Improving the prediction of files changed by programming tasks

João Pedro Santos
jpms2@cin.ufpe.br
Informatics Center, Federal University of Pernambuco
Recife, Brazil

Thaís Rocha
tabr@cin.ufpe.br
Informatics Center, Federal University of Pernambuco
Recife, Brazil

Paulo Borba
phmb@cin.ufpe.br
Informatics Center, Federal University of Pernambuco
Recife, Brazil

ABSTRACT

Integration conflicts often damage software quality and developers' productivity in a collaborative development environment. For reducing merge conflicts, we could avoid asking developers to execute potentially conflicting tasks in parallel, as long as we can predict the files to be changed by each task. As manually predicting that is hard, the TAITI tool tries to do that in the context of BDD (*Behaviour-Driven Development*) projects, by statically analysing the automated acceptance tests that validate each task. TAITI computes the set of files that might be executed by the tests of a task (a so called *test-based task interface*), approximating the files that developers will change when performing the task. Although TAITI performs better than a random task interface, there is space for accuracy improvements. In this paper, we extend the interfaces computed by TAITI by including the dependences of the application files reached by the task tests. To understand the potential benefits of our extension, we evaluate precision and recall of 60 task interfaces from 8 Rails GitHub projects. The results bring evidence that the extended interface improves recall by slightly compromising precision.

CCS CONCEPTS

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

KEYWORDS

Collaborative development, Behaviour-Driven Development, File change prediction

ACM Reference Format:
João Pedro Santos, Thaís Rocha, and Paulo Borba. 2019. Improving the prediction of files changed by programming tasks. In *XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '19), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357141.3357145

1 INTRODUÇÃO

A utilização de sistemas de controle de versão é uma prática comum no ambiente de desenvolvimento de software, permitindo a seus usuários gerenciar mudanças no código-fonte de um programa. A realização dessas mudanças, quando feitas paralelamente em versões do mesmo arquivo, pode provocar conflitos. Especificamente, se a mesma área de um documento for alterada por dois ou mais desenvolvedores, a junção automática das edições requer que o impasse seja resolvido manualmente.

Conflitos de *merge* como esse são frequentes, consomem tempo para serem resolvidos [19] e têm sua resolução propensa à erros [3, 7, 11, 17]. Isso afeta negativamente a produtividade no ambiente de desenvolvimento, tendo em vista que uma quantidade relevante de *merges* resultam em conflito [6, 13, 19]. Os desenvolvedores têm receio de encontrá-los durante o desenvolvimento e, por isso, muitas vezes empregam técnicas arriscadas para evitá-los, o que aumenta as chances de ocorrência destes [5, 8, 9]. Em particular, esse problema pode inviabilizar o desenvolvimento modular, independente, das tarefas de desenvolvimento de um sistema, principalmente quando essas não têm um alinhamento claro com os componentes da arquitetura do sistema. Essa falta de alinhamento é comum quando a definição de tarefas é direcionada por conceitos como *feature* e *user stories*, que são comumente adotados por processos modernos de desenvolvimento de software.

A escolha criteriosa das tarefas de programação que cada desenvolvedor irá realizar, bem como a ordem em que ele vai executá-las, pode reduzir a incidência de conflitos, desde que as tarefas realizadas em paralelo foquem em funcionalidades de sistema distintas, exigindo mudanças de código em arquivos diferentes. No entanto, essa escolha pode ser feita de forma menos apropriada quando realizada manualmente e baseada em critérios *ad hoc*. Prever automaticamente quais arquivos serão modificados pelas tarefas também é difícil, mas pode se tornar plausível em um contexto específico [4]. A ferramenta TAITI [14] tenta prever os arquivos que uma tarefa irá alterar no contexto de *Behaviour-Driven Development* (BDD) [18], que recomenda a descrição de requisitos em linguagem não ambígua, subconjunto da linguagem natural, especificando o comportamento esperado de funcionalidades de sistema em formato de cenários de teste. Com base nessas descrições, os desenvolvedores implementam código que automatiza os testes, a fim de validar o comportamento do sistema. Como os testes são implementados antes mesmo das funcionalidades e utilizam código de aplicação, inicialmente eles falham. À medida que o desenvolvimento do sistema progride, as falhas nos testes são reduzidas, e o esperado é que, uma vez que as funcionalidades sejam desenvolvidas, os testes passem.

Como em BDD os testes da tarefa são desenvolvidos antes do código da aplicação, TAITI utiliza-os como entrada, buscando por chamadas à métodos, construtores e acessos à constantes, a fim de ligá-los aos arquivos de projeto que os declaram. Esses arquivos, por terem sido referenciados nos testes, aproximam o comportamento da funcionalidade, e possivelmente serão modificados durante a

SBCARS '19, September 23–27, 2019, Salvador, Brazil

ACM ISBN 978-1-4503-7637-2/19/09…$15.00
https://doi.org/10.1145/3357141.3357145

realização da tarefa, constituindo sua interface (especificamente, a interface baseada em testes). Tendo como referência as tarefas em execução e as tarefas a serem realizadas em um projeto, é possível usar TAITI para calcular interfaces de tarefa baseadas em testes e avaliar o risco de conflito entre tarefas, segundo a interseção entre suas interfaces. Com base nessa informação, é possível então ajudar o desenvolvedor a escolher a próxima tarefa para executar, visando reduzir a ocorrência de conflitos com as tarefas em andamento. Dessa forma, o ponto crítico é garantir que as interfaces de tarefa são bons preditores de mudança de código; caso contrário, a análise de risco de conflito torna-se infundada.

Rocha et al. [14], em seu estudo avaliativo, mostrou que TAITI consegue prever tantos arquivos alterados quanto uma ferramenta que prevê esses arquivos aleatoriamente, porém com mais precisão. Os resultados do estudo também indicam que, para funcionar bem, TAITI requer tarefas com alta cobertura de testes (o quanto do código é exercitado pelos testes), uma vez que estes são o ponto de partida para o cálculo da interface da tarefa.

Uma análise mais cuidadosa dos resultados de avaliação de TAITI evidenciou que ajustes podem ser feitos a fim de melhorar sua capacidade preditiva. Uma possível melhoria consiste em aumentar o alcance da análise. Neste artigo, TAITI é estendida através da adição de arquivos às interfaces de tarefa baseadas em testes. Os novos arquivos são descobertos por meio da análise estática das dependências entre os arquivos originalmente existentes na interface da tarefa e demais arquivos de aplicação. Essa abordagem busca diminuir a quantidade de falsos negativos (arquivos que deveriam fazer parte da interface da tarefa e não fazem por limitação de TAITI), bem como entender se arquivos que não são explicitamente exercitados pelos testes da tarefa também são úteis ao propósito de predizer mudanças e evitar conflitos. Apesar de existirem alternativas à análise estática de dependências no contexto de propagação de mudanças, como acoplamento conceitual e acoplamento de mudanças, optou-se por adotar as dependências estáticas por não exigir um histórico de projeto rico, possibilitando o uso da ferramenta nas diferentes fases de um projeto. A fim de avaliar o efeito da extensão proposta, comparamos os valores de precisão e revocação (também chamada sensibilidade) das interfaces de tarefa computadas por TAITI com e sem análise de dependência entre arquivos. Para tanto, avaliamos 60 interfaces de tarefa, provenientes de 8 projetos Ruby on Rails no GitHub. Os resultados evidenciam que a inclusão de arquivos não exercitados diretamente pelos testes aumenta a revocação das interfaces originalmente calculadas por TAITI em 30%. Além disso, foi constatado que a cobertura dos testes parece não afetar o aumento ou a diminuição da precisão das interfaces.

O restante do artigo está organizado conforme descrito a seguir. A Seção 2 explica como a ferramenta TAITI e a extensão proposta podem ajudar a evitar conflitos de *merge* ao predizer os arquivos que serão alterados pelas tarefas, estendendo o exemplo apresentado por Rocha et al. [14]. A Seção 3 explica o funcionamento de TAITI e a Seção 4 explica o funcionamento da extensão à TAITIA Seção 5 descreve o estudo empírico realizado para avaliar o efeito da extensão proposta sobre a capacidade de predizer os arquivos alterados por uma tarefa. A Seção 6 apresenta e discute os resultados. A Seção 7 discute as ameaças à validade do estudo avaliativo. A Seção 8 apresenta os trabalhos relacionados. Por fim, a Seção 9 apresenta as considerações finais.

2 MOTIVAÇÃO

Para entender como interfaces de tarefa baseadas em testes (*TestI*) podem ser úteis ao propósito de prever mudanças em arquivos de código-fonte e evitar conflitos, considere que Adam e Betty são membros de uma equipe de desenvolvimento ágil que está desenvolvendo um sistema Rails de gerenciamento escolar, que armazena as notas dos alunos e permite aos professores visualizá-las. Em uma dada iteração, suponha que Adam é responsável por desenvolver a funcionalidade de avaliação de uma turma (tarefa T_1). Então, ele cria um método para computar a média das notas dos alunos e adiciona código para exibir essa informação na página de visualização da turma. Enquanto isso, Betty escolhe desenvolver a funcionalidade referente à identificação dos alunos com notas baixas pelos professores (tarefa T_2). Para tanto, Betty desenvolve um método para retornar uma lista de alunos com notas abaixo de um limiar e também o código que mostra essa informação na página de visualização da turma. O *backlog* da iteração ainda inclui outras tarefas, como corrigir o *feed* de notícias para exibir apenas notícias recentes (tarefa T_3).

Ao trabalhar independentemente em T_1 e T_2, cada um em seu repositório local do sistema de controle de versão, Adam e Betty adicionam diferentes métodos (*get_grade_average* e *low_perf_students*) ao fim do mesmo arquivo (*app/controllers/classes_controller.rb*). Posteriormente, a integração dessas contribuições resulta no conflito de *merge* ilustrado pela Figura 1. O conflito acontece porque as tarefas T_1 e T_2 estão associadas à mesma funcionalidade de sistema (avaliação de turma). Esse conflito poderia ser evitado caso Betty tivesse optado por executar a tarefa T_3 ao invés de T_2, já que T_1 e T_3 estão associadas à funcionalidades independentes entre si (note que T_3 está relacionada à funcionalidade de notificação).

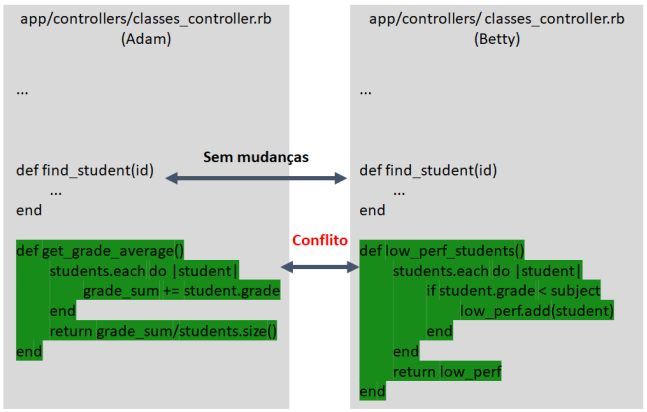

Figura 1: Exemplo de conflito de *merge* que poderia ser evitado usando a versão original de TAITI.

A ferramenta TAITI poderia ser usada para alertar sobre o risco de conflito ao calcular as interfaces das tarefas T_1, T_2 e T_3. Conforme ilustrado pela Figura 2, as interfaces de T_1 e T_2 são idênticas, ou seja, os testes dessas tarefas exercitam os mesmos arquivos de aplicação,

enquanto que a interface da tarefa T_3 é completamente diferente, sinalizando que seus testes exercitam outros arquivos. Dado que a interface representa o conjunto de arquivos que TAITI prevê que serão alterados pelas tarefas, é intuitivo concluir que tarefas que irão alterar arquivos diferentes não causarão conflito de *merge*.

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

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

Figura 2: Interfaces das tarefas T_1, T_2 e T_3 geradas por TAITI.

Apesar de TAITI poder ajudar a evitar o conflito apresentado, anteriormente discutido por Rocha et al. [14], ela pode falhar quando um arquivo que não é explicitamente exercitado no teste precisa ser modificado pelos desenvolvedores. Por exemplo, considere agora que enquanto Adam desenvolve a tarefa T_4, referente à listagem de alunos por regime de horário (se em tempo integral ou não), Betty desenvolve a funcionalidade que permite ao professor editar as notas dos alunos bolsistas (tarefa T_5), um tipo especial de aluno representado via código através do mecanismo de herança. Para isso, Adam cria um método que informa o nome e o horário de cada aluno, bem como um método que devolve a lista de alunos da turma com informação de horário, além de ajustar as páginas de visualização de aluno e de turma. Já Betty identifica que o mecanismo para atualizar as notas de um aluno não depende do fato do aluno ser bolsista ou não, e então cria um método de atualização na superclasse, além de gerar uma página para edição de aluno e deixá-la acessível por outras páginas do sistema.

As interfaces geradas por TAITI sugerem que não há risco de conflito entre as tarefas T_4 e T_5, pois os testes referentes à T_4 lidam com aluno, enquanto que os testes de T_5 lidam com um subtipo de aluno. Porém, ao trabalhar independentemente em T_4 e T_5 em seus repositórios locais, Adam e Betty adicionam métodos diferentes (*get_basic_info e edit_grade*) ao fim do mesmo arquivo (*app/models/student.rb*), o que posteriormente causa, no repositório central, o conflito de *merge* ilustrado pela Figura 3, análogo ao conflito anteriormente apresentado.

A falha de TAITI consiste em não incluir o arquivo *app/models/student.rb* na interface da tarefa T_5, o que seria determinante para sinalizar o risco de conflito entre ela e T_4. Como os testes de T_5 não exercitam explicitamente o arquivo com conflito, TAITI precisaria ir além dos testes para alcançá-lo. Neste artigo, é proposta a extração das dependências dos arquivos exercitados pelos testes das tarefas para esse fim, partindo da premissa de que dependências

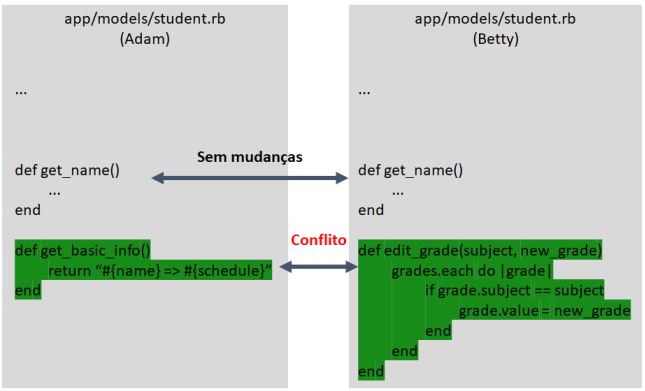

Figura 3: Exemplo de conflito de *merge* que só poderia ser evitado pela versão estendida de TAITI.

podem causar a propagação de mudanças de código. No exemplo em questão, como a interface de T_5 possui o arquivo *app/models/sponsored_student.rb*, que é subclasse da classe definida no arquivo *app/models/student.rb*, caracterizando uma dependência, este último arquivo também seria adicionado à interface da tarefa. A interseção entre as interfaces de T_4 e T_5 seria um indicador de risco de conflito, o que poderia levar Adam e Betty a evitarem o desenvolvimento paralelo dessas tarefas, ou prosseguirem de forma mais cuidadosa.

3 A FERRAMENTA TAITI

Antes de explicar a extensão da ferramenta TAITI, é necessário esclarecer seu funcionamento original. Com o propósito de prever os arquivos de aplicação modificados por uma tarefa, TAITI lida com projetos Ruby on Rails que usam a ferramenta Cucumber para elaborar e executar testes de aceitação. Dado o conjunto de testes de aceitação que validam o comportamento esperado de uma tarefa, escritos em Gherkin (linguagem usada pelo Cucumber para descrever cenários de teste em alto nível), TAITI faz o mapeamento destes para o código Ruby que automatiza a execução dos testes (na terminologia do Cucumber, os *step definitions*). O mapeamento é feito via expressões regulares. A Figura 4 ilustra um cenário de teste que valida o comportamento da tarefa T_1 relacionada à funcionalidade de avaliação de turma (vide o bloco de código iniciado com a palavra-chave **Scenario**). Os termos em negrito são palavras-chave em Gherkin, sendo os mais relevantes aqueles que delimitam a etapa de preparação para o teste (**Given**), a ação principal do teste (**When**) e o resultado esperado do teste (**Then**). Cada uma das sentenças iniciadas pelas referidas palavras-chave são automatizadas. A Figura 5 exibe um fragmento do código do *step definition* que automatiza a etapa de preparação do cenário de teste da Figura 4. Observe que ele é identificado por uma expressão regular definida pelo desenvolvedor, que é usada por TAITI para mapear o código à descrição do teste em alto nível.

Em seguida, TAITI analisa estaticamente o corpo dos *step definitions* a fim de coletar referências à elementos de programação, tais como métodos, referências à páginas *web* e, recursivamente, elementos e arquivos adicionais referenciados pelas páginas *web*. Na Figura 5 o uso da classe *Teacher*, destacado em azul, é um exemplo de referência coletável por TAITI. Por fim, TAITI tenta inferir os

class_evaluation.feature

```
Feature: Class evaluation
As a teacher
I want to evaluate the performance of my class
So that I can compare classes over time
And I can improve my teaching strategy

Scenario: Grades mean
Given I am logged in as a teacher
And my class contains these students:
        | name    | grade   |
        | John    | 7       |
        | Mary    | 5       |
        | Alex    | 9       |
When I request the class evaluation
Then the result should be "7"
```

Figura 4: Exemplo de cenário de teste referente à tarefa T_1.

arquivos de aplicação que declaram os elementos e páginas referenciados pelos testes. O resultado final gerado por TAITI consiste em um conjunto de arquivos (vide Figura 2 como exemplo) que provavelmente serão exercitados pelos testes da tarefa e poderão ser alterados a fim da tarefa ser concluída, já que delimitam o comportamento esperado desta, denominado interface da tarefa baseada em testes (*TestI*).

class_evaluation_steps.rb

```
Given /^I am logged in as teacher$/ do
  @teacher = Teacher.create("Mary", "mary@example.com")
  @current_class = @teacher.classes.last
  ....
end
```

Figura 5: Exemplo de *step definition* referente à tarefa T_1.

Devido à natureza dinâmica de Ruby, identificar os arquivos que declaram os elementos de programação referenciados nos testes não é trivial. Portanto, TAITI adota uma solução simplificada e predominantemente conservativa. Se um método é chamado por intermédio de uma classe, TAITI busca por arquivos que correspondem ao identificador da classe. Por exemplo, a chamada *Student.find_by(name:"Paul")*[1] leva à busca por um arquivo com o sufixo *student.rb*. Já em caso de chamadas de método sem informação sobre o tipo do objeto alvo, TAITI busca por uma declaração de método que corresponde ao identificador do método chamado e ao número de argumentos usados. Nesse caso, quando possível, TAITI também tenta aplicar convenção de nomenclatura tal como faz com classes. Por exemplo, a chamada *@student.edit_grade(s,ng)* leva à busca pela classe *Student*. Se essa classe existe no projeto, TAITI verifica se ela declara um método *edit_grade*, caso contrário, TAITI procura pela declaração do método *edit_grade* que aceita dois argumentos nos demais arquivos da aplicação. O tipo dos argumentos usados na chamada não são considerados. Em caso de haver múltiplas declarações de método compatível com a chamada, TAITI considera todas elas.

Ao longo desse processo, TAITI ignora arquivos que correspondem à código auxiliar usado nos *step definitions*. A diferenciação entre arquivos de teste e arquivos de aplicação se dá com base na estrutura de diretórios do projeto e na extensão dos arquivos. Por padrão, arquivos de teste são arquivos Gherkin localizados na pasta *features* e arquivos Ruby existentes na pasta *features/step_definitions*. Arquivos de aplicação são arquivos Ruby e arquivos HTML (e variações, incluindo arquivos ERB e HAML) existentes na pasta *app* ou na pasta *lib*.

Para localizar as páginas *web* referenciadas pelos testes, TAITI inspeciona chamadas ao método *visit* [2]. Esse método recebe como argumento um caminho que Rails mapeia para uma rota a fim de disparar uma ação de controlador, evitando o uso de strings URL diretamente nos testes. Dessa forma, a análise das chamadas ao método *visit* possibilita identificar as páginas *web* e os controladores usados nos testes. Existem diferentes formas de chamar esse método, que se diferenciam pela natureza do argumento utilizado. Além disso, Rails possui um mecanismo próprio para configuração de rotas e também gera automaticamente métodos que facilitam o uso destas via testes. TAITI tenta lidar com as formas mais usuais de chamadas de *visit*. Uma vez que as páginas *web* usadas nos testes são identificadas, TAITI percorre tais arquivos buscando acesso à variáveis de instância e chamadas de métodos referentes à formulários, botões, *links* e renderização de arquivos, dado que estes estão associados às ações do usuário e podem ser exercitados pelos testes também. Isso é feito de forma análoga à análise de *step definitions* já explicada, com a diferença de que a entrada não é arquivo Ruby e sim, arquivo HTML (e suas variantes).

Observe que TAITI não pode analisar os testes de uma tarefa de forma dinâmica, o que pressupõe a execução dos testes, pois no contexto de BDD, considerando que as tarefas ainda não foram concluídas, os testes falham porque o código de aplicação por eles exercitado ainda não está pronto.

4 EXTENSÃO DE TAITI

Com revocação de 0, 48±0, 32(0, 45) [14] no melhor caso, TAITI é um preditor promissor, que possui maior precisão do que um preditor aleatório, e que tem melhor performance quando a cobertura de testes da tarefa é alta. Apesar disso, TAITI ainda pode ser melhorada. Com o intuito de diminuir a quantidade de falsos negativos, ou seja, a omissão de arquivos que deveriam fazer parte da interface da tarefa, esta ferramenta foi estendida de forma a também extrair as dependências dos arquivos da interface originalmente calculada. A premissa é que as dependências entre arquivos podem motivar a propagação de mudanças de código.

A Figura 6 a seguir ilustra a ideia de maneira simplificada. Nesta figura, o arquivo destacado em verde (*identify_user_steps.rb*) contém os *step definitions* dos testes de uma dada tarefa. Ao analisá-los, TAITI identifica que eles exercitam os arquivos destacados em azul (*search_controller.rb* e *user.rb*). A ideia consiste em adicionar à interface da tarefa, até então constituída pelos arquivos destacados em azul, os arquivos de aplicação dos quais eles dependem, destacados em roxo. Logo, a interface que tinha 2 arquivos passa a ter 6.

Em sua versão estendida [16], TAITI contempla três novas etapas que complementam o cálculo de *TestI*, detalhadas a seguir: Pré-processamento de dependências do projeto, processamento de dependências do projeto, e extração de dependências de *TestI*.

[1] *find_by* é um método de busca gerado por Rails de acordo com o padrão *Active Record*.

[2] Método da biblioteca Capybara, que é comumente usada pelos testes do Cucumber para simular a interação do usuário com a GUI.

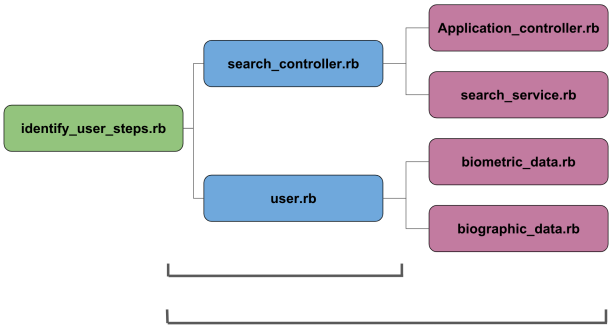

Figura 6: Exemplo de conjunto de arquivos obtido pela versão estendida de TAITI.

4.1 Pré-processamento de dependências do projeto

A fase de pré-processamento identifica cada classe declarada no projeto e suas respectivas dependências em relação à outras classes. O arquivo em que a classe está declarada também é identificado, o que é necessário para complementar a interface da tarefa, já que TAITI considera os arquivos do projeto. A identificação das classes e suas dependências é feita via análise sintática pela biblioteca Rubrowser [15], que gera um grafo de dependências, tal como ilustrado pela Figura 7, onde os nós são as classes declaradas no sistema e as setas representam as dependências entre as classes, em que a seta parte da classe dependente. O tamanho do nó indica o tamanho da classe em linhas.

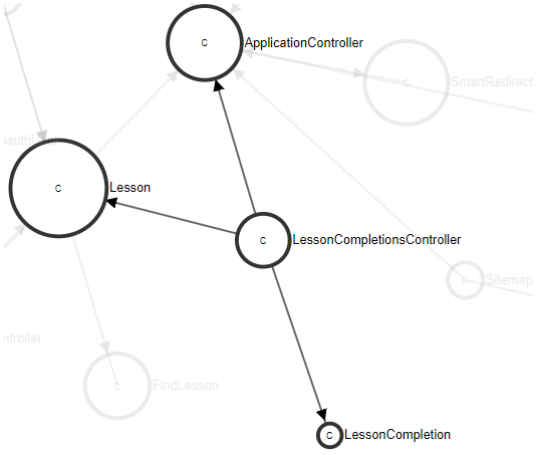

Figura 7: Exemplo de grafo de dependências gerado pela biblioteca Rubrowser.

O grafo também pode ser representado por um arquivo JSON, facilitando a manipulação do resultado. A Figura 8 ilustra um fragmento de um arquivo JSON informando que a classe *AbuseReportsController* depende da classe *ApplicationController*. A lista identificada como *definitions* identifica todas as classes declaradas no projeto. Dentre os atributos, os principais são *namespace*, que consiste no identificador da classe e *file*, o arquivo que contém a declaração. Já a lista *relations* identifica a dependência entre duas classes. Nesse caso, é relevante saber se a dependência é circular, o identificador da classe que gera a dependência (atributo *namespace*), o identificador da classe dependente (atributo *caller*), o caminho absoluto para a classe dependente (atributo *file*) e a linha em que a dependência é identificada na classe dependente (atributo *line*). Logo, na representação em JSON do grafo ilustrado pela Figura 7, todas as classes estariam em *definitions*, e *LessonCompletionsController* apareceria três vezes na lista de *relations*, uma vez que depende de três classes (*Lesson*, *ApplicationController* e *LessonCompletion*).


```
{
  "definitions": [
    {
      "type": "Class",
      "namespace": "AbuseReportsController",
      "circular": false,
      "file": "app/controllers/abuse_reports_controller.rb",
      "line": 1,
      "lines": 35
    }
  ],
  "relations": [
    {
      "type": "Base",
      "namespace": "ApplicationController",
      "resolved_namespace": "ApplicationController",
      "caller": "AbuseReportsController",
      "file": "app/controllers/abuse_reports_controller.rb",
      "circular": false,
      "line": 1
    }
  ]
}
```


Figura 8: Exemplo de arquivo de dependências entre classes.

A biblioteca Rubrowser lida apenas com arquivos Ruby, mas como TAITI extrai dependências de páginas *web*, essa restrição não é um problema. Além disso, a biblioteca só lida com arquivos sintaticamente corretos e ignora o uso de *metaprogramming* (recurso que permite ao desenvolvedor escrever código que gera mais código em tempo de execução).

4.2 Processamento de dependências do projeto

Na fase de processamento, o arquivo JSON com as classes declaradas no projeto e suas dependências é refinado para evitar que dependências sejam registradas de forma incompleta, quando possível. Isso é necessário porque nem sempre a biblioteca Rubrowser consegue obter o identificador da classe dependente (atributo *caller* na lista *relations*), deixando esse campo vazio. Isso acontece quando a classe dependente é uma classe de biblioteca ou qualquer outra que não seja declarada no projeto, ou quando se trata de uma classe sintaticamente incorreta, ou quando é uma classe que contém atribuições do operador "::" (operador que permite acesso à constantes estáticas fora de seu escopo).

Para obter o identificador da classe dependente, são verificados os atributos *file* e *line* de cada objeto em *relations* cujo atributo *caller* é vazio. Com base nesses atributos, é verificada a ocorrência da dependência no arquivo que a possui e, por meio de expressões regulares, tenta-se mapeá-la para alguma declaração de classe existente na lista de *definitions*, especificamente com base no atributo *namespace*.

4.3 Extração de dependências de *TestI*

Uma vez que as dependências entre as classes do projeto são devidamente registradas, são extraídas as dependências de cada arquivo de interesse, que são os arquivos em *TestI*. Para tanto, são verificados todos os objetos da lista de *relations* cujo atributo *file* corresponde à um dado arquivo para o qual se deseja saber as dependências, e dele se extrai o atributo *namespace*. Em seguida, verifica-se na lista de *definitions*, o objeto que corresponde ao mesmo *namespace* e seu respectivo arquivo (atributo *file*). Ao término da extração, são eliminadas eventuais dependências repetidas.

Retomando à Figura 7, a busca pelas dependências do arquivo *app/controllers/lesson_completions_controller.rb* retornaria três arquivos: *app/models/lession_completion.rb*, *app/models/lesson.rb* e *app/controllers/application_controller.rb*.

4.4 Integração à TAITI

A integração com a versão original de TAITI é feita em dois momentos: Antes do cálculo de *TestI*, a fim de gerar o arquivo JSON identificando todas as classes do projeto e suas dependências (etapas de pré-processamento e processamento de dependências); e após o cálculo de *TestI*, quando são localizadas as dependências dos arquivos que constituem a interface (etapa de extração de dependências). O resultado final é denominado *TestIDep* e consiste na interface originalmente calculada por TAITI acrescida das suas dependências.

5 ESTUDO EMPÍRICO

Com o objetivo de investigar se a extensão proposta à TAITI aumenta sua capacidade preditiva, foi realizado um estudo empírico que compara as diferenças de precisão e revocação das interfaces de tarefa geradas pela versão original de TAITI (*TestI*) e a versão estendida pela extração de dependências entre arquivos (*TestIDep*). Especificamente, o estudo investiga duas questões de pesquisa:

- *TestIDep* possui maior valor de precisão do que *TestI*?
- *TestIDep* possui maior valor de revocação do que *TestI*?

Dado que a interface real da tarefa (*TaskI*) é o conjunto de arquivos alterados pela tarefa durante sua execução, ou seja, o oráculo, a revocação de uma tarefa t é a fração de arquivos alterados por t que TAITI consegue prever que irá mudar (Equação 1).

$$revocação(t) = \frac{|TestI(t) \cap TaskI(t)|}{|TaskI(t)|} \quad (1)$$

Já a precisão de uma tarefa t é a fração de arquivos que TAITI sinaliza que t irá mudar e que realmente muda (Equação 2).

$$precisão(t) = \frac{|TestI(t) \cap TaskI(t)|}{|TestI(t)|} \quad (2)$$

Dessa forma, a revocação e a precisão foram avaliadas duas vezes para cada tarefa: A primeira vez considerando *TestI*, e a segunda, *TestIDep*. A comparação final dos resultados foi feita na perspectiva da média do conjunto de tarefas de entrada, sendo adotada a notação **média ± desvio padrão (mediana)**. De maneira complementar, também foi calculada a métrica F_2, uma variação de *F-measure* em que a revocação tem maior peso que a precisão. No contexto de uso de TAITI um baixo valor de revocação representa risco de conflito ignorado, enquanto que um baixo valor de precisão representa falso risco de conflito. Dado que a ideia da ferramenta é ajudar o desenvolvedor a decidir qual a próxima tarefa que irá realizar com base no risco de conflitos, ignorar risco parece pior. Nesse sentido, foi dada prioridade à revocação. De toda forma, a ideia não é impossibilitar o desenvolvimento paralelo de tarefas, mas torná-lo mais efetivo, deixando o desenvolvedor ciente das possíveis consequências de suas ações.

O estudo foi conduzido de forma automatizada, mas também foram analisadas manualmente pouco mais de 20 tarefas para entender melhor os resultados. Note que trata-se de um estudo retroativo, ou seja, as tarefas avaliadas já foram concluídas, pois caso contrário, não seria possível delimitar o oráculo. As tarefas foram extraídas do histórico dos projetos e consistem em um conjunto de *commits*. A premissa subjacente é que os testes adicionados ou alterados pelos *commits* da tarefa delimitam o conjunto de testes que validam o seu comportamento, sendo ele a entrada para o cálculo de *TestI* (e *TestIDep*, por consequência).

O estudo usou como amostra de dados um conjunto de 60 tarefas provenientes de 8 projetos Rails no GitHub, que consiste em uma subamostra aleatória da amostra utilizada no estudo anterior de avaliação de TAITI [14]. A seguir resumimos a montagem da amostra de tarefas utilizada, descrevendo brevemente a amostra original a partir da qual ela foi extraída. Por questão de clareza, foram omitidos critérios de inclusão e exclusão relevantes para a amostra original que não são relevantes para o estudo em questão. Portanto, para obter uma visão detalhada do processo de montagem da amostra usada, é recomendável a leitura da Seção 5 do primeiro estudo de avaliação de TAITI.

5.1 Seleção de projetos

As tarefas da amostra foram extraídas de projetos Rails no GitHub, uma vez que TAITI foi concebida para lidar com projetos nesta plataforma, dada a suposta popularidade de BDD e Cucumber na comunidade Rails. Para buscar projetos Rails que usam o Cucumber, foi utilizado um *script* que utiliza a API do GitHub em Java [2], uma vez que a busca padrão do GitHub não suporta a filtragem de projetos de acordo com as ferramenta em uso. Em síntese, o *script* busca por projetos Ruby, faz o *download* de cada um deles e verifica se o *gemfile*, arquivo que lista todas as dependências do projeto, referencia as bibliotecas do Rails e do Cucumber.

De forma a ampliar a chance de encontrar projetos de interesse, apenas foram considerados projetos criados a partir de 2010, pois Cucumber e BDD eram menos populares antes disso. Foram realizadas três rodadas de busca por projetos. Na primeira rodada, foi restringido o número máximo de estrelas do projeto e os projetos obtidos foram ordenados em ordem decrescente do número de estrelas, na esperança de serem localizados projetos mais relevantes e populares. Na segunda rodada, a busca se deu considerando a data da última atualização dos projetos, visando alcançar projetos mais ativos. Por fim, a terceira rodada de busca consistiu em selecionar os projetos Rails dentre os projetos Ruby indicados no site do Cucumber. Como resultado final, foram selecionados 61 projetos.

5.2 Extração de tarefas e suas interfaces

Em seguida, os projetos selecionados foram filtrados, de maneira a excluir projetos cujas tarefas não possibilitam o cálculo de *TestI* (e

TestIDep, por consequência) e *TaskI*, ou seja, tarefas que não contribuem simultaneamente com código de aplicação (especificamente arquivos Ruby e páginas *web*) e testes do Cucumber.

Dado que nem todo projeto usa padrões (identificadores ou algo do tipo) em mensagens de *commit*, para relacionar *commits* à tarefas foi assumido que: (i) tarefas são integradas através de *commits* de *merge*; (ii) a contribuição de uma tarefa consiste nos *commits* entre o *commit* de *merge* e o ancestral em comum com as outras contribuições integradas; (iii) as mudanças de código causadas pela tarefa são necessárias para a sua conclusão; e (iv) os testes adicionados ou alterados por uma tarefa validam o seu comportamento.

A extração das tarefas de um projeto se deu através de um *script* que utiliza a API JGit [10]. O *script* clona o repositório do projeto, lista os *commits* de *merge* (excluindo *merges fast-fowarding*) e para cada um deles delimita duas tarefas, cada uma delas correspondendo às contribuições integradas. Tarefas que não alteram arquivos de aplicação e arquivos Gherkin foram excluídas. Nessa etapa, de 61 projetos de entrada restaram 31 projetos.

Em seguida, TAITI foi utilizada para calcular *TestI* e *TaskI* de cada tarefa. Dentre os critérios de exclusão adotados em seguida, é relevante saber que foram descartadas tarefas cujos testes não foram implementados ou foram parcialmente implementados, tarefas cujos testes contêm erros de sintaxe, e tarefas cujo *TestI* é vazio.

Ao final do processo, foi obtida uma amostra de 463 tarefas de 18 projetos para o primeiro estudo de avaliação de TAITI. Dessa amostra, foi extraída aleatoriamente uma amostra de 60 tarefas de 8 projetos distintos.

6 RESULTADOS

Nesta seção, são apresentados e discutidos os resultados do estudo empírico realizado, sob três perspectivas complementares: Na perspectiva de projeto, de tarefas e das interfaces de tarefa baseadas em testes estendidas pelas dependências entre arquivos (*TestIDep*). Como explicado, os valores de precisão e revocação avaliam se a solução proposta melhora a capacidade de TAITI de prever as mudanças de código necessárias para uma tarefa ser concluída. A ideia é que quanto maior a capacidade preditiva, maior o potencial de auxiliar os desenvolvedores a evitar conflitos de *merge*. Os resultados estão disponíveis *online* [16], em conjunto com a versão estendida de TAITI.

6.1 Resultados por projeto

A Tabela 1 a seguir resume os resultados obtidos para 60 tarefas de 8 projetos Rails distintos. Conforme pode ser observado com o auxílio da Tabela 2, a versão estendida de TAITI promove o aumento da revocação das interfaces de tarefa para todos os projetos analisados. O referido aumento varia entre 3,5% e 16,5%, com média de 8% por projeto. No entanto, na maioria dos casos acontece também perda de precisão, cuja taxa varia entre 0,2% e 9%, sendo a perda média em torno de 3,5% por projeto.

Observando os resultados mostrados na Tabela 2, alguns projetos se sobressaem: Os projetos destacados em vermelho (*tracks* e *diaspora*) sofreram a maior perda em precisão; o projeto destacado em verde (*odin*), por ter sido o único em que houve aumento de precisão; e o projeto *otwarchive*, destacado em amarelo, por parecer ter um bom resultado em um primeiro momento, mas que após análise cuidadosa, não se mostrou exatamente assim.

A diminuição da precisão se dá por vários fatores que afetam o resultado de *TestIDep*. Com base na análise manual de algumas tarefas dos projetos *tracks*, *diaspora* e *otwarchive*, foi possível identificar os principais. Primeiramente, a qualidade de *TestI*, ou seja, quando *TestI* é muito generalista (e portanto, contém muitos falsos positivos), *TestIDep* contém inúmeros falsos positivos também. Esse fenômeno foi observado em todas as tarefas analisadas do projeto *otwarchive*. Por exemplo, para uma dada tarefa cujo *TaskI* possui apenas 1 arquivo, *TestIDep* contém 86 falsos positivos, enquanto que *TestI* contém 182 falsos positivos. Os projetos *tracks* e *diaspora* também são afetados por esse fenômeno, mas em menor escala. O pior resultado de *diaspora* possui 104 falsos positivos, e o de *tracks*, 25 falsos positivos.

Um outro fator que causa perda de precisão por parte de *TestIDep* é quando *TestI* simplesmente erra a predição porque os testes não cobrem satisfatoriamente a funcionalidade subjacente à tarefa. Foram identificadas ocorrências desse fenômeno em tarefas do projeto *diaspora*. Por exemplo, uma dada tarefa possui testes no arquivo *conversations.feature* que exercitam os arquivos *app/models/visibilities.rb* e *app/models/conversation.rb*, segundo *TestI*, porém a tarefa em si não altera tais arquivos. Ao que parece, o teste não está alinhado ao objetivo da tarefa em si.

Uma hierarquia de dependências mais complicada também parece afetar negativamente a precisão de *TestIDep*, pois introduz falsos positivos que até poderiam ser evitados, uma vez que os verdadeiros positivos estão presentes em algum lugar na hierarquia de dependências calculada. Ao que parece, esse fenômeno deve-se à maturidade dos projetos, que já possuem várias funcionalidades implementadas. Um exemplo concreto acontece com a tarefa de ID 57 do projeto *tracks*. TAITI identificou que o arquivo *app/controllers/contexts_controller.rb* deveria fazer parte de *TestI*, dentre outros arquivos. O referido arquivo depende de 2 outros (*app/controllers/application_controller.rb* e *app/models/context.rb*). Os arquivos identificados pela análise de dependência, por sua vez, não foram alterados pela tarefa em si. No entanto, a tarefa alterou o arquivo *app/models/null_context.rb*, que é uma dependência da classe *Context* e que poderia ter sido incluída em *TestIDep* caso a solução proposta lidasse com dependências indiretas ou transitivas entre arquivos.

Já o aumento de precisão foi identificado em apenas um projeto da amostra, *odin*. O aumento de 4% observado não está relacionado diretamente à nenhum dos pontos citados anteriormente no que diz respeito à diminuição de precisão, pois se estivesse, outros projetos cujas tarefas possuem boa cobertura de testes e uma hierarquia de dependências pouco complexa (com poucas dependências indiretas) também se beneficiariam. Por exemplo, o projeto *tip4commit* possui as mesmas qualidades de *odin*, com exceção da boa cobertura dos testes. Mesmo com uma cobertura mais baixa, esse projeto consegue ter mais de 50% de precisão e 80% de revocação, o que significa que a cobertura de testes não é determinante para a qualidade de *TestI* (e *TestIDep*).

A explicação mais lógica para o aumento de precisão do projeto *odin* e a diminuição de precisão do projeto *tip4commit* foi deduzida a partir da análise do grafo de dependências das tarefas. A partir de tal grafo, foi observado que o projeto *odin* é bem menor do que o projeto *tip4commit*, tanto em termos do número total de

Tabela 1: Resultados de precisão, revocação e F_2 por projeto, antes e depois da extração de dependências

Projeto	Precisão antes %	Precisão depois %	Revocação antes %	Revocação depois %	F_2 antes %	F_2 depois %
Diaspora	19, 2 ± 21, 1(10, 7)	12, 1 ± 12, 1(8, 2)	26, 1 ± 27, 8(22, 2)	34, 5 ± 27, 7(27, 4)	16, 2 ± 9, 9(17, 1)	16, 9 ± 11(16, 2)
Odin	25, 5 ± 25(35, 3)	29, 9 ± 26, 5(40, 9)	14, 6 ± 14, 3(16)	31, 1 ± 20, 8(42, 9)	15, 6 ± 15(18, 7)	22, 7 ± 15, 7(23)
oneclickorgs	75 ± 19, 6(76, 1)	70, 7 ± 21, 6(72, 2)	43, 1 ± 4, 4(44, 4)	46, 3 ± 3, 3(47, 6)	46, 9 ± 6(49, 2)	48, 8 ± 6, 2(50, 3)
otwarchive	1, 1 ± 0, 3(1, 0)	0, 9 ± 0, 5(0, 6)	67, 6 ± 26, 5(50)	82, 4 ± 22, 2(100)	4, 9 ± 1, 5(4, 6)	4, 5 ± 1, 7(4, 2)
rapidFTR	11, 7 ± 6, 6(11, 4)	10, 8 ± 6, 1(10, 5)	63, 4 ± 16, 4(61, 2)	70, 4 ± 13, 7(65, 8)	30 ± 13, 2(32, 9)	30, 4 ± 13, 3(33, 2)
tip4commit	52, 9 ± 28, 6(57, 1)	51, 2 ± 27, 8(57, 1)	80 ± 14, 8(76, 2)	83, 2 ± 12, 8(81)	64 ± 12, 6(70, 8)	64, 7 ± 14, 5(73, 8)
tracks	31, 7 ± 30, 6(31, 7)	22, 3 ± 26, 1(22, 3)	58, 5 ± 58, 7(58, 5)	61, 7 ± 54, 2(61, 7)	27, 7 ± 11, 3(27, 7)	18 ± 1, 8(18)
whitehall	19, 1 ± 14, 6(12, 6)	14, 2 ± 12, 3(9, 3)	13, 1 ± 11, 1(10, 4)	20, 3 ± 12, 4(20, 7)	12, 5 ± 9, 3(11, 9)	16, 4 ± 9, 7(15, 9)

Tabela 2: Média de variação de precisão, revocação e F_2

Projeto	Precisão	Revocação	F_2
Diaspora	-7%	+8,4%	+0,7%
Odin	+4,3%	+16,5%	+7,1%
oneclickorgs	-4,3%	+3,1%	+1,9%
otwarchive	-0,2%	+14,8%	-0,5%
rapidFTR	-0,9%	+6,9%	+0,4%
tip4commit	-1,7%	+3,2%	+0,7%
tracks	-9,4%	+3,2%	-9,7%
whitehall	-4,8%	+7,2%	+3,9%

classes, quanto em termos de dependências. De todos os projetos deste estudo, *odin* é o menor, sendo expressiva a sua diferença de tamanho em relação aos demais. Para se ter ideia do impacto do tamanho, o cálculo aleatório da dependência para um arquivo desse projeto teria 1/27 de chance de acerto. O segundo menor projeto da amostra é *otwarchive*, que possui muito mais que o dobro de classes e dependências de *odin*. Apesar disso, novamente o tamanho do projeto não se mostrou determinante para o aumento da precisão; os fatores anteriormente citados também devem estar presentes, como alta taxa de cobertura dos testes, testes bem projetados e uma hierarquia de dependências direta.

Por fim, dentre os resultados, chama a atenção o projeto *otwarchive*. A revocação de *TestIDep* de todas as suas tarefas é superior à 50%. A contrapartida disso é justamente a precisão que, no melhor caso, chega à 1%. Esse resultado se explica, inicialmente, ao analisar a composição de *TestI* e de *TestIDep*. Por exemplo, foi observado que o número de falsos positivos produzidos por *TestI* é enorme: *TestI* possui entre 164 e 233 arquivos e as tarefas compreendem entre 2 e 55 *commits*. Em outros projetos, foi observado que o tamanho de *TestI* extrapola 50 arquivos apenas em caso da tarefa ser constituída por mais de 100 *commits*. Se *TestI* possui muitos falsos positivos, é grande a chance de *TestIDep* também possuir. Ao que parece, o tamanho elevado de *TestI* deve-se ao das tarefas possuírem elevado volume de testes, ou seja, a maioria dos *commits* consistem na adição ou edição de testes. Inclusive, dentre as tarefas analisadas manualmente, foi observado que 50% ou mais dos arquivos alterados pelos *commits* eram arquivos de testes. Ao ter mais conteúdo de entrada para analisar, aumenta a chance de TAITI produzir interfaces maiores, consequentemente aumentando também o tamanho de *TestIDep* em termos de quantidade de arquivos.

Por outro lado, analisando um pouco mais à fundo o projeto *otwarchive*, foi percebido que a cobertura dos testes em geral é baixa, cobrindo pouco mais da metade do código existente. Além disso, ao inspecionar alguns *commits* de suas tarefas, foi percebido que os testes alterados não exercitam nenhum dos arquivos que foram modificados pela tarefa. Logo, é possível que o baixo valor de precisão seja devido ao uso incorreto de BDD, sob a perspectiva de TAITI. Por exemplo, uma dada tarefa possui um teste no arquivo *tag_wrangling_admin.feature*, que exercita várias páginas *web* e os arquivos *app/controllers/user_controller.rb*, *app/models/fandom.rb*, porém nenhum deles é alterado pela tarefa. Ora, sendo a tarefa a criação da funcionalidade validada pelo teste, seria esperado que algum desses arquivos fossem alterados.

6.2 Resultados por tarefa

Através da análise dos resultados na perspectiva das tarefas, foi comprovada diferença estatisticamente significativa entre as médias de precisão, revocação e F_2 referentes à *TestI* e *TestIDep*, conforme resumido na Tabela 3 a seguir, onde p é o *p-value* e r é o tamanho do efeito. Para tanto, foi usado o teste pareado de Wilcoxon, uma vez que os dados são pareados e desviam da normalidade, com $\alpha = 0,05$, bem como o teste de Cohen para avaliar o tamanho de efeito (pequeno=0,10, médio=0,30 e grande=0,50).

Tabela 3: Avaliação da significância estatística dos resultados

Resultado	p	r
Precisão	0,00	0,55
Revocação	0,00	0,76
F_2	0,03	0,29

Além disso, também foi possível constatar uma mudança constante. A maioria das tarefas continuaram com valores de precisão entre 0% e 10%, mas essa taxa foi ligeiramente aumentada: Antes haviam 22 tarefas da amostra com precisão abaixo de 10% e depois passaram a existir 28 tarefas nessa situação. Tarefas com alta precisão também não sofreram mudança significativa: O número de tarefas no intervalo de 60% à 100% de precisão caiu de 12 para 11. Além disso, houve diminuição da quantidade de tarefas com precisão entre 70% e 80% e houve aumento da quantidade de tarefas com precisão entre 60% e 70%. A análise desses valores demonstra que a diminuição da precisão é mais ou menos constante em relação à TAITI e que são poucos os casos em que há uma piora abrupta,

como no caso do projeto *otwarchive*, cuja perda de precisão chega a ser de mais de 30% em certas tarefas.

Em se tratando de revocação, foi percebida a diminuição significativa e constante da quantidade de tarefas com valor abaixo de 30%: Antes haviam 25 tarefas nessa margem e depois restaram 16. Além disso, o valor de revocação das 9 tarefas que antes eram inferiores à 30% passou para valores entre 40% e 50%. Já o número de tarefas com revocação entre 60% e 100% aumentou, passando de 25 para 29, enquanto que o número de tarefas com 100% de revocação passou de 8 para 10.

Os resultados apresentados evidenciam a melhora da revocação, principalmente para tarefas que tem esse valor abaixo de 40%, confirmando que a solução proposta cumpre seu propósito, apesar de diminuir a precisão. Isso significa que a inclusão das dependências em *TestI* diminui o número de falsos negativos porém adiciona falsos positivos. Além disso, os resultados apontam também para um aumento ou diminuição constante dos valores analisados, o que demonstra que, em termos gerais, a solução se comporta de forma similar entre os projetos, mesmo quando estes possuem cobertura de testes satisfatória ou um número reduzido de dependências.

6.3 Análise manual de *TestIDep*

Com base na análise manual dos resultados de mais de 20 tarefas, é possível organizar os arquivos em *TestIDep* em três grupos. O primeiro grupo é composto dos arquivos em pastas *MVC*, ou seja, qualquer arquivo existente nos diretórios *app/controllers*, *app/models* e *app/views*. O segundo grupo é constituído de arquivos de configuração no diretório *config*. Já o último grupo é composto, em sua maioria, por *helpers* e classes de suporte ao código principal existentes no diretório *lib*.

Dessa análise é possível afirmar que o grupo de configurações não beneficia a solução proposta em nenhum momento, já que nunca aparece em *TaskI* (por restrição de TAITI, que já admite que mudanças em tais arquivos não podem ser previstas através de testes de aceitação). Também é possível afirmar que os arquivos do terceiro grupo são responsáveis por poucos acertos de previsão se comparados aos arquivos do primeiro grupo. No geral, arquivos MVC chegam a representar 89% dos arquivos contidos em *TestIDep*, podendo chegar a representar 75% de *TaskI*. A conclusão que se chega é de que a remoção de arquivos da interface que estão contidos na pasta *config* não impacta na revocação e provavelmente melhora a precisão da ferramenta. No mais, a análise de mais casos poderia explicar se manter apenas arquivos MVC na interface melhoraria a precisão sem maiores impactos na revocação.

7 AMEAÇAS À VALIDADE

Nesta seção são discutidas as potenciais ameaças à validade do estudo.

7.1 Validade interna

Como a solução proposta não lida com *metaprogramming*, é possível que nem todas as dependências dos arquivos que constituem a interface da tarefa originalmente calculada por TAITI sejam extraídas. Além disso, devido às limitações descritas na Seção 4, também pode acontecer de serem extraídas menos ou mais dependências do que um arquivo realmente possui. Esse problema é parcialmente solucionado pelo refinamento descrito na Subseção 4.2 referente à identificação de dependências não extraídas corretamente pela biblioteca externa utilizada para esse fim.

7.2 Validade externa

Para esse estudo foi utilizada uma amostra reduzida de 60 tarefas, considerando apenas projetos Rails hospedados no GitHub que utilizam Cucumber como ferramenta para criação de testes de aceitação. Portanto, não é possível generalizar os resultados obtidos. Além disso, a análise de projetos em outra linguagem que não Ruby necessitaria de ajustes na versão original de TAITI a fim de ser utilizado um *parser* adequado e também a sobrescrita de alguns métodos, bem como a remodelagem completa da extensão apresentada neste artigo. Apesar disso, é provável que os resultados obtidos sejam compatíveis com os resultados de outras linguagens estaticamente tipadas.

8 TRABALHOS RELACIONADOS

Muitos estudos dão suporte ao desenvolvimento colaborativo. Ferramentas como FSTMerge [1], JDime [12] e JFSTMerge [7] visam diminuir o trabalho para realizar a integração de código tentando resolver automaticamente alguns conflitos de *merge* que desnecessariamente requerem intervenção manual. Outras ferramentas tentam auxiliar os desenvolvedores a detectar conflitos durante a execução das tarefas, antes mesmo que se concretizem no repositório central, de forma a facilitar a resolução destes, como Palantír [17] e Crystal [5].

Com o objetivo de evitar conflitos, a ferramenta Cassandra [11] analisa um conjunto de restrições para recomendar uma ordem otimizada na qual as tarefas devem ser executadas. Suas restrições são baseadas nos arquivos que cada tarefa deve modificar, que são identificados pelos desenvolvedores, os arquivos que dependem destes e que são automaticamente identificados a partir de *call-graphs*, e a preferência do desenvolvedor.

Nesse sentido, a solução proposta neste artigo se assemelha à Cassandra no sentido de também se valer da análise de dependências no contexto de prevenção de conflitos. No entanto, Cassandra requer que o desenvolvedor informe os arquivos que pretende alterar e, como já discutido, prever manualmente os arquivos que serão alterados pode ser difícil e trabalhoso, levando à erros. A identificação errada dos arquivos a serem modificados pela tarefa implica em previsões erradas sobre o risco de conflitos. O objetivo de TAITI é justamente tentar inferir quais arquivos serão modificados durante o desenvolvimento de uma tarefa, tendo como base os testes de aceitação. Logo, uma possibilidade seria combinar TAITI e Cassandra de maneira que TAITI seria usada para identificar os arquivos que serão alterados pela tarefa e estes, por sua vez, seriam tomados como entrada para Cassandra sugerir uma ordem para os desenvolvedores executarem as tarefas a fim de evitar conflitos. Ao estender TAITI, a solução proposta visa aprimorar a capacidade preditiva desta ferramenta, o que consequentemente também beneficiaria as sugestões geradas por Cassandra quando combinada com TAITI.

9 CONSIDERAÇÕES FINAIS

Neste artigo, foi apresentada uma solução para melhorar as interfaces de tarefas baseadas em testes (*TestI*) calculadas por TAITI na perspectiva de melhorar a capacidade desta ferramenta de predizer os arquivos que os desenvolvedores irão alterar para realizar uma tarefa, assumindo o contexto de BDD. A solução consiste em incluir em *TestI* as dependências dos arquivos que originalmente a constituem, partindo da premissa de que há propagação de mudanças entre arquivos estruturalmente dependentes. A referida solução é importante porque, ao melhorar a capacidade de predizer os arquivos que serão alterados por uma tarefa, torna possível também aprimorar a capacidade de avaliar o risco de conflitos entre tarefas, minimizando a ocorrência de situações em que esse risco é ignorado ou despercebido.

O estudo de avaliação realizado com 60 tarefas de 8 projetos distintos confirma que a solução proposta ajuda a diminuir falsos negativos, ou seja, o número de arquivos que deveriam fazer parte de *TestI*, uma vez que são alterados quando a tarefa é realizada, e não fazem por limitação de TAITI. Especificamente, a revocação das interfaces aumentou em média 32,5% quando comparado com as interfaces geradas por TAITI em sua versão original, enquanto que a precisão reduziu 13%, em média, considerando os resultados na perspectiva do conjunto total de tarefas. O estudo também possibilitou identificar fatores que prejudicam e/ou beneficiam os resultados. Por exemplo, a qualidade de *TestI* afeta diretamente os resultados de *TestIDep*: Se os testes da tarefa não foram desenvolvidos de forma satisfatória ou mesmo se não cobrem adequadamente a funcionalidade subjacente à tarefa, *TestI* é bastante limitada e *TestIDep* provavelmente não tem melhor capacidade preditiva do que *TestI*. Outros fatores, como a complexidade da hierarquia de dependências entre os arquivos e o tamanho do projeto também podem afetar *TestIDep*.

Apesar dos resultados promissores, há melhorias a serem feitas na solução proposta. Por exemplo, conforme explicado na Subseção 4.2, foi adotada uma estratégia baseada em expressões regulares para recuperar o identificador das classes de um resultado de dependências quando este não é devidamente recuperado pela biblioteca externa utilizada. Ao se adotar uma estratégia baseada em árvore sintática, seria possível melhorar a precisão desse processo. Também seria possível adicionar um filtro sobre *TestIDep* para limitar a quantidade máxima de arquivos identificados como dependências, com base no total de arquivos do projeto. Essa medida ajudaria a evitar casos extremos em que o tamanho de *TestIDep* se aproxima do total de arquivos de *models*, *views* e *controllers* do projeto, ou seja, casos em que claramente o resultado não parece condizer com a funcionalidade subjacente à tarefa cuja a interface está associada, devido ao fato de *TestI* ser muito genérica.

De maneira similar, uma outra forma de restringir o conteúdo de *TestIDep* visando melhorar a precisão seria filtrar os arquivos com base no diretório em que estes se encontram. Por exemplo, os arquivos relacionados à configuração do projeto (pasta *conf*), poderiam ser descartados do resultado de dependências. Os resultados obtidos mostram que eles não melhoram a revocação e só pioram a precisão. E, de fato, na perspectiva de tarefas de desenvolvimento, eles não são significativos.

Por fim, tanto *TestIDep* como *TestI* representam um conjunto de arquivos relevantes para a execução de uma tarefa, porém não é possível esperar que todos eles sejam modificados pela tarefa. Os valores de precisão apresentados demonstram isso. Além disso, mudanças de código nem sempre são coesas, ou seja, um desenvolvedor pode modificar um arquivo que não está no escopo de sua tarefa e que, portanto, não é coberto pelos seus testes. Dessa forma, TAITI não consegue alcançar o arquivo, nem tampouco a solução proposta via análise de dependências, dando espaço para a ocorrência de conflitos.

ACKNOWLEDGMENTS

Esta pesquisa foi parcialmente financiada pelo INES 2.0, projetos FACEPE PRONEX APQ 0388-1.03/14 e APQ-0399-1.03/17, projeto CAPES 88887.136410/2017-00, e projeto CNPq 465614/2014-0.

REFERENCES

[1] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 190–200.

[2] Java GitHub API. 2011. Java GitHub API. https://github.com/eclipse/egit-github/tree/master/org.eclipse.egit.github.core. Acessado em: Junho de 2019.

[3] Christian Bird and Thomas Zimmermann. 2012. Assessing the Value of Branches with What-if Analysis. In *20th International Symposium on the Foundations of Software Engineering*. 45:1–45:11.

[4] Lionel C. Briand, Domenico Bianculli, Shiva Nejati, Fabrizio Pastore, and Mehrdad Sabetzadeh. 2017. The Case for Context-Driven Software Engineering Research: Generalizability Is Overrated. *IEEE Software* 34, 5 (2017), 72–75.

[5] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *Transactions on Software Engineering* 39, 10 (2013), 1358–1375.

[6] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. 2015. Assessing semistructured merge in version control systems: A replicated experiment. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–10.

[7] Guilherme Cavalcanti, Paulo Borba, and Paola R. G. Accioly. 2017. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 59:1–59:27.

[8] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. 2003. "Breaking the Code", Moving Between Private and Public Work in Collaborative Software Development. In *International ACM SIGGROUP Conference on Supporting Group Work*. 105–114.

[9] Rebecca E. Grinter. 1996. Supporting Articulation Work Using Software Configuration Management Systems. *Computer Supported Cooperative Work (CSCW)* 5, 4 (1996), 447–465.

[10] JGit. 2010. JGit. https://www.eclipse.org/jgit/. Acessado em: Junho de 2019.

[11] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling. In *International Conference on Software Engineering (ICSE)*. 732–741.

[12] Olaf Leßenich, Sven Apel, and Christian Lengauer. 2015. Balancing precision and performance in structured merge. *Automated Software Engineering* 22, 3 (2015), 367–397.

[13] Dewayne E Perry, Harvey P Siy, and Lawrence G Votta. 2001. Parallel changes in large-scale software development: an observational case study. *Transactions on Software Engineering and Methodology (TOSEM)* 10, 3 (2001), 308–337.

[14] Thaís Rocha, Paulo Borba, and João Pedro Santos. 2019. Using acceptance tests to predict files changed by programming tasks. *Journal of Systems and Software* 154 (2019), 176–195.

[15] Rubrowser. 2017. Rubrowser. https://github.com/emad-elsaid/rubrowser. Acessado em: Junho de 2019.

[16] João Pedro Santos. 2018. Extended TAITI. https://github.com/jpms2/TestInterfaceEvaluationWithDeps. Acessado em: Junho de 2019.

[17] Anita Sarma, David Redmiles, and Andre van der Hoek. 2012. PalantíR: Early Detection of Development Conflicts Arising from Parallel Code Changes. *Transactions on Software Engineering* 38, 4 (2012), 889–908.

[18] John Ferguson Smart. 2014. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications.

[19] Thomas Zimmermann. 2007. Mining Workspace Updates in CVS. In *Fourth International Workshop on Mining Software Repositories*. 11–11.