The Journal of Systems and Software 154 (2019) 176–195

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Using acceptance tests to predict files changed by programming tasks

Thaís Rocha*, Paulo Borba, João Pedro Santos

Informatics Center, Federal University of Pernambuco, Recife, Brazil

ARTICLE INFO

Article history:
Received 5 November 2018
Revised 22 April 2019
Accepted 24 April 2019
Available online 30 April 2019

Keywords:
Collaborative development
Task scheduling
Behaviour-driven development
File change prediction

ABSTRACT

In a collaborative development context, conflicting code changes might compromise software quality and developers productivity. To reduce conflicts, one could avoid the parallel execution of potentially conflicting tasks. Although hopeful, this strategy is challenging because it relies on the prediction of the required file changes to complete a task. As predicting such 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 infers test-based interfaces (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 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. A hybrid approach for computing test-based interfaces is promising.

© 2019 Elsevier Inc. All rights reserved.

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 non small 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 et al., 2001; Zimmermann, 2007; Brun et al., 2013a; Kasi and Sarma, 2013). This has been observed even when using advanced merge tools (Apel et al., 2011; 2012; Cavalcanti et al., 2017; Accioly et al., 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 et al., 2012; Bird and 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 et al., 2012; Grinter, 1996), and partial check-ins (de Souza et al., 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 and McIntosh, 2016a; Henderson, 2017; Potvin and Levenberg, 2016) and feature toggles (Bass et al., 2015; Adams and McIntosh, 2016a; 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 infer which files will be changed by a given task. Automatically predicting such code changes, in general, is also hard. It's worth, though, to investigate whether this kind of automatic prediction is feasible for specific contexts (Briand et al., 2017). In particular, in a BDD (Smart, 2014) 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 infer parts of the application code that might be

* Corresponding author.
E-mail addresses: tabr@cin.ufpe.br (T. Rocha), phmb@cin.ufpe.br (P. Borba), jpms2@cin.ufpe.br (J.P. Santos).

https://doi.org/10.1016/j.jss.2019.04.060
0164-1212/© 2019 Elsevier Inc. All rights reserved.

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 build a 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 from 18 Ruby on Rails (2019) projects that use Cucumber (Wynne and Hellesøy, 2012) 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*).

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 that *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.

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 executed to complement the interface and improve recall. Concerning conflict avoidance, our results suggest that we should consider a notion of interface with weighted elements. Files inferred by analyzing certain parts of the test, such as the test setup, should have less importance than files inferred from other parts. Similarly, we should explore a hybrid solution involving *TestI* and *TextI*. Knowing that *TextI* is more precise than *TestI*, we might consider that if both *TestI* and *TextI* include a file, such a file is more likely to change and consequently to cause conflicts.

The rest of the paper is organized as follows. Section 2 illustrates how *TestI* could be used to avoid conflicts. Section 3 presents the *TestI* inference tool. Section 4 describes the empirical study and Section 5 provides information about the collection of our tasks sample. In Section 6 we present and discuss results. Finally, Section 7 discusses the threats to the validity of our study, Section 8 presents related work, and Section 9 brings final considerations. All data about the empirical study, as well our tool for computing *TestI*, and other complementary scripts are in our online appendix Rocha et al. (2019).

2. Motivating example

To illustrate how test-based task interfaces might be useful to predict changes and avoid conflicts, let's consider that Adam and Betty are members of an agile team that is developing a Rails (Ruby on Rails, 2019) 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).

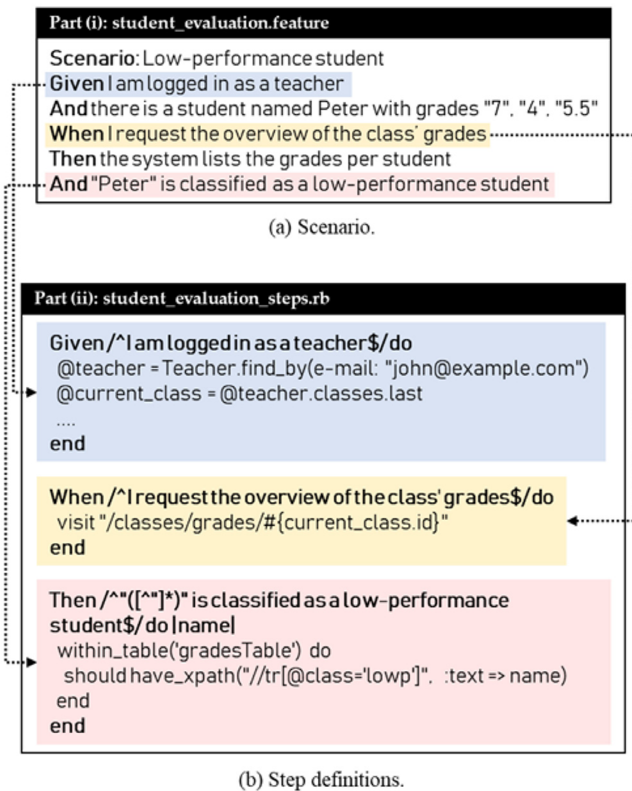
By independently working on T_1 and T_2 in their private repositories, Adam and Betty add different methods (`compute_grade_average` and `low_perf_students`) at the end of copies of the same file (app/controllers/classes_controller.rb), and change the same area of the class visualization page (app/views/classes/grades.html.erb), 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's 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 slice (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 (Smart, 2014) 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 Fig. 1 the Cucumber (Wynne and Hellesøy, 2012) automated test related to task T_2. The test has two parts: (i) a high-level concrete usage scenario written in (Gherkin, 2019), 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 the paper we use the term "Cucumber test" refering to the whole automated test, whereas "Gherkin scenarios" means the high-level usage scenario only.

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

[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.



(a) Scenario.

(b) Step definitions.

Fig. 1. Example of an automated acceptance test related to task T_2.

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 Fig. 2. 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 Fig. 2), 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.

ITest(T_1) and ITest(T_2)
app/models/class.rb
app/models/student.rb
app/models/teacher.rb
app/controllers/classes_controller.rb
app/views/classes/grades.html.erb

ITest(T_3)
app/models/principal.rb
app/models/notice.rb
app/controllers/notices_controller.rb
app/views/notices/index.html.erb
app/views/notices/new.html.erb
app/views/noticies/_form.html.erb

Fig. 2. Test-based task interfaces of tasks T_1, T_2, and T_3. The files in bold are the ones actually changed by developers.

Finally, someone might wonder 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[2] of a project on GitHub. The task was concluded by one commit that changed 15 application files (whether we consider the type file 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 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 et al., 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. 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[3] 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.

This way, any project can use our tool whereas 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 Fig. 3, and we provide information about its implementation. In 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

[2] https://github.com/AgileVentures/WebsiteOne/commit/ab1723df7878152b4a814fa5fbfe86c7076aecab.

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

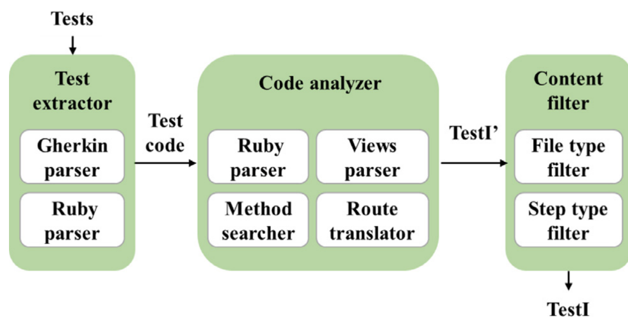


Fig. 3. TAITI architecture.

a initial version of *TestI*. Then, the *content filter* module refines *TestI* by filtering files according to different strategies.

3.1. 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. 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, 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 Section 3.3. 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'')`[4] 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[5] 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`.

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 there is a overridden library method. TAITI considers that the overridden 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 autogenerated 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.

3.3. 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 (2019), 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 Fig. 1 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

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

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

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 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 the paper.

3.4. 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.

3.5. 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.

3.5.1. Filtering by step type

Given different tests require similar `Given` (setup) steps, 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 Fig. 2, our tool would yield the interface with just the two files in bold at the left side of the figure.

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

To support the WF filter, our tool relies on algorithms 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.

3.5.2. Filtering by file type

As many tasks in the context we consider focus on changing a single MVC slice (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 slices. So, considering that controller files uniquely identify slices, 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 Fig. 2, our tool would yield the interface with just the first file in bold at the left side of the figure.

3.5.3. 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.

4. 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.

4.1. 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.

4.2. 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 3.3) 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.

4.3. 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 Section 8). 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.

4.4. 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 5.5 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 and McGill, 1986). In our context, the task specification is the Cucumber usage scenario written in Gherkin (as in the first part of Fig. 1), 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]

5. Study setup

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

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

5.1. 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, 2019) or (Coveralls, 2019), 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[8] we implemented to query GitHub's database using (GitHub Java API, 2019). 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 use 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 the Ruby projects referenced by Cucumber's site (Cucumber repository, 2019). As a result of this mining phase, we have a set of 61 selected Rails projects that use Cucumber, and a subset of 18 projects that additionally use the mentioned coverage tools.

5.2. 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 (JGit, 2019), 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 don't 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.

5.3. 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[9] 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) 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 6.1. 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.

[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] Available in our online Appendix Rocha et al. (2019).

[9] Please see our Appendix Rocha et al. (2019).

Table 2
Construction of the smaller sample.

Git Repository Name	Tasks	Cucumber and coverage tasks	Problem to compute *TestI*	Tests execution	Rails routes	Selected tasks
AgileVentures/MetPlus_PETS	231	13	YES	–	–	0
AgileVentures/WebsiteOne	1122	970	NO	PASSED	PASSED	10
alphagov/whitehall	1620	1202	NO	PASSED	PASSED	10
BTHUNTERCN/bsmi	193	193	NO	FAILED	–	0
diaspora/diaspora	1428	583	NO	PASSED	PASSED	10
iboard/CBA	297	138	NO	FAILED	–	0
moneyadviceservice/wpcc	47	46	NO	PASSED	FAILED	0
oneclickorgs/one-click-orgs	123	61	NO	PASSED	PASSED	10
opf/openproject	2567	2529	YES	–	–	0
otwcode/otwarchive	2602	1374	NO	PASSED	PASSED	10
rapidftr/RapidFTR	372	63	NO	PASSED	PASSED	10
TheOdinProject/theodinproject	44	44	NO	PASSED	PASSED	5
tip4commit/tip4commit	29	29	NO	PASSED	PASSED	7
TracksApp/tracks	89	67	NO	PASSED	PASSED	2
twers/re-education	1	1	NO	FAILED	–	0
Total	10,765	7,313	–	–	–	74

5.4. General exclusion criteria

Aiming to fairly evaluate task interfaces, we need to select 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. For such reason, we apply the following exclusion criteria before answering the research questions. First, we discard tasks 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 5.1). 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.

5.5. 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).

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).

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 et al., 2013), by inspecting our samples we observe some degree of diversity with respect to the dimensions in Table 5.[10] Although two projects have no stars, they are not toy

[10] Information collected on January 2019.

Table 3
Construction of the larger sample.

Git Repository Name	*TestI*			
	Entry tasks	Candidate tasks	Valid tasks	No empty *TextI*
8bitpal/hackful	7	4	4	3
AgileVentures/MetPlus_PETS	231	62	2	1
AgileVentures/WebsiteOne	1122	332	77	66
alphagov/whitehall	1620	370	215	175
BTHUNTERCN/bsmi	193	59	0	–
concord-consortium/rigse	399	49	0	–
cul-it/blacklight-cornell	464	102	32	27
daqing/rabel	4	1	0	–
dchbx/enroll	2903	399	0	–
diaspora/diaspora	1428	339	55	49
EFForg/action-center-platform	92	27	18	13
hashrocket/hr-til	6	2	0	–
iboard/CBA	297	41	21	11
jasonroelofs/raidit	2	2	0	–
jpatel531/folioapp	22	10	6	5
makrio/makrio	434	148	0	–
moneyadviceservice/wpcc	47	22	0	–
oneclickorgs/one-click-orgs	123	31	30	28
opengovernment/opengovernment	6	3	3	2
opf/openproject	2567	237	0	–
otwcode/otwarchive	2602	779	33	26
qiushibaike/moumentei	2	2	2	1
rails3book/ticketee	3	3	2	0
RailsApps/rails3-devise-rspec-cucumber	1	0	0	–
rapidftr/RapidFTR	372	67	48	43
sameersharma25/time_stack	17	6	6	5
TheOdinProject/theodinproject	44	31	4	3
tip4commit/tip4commit	29	3	3	1
TracksApp/tracks	89	38	5	4
twers/re-education	1	1	1	–
wearefriday/spectre	2	1	1	–
TOTAL	15,129	3171	568	463

Table 4
Tasks distribution per sample and project.

Git Repository Name	Smaller sample	Larger sample
8bitpal/hackful	0	3
AgileVentures/MetPlus_PETS	0	1
AgileVentures/WebsiteOne	10	66
alphagov/whitehall	10	175
cul-it/blacklight-cornell	0	27
diaspora/diaspora	10	49
EFForg/action-center-platform	0	13
iboard/CBA	0	11
jpatel531/folioapp	0	5
oneclickorgs/one-click-orgs	10	28
opengovernment/opengovernment	0	2
otwcode/otwarchive	10	26
qiushibaike/moumentei	0	1
rapidftr/RapidFTR	10	43
sameersharma25/time_stack	0	5
TheOdinProject/theodinproject	5	3
tip4commit/tip4commit	7	1
TracksApp/tracks	2	4
Total	74	463

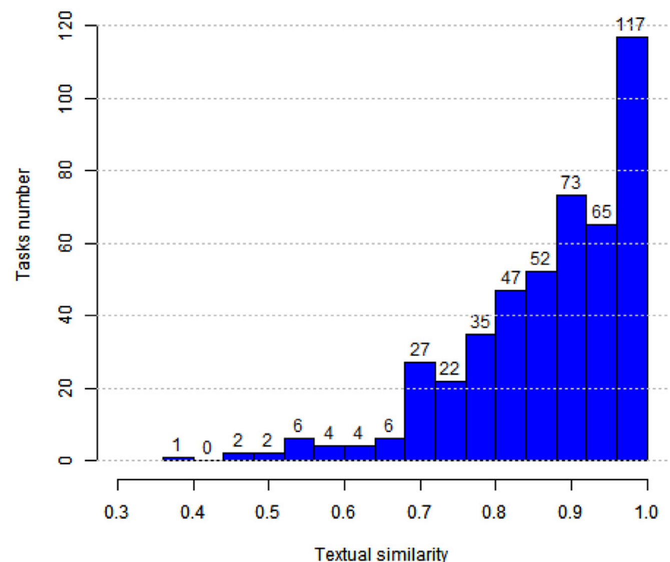


Fig. 4. Distribution of textual similarity related to the larger sample.

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 paper writing), but our results concern to historical data. Further information about our sample appears in the Appendix Rocha et al. (2019), including the complete name of git repositories.

For better characterizing our samples, in the following we present some complementary data. On average the tasks from the smaller sample contain $42.09 \pm 93.48(9)$[11] 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.

Specifically related to *TextI*, the histogram of Fig. 4 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 Section 3. “Ambiguous steps” is the number of steps that match with more

[11] 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	4168	9	18	155	10
MetPlus_PETS	A platform for searching and announcing job opportunities.	19	54,306	127	65	1935	46
WebsiteOne	A platform for promoting the agile methodology by the development of solutions to IT charities and nonprofits.	115	552,582	391	222	5556	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	5918	38
diaspora	A privacy-aware, distributed, open source social network.	12,126	190,259	265	2912	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	1172	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	2496	25
opengovernment	An application for aggregating and presenting US open government data.	200	13,431	11	117	2231	20
otwarchive	An application for hosting archives of fanworks, including fanfic, fanart, and fan vids.	368	289,490	1193	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	4939	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	2843	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	4056	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	$9.62 \pm 11.19(6)$

than one step definition, which possibly inflates *TestI* content, as explained in Section 3.1. “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 Section 3.2. 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. Section 3.3 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 did not can parse, as described in Section 3.4. The tests of our sample did not use a wildcard as step type, causing no noise related to WF filter (Section 3.5.1).

6. 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 and 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.

Table 7
Noise caused by TAITI.

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

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.

6.1. RQ1: how often does TestI predict file changes associated with a task?

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

Considering the larger sample of 463 tasks, *TestI-CF* recall is $0.48 \pm 0.32(0.45)$ (see Fig. 5),[12] 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 its 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 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

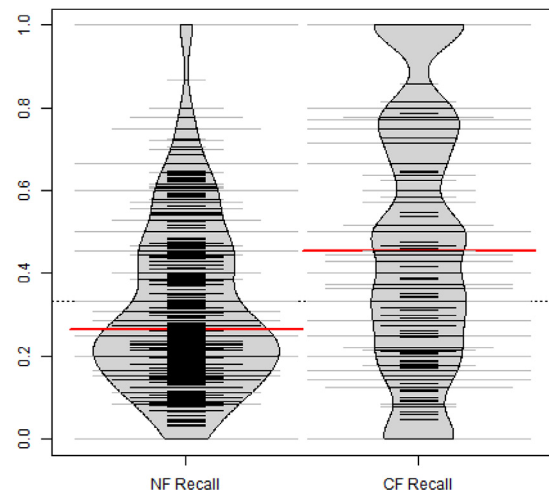


Fig. 5. Beanplots describing the recall value of *TestI-NF* and *TestI-CF* per task from the larger sample.

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 slice (related model, view, controller, and auxiliary files), one could also use *TestI-CF* to predict changes in slices. This is reinforced by the observation that, for 97% of the tasks in our sample, controllers predict changes in corresponding slice files; that is, when a controller appears in *TestI-CF, TaskI* quite often contains at least one file from the associated slice. So, in principle, *TestI-CF* could be used to reduce conflicts by avoiding the parallel execution of tasks that focus on common slices. 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 Fig. 6 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.

Under the slice perspective, we observe that, on average, 68% ($0.68 \pm 0.32(0.75)$) of the slices inferred from the controllers in

[12] Beanplots appear in the online appendix Rocha et al. (2019) for this and other results.

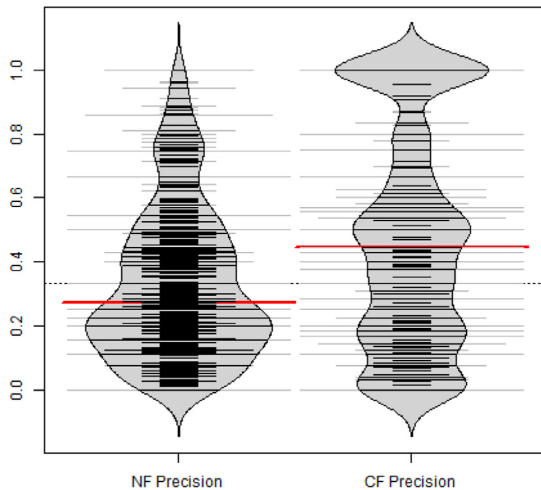


Fig. 6. Beanplots describing the precision value of *TestI-NF* and *TestI-CF* per task from the larger sample.

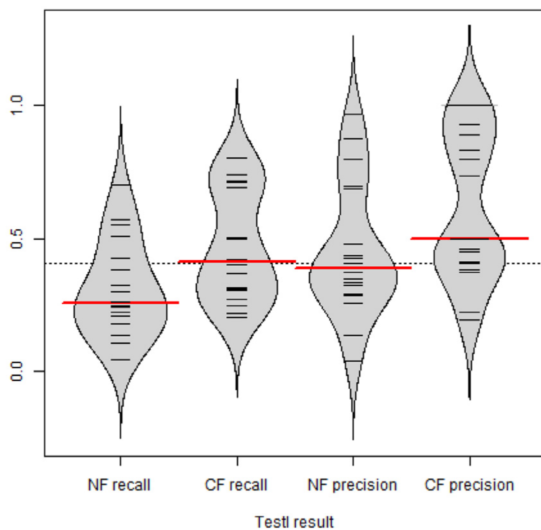


Fig. 7. Beanplots describing the results of *TestI-NF* and *TestI-CF* per project from the larger sample.

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 Fig. 7 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 deeper investigate 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 has 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 3.3) 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 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:

$$coverage(t) = \frac{|DTestI(t) \cap TaskI(t)|}{|TaskI(t)|} \quad (1)$$

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 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 2), 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 slices*

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 Section 3 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 et al. (2019), 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.

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

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 Fig. 8 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*.

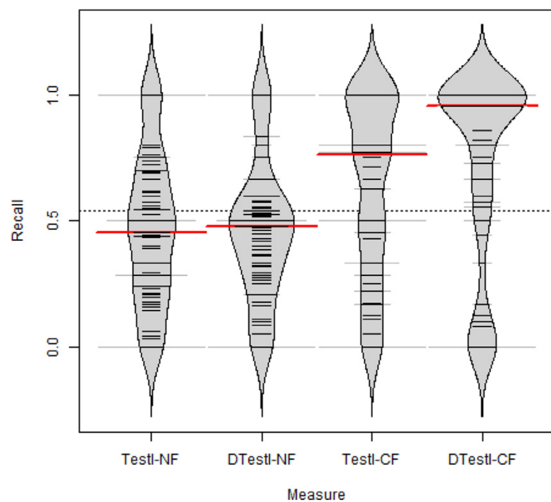


Fig. 8. Beanplots describing the recall value of *DTestI* per task from the smaller sample.

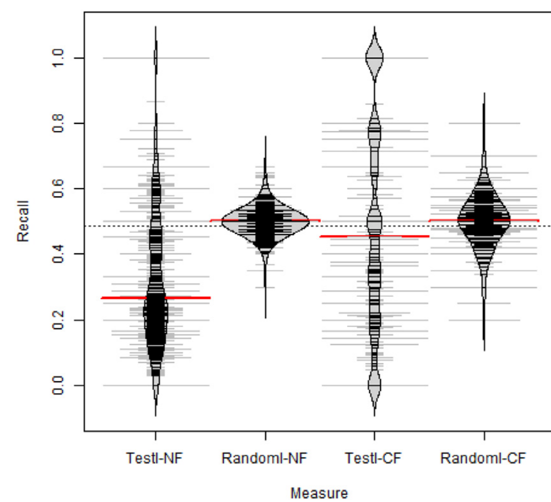


Fig. 9. Beanplots describing the recall value of *RandomI* per task from the larger sample.

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 doesn't compromise *TestI* improves its prognostics.

6.3. RQ3: is TestI *a better code change predictor than* RandomI*?*

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

We found no statistically significant difference between *TestI-CF* and *RandomI-CF* mean recalls for the larger sample ($p = 0.10$) (see Fig. 9). *TestI-CF* mean precision, however, is 69.7% higher than *RandomI-CF* precision (see Fig. 10): the first is $0.47 \pm 0.34(0.44)$, whereas the second is $0.28 \pm 0.27(0.17)$ ($p < 0.001$, $r = 0.72$). 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).

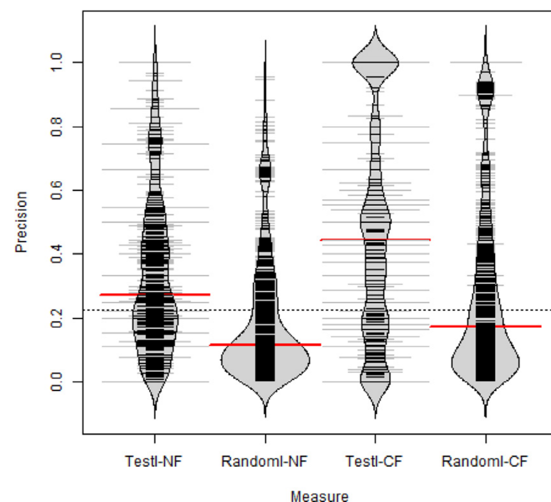


Fig. 10. Beanplots describing the precision value of *RandomI* per task from the larger sample.

These results reinforce our previous conclusion that *TestI* might help to predict changes in controllers and MVC slices. 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

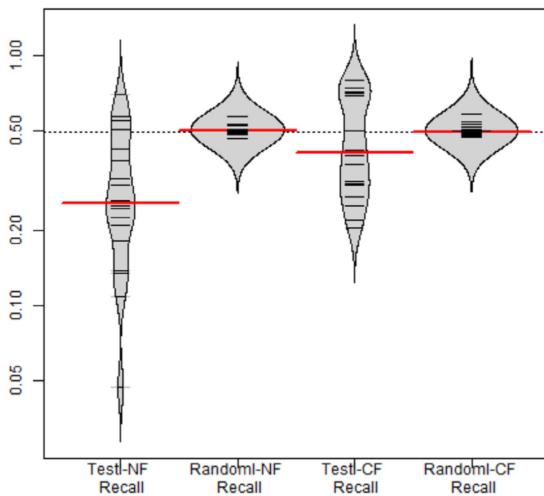


Fig. 11. Beanplots describing the recall value of *TestI* and *RandomI* per project from the larger sample.

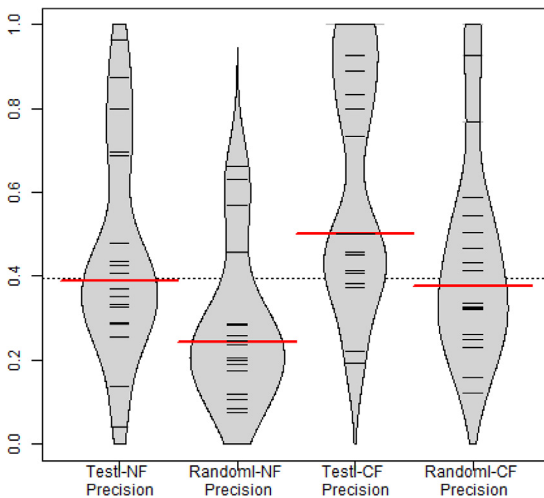


Fig. 12. Beanplots describing the precision value of *TestI* and *RandomI* per project from the larger sample.

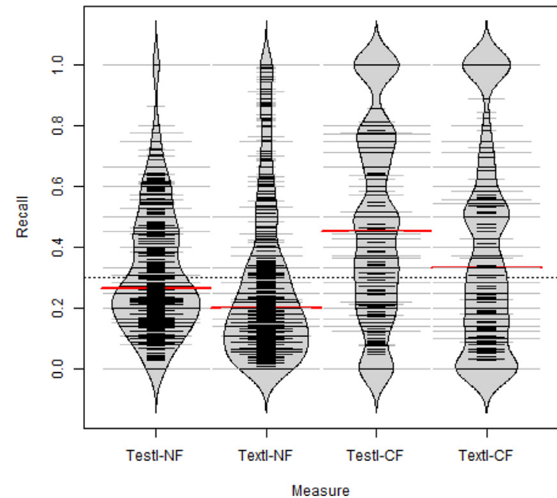


Fig. 13. Beanplots describing the recall value of *TextI* per task from the larger sample.

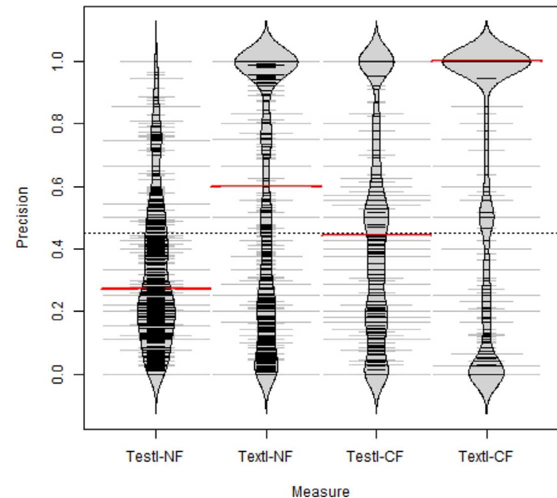


Fig. 14. Beanplots describing the precision value of *TextI* per task from the larger sample.

measure is intermediate (around 0.50) for all projects, as Fig. 11 illustrates.

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 Fig. 12 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.

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

TestI *has better recall but inferior precision than* TextI

As Fig. 13 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 Fig. 14): 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, 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 Fig. 4 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 7.2. 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) 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 et al., 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.

7. Threats to validity

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

7.1. Construct validity

Section 5.2 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.

Table 8
Average precision per project of the larger sample.

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

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

7.2. 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 and 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) 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 5.4), 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. 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.

7.3. 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. 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.

8. Related work

Many studies provide support to collaborative development. For example, tools like FSTMerge (Apel et al., 2011) and JDime (Apel et al., 2012) try to decrease integration effort by automatically solving or avoiding some spurious merge conflicts reported by state of the practice tools. Others assist developers to early detect conflicts during task execution aiming to facilitate conflict resolution, such as FastDash (Biehl et al., 2007), CollabVS (Dewan and Hegde, 2007), Palantír (Sarma et al., 2012), Crystal (Brun et al., 2013b), and WeCode (Guimarães and Silva, 2012). Basically, these workspace awareness tools monitor the developers' workspace and notify them about ongoing changes that are potentially conflicting. Similarly, development practices such as Continuous Integration (Fowler, 2009) and Continuous Delivery (Adams and McIntosh, 2016b) also support early conflict detection, as the code is frequently integrated, verified by build and test scripts, and released to production. Although useful, these studies still rely on conflicts occurrence (even if it only happens on the developer's workspace). In a complementary perspective, we are focused on avoiding conflicts by assisting developers to strategically select a new task to work on.

Aiming to avoid conflicts as well, Cassandra (Kasi and Sarma, 2013) is a tool that analyzes a set of constraints to recommend an optimum order of task execution per developer. The constraints relate to the files each task is supposed to edit according to the developers, their dependent files that are identified by call-graph analysis, and the developer's preferred sequence for executing tasks. Regarding task interfaces, the main topic of this work, 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. Our tool, TAITI, supported developers by computing test based interfaces that they can further refine, provided tests are associated with tasks. In this way, TAITI could maybe complement Cassandra and help avoid conflicts.

Alternatively, developers might use a search tool to assist them while defining task interfaces manually. 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.

As a tool to improve productivity, Mylyn (Kersten and Murphy, 2006) monitors developers' workspace to track relevant resources (e.g., selected or edited files) and updates the IDE accordingly. In this sense, by using a prioritizing policy for resources based on user interaction, Mylyn delineates a *task context* during task execution. Instead, we intend to infer the task context before task execution.

Other 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. 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 task interface. Although past-based predictions have other successful applications, the use of textual similarity to identify similar past tasks seems inappropriate whether using natural language due to its intrinsic imprecision. In such 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 to develop new code and tests 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. Finally, Hipikat requires projects' historical data. Even though it might benefit our strategy as well, the requirement limits the usage of this solution to predict task interfaces.

As said before, Cassandra verifies dependencies among tasks by a call-graph analysis to evaluate conflict risk. In fact, violated dependencies often lead to software faults (Cataldo et al., 2009) (a kind of conflict) and many studies (Zimmermann et al., 2004; Ying et al., 2004; Denninger, 2012; Giger et al., 2012; Bailey et al., 2012) rely on code dependencies for predicting code changes and bugs. We realize dependencies complement task interfaces, but the challenge for defining task interfaces remains, as the developers still need to identify a start point for the prediction.

To conclude, despite being possible to reduce conflicts by planning the parallel execution of tasks, other factors might cause conflicts, such as developer skill, task requirements, and projects' restrictions like duration and costs. 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, we might extend TAITI to also evaluate such factors to recommend tasks.

Similarly, 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)). In such context, requirements' priority, interdependencies between requirements, costs, and customer satisfaction are relevant. Although the interdependence between system requirements might lead to conflicts (Carlshamre et al., 2001), this research field concerns to solve the optimization problem to balance 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

to plan tasks and TAITI to support decision making during task execution seems a promising idea to reduce conflicts.

9. Conclusions

In this paper, we investigate a strategy for inferring task interfaces based on acceptance tests (*TestI*), assuming the specific BDD context. Such a strategy is relevant because it enables the development of a tool for computing the conflict risk among tasks. 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, 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 *RandomI*. Nevertheless, *TestI* is always more precise than *RandomI*. Compared to *TextI, TestI* covers more code changes, although it is less precise.

Our results also confirm that for 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.

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 might define a notion of interface with weighted elements. For example, we might consider a file more vulnerable to changes and, as a consequence, to cause conflicts than other files according to the phase a test accesses it. Alternatively, we might also define a hybrid solution involving *TestI* and *TextI*. This time, if both *TestI* and *TextI* include a file, we might conclude such a file is more relevant for conflicts occurrence. Another possibility might be to mapping syntactic or logical dependencies of each file in *TestI* as proposed by related works and confirmed by manual task analysis, aiming to capture implicit relationships among files that tests are unable to detect.

Finally, although we believe 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 reinforces such understanding. At last, usually code changes are not 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.

This work was an exploration into the potential of test-based task interfaces for supporting developers to avoid conflicts. However, to effectively evaluate the capacity of *TestI* to reduce conflicts it is necessary to correlate it to true conflict occurrence. For instance, does the integration of tasks with similar *TestI* fail? Can we successfully integrate tasks with disjoint *TestI*? Answering such questions is not trivial, given it depends on the definition of a similarity measure among interfaces that approximates the conflict risk as well as the construction of an appropriate dataset. And given the other factors that cause conflicts during tasks integration that we previously mentioned, a field study to evaluate the human aspects involved in the acceptance of solution to reduce conflicts might be interesting.

Acknowledgments

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

References

Accioly, P., Borba, P., Cavalcanti, G., 2017. Understanding semi-structured merge conflict characteristics in open-source java projects. Empir. Softw. Eng. doi:10.1007/s10664-017-9586-1.

Adams, B., McIntosh, S., 2016a. Modern release engineering in a nutshell – why researchers should care. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 5, pp. 78–90.

Adams, B., McIntosh, S., 2016b. Modern release engineering in a nutshell – why researchers should care. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 5, pp. 78–90.

Almeida, J.C., Pereira, F.d.C., Reis, M.V., Piva, B., 2018. The next release problem: Complexity, exact algorithms and computations. In: International Symposium on Combinatorial Optimization. Springer, pp. 26–38.

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

Apel, S., Liebig, J., Brandl, B., Lengauer, C., Kästner, C., 2011. Semistructured merge: Rethinking merge in revision control systems. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 190–200. doi:10.1145/2025113.2025141.

Bailey, M., Lin, K.-I., Sherrell, L., 2012. Clustering source code files to predict change propagation during software maintenance. In: Proceedings of the 50th Annual Southeast Regional Conference. ACM, New York, NY, USA, pp. 106–111. doi:10.1145/2184512.2184538.

Baldwin, C.Y., 2000. Design Rules: The Power of Modularity. The MIT Press.

Bass, L., Weber, I., Zhu, L., 2015. Devops: A Software Architect's perspective. Addison-Wesley Professional. http://cds.cern.ch/record/2034028.

Biehl, J.T., Czerwinski, M., Smith, G., Robertson, G.G., 2007. Fastdash: a visual dashboard for fostering awareness in software teams. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, New York, NY, USA, pp. 1313–1322. doi:10.1145/1240624.1240823.

Binamungu, L.P., Embury, S.M., Konstantinou, N., 2018. Detecting duplicate examples in behaviour driven development specifications. In: 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST), pp. 6–10.

Bird, C., Zimmermann, T., 2012. Assessing the value of branches with what-if analysis. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, p. 45.

Briand, L., Bianculli, D., Nejati, S., Pastore, F., Sabetzadeh, M., 2017. The case for context-driven software engineering research: generalizability is overrated. IEEE Softw. 34 (5), 72–75.

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

Brun, Y., Holmes, R., Ernst, M.D., Notkin, D., 2013b. Early detection of collaboration conflicts and risks. IEEE Trans. Softw. Eng. 39 (10), 1358–1375.

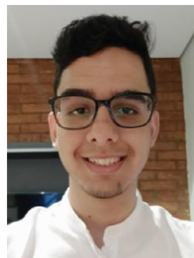
Capybara. http://teamcapybara.github.io/capybara/. Accessed: April 2019.

Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B., och Dag, J.N., 2001. An industrial survey of requirements interdependencies in software product release planning. In: Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on. IEEE, pp. 84–91.

Cataldo, M., Mockus, A., Roberts, J.A., Herbsleb, J.D., 2009. Software dependencies, work dependencies, and their impact on failures. IEEE Trans. Softw. Eng. 35 (6), 864–878.
Cavalcanti, G., Borba, P., Accioly, P., 2017. Evaluating and improving semistructured merge. Proc. ACM Program. Lang. 1, 59:1–59:27. doi:10.1145/3133883. (OOPSLA)
Coveralls. https://coveralls.io/. Accessed: April 2019.
Cubranic, D., Murphy, G.C., Singer, J., Booth, K.S., 2005. Hipikat: a project memory for software development. IEEE Trans. Softw. Eng. 31 (6), 446–465. doi:10.1109/TSE.2005.71.
Cucumber repository. https://github.com/cucumber/cucumber/wiki/Projects-Using-Cucumber. Accessed: April 2019.
Denninger, O., 2012. Recommending relevant code artifacts for change requests using multiple predictors. In: Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 78–79. http://dl.acm.org/citation.cfm?id=2666719.2666737.
Dewan, P., Hegde, R., 2007. Semi-synchronous conflict detection and resolution in asynchronous software development. In: ECSCW 2007. Springer, pp. 159–178.
Dias, M., Bacchelli, A., Gousios, G., Cassou, D., Ducasse, S., 2015. Untangling fine-grained code changes. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, pp. 341–350.
Fowler, M., 2009. Feature Branch. https://martinfowler.com/bliki/FeatureBranch.html. Accessed: April 2018
Fowler, M., 2016. Feature Toggle. Technical Report. ThoughtWorks Insights. https://martinfowler.com/bliki/FeatureToggle.html.
Gherkin. https://github.com/cucumber/cucumber/tree/master/gherkin. Accessed: April 2019.
Giger, E., Pinzger, M., Gall, H.C., 2012. Can we predict types of code changes? an empirical analysis. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on, pp. 217–226.
GitHub Java API. https://github.com/eclipse/egit-github/tree/master/org.eclipse.egit.github.core. Accessed: April 2019.
Grinter, R.E., 1996. Supporting articulation work using software configuration management systems. Comput. Supported Coop. Work.
Guimarães, M.L., Silva, A.R., 2012. Improving early detection of software merge conflicts. In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 342–352. http://dl.acm.org/citation.cfm?id=2337223.2337264.
Henderson, F., 2017. Software Engineering at Google. Technical Report. CoRR. arXiv:1702.01715.
Herzig, K., Zeller, A., 2013. The impact of tangled code changes. In: Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, Piscataway, NJ, USA, pp. 121–130. http://dl.acm.org/citation.cfm?id=2487085.2487113.
Hodgson, P., 2017a. Feature branching vs. feature flags: what's the right tool for the job? Technical Report. DevOps Blog. https://devops.com/feature-branching-vsfeature-flags-whats-right-tool-job/.
Hodgson, P., 2017b. Feature Toggles (aka Feature Flags). https://martinfowler.com/articles/feature-toggles.html. Accessed: April 2018
JGit. https://www.eclipse.org/jgit/. Accessed: April 2019.
Kasi, B.K., Sarma, A., 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 732–741. http://dl.acm.org/citation.cfm?id=2486788.2486884.
Kersten, M., Murphy, G.C., 2006. Using task context to improve programmer productivity. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 1–11. doi:10.1145/1181775.1181777.
McKee, S., Nelson, N., Sarma, A., Dig, D., 2017. Software practitioner perspectives on merge conflicts and resolutions. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 467–478.
Nagappan, M., Zimmermann, T., Bird, C., 2013. Diversity in software engineering research. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, pp. 466–476. doi:10.1145/2491411.2491415.
Perry, D.E., Siy, H.P., Votta, L.G., 2001. Parallel changes in large-scale software development: an observational case study. ACM Trans. Softw. Eng.Method. (TOSEM) 10 (3), 308–337.
Potvin, R., Levenberg, J., 2016. Why google stores billions of lines of code in a single repository. Commun ACM 59, 78–87. http://dl.acm.org/citation.cfm?id=2854146.
Rocha, T., Borba, P., Santos, J., Online Appendix. https://thaisabr.github.io/task-interface-study-site/. Accessed: April 2019.
Ruby on Rails. http://rubyonrails.org/. Accessed: April 2019.
Salton, G., McGill, M.J., 1986. Introduction to Modern Information Retrieval. McGraw-Hill, Inc., New York, NY, USA.
Sarma, A., Redmiles, D., van der Hoek, A., 2012. Palantír: early detection of development conflicts arising from parallel code changes. IEEE Trans. Softw. Eng. 38 (4), 889–908. doi:10.1109/TSE.2011.64.
Savage, T., Dit, B., Gethers, M., Poshyvanyk, D., 2010. Topicxp: exploring topics in source code using latent dirichlet allocation. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, pp. 1–6. doi:10.1109/ICSM.2010.5609654.
Shen, X., Minku, L.L., Bahsoon, R., Yao, X., 2016. Dynamic software project scheduling through a proactive-rescheduling method. IEEE Trans. Software Eng. 42 (7), 658–686.
SimpleCov. https://github.com/colszowka/simplecov. Accessed: April 2019.
Smart, J., 2014. BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle. Manning Publications Company. https://books.google.com.br/books?id=2BGxngEACAAJ.
de Souza, C.R.B., Redmiles, D., Dourish, P., 2003. "Breaking the code", moving between private and public work in collaborative software development. In: Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work. ACM, pp. 105–114.
Stray, V., Sjøberg, D.I., Dybå, T., 2016. The daily stand-up meeting: a grounded theory study. J Syst Softw 114, 101–124. http://www.sciencedirect.com/science/article/pii/S0164121216000066.
Wilcoxon, F., Wilcox, R.A., 1964. Some rapid approximate statistical procedures. Lederle Laboratories. https://books.google.com.br/books?id=aBU8AAAAIAAJ.
Wynne, M., Hellesøy, A., 2012. The cucumber book: behaviour-driven development for testers and developers. Pragmatic programmers. Pragmatic Bookshelf. https://books.google.com.br/books?id=oMswygAACAAJ.
Ying, A.T.T., Murphy, G.C., Ng, R., Chu-Carroll, M.C., 2004. Predicting source code changes by mining change history. IEEE Trans. Softw. Eng. 30 (9), 574–586.
Zimmermann, T., 2007. Mining workspace updates in cvs. In: Proceedings of the Fourth International Workshop on Mining Software Repositories. IEEE Computer Society, Washington, DC, USA, p. 11. doi:10.1109/MSR.2007.22.
Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A., 2004. Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 563–572. http://dl.acm.org/citation.cfm?id=998675.999460.

Thaís Rocha is a PhD student at the Informatics Center of the Federal University of Pernambuco and a member of the Software Productivity Group group. Her research interests are software modularity, collaborative software development, and empirical software engineering.

Paulo Borba is Professor of Software Development at the Informatics Center of the Federal University of Pernambuco, Brazil, where he leads the Software Productivity Group. His main research interests are in the following topics and their integration: software modularity, software product lines, and refactoring.



João Pedro Santos is a graduate student at Federal university of Pernambuco and a software engineer at Griaule biometrics. His research interests are data science, distributed computing, software product lines and acceptance testing.