20$). Similar results apply for the smaller sample, but with no significant difference in precision and a slightly smaller decrease rate in recall.

By analysing concrete scenarios, we observe that the focus on `When` steps increases the chances of reducing false positives because code in `Given` and `Then` is not always related to the task functionality. For example, many tests we analyzed have authentication related `Given` steps that have no direct relation with the functionality of interest to the test. Files inferred from these steps could be safely removed from interfaces. On the other hand, we have also observed `Given` and `Then` steps exercising functionality closely related to the task under study. In these cases, considering only `When` steps in the analysis might makes us miss true positives. We have no guarantees, though. For instance, in our motivating example (see Section 3.1), we have a *Given* step that configures crucial information to the feature under testing, but none of the related classes were changed by the task.

Given the involved uncertainty, and the greater importance of recall in our context, one should opt for interfaces that consider all test steps. Nevertheless, our observations suggest that an alternative strategy to refine *TestI* could weight differently files inferred from different kinds of steps. For example, files inferred from *When* steps should weight more than files inferred from *Then* steps, which should weight more than files inferred from

Given steps. Interfaces would then be sets of weighted files, with alternative comparison criteria. This is, however, left as future work.

Discarding changed tests compromises *TestI* recall with slight improvement in precision

As explained before, to compute test interfaces for a task t we analyze the tests associated with t. In principle, and in our study so far, this corresponds to the tests *created* or *changed* with the aim of validating the task results. However we could simply feed TAITI with the tests created for the task, as this could be more strongly related to the task. So we explored this possibility to assess the effect it might have. Considering only the created tests, we improve average precision in 9% in relation to *TestI-NF* precision, obtaining $0.37 \pm 0.26(0.32)$ ($p < 0.001$, $r = 0.36$). However, the average recall is reduced by 13.13%, obtaining $0.29 \pm 0.21(0.22)$ ($p < 0.001$, $r = 0.54$). Similar results apply for the smaller sample, but with slightly smaller increase rate for precision and decrease rate for recall. Considering these results and our analysis, and given that our context favours recall, one should opt for test interfaces that consider both created and changed tests associated with a task.

TestI performs better as a predictor of changes in controllers and MVC segments

As briefly mentioned at the beginning of the section, *TestI-CF* presents the best recall results among all analyzed *TestI* configurations, in both samples. To conclude that, we consider eight valid configurations for *TestI*, derived from the usage of the four filters presented in Sec. 3.2 under two different conditions— when considering all tests related to the tasks, and when considering only created tests. Differently from the previous results, where we compare just the absence and the presence of a filter (i.e., two measures), this time we examine a number of precision and recall measures. Considering the eight configurations, we evaluate a final set of 28 configuration pairs. We use the paired Wilcoxon Signed-Rank test combined with the Bonferroni correction method. In the following, consider that CTCF (created-tests-controller-filter) represents an analysis that considers only created tests, filtering *TestI* content by controllers, and CTWCF (created-tests-when-controller-filter) is an extension of CTCF that also filters *TestI* content by step type.

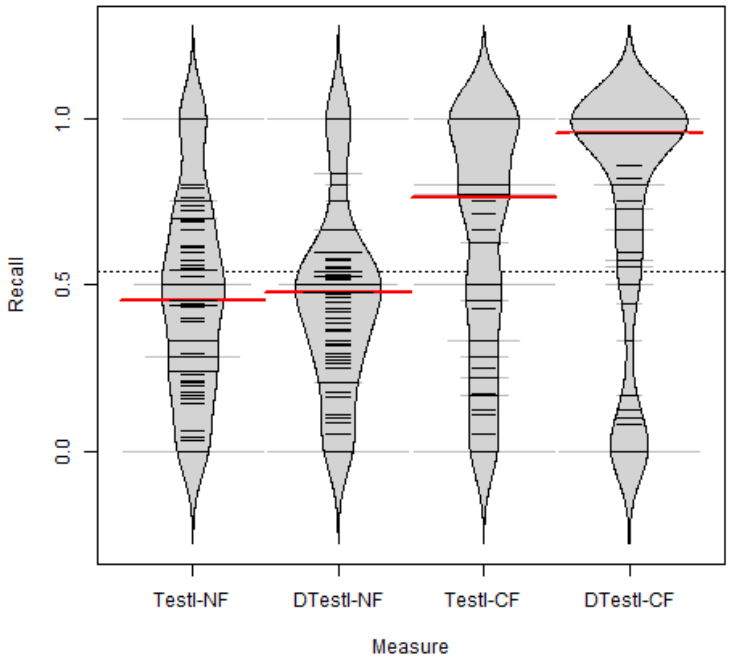
In the larger sample, we observe that *TestI-CF* has the best overall result. It has the best average recall with a significant difference from the other filters ($p < 0.05$). Regarding precision, CTCF, WCF, and CTWCF have the highest mean values, with no significant difference among them. The detailed result appears in the Appendix (ROCHA; BORBA; SANTOS, 2018), including effect size measure. Similar results apply for the smaller sample, but CTCF and CF ($p = 0.11$) have the highest mean precision. Given we prioritize recall over precision, one should opt for the CF configuration. Furthermore, both filters with the best precision are just CF combined with other filters that favour precision.

3.5.2 RQ2: Is static code analysis suitable to compute *TestI*?

When passing tests are available, *DTestI* is a better predictor of changes in controllers than *TestI*

Test-based task interfaces computed by using test coverage reports (*DTestI*), not step definitions, increase the average recall in 14.55% when dealing with controllers. As Figure 14 illustrates, for the smaller sample, *DTestI-CF* recall is $0.71 \pm 0.38(0.95)$ with a significant difference from *TestI-CF* ($0.62 \pm 0.35(0.76)$), with $p < 0.01$ and $r = 0.32$. Contrasting, we observe no significant difference in recall when comparing *DTestI-NF* with *TestI-NF*. Similarly, we observe no significant difference in precision values when comparing *DTestI* with *TestI*.

Figure 14 – Beanplots describing the recall value of *DTestI* per task from the smaller sample.



At first thought, the precision findings might be surprising. Our static analysis tries to infer the files a test might execute, while test coverage information (and *DTestI*) accurately detects the executed files. So one could initially expect a reduction of false positives by considering test execution. However, although all files detected by *DTestI* are executed by the tests, not all of them are related to the task (as explained before) or changed by developers, affecting precision measures. So, while *DTestI* might increase precision by avoiding typically conservative choices of a static analysis, it might also decrease precision by going deeper (beyond controller code) and capturing a larger number of files that are not changed by the tasks. Given that, and the recall results, in our small sample, *DTestI* has more chances to provide better predictions than the static analysis based interfaces we discuss here.

Static analysis is suitable to compute *TestI*

In spite of *DTestI* advantage, due to the small observed recall improvement, and the fact that *DTestI* cannot often be computed in the BDD context we assume (one cannot generate test coverage report for failing or empty tests), we consider static code analysis as a suitable option to compute *TestI*.

Our observations also suggest that one could maybe adopt a hybrid (static and dynamic) approach for dealing with tasks that already have tests and the corresponding implemented functionality. For instance, suppose a bug fix task that is associated with 6 tests, but only 2 of them are failing because the associated functionality has not been fixed yet. In such situation, we could compute *TestI* for the 2 failing tests and *DTestI* for the other 4 tests. Moreover, the evidence that our routing mechanism based on static code analysis does not compromise *TestI* improves its prognostics.

3.5.3 RQ3: Is *TestI* a better code change predictor than *RandomI*?

TestI outperforms *RandomI* for predicting changes in controllers and MVC segments

We found no statistically significant difference between *TestI-CF* and *RandomI-CF* mean recalls for the larger sample ($p = 0.10$) (see Figure 15). *TestI-CF* mean precision, however, is 69.7% higher than *RandomI-CF* precision (see Figure 16): the first is $0.47 \pm 0.34(0.44)$, whereas the second is $0.28 \pm 0.27(0.17)$ ($p < 0.001$, $r = 0.72$).

Figure 15 – Beanplots describing the recall value of *RandomI* per task from the larger sample.

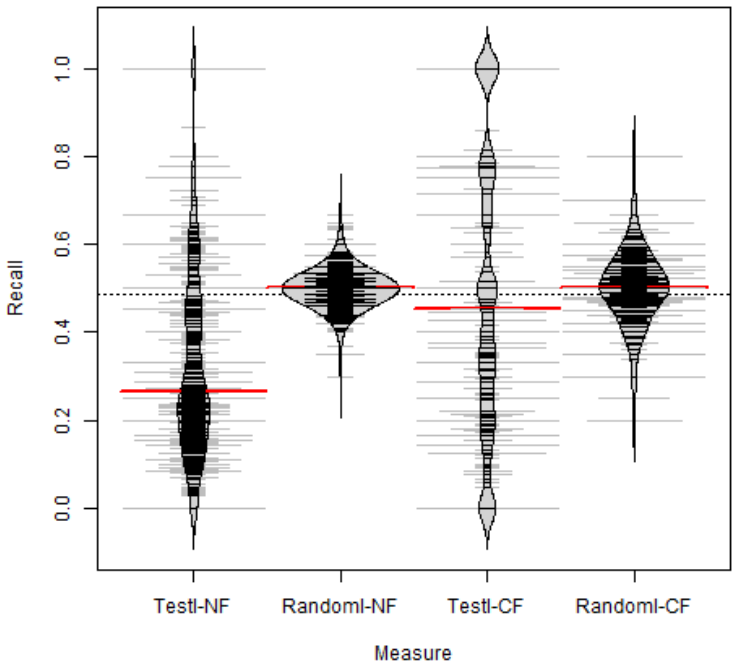
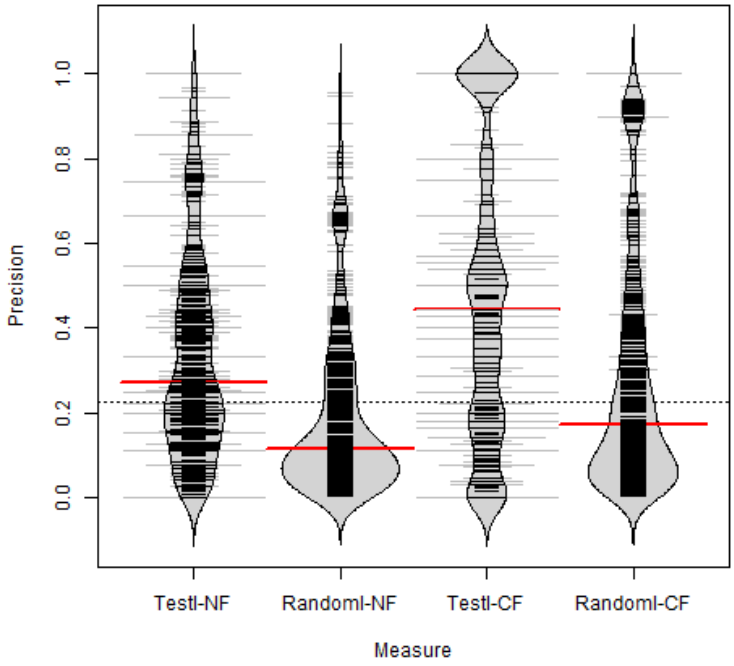


Figure 16 – Beanplots describing the precision value of *RandomI* per task from the larger sample.



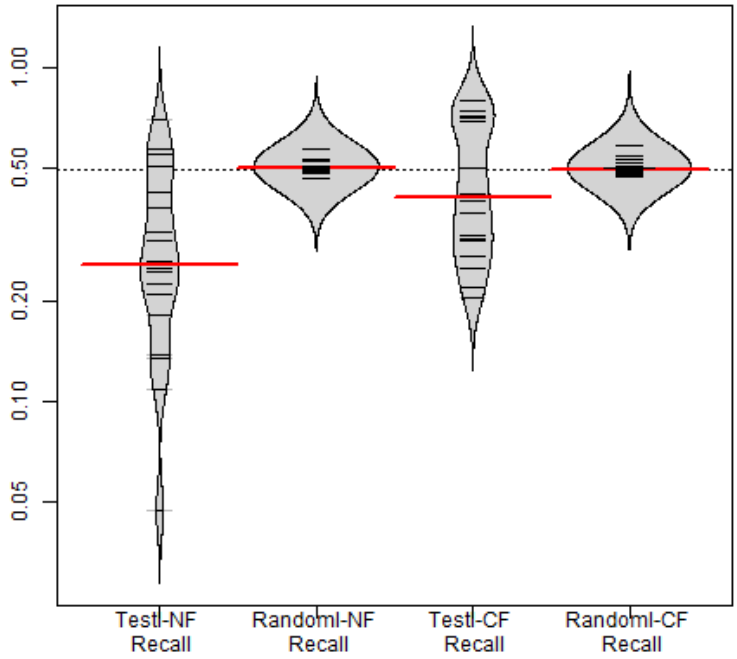
For the smaller sample, we observe similar precision results (increase of 41.1% in favour of *TestI-CF*), but we also observe a statistically significant mean recall difference in favour of *TestI-CF* (27.8% higher than *RandomI-CF* recall).

These results reinforce our previous conclusion that *TestI* might help to predict changes in controllers and MVC segments. Given the superiority in precision in the two samples, and no evidence of a loss in recall (including superiority in one of the samples), we consider that *TestI-CF* outperforms *RandomI-CF*.

Considering all kinds of files, we have contrasting recall results. They favour *RandomI-NF* (54.1% higher than *TestI-NF*, with $p < 0.001$ and $r = 0.64$) in the larger sample, but report no statistically significant difference in the smaller sample ($p = 0.33$). The precision results are similar to when considering only controllers. *TestI-NF* mean precision is 70.9% higher than *RandomI-NF* precision ($p < 0.001$, $r = 0.84$). A similar result applies for the smaller sample, but with a smaller increase rate (45.6%).

Contrasting from the other interfaces discussed so far, we computed *RandomI* as a mean of multiple results per task. So, to better understand the results, we further investigate them from a project perspective. As discussed in the literature, random data usually leads to intermediate results. In the context of task interfaces, it means that half of the candidate files of a project are part of *RandomI*, which explains the high size of such interface: on average, *RandomI-NF* has $238.46 \pm 184.47(185.30)$ files. The size of *RandomI* varies per project according to the overall project size, but the proportion of selected files does not. As a consequence, the recall measure is intermediate (around 0.50) for all projects, as Figure 17 illustrates.

Figure 17 – Beanplots describing the recall value of *TestI* and *RandomI* per project from the larger sample.

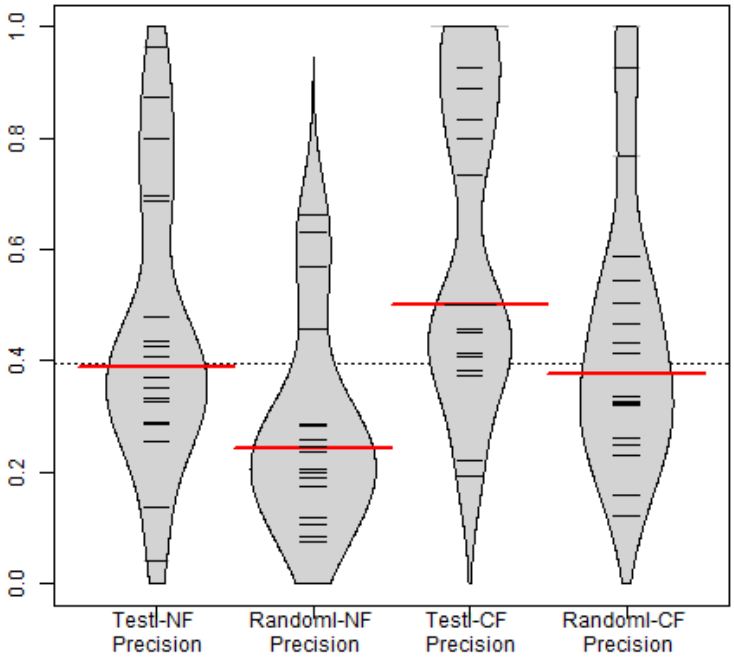


As expected, we can observe a similar effect over *RandomI-CF*, given the relationship between *RandomI-NF* and *RandomI-CF*: *RandomI-CF* receives as input the files set of *RandomI-NF* and applies the CF filter. This way, around half of the controllers of a project is part of *RandomI-CF*, resulting in smaller interfaces: On average, *RandomI-CF* has $28.11 \pm 22.73(18.90)$ files.

In turn, *TestI* depends on a set of variables, as previously discussed: the coding style of a project, the nature of the tasks (e.g., some tasks require changes on controllers, but others do not), the test coverage, the limitations of our tool and the static analysis, and the cohesion of code changes related to tasks. Therefore, by indiscriminately makes suggestions, *RandomI-NF* expands its chance of make a right guess. Even so, when considering *RandomI-CF* such a chance is reduced.

Concerning the precision result, summarized by Figure 18 from a project perspective, we observed a negative relationship among the task cohesion and the performance of *RandomI*: It seems that less cohesive tasks have higher *RandomI* precision values. For example, the project *tip4commit/tip4commit* has the higher precision value related to *RandomI-NF*: 0.66. This project has only 1 task that has 203 commits and 40 changed files, whereas the size of its *RandomI-NF* is around 32 files. Luckily almost half of the changed files were included into *RandomI-NF*. Similarly, the project *oneclickorgs/one-click-orgs*, which has the third better result of *RandomI-NF* precision (the second is another project with only 1 task), has 28 tasks and 4 of them has 109, 111, 392 e 469 commits, respectively, and 105, 103, 143, and 314 changed files. Considering that the lack of cohesion is an exception instead of a rule, the *RandomI* precision is not a surprise.

Figure 18 – Beanplots describing the precision value of *TestI* and *RandomI* per project from the larger sample.



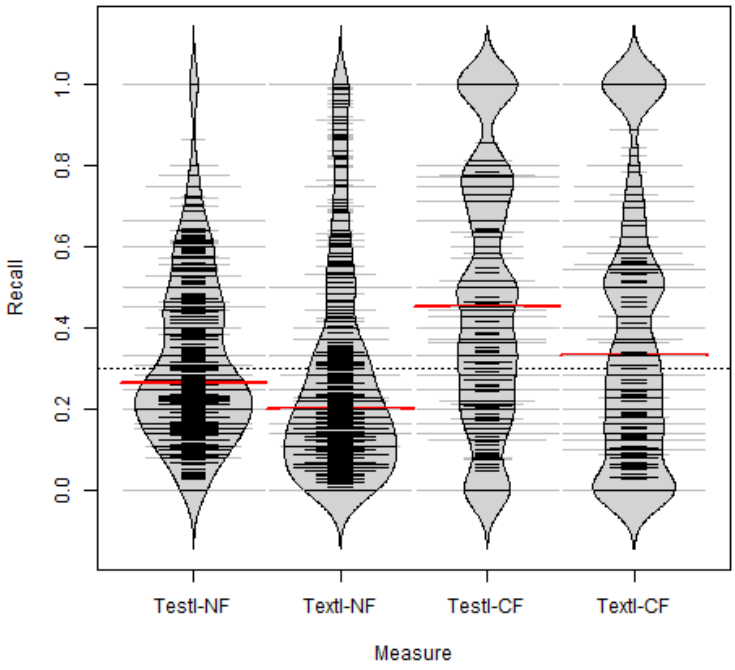
3.5.4 RQ4: Is *TestI* a better code change predictor than *TextI*?

TestI has better recall but inferior precision than *TextI*

As Figure 19 illustrates, *TestI-CF* has 20.1% higher mean recall than *TextI-CF*: the first is $0.48 \pm 0.32(0.45)$, whereas the second is $0.40 \pm 0.34(0.33)$ ($p < 0.001$, $r = 0.23$). Contrasting, *TextI-CF* precision is 32.7% higher than *TestI-CF* precision (see Figure 20): the first is $0.63 \pm 0.42(1.00)$ and the second is $0.47 \pm 0.34(0.43)$ ($p < 0.001$ and $r = 0.36$). Similar results apply if we consider all files (*TestI-NF* versus *TextI-NF*).

A more in-depth investigation of the results reveals that it is not surprising that *TextI* precision outperforms *TestI*. The reason is *TextI* reduces the number of false positives by 50.5% for NF and 50.4% for CF, whereas it slightly decreases the number of false negatives by 9.2% for NF and increases the number of false negatives by 8% for CF. In practice, it means that the overall idea of investigating the past for predicting the future makes sense when dealing with known code changes: If task A changed file X, we can guess task B will change file X too because tasks A and B are similar. However, subtle differences among tasks, such as performance requirements, for instance, might motivate new code changes that a past-based analysis like the one we implemented cannot deduce. As a consequence, it increases false negatives and reduces recall. For anticipating more code changes, e.g., to guess that file Y changes when file X changes (the notion of change propagation), maybe a past-based analysis should also verify the similarity and dependencies among past code changes besides the similarity among their underlying tasks. Even so, someone would apply the current version of *TextI* for assisting developers to define task interfaces manually,

Figure 19 – Beanplots describing the recall value of *TextI* per task from the larger sample.



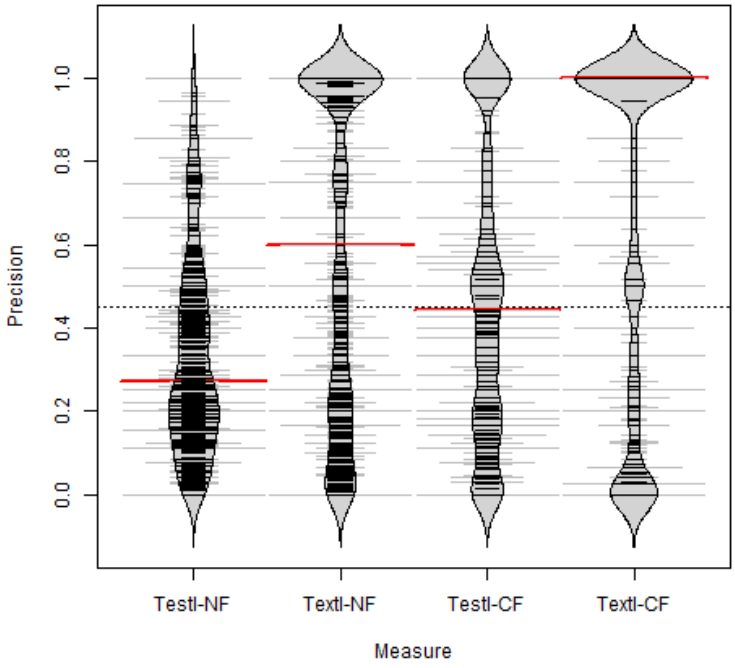
according to a "learning by example" dynamics. That is, by analyzing the artifacts of a similar past task, the developer might gain insight into the code of her new task. However, our vision is that an automatic solution for computing task interfaces is preferable.

We also speculate that the test coverage affects *TextI* recall as well, given that if the tests cover only a subset of the task code, the similarity among tasks might be compromised, increasing the rate of false negatives in *TextI*. Unfortunately, as we do not have data about the test coverage for our larger sample, we cannot evaluate the correlation among tests coverage and *TextI* recall.

Finally, as Figure 10 illustrates, the similarity among the most similar past tasks is predominantly high, so we cannot deduce anything when confronting such property with the results, even when we analyze it from a project perspective; the average values per project vary in [0.71, 1.0]. To understand the cause of such a phenomenon, we manually inspect some tasks. At the first moment, we imagine that there are many intersections between the test set, leading us to question why different tasks would be validated by the same tests. We observed that sometimes a task creates a new functionality and other task fixes it; in this cases the values of precision and recall reached by *TextI* are high. However, given that the teams intuitively avoid such an obvious dependency among tasks, the potential of conflict avoidance is minimized.

Other times a task integrates code from other tasks, i.e., the task did not project all tests from its test set; a limitation of our strategy for delimiting tasks based on merge commits that we mentioned in Section 3.6.2. But we also observed that sometimes there is no intersection between the test set and the similarity is still very high. This time, it is not trivial to evaluate whether the tests have similar meanings (i.e., they are clones)

Figure 20 – Beanplots describing the precision value of *TextI* per task from the larger sample.



or our strategy for evaluating similarity among tests is fragile. As known, most projects adopt a pattern to write Cucumber tests that rely on access to views and the verification of its content. Although we preprocessed the Gherkin files, such a pattern might induce a "false similarity". Furthermore, steps are widely reused accross scenarios, so it is expected that well written test suites will exhibit high degrees of textual similarity (BINAMUNGU; EMBURY; KONSTANTINOU, 2018). As a consequence, the predictions provided by *TextI* do not apply.

Based on the exposed, we conclude that given our context values more recall than precision, one should opt for *TestI*. However, the results suggest an hybrid solution involving *TestI* and *TextI* might be a promising predictor. For each file in *TestI*, if *TextI* includes it too, it has less chances of being a false positive. Such perception is confirmed by the low intersection rate between both interfaces. On average, 21% of the files in *TestI* intersects with the files in *TextI* ($0.21 \pm 0.21(0.13)$). Considering only controllers, the rate increases to 32% ($0.32 \pm 0.31(0.25)$).

In order to complement the overall discussion of our results and to provide a comparative overview, Table 8 and Table 9 summarize the values of precision and recall of the larger sample for *TestI*, *RandomI*, and *TextI*, considering the main variations (NF and CF), under a project perspective. Similarly, Table 10 and Table 11 summarize the results of the smaller sample, this time also including *DTestI*. The highlights are the best results for each project.

Table 8 – Average precision per project of the larger sample.

Repository	*TestI-NF*	*TestI-CF*	*RandomI-NF*	*RandomI-CF*	*TextI-NF*	*TextI-CF*
hackful	0.29 ± 0.27(0.33)	0.50 ± 0.50(0.50)	0.29 ± 0.23(0.32)	0.55 ± 0.33(0.42)	0.70 ± 0.51(1.00)	0.89 ± 0.19(1.00)
MetPlus_PETS	0.29 ± 0.00(0.29)	0.50 ± 0.00(0.50)	0.26 ± 0.00(0.26)	0.43 ± 0.00(0.43)	1.00 ± 0.00(1.00)	1.00 ± 0.00(1.00)
WebsiteOne	0.33 ± 0.28(0.21)	0.45 ± 0.29(0.43)	0.20 ± 0.24(0.09)	0.34 ± 0.27(0.25)	0.61 ± 0.37(0.67)	0.71 ± 0.39(1.00)
whitehall	0.26 ± 0.16(0.23)	0.37 ± 0.28(0.33)	0.11 ± 0.10(0.09)	0.12 ± 0.13(0.08)	0.48 ± 0.38(0.42)	0.51 ± 0.43(0.50)
blacklight-cornell	0.41 ± 0.25(0.33)	0.93 ± 0.23(1.00)	0.21 ± 0.17(0.14)	0.41 ± 0.23(0.39)	0.71 ± 0.28(0.75)	0.82 ± 0.28(1.00)
diaspora	0.33 ± 0.22(0.36)	0.41 ± 0.30(0.38)	0.24 ± 0.19(0.19)	0.33 ± 0.28(0.24)	0.58 ± 0.41(0.75)	0.62 ± 0.43(1.00)
action-center-platform	0.44 ± 0.21(0.55)	0.73 ± 0.37(1.00)	0.17 ± 0.12(0.14)	0.26 ± 0.16(0.22)	0.61 ± 0.45(0.88)	0.69 ± 0.43(1.00)
CBA	0.37 ± 0.33(0.20)	0.41 ± 0.41(0.50)	0.19 ± 0.22(0.06)	0.25 ± 0.20(0.16)	0.27 ± 0.29(0.20)	0.39 ± 0.44(0.20)
folioapp	0.80 ± 0.05(0.81)	1.00 ± 0.00(1.00)	0.46 ± 0.10(0.51)	0.51 ± 0.18(0.56)	0.67 ± 0.30(0.52)	0.60 ± 0.37(0.40)
one-click-orgs	0.70 ± 0.17(0.75)	0.89 ± 0.23(1.00)	0.57 ± 0.23(0.66)	0.77 ± 0.30(0.91)	0.89 ± 0.24(1.00)	0.94 ± 0.18(1.00)
opengovernment	0.04 ± 0.06(0.04)	0.22 ± 0.31(0.22)	0.08 ± 0.04(0.08)	0.23 ± 0.03(0.23)	0.50 ± 0.71(0.50)	0.50 ± 0.71(0.50)
otwarchive	0.14 ± 0.15(0.07)	0.19 ± 0.25(0.06)	0.08 ± 0.11(0.02)	0.16 ± 0.25(0.04)	0.41 ± 0.45(0.14)	0.44 ± 0.44(0.30)
moumentei	0.88 ± 0.00(0.88)	1.00 ± 0.00(1.00)	0.63 ± 0.00(0.63)	1.00 ± 0.00(1.00)	0.99 ± 0.00(0.99)	1.00 ± 0.00(1.00)
RapidFTR	0.35 ± 0.19(0.38)	0.38 ± 0.23(0.44)	0.28 ± 0.17(0.29)	0.33 ± 0.22(0.31)	0.71 ± 0.34(0.90)	0.70 ± 0.40(1.00)
time_stack	0.69 ± 0.30(0.76)	0.80 ± 0.27(1.00)	0.26 ± 0.13(0.30)	0.47 ± 0.29(0.43)	0.85 ± 0.34(1.00)	1.00 ± 0.00(1.00)
theodinproject	0.43 ± 0.13(0.35)	1.00 ± 0.00(1.00)	0.25 ± 0.02(0.25)	0.59 ± 0.15(0.67)	0.75 ± 0.43(1.00)	1.00 ± 0.00(1.00)
tip4commit	0.97 ± 0.00(0.97)	0.83 ± 0.00(0.83)	0.66 ± 0.00(0.66)	0.93 ± 0.00(0.93)	0.81 ± 0.00(0.81)	1.00 ± 0.00(1.00)
tracks	0.48 ± 0.23(0.42)	0.46 ± 0.42(0.42)	0.12 ± 0.06(0.10)	0.32 ± 0.13(0.30)	0.52 ± 0.34(0.41)	0.66 ± 0.23(0.58)

Table 9 – Average recall per project of the larger sample.

Repository	*TestI-NF*	*TestI-CF*	*RandomI-NF*	*RandomI-CF*	*TextI-NF*	*TextI-CF*
hackful	0.14 ± 0.13(0.17)	0.25 ± 0.22(0.33)	0.57 ± 0.07(0.56)	0.59 ± 0.05(0.57)	0.59 ± 0.54(1.00)	0.72 ± 0.48(1.00)
MetPlus_PETS	0.18 ± 0.00(0.18)	0.50 ± 0.00(0.50)	0.51 ± 0.00(0.51)	0.50 ± 0.00(0.50)	1.00 ± 0.00(1.00)	1.00 ± 0.00(1.00)
WebsiteOne	0.55 ± 0.20(0.55)	0.80 ± 0.27(1.00)	0.50 ± 0.05(0.50)	0.51 ± 0.12(0.50)	0.36 ± 0.24(0.33)	0.50 ± 0.33(0.50)
whitehall	0.22 ± 0.13(0.22)	0.31 ± 0.22(0.33)	0.50 ± 0.02(0.50)	0.50 ± 0.07(0.50)	0.19 ± 0.17(0.15)	0.26 ± 0.26(0.19)
blacklight-cornell	0.13 ± 0.07(0.11)	0.42 ± 0.24(0.40)	0.50 ± 0.03(0.50)	0.50 ± 0.08(0.50)	0.31 ± 0.16(0.25)	0.51 ± 0.33(0.50)
diaspora	0.21 ± 0.10(0.20)	0.32 ± 0.26(0.22)	0.50 ± 0.04(0.50)	0.49 ± 0.07(0.50)	0.22 ± 0.27(0.11)	0.32 ± 0.30(0.23)
action-center-platform	0.26 ± 0.14(0.22)	0.30 ± 0.13(0.33)	0.49 ± 0.05(0.50)	0.48 ± 0.09(0.50)	0.26 ± 0.20(0.28)	0.39 ± 0.26(0.50)
CBA	0.32 ± 0.29(0.30)	0.37 ± 0.40(0.43)	0.53 ± 0.05(0.52)	0.52 ± 0.12(0.50)	0.24 ± 0.25(0.14)	0.46 ± 0.46(0.67)
folioapp	0.51 ± 0.14(0.55)	0.74 ± 0.13(0.80)	0.51 ± 0.02(0.51)	0.47 ± 0.05(0.50)	0.62 ± 0.29(0.68)	0.66 ± 0.31(0.50)
one-click-orgs	0.42 ± 0.07(0.45)	0.69 ± 0.17(0.77)	0.50 ± 0.02(0.50)	0.49 ± 0.06(0.50)	0.70 ± 0.37(0.89)	0.80 ± 0.33(1.00)
opengovernment	0.11 ± 0.15(0.11)	0.40 ± 0.57(0.40)	0.49 ± 0.01(0.49)	0.53 ± 0.01(0.53)	0.09 ± 0.13(0.09)	0.25 ± 0.35(0.25)
otwarchive	0.38 ± 0.22(0.39)	0.71 ± 0.33(0.81)	0.48 ± 0.05(0.50)	0.48 ± 0.09(0.49)	0.22 ± 0.18(0.23)	0.39 ± 0.31(0.37)
moumentei	0.05 ± 0.00(0.05)	0.20 ± 0.00(0.20)	0.49 ± 0.00(0.49)	0.48 ± 0.00(0.48)	0.99 ± 0.00(0.99)	1.00 ± 0.00(1.00)
RapidFTR	0.57 ± 0.17(0.59)	0.74 ± 0.19(0.71)	0.50 ± 0.03(0.50)	0.52 ± 0.10(0.50)	0.36 ± 0.21(0.39)	0.47 ± 0.33(0.50)
time_stack	0.30 ± 0.17(0.38)	0.50 ± 0.29(0.40)	0.52 ± 0.03(0.51)	0.49 ± 0.10(0.51)	0.09 ± 0.10(0.04)	0.34 ± 0.37(0.20)
theodinproject	0.24 ± 0.07(0.29)	0.27 ± 0.02(0.29)	0.53 ± 0.04(0.54)	0.55 ± 0.04(0.55)	0.09 ± 0.06(0.05)	0.18 ± 0.06(0.14)
tip4commit	0.70 ± 0.00(0.70)	0.71 ± 0.00(0.71)	0.53 ± 0.00(0.53)	0.50 ± 0.00(0.50)	0.33 ± 0.00(0.33)	0.29 ± 0.00(0.29)
tracks	0.25 ± 0.04(0.25)	0.22 ± 0.18(0.24)	0.47 ± 0.05(0.47)	0.48 ± 0.03(0.48)	0.49 ± 0.44(0.43)	0.77 ± 0.29(0.83)

Table 10 – Average precision per project of the smaller sample.

Repository	*TestI-NF*	*TestI-CF*	*RandomI-NF*	*RandomI-CF*	*TextI-NF*	*TextI-CF*	*DTestI-NF*	*DTestI-CF*
WebsiteOne	0.13 ± 0.13(0.11)	0.25 ± 0.29(0.16)	0.05 ± 0.03(0.04)	0.11 ± 0.07(0.10)	0.45 ± 0.50(0.25)	0.30 ± 0.48(0.00)	0.14 ± 0.16(0.09)	0.17 ± 0.18(0.17)
whitehall	0.19 ± 0.15(0.13)	0.42 ± 0.40(0.35)	0.09 ± 0.10(0.03)	0.21 ± 0.29(0.03)	0.27 ± 0.39(0.09)	0.30 ± 0.44(0.00)	0.13 ± 0.15(0.06)	0.31 ± 0.26(0.29)
diaspora	0.19 ± 0.21(0.11)	0.23 ± 0.23(0.12)	0.12 ± 0.14(0.06)	0.14 ± 0.11(0.15)	0.29 ± 0.42(0.03)	0.35 ± 0.42(0.11)	0.11 ± 0.10(0.09)	0.24 ± 0.18(0.31)
one-click-orgs	0.76 ± 0.16(0.78)	0.88 ± 0.23(1.00)	0.67 ± 0.19(0.72)	0.81 ± 0.27(0.89)	0.88 ± 0.29(1.00)	0.89 ± 0.30(1.00)	0.75 ± 0.25(0.79)	0.83 ± 0.29(0.95)
otwarchive	0.01 ± 0.00(0.01)	0.04 ± 0.02(0.04)	0.00 ± 0.00(0.00)	0.02 ± 0.01(0.01)	0.50 ± 0.53(0.50)	0.50 ± 0.53(0.50)	0.01 ± 0.00(0.01)	0.02 ± 0.01(0.01)
RapidFTR	0.12 ± 0.07(0.11)	0.10 ± 0.06(0.11)	0.10 ± 0.06(0.09)	0.09 ± 0.05(0.09)	0.77 ± 0.36(1.00)	0.15 ± 0.34(0.00)	0.13 ± 0.06(0.14)	0.11 ± 0.05(0.11)
theodinproject	0.26 ± 0.25(0.35)	0.60 ± 0.55(1.00)	0.14 ± 0.12(0.21)	0.40 ± 0.33(0.42)	0.04 ± 0.09(0.00)	0.07 ± 0.15(0.00)	0.44 ± 0.31(0.64)	0.67 ± 0.47(1.00)
tip4commit	0.53 ± 0.29(0.57)	0.54 ± 0.24(0.67)	0.31 ± 0.19(0.33)	0.52 ± 0.28(0.62)	0.88 ± 0.21(1.00)	0.90 ± 0.25(1.00)	0.53 ± 0.27(0.59)	0.61 ± 0.24(0.71)
tracks	0.32 ± 0.31(0.32)	1.00 ± 0.00(1.00)	0.11 ± 0.15(0.11)	0.27 ± 0.29(0.27)	0.51 ± 0.69(0.51)	0.56 ± 0.63(0.56)	0.28 ± 0.33(0.28)	0.75 ± 0.35(0.75)

Table 11 – Average recall per project of the smaller sample.

Repository	*TestI-NF*	*TestI-CF*	*RandomI-NF*	*RandomI-CF*	*TextI-NF*	*TextI-CF*	*DTestI-NF*	*DTestI-CF*
WebsiteOne	$0.60 \pm 0.26(0.59)$	$0.82 \pm 0.34(1.00)$	$0.52 \pm 0.06(0.52)$	$0.51 \pm 0.13(0.53)$	$0.07 \pm 0.11(0.04)$	$0.25 \pm 0.42(0.00)$	$0.18 \pm 0.18(0.18)$	$0.35 \pm 0.42(0.17)$
whitehall	$0.13 \pm 0.11(0.10)$	$0.24 \pm 0.21(0.17)$	$0.49 \pm 0.03(0.49)$	$0.49 \pm 0.10(0.51)$	$0.06 \pm 0.04(0.04)$	$0.04 \pm 0.08(0.00)$	$0.23 \pm 0.11(0.27)$	$0.36 \pm 0.40(0.15)$
diaspora	$0.26 \pm 0.28(0.22)$	$0.32 \pm 0.30(0.28)$	$0.48 \pm 0.04(0.48)$	$0.41 \pm 0.13(0.42)$	$0.18 \pm 0.16(0.14)$	$0.31 \pm 0.34(0.28)$	$0.28 \pm 0.18(0.25)$	$0.51 \pm 0.39(0.58)$
one-click-orgs	$0.43 \pm 0.04(0.44)$	$0.73 \pm 0.15(0.77)$	$0.49 \pm 0.02(0.50)$	$0.50 \pm 0.02(0.50)$	$0.68 \pm 0.36(0.83)$	$0.77 \pm 0.37(1.00)$	$0.57 \pm 0.09(0.54)$	$0.96 \pm 0.05(0.95)$
otwarchive	$0.67 \pm 0.25(0.55)$	$0.91 \pm 0.22(1.00)$	$0.49 \pm 0.13(0.50)$	$0.49 \pm 0.12(0.50)$	$0.38 \pm 0.46(0.13)$	$0.45 \pm 0.50(0.25)$	$0.81 \pm 0.23(0.90)$	$1.00 \pm 0.00(1.00)$
RapidFTR	$0.63 \pm 0.16(0.61)$	$0.80 \pm 0.18(0.80)$	$0.49 \pm 0.07(0.51)$	$0.50 \pm 0.13(0.50)$	$0.20 \pm 0.15(0.16)$	$0.07 \pm 0.16(0.00)$	$0.60 \pm 0.19(0.56)$	$0.94 \pm 0.10(1.00)$
theodinproject	$0.15 \pm 0.14(0.16)$	$0.16 \pm 0.15(0.25)$	$0.46 \pm 0.09(0.50)$	$0.48 \pm 0.11(0.50)$	$0.10 \pm 0.22(0.00)$	$0.20 \pm 0.45(0.00)$	$0.46 \pm 0.36(0.48)$	$0.58 \pm 0.37(0.57)$
tip4commit	$0.80 \pm 0.15(0.76)$	$0.80 \pm 0.17(0.80)$	$0.49 \pm 0.07(0.51)$	$0.49 \pm 0.11(0.50)$	$0.57 \pm 0.36(0.50)$	$0.68 \pm 0.34(0.71)$	$0.52 \pm 0.12(0.48)$	$0.98 \pm 0.05(1.00)$
tracks	$0.59 \pm 0.59(0.59)$	$0.61 \pm 0.55(0.61)$	$0.53 \pm 0.09(0.53)$	$0.55 \pm 0.07(0.55)$	$0.51 \pm 0.69(0.51)$	$0.56 \pm 0.63(0.56)$	$0.66 \pm 0.48(0.66)$	$0.78 \pm 0.31(0.78)$

3.6 THREATS TO VALIDITY

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

3.6.1 Construct validity

Section 3.4.2 describes the assumptions we make about the code contributions we analyze. They might actually not correspond to formal programming tasks defined by a manager or team, and performed accordingly to BDD. Nevertheless, considering the nature of our retrospective analysis, this does not compromise our code change prediction conclusions. They do, however, impair us to better understand the limitations of our tool. Since we do not also have task descriptions, we use Gherkin scenario descriptions as proxies, but these often do not correspond to task descriptions in practice.

3.6.2 Internal validity

As our tool does not consider tests that use Cucumber's alternative (non regex based) syntax to identify step definitions, we might have discarded consistent tasks from our sample because we wrongly consider they have undefined step definitions (that is, partially implemented tests). We might also miss task related tests because we only consider tests created or changed by a code contribution that is interpreted as a task. A previously created and contributed test could well be related to a task, especially in case BDD was not fully adopted by the analyzed project. Similarly, we miss tests having scenarios that were not changed by a contribution that changes these scenarios steps.

Besides missing relevant tests as just described, and consequently deflating test interfaces, we might also inflate task interfaces (*TaskI*) because we consider all changes in a contribution, no matter if some of them are reverted. Again, this leads to results that might artificially downgrade the test interfaces conclusions we achieve here. Similarly, task interfaces, and test interfaces to a lesser extent, might be inflated by tangled (HERZIG; ZELLER, 2013; DIAS et al., 2015) contributions that include non task related changes. However, since tangled commits and contributions are not rare, this does not make our results less realistic.

Due to the previously discussed (see Section 3.2) design decisions and limitations of our tool, interfaces might include files that should not be included, and miss files that should be included. This is reflected in our analysis of false positives and false negatives. So our assessment actually focuses on the current version of our tool, not on the overall idea of test interfaces. As previous explained in Section 3.4.4, we also discard tasks with empty test interfaces, as this is often caused by limitations in our tool.

Finally, the selected GitHub projects from which we extract tasks might use Git mechanisms such as rebase, squash, stash apply, and cherry-pick. This way, we might have missed integration scenarios, causing the reduction of tasks we analyze. Given that the good practice is to rebase locally only and we extracted tasks from the master branch, we expect an insignificant effect over the construction of our task samples, as the number of possible lost tasks is small. Furthermore, while extracting tasks from merges, we consider all merges in a project; we are aware that some merges might be related to others, which disrupts the criteria of delimiting an independent data sample. We tried to mitigate such problem by excluding tasks of the larger sample whose commits set is a subset of the commit set of other tasks. However, 23 tasks of our smaller sample are dependent on others. As this might impact *TextI*, given it depends on the evaluation of similarity among tasks, we only compute *TextI* for the larger sample.

3.6.3 External validity

We analyzed a reduced number of tasks (513 in total), and only considered GitHub Rails projects that use Cucumber. Maybe *TestI* might be successfully applied in other contexts instead of BDD and Cucumber tests, but we did not evaluate it, and BDD itself may have eased the process of predicting code change. Also, as previously explained, our tool for inferring test interfaces is language-specific for both test and application code. So it would be hard to consider projects in other languages. Even so, we design TAITI by providing facilities to adapt it to other technologies. First, it is compatible with Gherkin-based acceptance test tools, supporting other test tools besides Cucumber. Second, it supports other programming languages and frameworks besides Rails. Given we implemented the *code analyzer* module (see Figure 9) by using the *Template Method design pattern*, there is a basic algorithm to guide the overall analysis process and specific configuration steps that require a language-specific parser. Although we cannot generalize our results, we expect that it would be easier to more accurately compute interfaces from code written in statically typed languages, and frameworks with less sophisticated web routing mechanisms. In this sense, the Rails results we obtain might be close to the minimum potential one can achieve with the *TestI* idea.

4 PREDICTING RISK OF MERGE CONFLICTS

We found evidence that test-based task interfaces are a promising predictor of file changes in Chapter 3. Even so, we have no evidence that this might actually avoid conflicts. In this chapter, we go further and assess whether it is possible to predict the risk of merge conflicts between two tasks based on the intersection between their *TestI*. Following, we present the second empirical study we conducted.

4.1 MOTIVATING EXAMPLE

As a demonstration of how test-based task interfaces might support developers for avoiding conflicts, let us see a simplified example of a merge conflict from project *allourideas/allourideas.org*.[1] The project is a web system for collecting and prioritizing ideas by a kind of collaborative survey that evolves by contributions from respondents. Andrew concludes task T_{175}, which consists of a set of refactorings to improve code quality. One day later, Becca concludes task T_{176}, which fixes a set of bugs and enhances the GUI layout. When Andrew integrates Becca's contributions, the conflicts illustrated in Figure 21 are reported. The first conflict (21a) occurs because task T_{175} excludes a set of comments whereas task T_{176} adds a method just above the comments, both affecting the same area of the file. The second conflict (21b) happens because both tasks change, in the same area of another file, the body of the `results` method.

If Andrew and Becca would predict the files they will change, they could prevent these conflicts by avoiding to perform these tasks in parallel. In the case of refactorings, as Andrew's task, predictions might be more straightforward, given there is a revision planning, and an expectation to review a file subset. However, predictions related to the development of new system functionalities or bug fixes, like Becca's task, might be more complicated. In the particular context of BDD, developers write automated acceptance tests before implementing features, and each feature is associated with test scenarios. So, we can use the TAITI tool to systematically analyze the tests, and infer the application files that would be executed by them, approximating the files that would be changed by the tasks.

For instance, by inspecting the tests related to each task (3 Cucumber tests[2] and other 5 step definitions related to task T_{175}, and 7 Cucumber tests and another step definition relate to task T_{176}), TAITI would search for references to programming elements, such as fields, methods, references to web pages and, recursively, elements, and additional files referenced by web pages, obtaining the file sets (test-based task interfaces) in Figure 22.

[1] The project is part of our sample, but task T_{175} is not because it does not satisfy all selection criteria explained in Section 4.3. Also, we present the merge conflict example by using fictitious developers.

[2] By "cucumber test", we mean a scenario and its step definitions.

Figure 21 – Merge conflicts caused by the integration of tasks T_{175} and T_{176}.

(a) Conflicting model file.

app/model/question.rb

```
<<<<<<< Task 175
=======
  def active_choices
    self.choices_count - self.inactive_choices_count
  end
  #
  # def items_url
  #   Item.collection_path(:question_id => self.id)
  # end
  #
  # def items
  #     item_ids = Item.find(:all, :select => "user_id")
  #     users = user_ids.collect { |projects_users| User.find(projects_users.user_id) }
  # end
>>>>>>> Task 176
end
```

(b) Conflicting controller file.

app/controllers/questions_controller.rb

```
  def results
    @meta = '<META NAME="ROBOTS" CONTENT="NOINDEX, NOFOLLOW">'
<<<<<<< Task 175
    logger.info "@question = Question.find_by_name(#{params[:id]}) ..."
    @earl = Earl.find params[:id]

    @question = Question.find(@earl.question_id)
    @question_id = @question.id
=======
    @question = Question.find_by_name(params[:id], true)
    @question_id = @question.id
    @earl = Earl.find params[:id]

    current_page = params[:page] || 1
    current_page = current_page.to_i
    per_page = 50

    logger.info "current page is #{current_page} but params is #{params[:page]}"

>>>>>>> Task 176
```

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 Figure 21 are part of the intersection.

Thus, assuming that Andrew and Becca have developed the Cucumber tests before the application code, according to the BDD dynamics, they could avoid the conflicts by using TAITI. 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, one with a $TestI$ that does not intersect with $TestI(T_{175})$, or at least one having a smaller intersection size, suggesting an inferior 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 $TestI$, as developers will change more files in common, increasing

Figure 22 – Test-based task interfaces of conflicting tasks from project *allourideas/allourideas.org*. The files in red are the intersection between the interfaces, and the underlined files are the conflicting ones.

(a) $TestI(T_{175})$

TestI(T_{175})
app/controllers/home_controller.rb
app/controllers/questions_controller.rb
app/models/choice.rb
app/models/earl.rb
app/models/item.rb
app/models/question.rb
app/views/abingo_dashboard/_experiment_row.html.haml
app/views/abingo_dashboard/index.html.haml
app/views/home/about.html.haml
app/views/home/index.html.haml
app/views/home/privacy.html.haml
app/views/questions/_idea.html.haml
app/views/questions/about.html.haml
app/views/questions/admin.html.haml
app/views/questions/new.html.haml
app/views/questions/results.html.haml
app/views/questions/voter_map.html.erb
app/views/questions/word_cloud.html.erb
app/views/shared/_google_jsapi.html.haml
app/views/shared/_header_vote.html.haml
app/views/shared/_highcharts_header.html.haml

(b) $TestI(T_{176})$

TestI(T_{176})
app\controllers\questions_controller.rb
app\models\choice.rb
app\models\earl.rb
app\models\question.rb
app\views\abingo_dashboard_experiment_row.html.haml
app\views\abingo_dashboard\index.html.haml
app\views\questions\new.html.haml

the chance that parallel changes affect the same file hunk.

TAITI works for Ruby on Rails projects using Cucumber as an acceptance test tool. As Figure 9[3] illustrates, given the set of Cucumber scenarios that verify the expected behavior of a task, TAITI links them to the Ruby code that automates the tests (called *step definitions* by Cucumber). The mapping process relies on regular expressions used as the identifier of the step definitions, as defined by Cucumber syntax (the most popular one). Next, TAITI parses each step definition and statically analyzes its body searching for references to programming elements, references to web pages, and its additional files. This way, TAITI analyzes web pages as well (module *Views parser* in Figure 9). In this process, it collects the set of application files that declare the programming elements

[3] Chapter 3 presented TAITI in detail. In the current chapter, we briefly review TAITI, and we do not filter its content, meaning the *content filter* module does not apply in our second empirical study.

referred by the Cucumber tests, generating *TestI*. Thus, TAITI excludes from *TestI* any auxiliary code used by step definitions, based on the project's directory structure and file extension. Also, TAITI does not analyze application files other than views, acting like almost there is no code to support tasks.

By relying on static analysis, TAITI cannot accurately identify the files executed by the tests. For example, due to Ruby's dynamic nature, it is hard to match a method call to a method declaration, and consequently, the file in which it appears. In this process, when necessary, TAITI might search for matching methods declaration with the same method name and number of arguments, or it might apply naming conventions (module *Method searcher* in Figure 9). Similarly, it is hard to identify the web pages rendered by the tests, which depends on a controller file and a set of view files. The tests usually refer to them as a path or URL that the Rails routing mechanism translates to application files based on a set of configuration properties. Given that the Rails routing mechanism requires running the system being developed, TAITI implements a simplified version of it (module *Route translator* in Figure 9), which have limitations related to static analysis and the absence of dynamic information, as the value of arguments. Also, TAITI only can reach files that are explicitly executed by the tests. These kinds of limitations might cause the inclusion of files into *TestI* that the tests cannot reach when executing, as well as prevent the inclusion of relevant files into *TestI*.

Even if TAITI does not have these limitations and the *TestI* interfaces have all relevant files for the tasks, the intersection between two interfaces does not guarantee conflict occurrence; we are dealing with predictions. First of all, the files in *TestI* might not require changes because part of the functionality might have been implemented before, as expected for tasks as refactoring and bug fix. For instance, *TestI* (T_{175}) has 21 files, and only 4 of them changed, whereas *TestI* (T_{176}) has 7 files, and 3 of them changed. As a consequence, the tasks might not change all files in the intersection between *TestI*. In the presented case, the intersection between *TestI* has 7 files, only 2 of them changed, and both have a conflict, meaning *TestI* predicts the conflicting files. But conflicts may occur in a file that is not in the intersection between *TestI* and the prediction of conflict risk might applies as well. Finally, the tests might not sufficiently cover the tasks. As a consequence, *TestI* might not contain all relevant files for a given task. 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 our evaluation study about the viability to use *TestI* as a predictor of conflict risk between programming tasks.

4.2 RESEARCH QUESTIONS

To evaluate whether *TestI* helps to predict the risk of a merge conflict when integrating the code produced by two programming tasks, we conduct an empirical study, answering some research questions.

First, given that *TestI* approximates the tasks' changed files set, we investigate whether the intersection between *TestI* interfaces points out that the tasks related to them might cause a merge conflict. So, we ask the following question.

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 possibly integrated (merged). For each pair, we use TAITI to compute *TestI* for both tasks and check whether the interfaces are disjoint. We assume there is conflict risk whenever the tasks in a pair change files in common, that is, their set of changed files are not disjoint. Knowing, for a number of task pairs, if the interfaces are disjoint and if there is conflict risk, we use a logistic regression model for assessing the association between the two variables: conflict risk (the dependent and binary variable) and non-disjoint *TestI* (the independent and binary variable).

We know a conflict only occurs when the tasks change the same hunk of a file, indeed. Even so, the prediction of conflict risk between tasks might help developers to coordinate the tasks better, avoiding the risk when possible, or at least mitigating potential adverse effects.

Although the answer to *RQ1* allows us to verify whether the intersection between *TestI* relates to the potential occurrence of merge conflicts, it does not allow us to assess the frequency in which the predictions of conflict risk apply. So, we ask the following question to evaluate the effectiveness of *TestI* as a predictor of conflict risk between programming tasks.

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, the precision is the proportion of predictions that truly applies: the ratio between the number of results that both the tasks' changed files set and *TestI* of two tasks intersect (true positives), and the number of results that *TestI* of two tasks intersect (true positives and false positives). The recall is the proportion of conflict risk that *TestI* predicts. That is, the ratio between the number of results that both the tasks' changed files set and *TestI* of two tasks intersect (true positives) and the number of results whose tasks' changed files set intersect (true positives and false negatives).

In the context of conflict risk, recall is more relevant than precision. A lower recall means unexpected conflicts might happen, and the development team might not be prepared to deal with it. In contrast, a lower precision discourages (but not prevents) parallel execution of tasks that would not conflict. Complementarily, we evaluate F_2, a variation of F-measure that weights recall higher than precision.

Next, if there is conflict risk between all new tasks and the ones under development, we desire to provide criteria for developers to compare the conflict risk between task pairs and choose a new task to work on that has the smallest chance of conflict occurrence with others. So, we ask the following question.

Research Question 3 (RQ3): Is the size of the intersection 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 *TestI*. An affirmative answer suggests that it is possible to reduce the conflict likelihood if a task coordination solution prioritizes a task choice that minimizes the intersection between *TestI*.

The questions so far assess the potential of *TestI* for predicting conflict risk between tasks. But to better assess its performance, we compare it with a predictor based on task interfaces obtained by observing textual similarity of test specifications with past tasks, which we call as *TextI*. We design *TextI* based on Hipikat (CUBRANIC et al., 2005), a tool that predicts artifacts related to tasks by investigating the past. Ideally, we would compare TAITI and Hipikat, but they require different inputs. Indeed, a previous evaluation study (ROCHA; BORBA; SANTOS, 2019) shows that *TextI* is more precise than *TestI* when predicting the changed files by a given task. Thus, we compare the ability of these interfaces to predict the risk of merge conflicts, according to the following question.

Research Question 4 (RQ4): Is *TestI* a better predictor of conflict risk than *TextI*?

We answer that by checking whether *TestI* has better precision, recall, and F_2 measures than *TextI*. As a prerequisite, we verify the intersection between *TextI* relates to the potential occurrence of merge conflicts by using a logistic regression model, like *RQ1*. This time, the two binary variables are conflict risk (the dependent variable) and non-disjoint *TextI* (the independent variable).

The *TextI* of a task t is the intersection between the set of files changed[4] by the three past tasks with the most similar test specifications to t. The similarity between test specifications is the cosine similarity between vectors of TF-IDF values (SALTON; MCGILL, 1986). As part of the process of computing similarity, we preprocess the test specifications by tokenizing the text using spaces and punctuation, stemming it, and eliminating English and Gherkin keywords, such as *Given*, *When* and *Then*.

[4] The previous evaluation study limited such a set to application files reachable by *TestI*. Here we investigate the original set of changed files as well as the limited set, as discussed in Section 4.4.4.

4.3 STUDY SETUP

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

4.3.1 Initial project selection

Given that TAITI is specific for Rails projects that use Cucumber for specifying acceptance tests, we use a script[5] to mine GitHub repositories looking for projects that satisfy these requirements. Basically, the script queries Ruby projects, downloads the latest version of each project, and based on the gemfile content (a file that lists all project dependencies), verifies whether the project uses libraries related to Rails and Cucumber. We restrict the search by avoiding projects created earlier than 2010, as Cucumber and BDD were less popular before that, and TAITI might not be compatible with older versions of Ruby and Rails.

Hoping to find a significant number of projects, we perform the search in three steps, changing other parameters. First, aiming to find the most active projects, we sort results by the date of the last update. Second, aiming to find projects with a significant test dataset, we select the Ruby projects referenced by Cucumber's site[6]. Third, aiming to find popular projects, we limited the project's maximum number of stars, starting with 15,000 and going up to 50, and sorted results by descending order of stars number. In the end, we find 1,164 Rails projects (461 projects in the first step, 26 projects in the second step, and 898 projects in the third step), 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.

4.3.2 Task extraction and further project selection

From the 79 selected projects, we try to extract tasks with associated Cucumber tests[7]. We consider a task consists of the commit set between a merge commit and the common ancestor with the other contribution the merge integrates. This way, we clone each project and search for merge commits performed until June 2019, excluding fast-forwarding merges. From each merge commit, we try to extract two tasks (which we call as "merge tasks"). During such a process, we discard tasks that contain intermediate merges as a strategy to construct an independent sample. Also, sometimes there are successive merge commits, which disables task extraction. As a consequence, we do not always extract two tasks from a merge commit.

Next, we discard tasks that do not contribute with both application code and Cucumber tests, as we would have no evidence that they could have been developed accordingly

[5] Available in our online Appendix (ROCHA; BORBA, 2020).
[6] <https://cucumber.io/docs/community/projects-using-cucumber/>
[7] By "cucumber test", we mean a scenario and its step definitions.

to BDD practices. 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 "independent tasks".

Finally, given we deal with 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: 4 projects only have fast-forwarding merges, and 7 projects have no merge commits because they have only one active contributor per time. Also, 24 projects do not have tasks that contribute with both application code and Cucumber tests (there are no tests related to the task, or they were added to the project by different merge commits). And four projects have less than two eligible tasks.

4.3.3 Collecting task data

Finally, we collect the set of changed files, the *TestI*, and the *TextI* of each task. The set of changed files by a task is the union of the files modified by its commits. We use TAITI for computing *TestI* (as described in Section 4.1). As part of this process, TAITI collects the tests of 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. For avoiding inconsistencies, such as the case a commit deletes a Cucumber test created by a previous commit, TAITI consolidates the result by selecting the ultimate version of the Cucumber test (i.e., the version from the last commit).

We need tasks for which we can compute *TestI*. Thus, while collecting task data, we discard tasks with no implemented Cucumber tests, or partially implemented ones (not all step definitions related to a scenario are implemented), as well as tasks with step definitions that TAITI can not parse. We also discard tasks whose *TestI* is empty, as this is often associated with TAITI limitations or limited test coverage, delimiting a set of "relevant tasks". Given we deal with 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.

Next, for answering *RQ4*, we try to compute a non-empty *TextI* for the 1,762 relevant tasks. Given *TextI* depends on the project history, we cannot find three similar past tasks (as explained in Section 4.2) for 36 tasks[8]. Also, there is no intersection between the changed files set of the three most similar past tasks for 665 tasks. Given that we also

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

filter out projects with less than two tasks with non-empty *TextI*, we get a preliminary sample of 1,057 tasks from 22 projects after computing *TextI*.

At last, we simulate the integration of likely concurrent tasks. Thus, we compute all task pairs per project by grouping tasks that were concluded with no more than one month of difference. This way, we might group tasks extracted from different merge commits. 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.

4.3.4 Task pair sample

Table 12 summarizes the steps for constructing our task sample, but just for the final project set. As previously explained, "Merge tasks" is the number of tasks extracted from merge commits that do not contain intermediate merges. "Candidate tasks" is the number of merge tasks that contribute with both application code and Cucumber tests. "Independent tasks" is the number of candidate tasks that are not a subtask of other tasks, i.e., tasks whose commit set is not a subset of other tasks. We compute *TestI* for all independent tasks. "Relevant tasks" is the number of independent tasks for which we can successfully compute a non-empty *TestI*. "*TextI*" is the number of relevant tasks that have a non-empty *TextI*. At last, "Concurrent tasks" is the number of non-empty *TextI* tasks that have at least one other task that is less than one month apart, which we classify as possible concurrent tasks, and "Pairs" is the number of integrations 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.

Note that although the overall process for constructing the task sample is similar to the one from our first empirical study, their selection and exclusion criteria are different, resulting in the selection of distinct task sets that have only one task in common.

While constructing our task sample, we do not systematically target representativeness and diversity (NAGAPPAN; ZIMMERMANN; BIRD, 2013). Even so, we observe some variety concerning the attributes in Table 13,[9] which presents six projects from our sample. Note that we collected these attributes from the project perspective in October 2019, but these might not apply during the time the developers concluded the tasks of our sample. Two projects have no stars, and one project has three collaborators, but they are not toy systems (a system with no practical usage that someone creates for study purpose). Also, two projects do no have Cucumber tests, which means they gave up using Cucumber. This phenomenon does not compromise our results because the projects used Cucumber during the time the extracted tasks were concluded. All data related to our sample is available in the online Appendix (ROCHA; BORBA, 2020).

[9] Our online Appendix presents the complete table, characterizing the 19 projects of our sample.

Table 12 – Construction of the task pair sample.

GitHub 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

Table 13 – Diversity of projects in our task pair sample.

Repository Name	Description	#Stars	#LOC	#Tests	#Commits	#Authors
allourideas.org	A tool for groups to collect and priorize information.	134	56,884	126	2,372	20
whitehall	A content management app for the UK Government.	549	217,235	263	23,435	336
diaspora	A privacy-aware, distributed, open source social network.	12,296	195,654	277	19,915	585
gitlabhq	A DevOps platform.	22,020	2,346,933	0	149,603	3,282
jekyll	A simple, blog-aware, static site generator.	38,807	59,733	259	11,093	1,063
RapidFTR	An app for sharing info about children in emergencies.	287	98,820	274	4,939	252

4.4 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; BORBA, 2020) provides detailed information.

4.4.1 RQ1: Are tasks with non-disjoint *TestI* interfaces associated with higher merge conflict risk?

Tasks with non-disjoint *TestI* interfaces are more likely to have modified files in common

Tasks with non-disjoint *TestI* interfaces are 2.07 times (the odds ratio of the logistic regression, as explained in Section 4.2) 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 detailed explained in Section 4.1.

Because *TestI* only contains some specific types of application files (i.e., Ruby files, .html files, and its variants, such as .haml and .erb files), we wonder if *TestI* performs better when predicting merge conflict risk exclusively in files reachable from *TestI*. So, we repeat the same kind of analysis as before, but we filter the task's changed files set by excluding configuration files, test files, and any other programming files such as JavaScript files[10]. This time, we find that when the *TestI* interface of two tasks intersects, 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 in this case, because the restricted set of changed files by a task probably relates more 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 has a general purpose in the project, given configuration files are not part of *TestI*. Thus, by restricting the tasks' changed files set, we eliminate such a fail.

4.4.2 RQ2: How often does *TestI* predict conflict risk between two tasks?

A minimal intersection between *TestI* 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 performance, 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. Table 14

[10] In other words, we restrict the set of changed files by a task to Ruby and .html files (and common variations) into `app` or `lib` folders.

summarizes the results. We experimentally delimit this small value range 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 3,712 integrations are potentially conflicting (58.4%). Such a number seems incompatible with other results about the frequency of conflict occurrence presented in Section 2.2.1, which is near to 10% of all merges per project, but it is important to note that it just refers to potential conflicts. Also, almost half of the potential conflicting integrations (49%) have up to 10 files in the intersection between *TestI*. So, we compute precision and recall by using an intersection limit ranging from 1 to 10, but we observe a substantial reduction in results quality (mainly in the recall measure) when such a limit is bigger than 5. As in Section 4.4.1, we also investigate the alternative result that restricts a task's changed files set by excluding files not reachable by *TestI*.

Table 14 – Precison and recall measures of the *TestI* intersection predictor. "All files" is the result when considering all files changed by tasks, and "Files reachable from *TestI*" is the result when restricting a task's changed files set by excluding files not reachable by *TestI*.

min intersection size	Risky pairs (%)	All files			Files reachable from *TestI*		
		Precision	Recall	F_2	Precision	Recall	F_2
1	96.70	0.59	0.98	0.86	0.37	0.98	0.74
2	92.41	0.60	0.95	0.85	0.38	0.95	0.73
3	88.07	0.60	0.91	0.83	0.38	0.92	0.71
4	78.90	0.62	0.84	0.78	0.39	0.85	0.69
5	71.07	0.63	0.77	0.74	0.40	0.78	0.66

According to Table 14, we find that a minimal intersection between *TestI* (1 file) is the best predictor of conflict risk, and especially when we do not restrict the task's changed files set, contrasting our expectation that such a strategy might benefit the results by reducing false negatives (FN). We can also observe that there is a subtle variation[11] in precision when we change the value of the minimal intersection size. The reason is when we increase the value of the minimal intersection, the false negatives also increase, whereas both the true positives (TP) and the false positives (FP) might decrease. The final balance impacts more in recall than precision because the first depends on TP and FN (which inversely vary according to different rates), whereas the second depends on TP and FP (which decrease). For example, when the minimum intersection size is 4, there is 3,106 TP, 1,912 FP, and 606 FN. When the minimum intersection size is 5, there is 2,855 TP, 1,665 FP, and 857 FN.

[11] For simplicity, we round the values, but they are not identical.

Next, we find that recall outperforms precision in the overall result with a significant advantage. By considering a reduced minimal intersection size as one file, for example, we find the intersection between *TestI* points out conflict risk for 96.7% task pairs of our sample (the percentage of the predicted positive condition rate presented by column "Risky pairs" in Table 14). However, only 58.4% of them are risky in practice (the tasks did change at least one file in common). In this sense, there is much FP and few FN, which benefits recall and prejudice precision. This way, by also comparing the predicted positive condition rate, someone might consider the predictor with minimum intersection size of 4 files a better option, as it points out conflict risk for 79% task pairs and presents acceptable values of precision, recall, and F_2.

We wonder much of the intersection between *TestI* might be from files exercised by the test precondition (`Given` steps of Cucumber scenarios). Given different tests require similar setup, such as user authentication, they might not be relevant to most tasks, motivating the design of an alternative *TestI* that discards application files accessed by `Given` steps, as suggested by the previous study. On the other hand, the coding style varies among projects, and maybe it is necessary to verify the particularity of each project to design a refined solution to prioritize files to be included in *TestI*. Also, as already exposed, by adopting a mostly conservative solution based on static code analysis and naming conventions, TAITI might inflate *TestI*. For example, Cucumber tests usually use a method to decide the view to access by using a decision structure based on some argument. When TAITI cannot identify the specific accessed view, it captures all possible ones.

Finally, Table 14 also shows that precision severely reduces, and recall slightly improves when we restrict predictions to consider only files reachable from *TestI*, whether we compare results to those applied to all files. This time, we observed that 1,396 potential conflicting integrations (37.6% of all risky integrations) only affect files that are not reachable by *TestI*, decreasing TP, and increasing FP. Such a phenomenon affects precision. On the other hand, proportionally, we observed a reduction of FN, which promotes recall.

The intersection between *TestI* is a viable predictor of conflict risk between tasks, even when it does not predict the potential conflicting files

So far we discuss the prediction of conflict risk among tasks by abstracting the location of such a risk. To better evaluate our predictions, we investigate the frequency that we guess the conflicting files, that is, the files more vulnerable to conflicts because tasks concurrently changed them. So, 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 (which represents 20.9% of all integrations), the intersection between *TestI* predicts some potential conflicting file. Whether we consider files genuinely reachable by *TestI*, 36.4% of the integrations are risky. And as expected (the number of right 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.

Given that TAITI does not analyze other application files than views, *TestI* reaches only the surface of the code that could be exercised by the tests, and, as a consequence, potential conflicting files are not covered. By manually inspecting some task pairs, we also find that in some cases, *TestI* cannot guess the potential conflicting files because there is no clear relationship between them and the tasks, which suggests the tasks are not cohesive. In other cases, the potential conflicting file has an implicit relationship with other files in *TestI* that is defined by Rails, which is not reachable by the tests. Also, other limitations of TAITI might prevent the inclusion of the potential conflicting file in *TestI*. For example, its routing mechanism wrongly translates a URL into application files, impairing the identification of some files, or the tests adopt a deprecated syntax that TAITI does not support, limiting the analysis of the full test code.

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 reflect a degree of proximity between the parts of the code modified by both tasks, eventually leading to conflicts, even in files not directly reached by the interfaces. Therefore, even when *TestI* does not predict the potential conflicting files, it might predict the risk of conflicts.

4.4.3 RQ3: Is the size of the intersection between two *TestI* interfaces proportional to the number of files changed in common by the corresponding tasks?

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 (small = 0.10, medium = 0.30, and large = 0.50).

We find the larger the size of the intersection between *TestI*, the larger the number of files that are changed by both tasks, but the correlation is small ($p < 0.001$ and $\rho = 0.21$). The same applies when dealing only with files reachable by *TestI* too, with $\rho = 0.15$. Given that developers might not change all files into *TestI*, we cannot expect a large correlation.

In practice, this result suggests that developers can decrease the probability of merge conflicts by prioritizing tasks based on the size of the intersection among 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 dependent on the selection of the next task to perform by each developer. As illustrated in Section 4.1, Andrew and Becca could prevent conflicts by selecting a task whose *TestI* has a smaller intersection with the *TestI* of tasks T_{175} or T_{176}.

4.4.4 RQ4: Is *TestI* a better predictor of conflict risk than *TextI*?

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 4.2) 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[12] is lower, but logistic regression focus on TP and a complete comparison depends on precision, recall, and F_2. In the case of files reachable by *TestI*, the likelihood is 2.25, a smaller value, contrasting the *TestI* result. Such a phenomenon means many potential conflicting files according to *TextI* cannot be part of *TestI*. The overall favorable result aligns with the fact that *TextI* is more precise than *TestI*, and also motivates further investigation about its predictive ability.

TextI is a more precise and less complete predictor of conflict risk than *TestI*

Table 15 summarizes the evaluation results of *TextI*. We show only minimum intersection sizes 1 and 2 because the recall is insignificant for larger values. Although we expect a better precision, we can observe that the improvement in precision is expressive, but the recall is severely affected, worsen F_2 as well. In practice, if a file relates to the most similar past tasks, given the file was modified by all of them, such a file has more chance to indeed relates to the new task as well. Then, whether most files in *TestI* truly relates to the tasks, disjoint *TextI* interfaces have more chance to be a TP of conflict risk. This way, the number of FP decreases. However, given that recall is more relevant than precision in the context of conflict prevention, such a result discourages the usage of *TextI* as a predictor of conflict risk between programming tasks. Besides, the cost for computing *TextI* is superior to the cost for computing *TestI*, given it relies on text processing.

Table 15 – Precison and recall measures of *TextI* intersection predictor. "All files" is the result when considering all files changed by tasks, and "Files reachable from *TestI*" is the result when restricting a task's changed files set by excluding files not reachable by *TestI*.

min intersection size	Risky pairs (%)	All files			Files reachable from *TestI*		
		Precision	Recall	F_2	Precision	Recall	F_2
1	35.82	0.75	0.46	0.50	0.49	0.48	0.48
2	14.72	0.81	0.20	0.24	0.56	0.23	0.26

Finally, a previous study (ROCHA; BORBA; SANTOS, 2019) suggests that the files in the intersection between *TestI* and *TextI* are more vulnerable to change and, as a consequence, to lead to conflicts. Thus, we also evaluate a hybrid task interface based on the intersection

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

between *TestI* and *TextI*. The hybrid task interface further improves precision (0.86) and worsen 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 some noise from TAITI. So, FP further decreases, benefiting precision. However, most potential conflicting files are not covered, increasing FN, and compromising recall.

4.4.5 Other results

TestI cannot predict conflict risk in MVC segments

According to Rocha, Borba e Santos (2019), when dealing with an MVC-like application (e.g., web applications developed in Rails), *TestI* performs better when predicting changes in controller files. As controllers uniquely identify an MVC segment (related model, view, controller, and auxiliary files), one could also use *TestI* to predict changes in segments. When a controller appears in *TestI*, the task quite often changes at least one file from the associated segment (ROCHA; BORBA; SANTOS, 2019). Therefore, it would be interesting to investigate whether tasks with common controllers in their *TestI* are associated with higher chances of merge conflict risk in these controllers segments.

This way, 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 intersect with any file from the MVC segment identified by the common controller. Like answering *RQ1*, we use a logistic regression model for assessing the association between two binary variables. This time, the dependent variable tells whether there is any controller file in the intersection between *TestI* and the independent variable informs whether both tasks change the segment identified by the intersected controllers in *TestI*.

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. From this subset, the tasks from 26% integrations (19% of all integrations) change at least one file in common from some segment identified by the intersected controllers in *TestI*. But we do not find a statistical correlation providing evidence that controllers in *TestI* are predictors of conflict risk in their associated MVC segments. Thus, recommending developers to avoid the concurrent execution of tasks that focus on the controller of the same segments might be much restrictive.

TestI similarity is not often associated with higher conflict risk

Besides studying the relation of *TestI* intersection with conflict risk, we explored whether *TestI* similarity relates to higher conflict risk, wondering it might be an alternative predictor. So we computed *TestI* similarity by using the cosine similarity between vectors

of TF-IDF values (SALTON; MCGILL, 1986), in which the similarity is in the scale [0,1], zero meaning no similarity and one, maximum similarity. Then, we confirm there is a relation between *TestI* similarity and conflict risk by using a logistic regression model, similarly to *RQ1*. The results of precision and recall measures are very similar to the ones 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 on the similarity measure, is not significant for our context. It does not matter how different are two *TestI* interfaces; if they have common files, there is a risk of 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, 2 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.

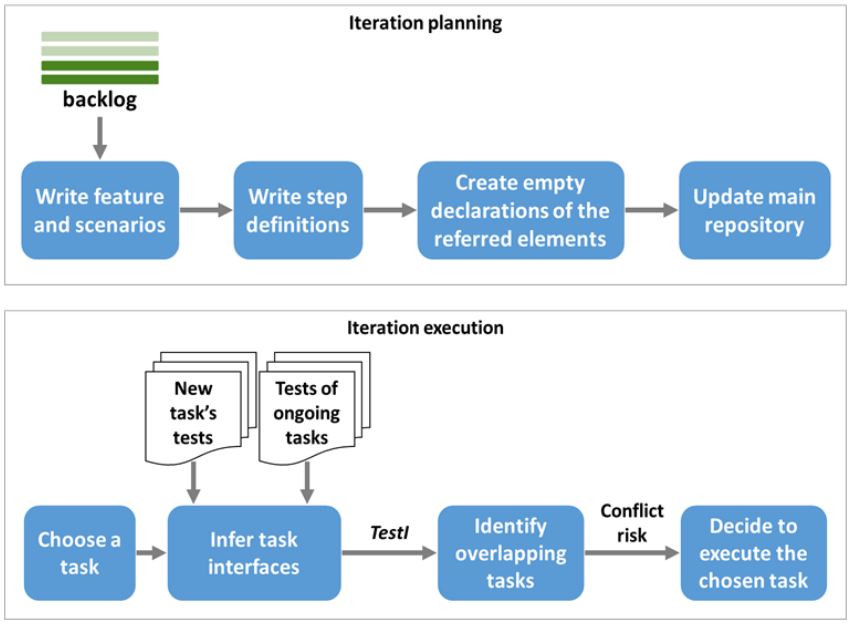
4.5 IMPLICATIONS

Prioritizing tasks based on *TestI*

Based on test-based task interfaces, we propose that, whenever possible, developers might dynamically schedule programming tasks to prevent conflicts, as Figure 23 illustrates. First, the development team plans the next iteration by selecting user stories of the product backlog, deriving programming tasks. The planning depends on factors like project restrictions concerned with time and resources, stakeholders' priority, task complexity, and developer skills. In such a process, the team projects the Cucumber tests that validate each task by writing features, scenarios, and step definitions. As expected, the tests initially fail because the application code is not complete. When dealing with new functionality that requires code creation, the developers must create empty declarations of the referred elements by the tests as well. It is necessary to enable the prediction of application files related to the task. For example, in our motivating example, both tasks T_{175} and T_{176} change the body of the `results` method in the file *app/controllers/questions_controller.rb* (Figure 21b). Because the file and the `results` method already exists, there is no need to create empty declarations. But let us imagine that the file *app/controllers/questions_controller.rb* does not exist when developers begin the execution of tasks T_{175} and T_{176}. How can the test code reach such a file? In this case, developers should create the file in their workspace and declare the `results` method with an empty body. In the future, to conclude the task, developers should update the body of such a method. In the end of the iteration planning, the team must update the central repository to publish the projected tests, given the analysis of conflict risk depends on them.

Next, in his workspace, when selecting a task to perform, each developer should require

Figure 23 – Prioritizing tasks based on *TestI*.



the conflict risk analysis based on *TestI* by using TAITI. TAITI receives as input the tests of the task a developer intends to work on and the tests of the ongoing tasks being executed by other developers. Tasks with overlapping interfaces, whose *TestI* intersects, are likely to change some file in common, increasing the conflict risk. This way, we hypothesize that tasks with less overlapping interfaces, when executed concurrently with the ongoing tasks, would be less likely to lead to conflicting code changes. The conflict risk of a new task is the sum of the size of its intersection values related to each ongoing tasks.

Finally, the developer decides about the task he will perform next based on the conflict risk with other tasks. In case it is not possible to avoid the concurrent development of the potential conflicting tasks, the knowledge about the conflict risk might support the developers to better deal with the situation by coordinating the test coverage planning or improving communication among team members.

To our proposal effectively works, the team must use a task management system, such as Pivotal Tracker[13], which integrates with a development platform such as GitHub, facilitating the verification of tasks' status and the connection among tests and tasks. Also, we should integrate TAITI with the development environment and the task management system.

Prioritizing tasks in a centralized management context

The whole idea of prioritizing tasks based on test-based task interfaces fits better on a BDD context, which presupposes agile teams that are self-managed by definition. However, a team that projects acceptance tests before application code can use our strategy as

[13] https://www.pivotaltracker.com/

well, even in the case that a project manager designates tasks to developers. The critical question now is that the project manager will be responsible for the iteration execution (see Figure 23), dynamically deciding the next task of each developer. Considering the decision process is supported by a tool, there is no overhead. The issue is that task allocation would not happen during the iteration planning, which is the standard practice in the context of centralized management. On the other hand, the project manager will control parallelism during software development, increasing or decreasing it when possible for avoiding conflicts, by indicating the next task a developer will perform and when he can do it.

Integrating tools that support software development

Continuous integration is a software development practice where developers integrate their work frequently, supported by automatic merge, build and test tools to detect integration errors as quickly as possible. Given its popularity in agile teams nowadays, when proposing the adoption of a new practice and tool for supporting software development, it is necessary to provide directions about how to fit the continuous integration dynamic.

As Figure 23 indicates, the development team should publish the tests (features, scenarios, and step definitions) related to the tasks planned for the next iteration, by open pull requests, which embraces one or more local commits. Then, the continuous integration dynamic proceeds by automatically evaluating the open pull requests. Initially, they will not pass because the tests fail, even if there is no syntactic or build error, as the application code is not complete. Nevertheless, the team should approve these pull requests, because it is necessary to provide visibility of the tests. From this point, the continuous integration dynamics should proceed as usual, and the team should only accept correct pull requests. The developers should locally use TAITI for selecting the next task to perform, considering the conflict risk among tasks. Alternatively, the project manager might designate tasks to developers as well.

Influencing factors

In our first empirical study, we collect evidence that the better the test coverage of a task, the better the *TestI* predictive power. Thus, whenever the whole project has adequate test coverage, developers can have more confidence in the conflict risk results: If *TestI* approximates the changed files, the intersection between *TestI* better approximates the concurrently changed files, which are more vulnerable to merge conflicts. Besides, our strategy applies for projects with many collaborators, for which we can expect much parallel work. Projects with geographically distributed collaborators might benefit as well, given the communication among team members is more complicated, impairing task management. In other scenarios, the teams might expect the occurrence of a few conflicts, leading someone to ponder about the overhead to implant a strategy for avoiding conflicts.

Precision versus recall

Previously we stated that when dealing with the prediction of conflict risk, recall is more relevant than precision. A lower recall rate implies a higher number of unexpected conflicts, and we wonder that such a surprise might compromise developers' productivity. The absence of risk alerts might lead to a false security sensation, making developers neglect communication needs and testing practices as strategies to mitigate integration problems. In turn, a lower precision discourages parallel execution of tasks that would not conflict. Given that an erroneous risk alert does not prevent task execution, we understand that developers might prioritize tasks according to the conflict risk rate concerning the other current tasks, favoring lower rates. This way, the risk rate of a task has a meaning only when compared with others.

On the other hand, a higher precision rate might motivate development teams to adopt a strategy for avoiding conflicts. In general, developers desire minimal intervention as possible during task execution. In this sense, they might perceive excessive alert of conflict risk (false positives) as unuseful. Also, knowing that some conflicts are inevitable because of stakeholders' priority and other factors, developers might prefer to adopt strategies only to avoid the most probable conflicts (true positives).

This way, 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 14), enabling an optimistic (emphasis on precision) or pessimist strategy (emphasis on recall).

4.6 THREATS TO VALIDITY

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

4.6.1 Construct validity

We assume that merged contributions correspond to programming tasks as defined by a BDD team, which might not always apply. An alternative solution would be to delimit a programming task as a set of commits whose messages refer to the same task id or consider each pull request as a task. Still, most projects do not systematically refer to task ids in commit messages or use pull requests to integrate contributions. Also, many projects use Cucumber as a test tool and do not adopt BDD. Even so, in the context of retrospective analysis, dealing with completed tasks, the fact that the developers write tests after concluding the application code does not compromise our conclusions.

Finally, we did not verify the occurrence of merge conflicts but only the risk of conflict based on the intersection between tasks' changed files set, given that a merge conflict might only happen whether the developers change the same file. As previously explained, we did not find a substantial number of merge conflicts to reproduce, given the requirements

to construct our task sample (Section 4.3). Thus, we simulate the integration of possible concurrent tasks, i.e., tasks that were concluded with no more than one month of difference. However, as *TestI* does not inform the methods or lines that will be changed in its files, in practice, it only can alert developers about potential conflicts. In this sense, our evaluation study aligns with the real usage context of *TestI*. Complementary, we would better restrict our sample of possible concurrent tasks based on the tasks' init date and its duration. Alternatively, we would verify if possible concurrent tasks change or add files in a common date, reinforcing the construction of 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.

In the ideal situation, we would have a task database that informs the task objective, the source files that contain the task implementation, the set of Cucumber tests that validate the task behavior, and if the task caused a merge conflict when integrated into the project. In the absence of such a database, we make approximations to evaluate the potential of test-based task interfaces for supporting developers to avoid conflicts.

4.6.2 Internal validity

On the one hand, by adopting a lightweight and mostly conservative solution that relies on static code analysis and naming conventions, TAITI might inflate *TestI*. Also, by considering all changes in a contribution, although some of them might be reverted, we might further inflate *TestI* and the set of changed files by a task. Tangled contributions (DIAS et al., 2015) that include non-task-related changes might inflate *TestI* as well. Given that tangled contributions are frequent in practice, such a limitation might turn our definition of programming task more realistic.

On the other hand, given that TAITI does not analyze other application files than views, among other design limitations, it might omit important files from *TestI*. This way, TAITI performs like we are dealing with a new project, and almost there is no code to support tasks, which might not be true. Also, TAITI does not deal with all possible Cucumber's syntax for identifying or reusing a step definition. Thus, we might have discarded consistent tasks from our sample because we wrongly consider they have undefined step definitions (i.e., partially implemented tests), or we might miss part of the tests related to a given task. Besides, we assume the Cucumber tests that are part of a contribution validate the expected behavior of the supposed task. 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 ignore relevant tests when computing *TestI*.

When dealing with conflict risk, we check whether the tasks changed any file in common by ignoring file renaming. As a consequence, we might miss some intersections between the tasks' changed files set, which impacts on 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 whether a task removes a file that is 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 over 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.

4.6.3 External validity

Our sample contains only GitHub Rails projects that use Cucumber because TAITI 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 the worst reference of the true potential of *TestI*.

5 CONCLUSIONS

In this work, we investigate a strategy for inferring task interfaces based on acceptance tests (*TestI*) and its usage for evaluating the risk of merge conflicts among programming tasks, assuming the specific BDD context. By knowing such a risk, a developer might wisely choose a task to work on in parallel with other ongoing tasks, reducing conflicts occurrence. Ideally, there is always a no risky task choice, which means developers might find a development path free of conflicts. However, in case all alternatives are risky, developers might compare them and choose for the less risky, for which we believe the integration effort is less too. Also, even when developers decide to execute a more risky task due to a project restriction, the upfront knowledge about conflict likelihood might be useful to coordinate test coverage planning to better check the risky code and to detect communication needs among team members.

By providing a tool for computing *TestI*, TAITI, first we conducted a retrospective analysis of Rails projects that confirmed *TestI* is a promising code change predictor and, as a consequence, it has the potential for avoiding conflicts. In sum, *TestI* can predict as many changes as random interfaces (*RandomI*). Nevertheless, *TestI* is always more precise than *RandomI*. Compared to a kind of task interface based on past similar tasks (*TextI*), *TestI* covers more code changes, although it is less precise.

Our results also confirm that, in order to be truly helpful, *TestI* requires a broad test coverage per task. Besides, although the intrinsic limitations of a static code analysis do not compromise *TestI*, they cause a little impact over recall, by affecting false positives and false negatives. This finding means the potential of *TestI* is underestimated and a more refined tool might benefit it. For example, by adopting a smarter static analysis with type inference and by analyzing step definitions according to its execution sequence, as well as by propagating data from one step to another, we might simulate test execution faithfully.

The study also brought us insight into improving *TestI* and its usage, besides the improvements in static analysis strategy. Concerning our tool, we might improve it to analyze application files other than views (when possible). For now, TAITI acts like we are dealing with a new project, and almost there is no code to support tasks, which is the worst context for its usage. As a consequence, *TestI* reaches only the surface of the code that could be exercised by the tests, impairing the prediction of changed methods instead of changed files, which might derive more refined predictions of conflict risk. For instance, let us suppose some tests explicitly calls the application methods m_1 and m_2, which are declared into different files and internally call the application method m_3. TAITI ignores such internal calls when analyzing the tests. Then, when evaluating the conflict risk between tasks t_1 and t_2, supposing that t_1'tests calls m_1 and t_2'tests call m_2, there is

no intersection among their *TestI*, alerting there is no conflict risk between these tasks. However, there is a conflict risk if both tasks change m_3, which *TestI* can not predict. On the other hand, let us now suppose that m_1 and m_2 are declared into the same file. The intersection among *TestI* would alert a conflict risk, but considering that m_1 and m_2 are in different areas of the same file, if task t_1 only changes m_1 and task t_2 only changes m_2, there is no conflict risk at all. We plan to fix such a limitation in the future, envisioning to improve *TestI* predictions and conflict risk predictions as well. Besides, we might adopt a hybrid approach for inferring *TestI* that uses static analysis when dealing with failing tests and a dynamic strategy when dealing with running tests.

Regarding conflict avoidance, we conducted a second retrospective analysis of Rails projects to evaluate predictions of merge conflict risk. Our results bring evidence that tasks with non disjoint *TestI* interfaces more likely 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 *TestI* does not always predict the potential conflicting files, *TestI* might suggest a degree of proximity between the parts of the code changed by both tasks, eventually leading to conflicts. Besides, although *TextI* is a more precise predictor of changed files and conflict risk, it cannot cover most potential conflicts (i.e., it has a low recall rate). Given that in the context of conflicts, recall is more important than precision, we conclude that *TestI* outperforms *TextI*.

By further investigating our results, we believe that it is possible to improve predictions whether we can refine *TestI* by discarding application files accessed by test preconditions, reducing false positives. Even so, note that more important than point out the existence of conflict risk, is the intensity of such a risk, the information that actually guides developers when prioritizing tasks. This way, developers might just ignore low risk rates. Also, predictions depend on task cohesion. If the tests do not relate to the system functionality underlying the task, predictions based on *TestI* do not apply.

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 occur in a subset of risk predictions, whether the tests truly relate to the tasks.

Finally, although we bring some evidence that task interfaces might help developers to avoid conflicts and this benefits the whole project, we also know that the effort to solve conflicts is more significant than the absolute conflicts number. In this sense, someone might consider *TestI* as a starting point to a robust solution for estimating resolution effort of potential conflicts. Moreover, the need for code changes depends on the features

already supported when a developer starts working on a new task. Thus, *TestI* represents the file set relevant for completing a task, but we cannot expect that all these files require changes. The low precision rates we found when evaluating the ability of *TestI* for predicting file changes reinforce such understanding. As a consequence, we cannot expect that the intersection between *TestI* interfaces predicts the location of conflicts. At last, code changes are not usually cohesive, meaning there is always a conflict risk among tasks, i.e., developers might change codes that do not align with their tasks and cause conflicts.

5.1 CONTRIBUTIONS

This work makes the following contributions:

- Propose a new strategy to predict task interfaces that relies on automated acceptance tests, which we call *test-based task interfaces* or *TestI*.
- Develop a prototype tool to compute test-based task interfaces by statically analyzing acceptance tests, TAITI.
- Gather evidence that test-based task interfaces are a promising predictor of changed files.
- Propose alternative strategies for test-based task interfaces, such as task interfaces computed by executing tests (*DTestI*) and task interfaces obtained by observing textual similarity of test specifications with past tasks (*TextI*), which enable us to evaluate better *TestI* and identify improvement opportunities.
- Propose the usage of test-based task interfaces to assess the risk of merge conflicts between programming tasks, supporting developers to dynamically schedule tasks, prioritizing them according to conflict likelihood.
- Gather evidence that test-based task interfaces might help to avoid conflicts by predicting when tasks are likely to change some file in common.
- Develop reusable scripts for mining GitHub repositories that other empirical studies can use.

5.2 FUTURE WORK

As the proposed studies described here are part of a broader context, a set of related aspects were left out of scope, as following described. Thus, we suggest them as future work.

- We plan to refine our second empirical study by discussing the results under the project perspective.

- We plan to evaluate conflict predictions using a filtered notion of *TestI* that excludes application files exercised by the test precondition (`Given` steps of Cucumber scenarios). We expect to reduce false positives derived from the fact that different tests might require a similar setup code, such as user authentication, that might not align to the task purpose.

- Alternatively, we also plan to evaluate conflict predictions using an extended notion of *TestI* that includes syntactic and logical dependencies (co-changes) of each file in *TestI*, as adopted by tools such as Cassandra (KASI; SARMA, 2013) and Hipikat (CUBRANIC et al., 2005). As proposed by related works and confirmed by manual task analysis, the dependencies might capture implicit relationships among files that tests are unable to detect. We performed a preliminary study (SANTOS; ROCHA; BORBA, 2019) assessing the predictive ability of such a task interface concerning changed files, considering syntactic dependencies. The study shows it is a promising strategy to improve *TestI*, and possibly the most promising among the related work. Also, such an extended notion of *TestI* enables the investigation of the prediction of build and test failures, besides merge conflicts. As known, concurrent changes on different (but related) files might cause indirect conflicts that also reduce developers' productivity.

- We intend to improve TAITI to simulate test execution faithfully, hoping to improve the predictive ability of *TestI*, and predictions about conflict risk as well. Possible resolutions include: adopting a smarter static analysis with type inference, analyzing step definitions according to its execution sequence, full propagating data from one step to another (we partially do it), and analyzing application files other than views.

- Once we have a new version of TAITI, we plan to refine *TestI* to predict method changes instead of file changes and investigate if this resolution improves predictions of conflict risk among programming tasks.

- We plan to conduct a case study to evaluate the potential of *TestI* for avoiding conflicts in the field, considering its influence on the dynamics of development teams and other human and social effects that might compromise its performance and acceptance. Such a study requires the development of a task planning tool for supporting developers to choose a task to perform based on the conflict risk with other tasks, integrating both TAITI and the development environment. Also, it requires a BDD project with adequate test coverage.

- We intend to evaluate a hybrid approach for inferring *TestI* that uses static analysis when dealing with failing tests, and a dynamic strategy when dealing with running tests. Similarly to the performed studies, we plan to assess the predictive ability of such a strategy concerning changed files and conflict risk.

- We believe that it is relevant to evaluate the ability of *TestI* for predicting merge conflicts as well. We informally investigated a reduced number of merge scenarios because we could not construct a larger sample. Hoping to find a larger sample of conflicting merge scenarios, we plan to extend TAITI to support other programming languages than Ruby.

- It is also interesting to evaluate the relationship between *TestI* and code quality metrics. For example, a low code coupling positively impacts *TestI* in terms of precision and recall measures? And what about the prediction of conflict risk among tasks: Do high coupled tasks are more vulnerable to cause a merge conflict than other tasks?

- Finally, concerning the practical usage of our approach, we plan to provide a confidence index of conflict risk prediction per project based on its test coverage. According to our results, we can expect better predictions when the project has adequate test coverage.

5.3 RELATED WORK

In this section, we present the related work. In Section 5.3.1, we discuss an alternative strategy to avoid conflicts by task scheduling. Subsequently, in Section 5.3.2, we discuss existing strategies related to the purpose of inferring task interfaces. Then, in Section 5.3.3, we review other tools and practices that might also avoid conflicts in collaborative software development.

5.3.1 Avoiding conflicts by task scheduling

Aiming to avoid conflicts as well, Cassandra (KASI; SARMA, 2013) is an Eclipse plugin that extends Mylyn (KERSTEN; MURPHY, 2006) to recommend an optimum order of task execution per developer. Cassandra works as follows. First, in their own workspace, developers create the tasks they intend to work on and identify the files they suppose each task will change. Then, developers sort their tasks according to their preferences. Next, when developers require conflict analysis, Cassandra tool models and evaluates a set of constraints, which represent the conflict risk between tasks. As a result, Cassandra suggests, for each developer, a task execution sequence that satisfies the constraints. The constraints consider the files each task is supposed to edit (F_e), their dependent files (F_d), and the developer's preferred execution sequence. The logic is to avoid the concurrent development of tasks that are supposed to edit the same file, assuming they are more vulnerable to cause merge conflicts. In the same way, to avoid the concurrent development of tasks that are supposed to edit a dependent file of other tasks, assuming they are likely to cause a build or test failure. Throughout this process, Cassandra also tries to satisfy the

developer's preference when possible. In such context, developers provide both F_e and the preferred execution sequence, whereas Cassandra automatically identifies F_d by executing a call-graph analysis using Dependency Finder.[1] Finally, during task execution, Cassandra reevaluates the constraints when a developer completes a task.

Although promising, regarding task interfaces, Cassandra relies on the developer's expertise. Although we believe developers are indispensable for predicting task interfaces, merely asking them to guess the file set they intend to modify might be quite challenging and error-prone. Such limitation is critical for the purpose of avoiding conflicts, given that if the tasks actually do not change or use the planned files, task scheduling would not help the team.

In addition, Cassandra assumes all tasks have the same fixed duration (KASI, 2014), which is unrealistic. As a consequence, the continuous reevaluation of the entire constraints might cause overhead. Furthermore, Cassandra tries to automatically deal with the impossibility of satisfying all constraints by defining a prioritizing policy to favor merge conflicts or build and semantic conflicts.

Contrasting, instead of asking for developers to guess the files a task will change, we propose a strategy to automatically infer task interfaces. Our strategy also requires the developers upfront knowledge about task behavior, since they have to write automated acceptance tests. On the other hand, it seems that guessing the files related to a task is a more risky activity than writing acceptance tests. The reason is test design is a systematic activity that leads developers to analyse system requirements, imagine alternative situations, question decisions and so on. However, there is no protocol to guess the files related to a task, the activity is subjective and depends on the developer expertise. Thus, TAITI could maybe complement Cassandra by predicting task interfaces that developers can further refine, provided tests are associated with tasks.

In the same way, we dynamically compute the risk of conflicts and we delegate for developers the responsibility to define the execution sequence to tasks, as they select a new task to perform. Thus, the duration of tasks is not relevant for us. Also, the cost to evaluate conflict risk according to our strategy seems reduced, given we evaluate the risk between the planned tasks of a developer and the current tasks of other developers, whereas Cassandra considers all planned tasks of all developers. At last, we do not speculate the relevance of conflicts, we just provide support for developers to select the next task to perform, assuming all potential conflicts are relevant.

5.3.2 Predicting task interfaces

As previously discussed, we rely on task interfaces to identify potentially conflicting tasks. In this sense, we propose a strategy to predict task interfaces based on acceptance tests. The following subsections present alternative strategies for test-based task interfaces.

[1] <http://depfind.sourceforge.net/>

5.3.2.1 Assisted manual definition of task interfaces

The most straightforward strategy to define task interfaces is to ask developers to guess the file set she intends to use or modify to perform a task. However, the identification of task interfaces without any support is challenging, as previously argued. For alleviating such activity, some tools provide search mechanisms, primarily based on textual information. For example, *TopicXP* (SAVAGE et al., 2010) receives as input a query using keywords and outputs relevant files for a task based on their vocabulary and static dependencies from code. Nevertheless, the search effectiveness depends on the quality of the task descriptions, which must clarify the task purpose. Also, the search requires the extra effort of formulating a query. Contrasting, we propose the automatic inference of task interfaces based on acceptance tests that developers are supposed to use to validate features.

5.3.2.2 Partial automated prediction of task interfaces

As said before, Cassandra verifies dependencies among tasks by a call-graph analysis to evaluate conflict risk. In fact, according to Cataldo et al. (2009), violated dependencies often lead to software faults. In such a context, the call-graph analysis is a strategy to capture *syntactic dependencies*, but there are alternatives. For example, a dataflow relationship to identify different methods that modify and use the same data structure. Another alternative is the *concern scattering*, which means the dispersion of a system requirement implementation. Furthermore, according to Cataldo et al. (2009), the *logical dependencies* (also called *co-changes*) between source files, which express the semantic relationship between files that the developers change together, is even more relevant for faults occurrence than the syntactic dependencies.

As can be seen, although code dependencies might explain conflict occurrence, assuming that faults are a kind of conflict, they cannot infer task interfaces alone. The developers still need to identify the source files they intend to edit, that is, the start point of a task. That is the case of several studies (ZIMMERMANN et al., 2004; YING et al., 2004; DENNINGER, 2012; GIGER; PINZGER; GALL, 2012; BAILEY; LIN; SHERRELL, 2012) that try to predict code changes or bugs to support developers during the execution of maintenance tasks. As an illustration, the ROSE tool (ZIMMERMANN et al., 2004) tries to guide the development of an ongoing task by analyzing the project's version history to identify co-changed elements. However, the developer first inputs the code element he intends to change. For this reason, we realize that dependencies complement *TestI*, motivating us to conduct a preliminary study (SANTOS; ROCHA; BORBA, 2019) that extend the interfaces computed by TAITI by including the syntactic dependences of the application files reached by the tests. The results bring evidence that, for a reduced sample of 60 tasks, the extended interface improves recall by slightly compromising precision. Even so, note that the challenge for defining task interfaces remains, as the developers still need to identify a start point for the prediction.

5.3.2.3 Predicting task interfaces by retrospective analysis

Another possibility is to predict task interfaces based on the project's version history, assuming that similar tasks are likely to change or use the same code elements. For instance, if task T_A is most similar to a past task T_B, T_A will change or use the same code elements that T_B altered or used. Accordingly, developers must provide a task description.

For example, Hipikat (CUBRANIC et al., 2005) recommends artifacts for supporting developers to start a new task. As an illustration, if the developer inputs a request ticket as search criteria, Hipikat searches for textually similar tickets. Then, the developer can search for the commits related to the most similar ticket and use their changeset as the task interface.

Hipikat constructs a project memory by using different artifacts, such as bug reports and feature requests (ticket) recorded in an issue tracking system, source file versions checked into a version control system, messages posted on newsgroups and mailing lists, and documents posted on the project's web site. The ticket is the primary artifact, given it represents the task and connects all others. The relationship between artifacts relies on the artifact's author, the ticket ID (used in commit messages and forum messages), the creation time or edition time of an artifact, as well the similarity between task descriptions or between documents, and the historic of message responses. As a consequence, the developer can use any of these artifacts as search criteria, not only a ticket, as we illustrate.

Although past-based predictions have other successful applications, like the logical dependency between code elements we mentioned, the use of textual similarity to identify similar past tasks seems inappropriate, whether using natural language, due to its intrinsic imprecision. Therefore, according to the text volume and the vocabulary, developers might describe tasks related to the same feature as different tasks, and vice versa. In such a context, a structured description like high-level tests in Gherkin might be promising, as our results suggest. Also, the overall idea seems more useful for assisting developers in developing new code and tests according to a "learning by example" dynamics. By analyzing the artifacts of a similar past task, the developer might gain insight into the code of her new task.

Thompson & Murphy (THOMPSON; MURPHY, 2014) extended Hipikat idea to recommend artifacts that a set of similar past tasks changed or used during their execution, assuming the project' repository provide data about modified and used resources per task. In any case, the evaluation results of both ideas discourage us from trying them. Hipikat reports precision of 11%, and Thompson & Murphy, 21%. Another example is the *DebugAdvisor* tool (ASHOK et al., 2009), which helps developers to identify prior issues potentially relevant to a given bug, reducing the effort for bug fixing. DebugAdvisor applies the reasoning of Hipikat, but it also uses structured text like code and stack traces. Anyway, such tools require projects' historical data. Even though it might benefit our strategy as well, the requirement limits the usage of this solution to predict task interfaces.

5.3.3 Other related work

Regarding conflicts, some studies bring evidence that conflicts occur frequently and damage developers' productivity as well as software quality (ZIMMERMANN, 2007; BIRD; ZIMMERMANN, 2012; BRUN et al., 2013; KASI; SARMA, 2013). Others investigate the cause of conflict occurrence (CATALDO; HERBSLEB, 2011), whereas others assist in conflict detection and resolution. However, few of them focus on avoiding conflicts. Similarly, work on software development planning proposes solutions to improve software quality and the usage of project (material and human) resources, aiming at on-time software delivery and customer satisfaction. However, they do not discuss conflicting code changes and how they affect software development. The following subsections briefly present these other works.

5.3.3.1 Predicting merge conflicts

Some studies investigate merge conflicts to understand their cause, and support development teams to avoid them. 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 do not found evidence that the indicators apply for the whole sample, but only on a per-project basis.

Dias, Borba e Barreto (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 occurrence significantly increases when contributions to be merged involve files from the same MVC segment. Also, bigger contributions involving more developers, commits, and changed files are more likely associated with merge conflicts, as well as contributions developed over more 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 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, development teams might combine our strategy and other practices, such as frequently commit and integrate contributions, avoiding some indicators discussed by these other studies during task execution.

5.3.3.2 Merge tools

Conventional version control systems provide facilities during artifacts integration by automatically detecting conflicts and sometimes solving them by diff and merge mechanisms. Generally, merge tools identify text lines that the developers insert, extract, or change independently. Still, they do not deal with changes at the same line, neither identify syntactic or semantic conflicts caused by changes in related lines. Besides, they might

identify false conflicts that disturb developers, such as changes in statement order of Java code and alterations related to indent style. As a consequence, developers often need to solve conflicts manually. Tools like FSTMerge (APEL et al., 2011) and JDime (APEL; LESSENICH; LENGAUER, 2012) evolve the state of the practice tools to automate the resolution of some spurious conflicts and improve conflict detection by using language-specific strategies. As developers might change files not aligned with their programming tasks, as well as 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.

5.3.3.3 Awareness tools

Other tools assist developers in detecting conflicts during task execution. Their assumption is the early detection facilitates conflict resolution, as the conflict did not propagate into the code, and the relevant changes did not fade away in the developers' memory yet. The so-called *workspace awareness tools* (or *change awareness*) monitor the developers' workspace and notify them about ongoing changes that are potentially conflicting. Thus, these tools still rely on conflicts occurrence, even if it only happens on the developer's workspace.

For example, FastDash (BIEHL et al., 2007) provides visibility of current file editions by a dashboard. Adopting a fine-grained solution, CollabVS (DEWAN; HEGDE, 2007) and Palantír (SARMA; REDMILES; HOEK, 2012) notify developers when they are editing not only the same file but also when they are editing program elements that are syntactic dependent on each other. Nevertheless, unnecessary notification due to false conflicts might disturb developers. Crystal (BRUN et al., 2013) tries to overcome such a limitation by performing version control and test operations in background to check conflict occurrence before notifying developers, as the developers update their local repositories. The problem is the notification is not accurate. The developer knows his code has some conflict (merge, build, or test) with the code of another developer. Still, the developer does not know which conflicts are exactly occurring, meaning he has to interrupt his work and try to localize conflicts by himself. WeCode (aES; SILVA, 2012) extends Crystal by doing a single merge of all developments of a team working on the same branch in the background and reporting precise details of the conflicts affecting the team as a whole. Even so, WeCode considers uncommitted and committed changes, which can lead to numerous false positives.

Considering predictions are imprecise by definition as well as detection of conflicts when dealing with ongoing tasks, we understand our strategy and workspace awareness tools complement each other. By assisting developers in selecting a new task to perform, we might avoid conflicts. As a consequence, we expect to reduce alerts from workspace awareness tools. On the other hand, by notifying developers about conflicting ongoing changes, workspace awareness tool might reduce integration effort, promoting productivity

and software quality (assuming the risk of defects during integration is also reduced).

5.3.3.4 Task context

Mylyn (KERSTEN; 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 task execution. Instead, we intend to predict the task context before task execution. 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 neither predict that planned tasks might cause conflicts.

5.3.3.5 Development practices

Similarly to awareness tools, development practices such as Continuous Integration (FOWLER; FOEMMEL, 2006) and Continuous Delivery (ADAMS; MCINTOSH, 2016) also support early conflict detection. 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 the increasing of conflict complexity, as previously argued, developers still will have to spend time to solve conflicts.

Besides, the practice of code review (BACCHELLI; 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. Finally, some agile practices, such as daily stand-up meetings, might prevent conflicts by promoting communication and clarifying ongoing tasks. Even so, communication might be more imprecise (STRAY; SJØBERG; DYBå, 2016). That's why we prioritize an automatic solution aiming to promote developers productivity and effectiveness. So, although BDD principles could help code change prediction, we expect more benefits when BDD is used together with a tool that computes test-based task interfaces. Given these points, we conclude development practices complement our strategy.

5.3.3.6 Project scheduling

Despite being possible to reduce conflicts by prioritizing task execution according to conflict likelihood, other factors besides the parallel execution of tasks influences on conflicting code changes, such as the alignment among developer skill, task requirements, and projects' restrictions like duration and cost. Several studies discuss this problem, the so-called *Project Scheduling Problem (PSP)* (SHEN et al., 2016), aiming to deliver high-quality systems that satisfy customer needs and fit on the project budget. Specifically, the investigation concerns to find a solution that satisfies a set of constraints, such as the estimated effort required per task and the dependence value between tasks. In this sense, in the future, a promising strategy might be to extend TAITI to evaluate such factors to recommend tasks.

5.3.3.7 Next release planning

Similarly to the PSP problem, other studies investigate the selection of system requirements that the developers will implement for the next release according to logical and business constraints (the *Next Release Problem* (ALMEIDA et al., 2018; CARLSHAMRE et al., 2001)). In such a context, requirements' priority, interdependencies between requirements, costs, and customer satisfaction are relevant.

As an illustration, let us consider a bug fixing task. The number of comments on the bug report defines the requirement's priority, whereas the bug severity relates to the estimated cost for the requirement. Regarding interdependence, frequently, there is no precedence relationship between bugs. Then, an optimization algorithm tries to find the best solution (or the nearest one) that satisfies a mathematical expression that combines these measures and others related to the project, aiming to favor higher priority requirements without violating requirements' interdependencies, while balancing the other criteria. Requirements' interdependence means that the absence of a condition impairs the development of another requirement.

Although the interdependence between system requirements might lead to conflicts (CARLSHAMRE et al., 2001), this research field concerns to solve the optimization problem for balancing multiple factors that affect release planning. As a result, reducing conflicts is only a possible side effect. Nonetheless, the combined usage of a tool for assisting teams in planning tasks and TAITI to support decision making during task execution seems a promising idea to reduce conflicts.

REFERENCES

ACCIOLY, P.; BORBA, P.; CAVALCANTI, G. Understanding semi-structured merge conflict characteristics in open-source java projects. *Empirical Software Engineering*, 2017. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-017-9586-1>.

ADAMS, B.; MCINTOSH, S. Modern release engineering in a nutshell – why researchers should care. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.: s.n.], 2016. v. 5, p. 78–90.

aES, M. L. G.; SILVA, A. R. Improving early detection of software merge conflicts. In: *Proceedings of the 34th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 342–352. ISBN 978-1-4673-1067-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2337223.2337264>.

AHMED, I.; BRINDESCU, C.; MANNAN, U. A.; JENSEN, C.; SARMA, A. 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)*. [S.l.: s.n.], 2017. p. 58–67.

ALMEIDA, J. C.; PEREIRA, F. d. C.; REIS, M. V.; PIVA, B. The next release problem: Complexity, exact algorithms and computations. In: SPRINGER. *International Symposium on Combinatorial Optimization*. [S.l.], 2018. p. 26–38. ISBN 978-3-319-96151-4.

AN, J.-h.; CHAUDHURI, A.; FOSTER, J. Static typing for ruby on rails. In: . [S.l.: s.n.], 2009. p. 590–594.

APEL, S.; LESSENICH, O.; LENGAUER, C. Structured merge with auto-tuning: Balancing precision and performance. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2012. (ASE 2012), p. 120–129. ISBN 978-1-4503-1204-2. Disponível em: <http://doi.acm.org/10.1145/2351676.2351694>.

APEL, S.; LIEBIG, J.; BRANDL, B.; LENGAUER, C.; KÄSTNER, C. 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*. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 190–200. ISBN 978-1-4503-0443-6. Disponível em: <http://doi.acm.org/10.1145/2025113.2025141>.

ASHOK, B.; JOY, J.; LIANG, H.; RAJAMANI, S. K.; SRINIVASA, G.; VANGALA, V. Debugadvisor: a recommender system for debugging. In: ACM. *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. [S.l.], 2009. p. 373–382.

BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. In: IEEE PRESS. *Proceedings of the 2013 international conference on software engineering*. [S.l.], 2013. p. 712–721.

BAILEY, M.; LIN, K.-I.; SHERRELL, L. Clustering source code files to predict change propagation during software maintenance. In: *Proceedings of the 50th Annual Southeast*

Regional Conference. New York, NY, USA: ACM, 2012. (ACM-SE '12), p. 106–111. ISBN 978-1-4503-1203-5. Disponível em: <http://doi.acm.org/10.1145/2184512.2184538>.

BALDWIN, C. Y. *Design Rules: The Power of Modularity*. [S.l.]: The MIT Press, 2000. ISBN 9780262267649.

BASS, L.; WEBER, I.; ZHU, L. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015. ISBN 0134049845, 9780134049847. Disponível em: <http://cds.cern.ch/record/2034028>.

BIEHL, J. T.; CZERWINSKI, M.; SMITH, G.; ROBERTSON, G. G. Fastdash: A visual dashboard for fostering awareness in software teams. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2007. (CHI '07), p. 1313–1322. ISBN 978-1-59593-593-9. Disponível em: <http://doi.acm.org/10.1145/1240624.1240823>.

BINAMUNGU, L. P.; EMBURY, S. M.; KONSTANTINOU, N. Detecting duplicate examples in behaviour driven development specifications. In: *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. [S.l.: s.n.], 2018. p. 6–10.

BIRD, C.; ZIMMERMANN, T. Assessing the value of branches with what-if analysis. In: ACM. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. [S.l.], 2012. p. 45.

BORICI, A.; BLINCOE, K.; SCHRÖTER, A.; VALETTO, G.; DAMIAN, D. Proxiscientia: Toward real-time visualization of task and developer dependencies in collaborating software development teams. In: *Proceedings of the 5th International Workshop on Co-operative and Human Aspects of Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012. (CHASE '12), p. 5–11. ISBN 978-1-4673-1824-2. Disponível em: <http://dl.acm.org/citation.cfm?id=2663638.2663641>.

BRIAND, L.; BIANCULLI, D.; NEJATI, S.; PASTORE, F.; SABETZADEH, M. The case for context-driven software engineering research: Generalizability is overrated. *IEEE Software*, v. 34, n. 5, p. 72–75, 2017. ISSN 0740-7459.

BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Early detection of collaboration conflicts and risks. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 39, n. 10, p. 1358–1375, out. 2013. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2013.28>.

BRUN, Y.; MUŞLU, K.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Predicting development trajectories to prevent collaboration conflicts. In: *The Future of Collaborative Software Development*. Bellevue, WA, USA: [s.n.], 2012.

CARLSHAMRE, P.; SANDAHL, K.; LINDVALL, M.; REGNELL, B.; DAG, J. N. och. An industrial survey of requirements interdependencies in software product release planning. In: IEEE. *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. [S.l.], 2001. p. 84–91.

CATALDO, M.; HERBSLEB, J. D. Factors leading to integration failures in global feature-oriented development: An empirical analysis. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA:

ACM, 2011. (ICSE '11), p. 161–170. ISBN 978-1-4503-0445-0. Disponível em: <http://doi.acm.org/10.1145/1985793.1985816>.

CATALDO, M.; MOCKUS, A.; ROBERTS, J. A.; HERBSLEB, J. D. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, v. 35, n. 6, p. 864–878, 2009. ISSN 0098-5589.

CAVALCANTI, G.; BORBA, P.; ACCIOLY, P. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.*, ACM, New York, NY, USA, v. 1, n. OOPSLA, p. 59:1–59:27, out. 2017. ISSN 2475-1421. Disponível em: <http://doi.acm.org/10.1145/3133883>.

COHN, M. *User stories applied: For agile software development*. [S.l.]: Addison-Wesley Professional, 2004.

CUBRANIC, D.; MURPHY, G. C.; SINGER, J.; BOOTH, K. S. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 31, n. 6, p. 446–465, jun. 2005. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2005.71>.

DENNINGER, O. Recommending relevant code artifacts for change requests using multiple predictors. In: *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012. (RSSE '12), p. 78–79. ISBN 978-1-4673-1759-7. Disponível em: <http://dl.acm.org/citation.cfm?id=2666719.2666737>.

DEWAN, P.; HEGDE, R. Semi-synchronous conflict detection and resolution in asynchronous software development. In: *ECSCW 2007*. [S.l.]: Springer, 2007. p. 159–178.

DIAS, K.; BORBA, P.; BARRETO, M. Understanding predictive factors for merge conflicts. *Information and Software Technology*, Elsevier, 2020.

DIAS, M.; BACCHELLI, A.; GOUSIOS, G.; CASSOU, D.; DUCASSE, S. Untangling fine-grained code changes. In: IEEE. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.], 2015. p. 341–350.

FOWLER, M. *Feature Branch*. 2009. <https://martinfowler.com/bliki/FeatureBranch.html>. Accessed: April 2018.

FOWLER, M. *Feature Toggle*. [S.l.], 2016. Disponível em: <https://martinfowler.com/bliki/FeatureToggle.html>.

FOWLER, M.; FOEMMEL, M. *Continuous integration*. [S.l.], 2006. 122 p.

FURR, M.; AN, J.-h. D.; FOSTER, J. S.; HICKS, M. Static type inference for ruby. In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2009. (SAC '09), p. 1859–1866. ISBN 9781605581668. Disponível em: <https://doi.org/10.1145/1529282.1529700>.

GIGER, E.; PINZGER, M.; GALL, H. C. Can we predict types of code changes? an empirical analysis. In: *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. [S.l.: s.n.], 2012. p. 217–226. ISSN 2160-1852.

GRINTER, R. E. Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work*, Kluwer Academic Publishers, 1996.

HENDERSON, F. *Software Engineering at Google*. [S.l.], 2017. Disponível em: <arXiv:1702.01715>.

HERZIG, K.; ZELLER, A. The impact of tangled code changes. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2013. (MSR '13), p. 121–130. ISBN 978-1-4673-2936-1. Disponível em: <http://dl.acm.org/citation.cfm?id=2487085.2487113>.

HODGSON, P. *Feature Branching vs. Feature Flags: What's the Right Tool for the Job?* [S.l.], 2017. Disponível em: <https://devops.com/feature-branching-vs-feature-flags-whats-right-tool-job/>.

HODGSON, P. *Feature Toggles (aka Feature Flags)*. 2017. <https://martinfowler.com/articles/feature-toggles.html>. Accessed: April 2018.

KASI, B. K. Minimizing software conflicts through proactive detection of conflicts and task scheduling. In: ACM. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.], 2014. p. 807–810.

KASI, B. K.; SARMA, A. Cassandra: Proactive conflict minimization through optimized task scheduling. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 732–741. ISBN 978-1-4673-3076-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2486884>.

KERSTEN, M.; MURPHY, G. C. Using task context to improve programmer productivity. In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2006. (SIGSOFT '06/FSE-14), p. 1–11. ISBN 1-59593-468-5. Disponível em: <http://doi.acm.org/10.1145/1181775.1181777>.

LESSENICH, O.; SIEGMUND, J.; APEL, S.; KÄSTNER, C.; HUNSEN, C. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering*, v. 25, n. 2, p. 279–313, 2018.

MCKEE, S.; NELSON, N.; SARMA, A.; DIG, D. Software practitioner perspectives on merge conflicts and resolutions. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2017. p. 467–478.

MIRANDA, S.; VALENTE, M. T.; TERRA, R. Inferência de tipos em ruby: Uma comparação entre técnicas de análise estática e dinâmica. In: *IV Workshop de Visualização, Evolução e Manutenção de Software (VEM)*. [S.l.: s.n.], 2016. p. 105–112.

NAGAPPAN, M.; ZIMMERMANN, T.; BIRD, C. Diversity in software engineering research. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013. (ESEC/FSE 2013), p. 466–476. ISBN 978-1-4503-2237-9. Disponível em: <http://doi.acm.org/10.1145/2491411.2491415>.

PERRY, D. E.; SIY, H. P.; VOTTA, L. G. Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 10, n. 3, p. 308–337, 2001.

POTVIN, R.; LEVENBERG, J. Why google stores billions of lines of code in a single repository. *Communications of the ACM*, v. 59, p. 78–87, 2016. Disponível em: <http://dl.acm.org/citation.cfm?id=2854146>.

ROCHA, T.; BORBA, P. *Online Appendix related to prediction of merge conflict risk*. 2020. <https://thaisabr.github.io/conflict-risk-prediction-study-site/>. Accessed: February 2020.

ROCHA, T.; BORBA, P.; SANTOS, J. *Online Appendix related to file change prediction*. 2018. <https://thaisabr.github.io/task-interface-study-site/>. Accessed: April 2018.

ROCHA, T.; BORBA, P.; SANTOS, J. P. Using acceptance tests to predict files changed by programming tasks. *Journal of Systems and Software*, v. 154, p. 176–195, 2019.

ROUNTEV, A.; YAN, D. Static reference analysis for gui objects in android software. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. New York, NY, USA: Association for Computing Machinery, 2014. (CGO '14), p. 143–153. ISBN 9781450326704. Disponível em: <https://doi.org/10.1145/2581122.2544159>.

SALTON, G.; MCGILL, M. J. *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, Inc., 1986. ISBN 0070544840.

SANTOS, J. a. P.; ROCHA, T.; BORBA, P. Improving the prediction of files changed by programming tasks. In: *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*. New York, NY, USA: ACM, 2019. (SBCARS '19), p. 53–62. ISBN 978-1-4503-7637-2. Disponível em: <http://doi.acm.org/10.1145/3357141.3357145>.

SARMA, A.; REDMILES, D.; HOEK, A. van der. Palantír: Early detection of development conflicts arising from parallel code changes. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 38, n. 4, p. 889–908, jul. 2012. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/TSE.2011.64>.

SAVAGE, T.; DIT, B.; GETHERS, M.; POSHYVANYK, D. Topicxp: Exploring topics in source code using latent dirichlet allocation. In: *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2010. (ICSM '10), p. 1–6. ISBN 978-1-4244-8630-4. Disponível em: <http://dx.doi.org/10.1109/ICSM.2010.5609654>.

SHEN, X.; MINKU, L. L.; BAHSOON, R.; YAO, X. Dynamic software project scheduling through a proactive-rescheduling method. *IEEE Transactions on Software Engineering*, v. 42, n. 7, p. 658–686, 2016. ISSN 0098-5589.

SMART, J. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications Company, 2014. ISBN 9781617291654. Disponível em: <https://books.google.com.br/books?id=2BGxngEACAAJ>.

SOUZA, C. R. B. de; REDMILES, D.; DOURISH, P. "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*. [S.l.]: ACM, 2003. (GROUP '03), p. 105–114.

STRAY, V.; SJøBERG, D. I.; DYBå, T. The daily stand-up meeting: A grounded theory study. *Journal of Systems and Software*, v. 114, p. 101 – 124, 2016. ISSN 0164-1212. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121216000066>.

THOMPSON, C. A.; MURPHY, G. C. Recommending a starting point for a programming task: An initial investigation. In: *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering*. New York, NY, USA: ACM, 2014. (RSSE 2014), p. 6–8. ISBN 978-1-4503-2845-6. Disponível em: <http://doi.acm.org/10.1145/2593822.2593824>.

WILCOXON, F.; WILCOX, R. A. *Some rapid approximate statistical procedures*. Lederle Laboratories, 1964. Disponível em: <https://books.google.com.br/books?id=aBU8AAAAIAAJ>.

YING, A. T. T.; MURPHY, G. C.; NG, R.; CHU-CARROLL, M. C. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, v. 30, n. 9, p. 574–586, Sept 2004. ISSN 0098-5589.

ZIMMERMANN, T. Mining workspace updates in cvs. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007. (MSR '07), p. 11–. ISBN 0-7695-2950-X. Disponível em: <http://dx.doi.org/10.1109/MSR.2007.22>.

ZIMMERMANN, T.; WEISGERBER, P.; DIEHL, S.; ZELLER, A. Mining version histories to guide software changes. In: *Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004. (ICSE '04), p. 563–572. ISBN 0-7695-2163-0. Disponível em: <http://dl.acm.org/citation.cfm?id=998675.999460>.