Melhorando a predição do risco de conflitos de merge em tarefas de desenvolvimento

João Pedro de Medeiros Santos
Recife, Brasil
Email: jpms2@cin.ufpe.br

***Abstract*—Num ambiente de desenvolvimento colaborativo, conflitos de merge podem prejudicar a produtividade de desenvolvedores e comprometer a qualidade do software. Para reduzir esses conflitos, uma maneira de se precaver seria evitando a execução paralela de tarefas conflitantes. Essa estratégia é difícil de ser usada porque depende de uma previsão dos arquivos que serão modificados. Usando BDD (Desenvolvimento baseado em comportamento) e testes de aceitação, uma ferramenta chamada TAITI tenta prever, de forma automática, a interface baseada em testes (arquivos que podem ser executados pelos testes) através de uma análise estática feita no código desses testes de aceitação. Apesar da ferramenta ter uma performance maior do que interfaces geradas aleatóriamente ainda há muito espaço para melhorias. A ferramenta desenvolvida nesse projeto tenta extender a análise estática existente, através da extração de dependências dos arquivos pertencentes à interface gerada, adicionando a ela novos arquivos. Para entender os benefícios dessa ferramenta, precisão e revocação das interfaces são medidas em 84 tarefas de 8 projetos Rails no GitHub. Os resultados dão evidência de que a adição desses arquivos, encontrados através da extração de dependências, impacta positivamente na revocação, aumentando o número de arquivos corretos previstos na interface.**

Keywords: Desenvolvimento colaborativo, Agendamento de tarefas, Desenvolvimento baseado em comportamento, Predição de mudanças em arquivos.

I. 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, é capaz de provocar conflitos entre versões do mesmo arquivo, ou seja, a mesma área de um documento ter sido alterada por dois ou mais desenvolvedores, tornando a junção automática das edições impossível até que esse impasse seja resolvido manualmente pelos autores dessas mudanças.

Conflitos como esse, de mesclagem, ou *merge*, são frequentes, consomem tempo para serem resolvidos [1] e tem sua resolução propensa a erros [2] [3] [4] [5]. Isso afeta negativamente a produtividade no ambiente de desenvolvimento, tendo em vista que uma quantidade relevante de *merges* resultam em conflito [1] [6] [7]. Desenvolvedores têm receio de encontrá-los durante o desenvolvimento, por isso se utilizam de técnicas arriscadas para evitar esses conflitos, aumentando ainda mais suas chances de acontecer [8] [9] [10].

Ao escolher bem as tarefas que cada desenvolvedor irá realizar, e a ordem em que ele vai realizar, é possível que a incidência desses conflitos diminua. Seguindo o pensamento de que é esperado que a quantidade de conflitos de integração diminua quando são realizadas tarefas paralelas que focam em funcionalidades diferentes. Essa escolha, porém, pode ser feita de forma menos apropriada, quando realizada manualmente e baseada em critérios *ad hoc*. Uma previsão automática de quais arquivos serão modificados por essas tarefas também é difícil, mas pode se tornar plausível ao se isolar um contexto específico [11]. No caso da ferramenta a ser estudada neste artigo, TAITI [12], o contexto analisado foi o de design baseado em comportamento (*BDD*) [13].

BDD é uma técnica de desenvolvimento ágil que recomenda a criação de descrições em linguagem ubíqua (Qualquer linguagem não ambígua que se assemelhe à linguagem natural e que possa ser lida por uma ferramenta). Essas descrições devem ser criadas para especificar o comportamento esperado de funcionalidades que serão implementadas durante o período de desenvolvimento de um programa. Baseado nessas descrições é feita a implementação de testes, que, de forma mais específica, irão reproduzir esses comportamentos, utilizando código que será escrito posteriormente.

Como TAITI precisa analisar o funcionamento de tarefas que ainda não foram implementadas, *BDD* acaba sendo uma forma de possibilitar essa análise. Ao utilizar os testes ligados à descrição de uma tarefa, a ferramenta busca por chamadas a métodos, construtores e acessos a constantes, tentando ligar esses valores extraídos à arquivos do projeto. Esses arquivos, por terem sido referenciados nos testes, provavelmente serão modificados na fase de desenvolvimento da tarefa.

Tendo como referência as tarefas sendo executadas e as tarefas a serem implementadas, TAITI busca sugerir quais delas possuem menor chance de gerar conflitos quando desenvolvidas em paralelo. Com o intuito de auxiliar na escolha de tarefas de desenvolvimento.

Em seu estudo, TAITI demonstra que é capaz de prever tantos arquivos quanto uma ferramenta que prevê esses arquivos aleatoriamente, porém com mais precisão. Os resultados do estudo também comprovam que, para funcionar bem, a ferramenta necessita que o máximo possível de funcionalidades existentes tenham seus testes completos. Essa necessidade de uma alta cobertura (quanto do código é exercitado pelos testes) é totalmente ligada ao modo com que é feita a análise, relacionando a qualidade dos testes criados à qualidade da previsão da ferramenta.

Os resultados vistos demonstram que a ferramenta se beneficiaria de refinos em suas técnicas de previsão [12]. Uma forma

de melhorar seria aumentando o alcance da análise e tornando-a um pouco menos dependente da qualidade dos testes. A ferramenta desenvolvida nesse projeto busca melhorar essa previsão através da adição de novos arquivos, descobertos através da análise das classes previstas por TAITI, utilizando árvores sintáticas para buscar por chamadas a definições de classe, módulos e constantes. Essa abordagem busca diminuir a quantidade de falsos negativos e, entender se a utilização de classes que não são exercitadas por testes é útil na predição.

A avaliação dos resultados da ferramenta foi feita comparando valores de precisão e revocação de TAITI sem a ferramenta e de TAITI com a ferramenta. Avaliando 84 tarefas de desenvolvimento, retiradas de 8 projetos Ruby on Rails encontrados no GitHub. Concluindo que a utilização de novas classes não exercitadas diretamente pelos testes aumenta a revocação em 30%, comparado com a ferramenta inicial. É descoberto também que a cobertura dos testes parece não afetar o aumento ou diminuição da precisão.

O restante desse trabalho está organizado da seguinte forma. A seção 2 explica a motivação para o desenvolvimento dessa ferramenta, se baseando principalmente na seção 2, *Motivating example* do artigo *Using acceptance tests to predict files changed by programming tasks* [12]. A seção 3 apresenta uma breve descrição do funcionamento da ferramenta TAITI. A seção 4 explica o funcionamento da ferramenta desenvolvida nesse trabalho. A seção 5 fala sobre os dados usados para a realização dessa pesquisa e o estudo realizado. A seção 6 detalha os resultados, a seção 7 apresenta ameaças à validade do experimento, a seção 8 demonstra possíveis melhorias a se fazer na ferramenta a seção 9 traz trabalhos relacionados. Por fim, a seção 10 traz considerações finais.

II. Motivação

Para entender como interfaces de tarefas baseadas em teste podem ser úteis na previsão de mudanças e na redução de conflitos, considere que Adam e Betty são membros de um time de desenvolvimento *Agile* que está desenvolvendo um sistema Rails [14] de gerenciamento escolar, que armazena a nota de estudantes e permite aos professores visualizá-las. Em uma dada iteração, suponha que Adam foi designado para desenvolver um funcionalidade de avaliação da classe (Tarefa T1). Ele então inicia a criação de um método para computar a média das notas dos estudantes e escreve o código que mostra essa informação extra na página de visualização da classe. Enquanto isso, Betty deve escolher uma nova tarefa. Ela acaba optando por desenvolver uma funcionalidade que ajuda professores a identificar estudantes com notas baixas (Tarefa T2). Começando, então, a desenvolver um método para retornar estudantes com notas abaixo de um limiar e também o código que mostra essa informação na página de visualização da classe. O *backlog* inclui tarefas como permitir ao feed de notícias exibir apenas tarefas recentes (Tarefa T3).

Ao trabalhar independentemente em T1 e T2, cada um em seu repositório local do sistema de controle de versão, Adam e Betty adicionam diferentes métodos (*get_grade_average*

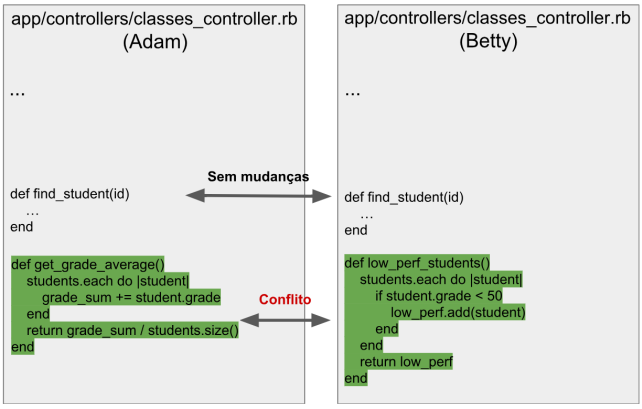

Fig. 1: Exemplo 1: Demonstração do conflito causado pela adição simultânea de métodos na mesma área do arquivo

e *low_perf_students*) ao fim de suas cópias do mesmo arquivo (*app/controllers/classes_controller.rb*), o que levaria, mais tarde, a conflitos de *merge* quando as contribuições fossem integradas, como é mostrado na Fig. 1. Esses conflitos aconteceriam porque ambas T1 e T2 estão associadas à mesma funcionalidade, a de avaliação de classe. Esse conflito poderia ter sido evitado, caso Betty tivesse optado por T3 ao invés de T2, já que T1 e T3 estão associadas a funcionalidades não relacionadas. Com T3 associada a funcionalidade de notificação.

Apesar de ajudar a selecionar uma tarefa que não apresentaria conflitos no exemplo mostrado, a ferramenta pode falhar quando um arquivo não exercitado no teste da funcionalidade precise ser modificado. Considere agora que Adam e Betty terminaram de desenvolver T1 e T2. Adam é designado para criar uma nova funcionalidade que coleta nome e média geral de todos os alunos de uma classe (T4), enquanto Betty escolhe desenvolver uma funcionalidade que permite ao professor editar as notas de seus alunos (T5). Ao consultar TAITI, Betty notaria que ambas as tarefas teriam baixa chance de conflito, pois T4 estaria associado à funcionalidade de avaliação de classe e T5 estaria associado à avaliação de aluno.

Novamente, ao trabalhar independentemente em T4 e T5, cada um em seu repositório local do sistema de controle de versão, Adam e Betty adicionam diferentes métodos (*get_basic_info e edit_grade*) ao fim de suas cópias do mesmo arquivo (*app/models/student.rb*). Levando a conflitos assim que as mudanças forem integradas ao código do repositório central, como ilustra a Fig. 2.

No exemplo visto, para escolher a funcionalidade com menor possibilidade de conflito, TAITI precisa abranger mais do que somente arquivos exercitados pelo conjunto de testes. Ao extrair as dependências dos arquivos exercitados, utilizando a extensão da análise, proposta pela ferramenta apresentada nesse trabalho, teríamos o acréscimo do arquivo *app/models/student.rb* à interface de T4. Pois o arquivo de controlador, *app/controllers/classes_controller.rb*, possui referência ao construtor de *app/models/student.rb*. Como mostrado na Fig 1 através de uma lista de alunos, tanto no código de

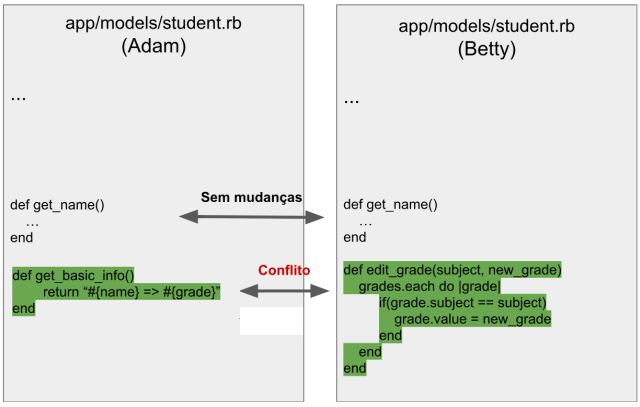

Fig. 2: Exemplo 2: Demonstração do conflito causado pela adição simultânea de métodos na mesma área do arquivo

Adam quanto no de Betty. Essa adição ao conjunto de classes previstas ajudaria TAITI a sugerir uma outra tarefa que não T5, retirando um falso negativo do conjunto de sugestões da ferramenta.

III. Contexto

Antes de explicar o funcionamento da ferramenta desenvolvida durante esse projeto é necessário esclarecer o funcionamento da ferramenta em que ela se baseia e o estudo envolvido no cálculo de sua qualidade.

Com o propósito de prever quais arquivos de produção foram modificados em um *merge*, TAITI atua sobre projetos que utilizam o framework Ruby on Rails e a ferramenta Cucumber como elaboradora de testes, recebendo como entrada tarefas contendo um conjunto de *commits* que resultou num *merge* em algum ponto da história do projeto. Nesse estudo, *commits* continham, primariamente, mudanças em arquivos de produção e em testes Gherkin (Linguagem ubíqua utilizada por Cucumber). Para cada um dos arquivos de teste modificados no *commit* foi feita uma comparação entre o antes e depois da mudança, utilizando uma ferramenta de diferenciação sintática para código Gherkin. O resultado dessa comparação leva à identificação dos cenários de teste atrelados ao *commit*, que por sua vez permitem inferir um conjunto de testes de aceitação.

Tendo em mãos o conjunto de testes de aceitação associado à tarefa, a ferramenta tenta, através da criação de uma árvore sintática, inferir os passos de cada cenário modificado. Esses passos, por sua vez, são a fonte de pesquisa para uma análise estática dos métodos e referências encontrados lá dentro. Não é possível analisar dinamicamente esse código, pois como o contexto trabalhado é baseado em *BDD* a implementação do código dos testes não teria sido concluída.

A análise estática busca por elementos coletados em cada passo dos testes, considerando chamadas de métodos, construtores e acessos a constantes. As chamadas são então mapeadas a uma classe que possua um nome semelhante ao elemento encontrado. Por exemplo, a chamada *@user.update_attributes(user_params)* levaria à busca por uma classe com sufixo

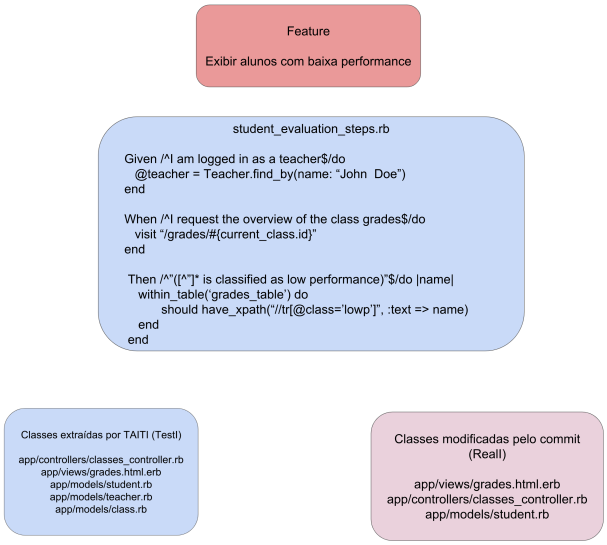

Fig. 3: Exemplo de *TestI* e *RealI*

user.rb. Para chamadas que não contém nenhuma informação sobre o objeto é feita uma busca por classes que contém métodos com o mesmo nome e mesmo número de parâmetros desejado.

Além de extrair referências para classes e seus arquivos, TAITI também conta com um módulo de extração de referências em arquivos com sufixo .html.erb e .haml, que representam as classes ligadas à interface gráfica de projetos Rails. Essa extração é feita utilizando arquivos do tipo *view*, identificados pela ferramenta na análise dos testes. Buscando nesses documentos chamadas a variáveis de instância, *forms, buttons, links* e comandos que renderizam algum arquivo. Coletar as referências exercitadas por esses comandos pode vir a ser útil, pois eles estão associados à ações realizadas por usuários e talvez sejam exercitados no momento da implementação dos passos dos testes.

O conjunto de arquivos identificados pela ferramenta é chamado de interface, mais especificamente *TestI*, no caso da interface gerada pelo conjunto de testes encontrados. No estudo realizado por TAITI, para gerar métricas de qualidade, *TestI* foi comparado à *RealI* com relação à precisão e revocação. *RealI* sendo o exato conjunto de classes modificadas pelos *commits* incluídos numa tarefa, como demonstrado pela Fig. 1.

Com um revocação de 0.49 ± 0.32 em sua amostra de testes e com grandes variações desse valor entre tarefas, TAITI entrega um preditor promissor, que possui maior precisão do que um preditor aleatório e que entrega resultados de qualidade quando a cobertura (quantidade de código exercitada pelos testes de um projeto) dos testes é alta. Essa ferramenta, porém, ainda pode ser melhorada, principalmente ao adotar uma estratégia mais interessante de análise estática.

Uma possível forma de refinar essa análise e melhorar

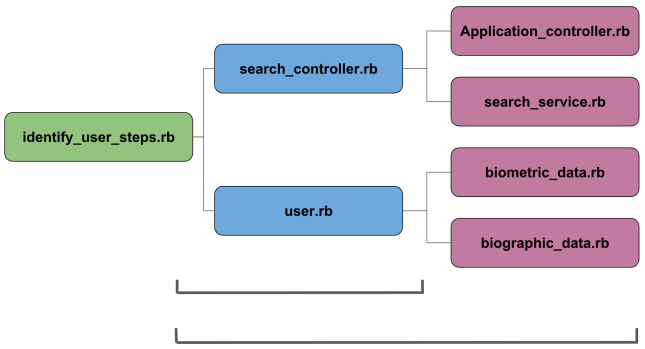

Fig. 4: Exemplo do aumento na quantidade de classes retornadas por TAITI, arquivos retornados normalmente em azul e arquivos adicionais retornados pela nova ferramenta em roxo

sua capacidade preditiva seria aumentando o número de arquivos retornados pela interface. Pelo fato da extração de dependências feita sobre arquivos do tipo *view* ser bem sucedida no experimento original, um passo lógico seria implementar essa extração em todos os arquivos da interface. Outro fator interessante nesse estudo é a possível melhora de precisão e revocação por usar um tipo de análise que não é baseada diretamente em testes.

IV. Desenvolvimento

Com o intuito de diminuir a quantidade de falsos negativos gerados por TAITI, a ferramenta apresentada neste documento busca expandir a análise estática já feita, como demonstrado na Fig. 2, através da extração de dependências dos arquivos retornados por sua interface. Esta seção busca explicar o funcionamento e as etapas do desenvolvimento desse projeto.

A ferramenta recebe como entrada um projeto Rails, que passa por três módulos: Pré processamento, Processamento e extração. Os seguintes termos serão simplificados para melhor entendimento: A classe, ou arquivo, que depende de alguma outra é chamada de **dependente** e a classe em que outra depende será chamada de classee **que gera a dependência**.

A. Pré processamento

Na fase de pré processamento a ferramenta utiliza uma biblioteca Ruby, chamada Rubrowser [15], para gerar um arquivo que contém um conjunto de dependências, demonstradas através de uma interface gráfica, como ilustra a Fig. 5. Para gerar esse resultado, a biblioteca recebe o caminho para a base do projeto em que se quer analisar as dependências, varrendo todos os diretórios, buscando arquivos que contém o sufixo.rb (Essa biblioteca não extrai código de views. Isso, no entanto, não é um problema, pois, como foi mostrado no funcionamento de TAITI, essa extração de dependências já é feita na ferramenta original). Para cada uma dessas classes é feito um *parse*, na tentativa de gerar árvores sintáticas. Caso o arquivo analisado esteja sintaticamente correto, uma árvore sintática é gerada a partir do conteúdo dessa classe. Essa árvore tem seus nós percorridos, permitindo listar chamadas a definições de classe, módulos e constantes encontradas. Permitindo também

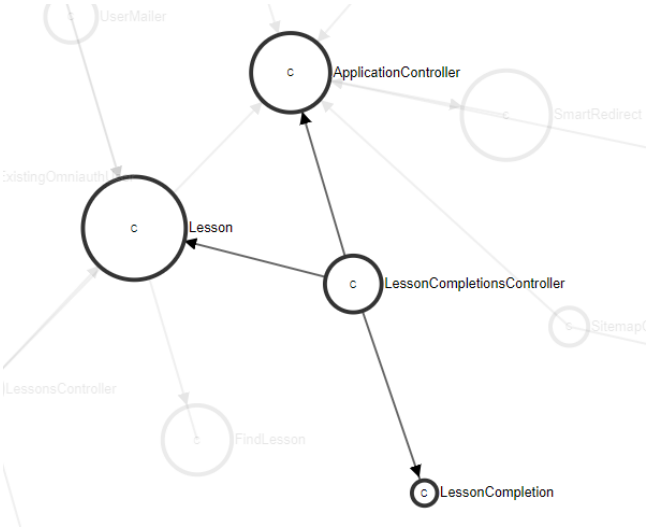

Fig. 5: Exemplo de interface gerada por Rubrowser. Círculos representam classes, setas pretas indicam a hierarquia de dependências

listar Heranças. Ao contrário de TAITI, que busca por todas as classes que contém um método de assinatura igual e as inclui na extração, Rubrowser não encontra dependências em que o nome da classe não foi encontrado no projeto. Também não consegue identificar chamadas Rails através de *meta programming* (Recurso que permite ao desenvolvedor escrever códigos que geram mais código em tempo de execução), pois a análise da biblioteca é feita em tempo de compilação.

A árvore sintática gerada tem seus nós percorridos, gerando uma lista de arquivos. Essas listas são então concatenadas e armazenadas num JSON contendo duas novas listas. A primeira lista, *definitions*, é responsável por enumerar todas as classes Ruby encontradas no projeto, com os atributos *namespace* e *filepath* em destaque. Já a segunda lista, *relations*, é responsável por indicar a relação de dependência entre duas classes, mostrando as classes envolvidas, se ela é circular e o momento onde essa dependência é criada. Seus atributos principais são *namespace* (nome da classe que gera a dependência), *caller* (nome do dependente), *filepath* (caminho absoluto para o dependente) e *line* (linha em que acontece a definição do arquivo que gera a dependência no arquivo do dependente).

Tomando como exemplo a Fig. 5, todas as classes na imagem estariam em *definitions*, e LessonCompletionsController apareceria três vezes na lista de *relations*, uma com cada classe que ela aponta. O tamanho de cada círculo indica o tamanho da classe em linhas.

B. Processamento

Com o JSON criado, a biblioteca gera um arquivo de saída que será usado pelo segundo módulo da ferramenta. Esse módulo lê o arquivo, o processa e extrai apenas o JSON. Armazenando-o em outro arquivo chamado dependencies.json, com formato demonstrado no exemplo da Fig. 6. Esse JSON passa por um processo de refinamento, pois grande parte dos campos *caller* na lista de *relations* chegam

```
{
  "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
    }
  ]
}
```


Fig. 6: Exemplo de JSON com *definitions* e *relations*, mostrando que o arquivo AbuseReportsController depende do arquivo ApplicationController

vazios. Esse problema acontece sempre que a biblioteca não consegue fazer uma conexão entre o nome real do arquivo e o valor encontrado na linha referente à possível dependência. Na sua maioria esses arquivos não encontrados são chamadas a bibliotecas e outros arquivos que não pertencem ao projeto. Porém, classes que contém atribuições do operador "::" (Operador que permite acesso a constantes estáticas fora de seu escopo) e algumas classes que tiveram problemas de compilação costumam aparecer com o problema.

A ferramenta então itera sobre todos os objetos contidos em *relations*, buscando por *callers* vazios. Ao encontrá-los, os campos *filepath* e *line* são utilizados para abrir o documento na linha em que o valor de cada um desses *callers* se encontra, isolando esse valor utilizando expressões regulares e buscando-o na lista de *definitions*, para garantir a precisão. Valores que coincidem com algum campo *namespace* dentro de *definitions* substituem o campo vazio deixado pela biblioteca. Esse refino se faz necessário para remover possíveis chamadas à arquivos que não foram desenvolvidos no escopo do projeto, como chamadas a bibliotecas e também para adicionar possíveis arquivos que tiveram problemas de compilação.

C. Extração

O último módulo é o de extração. A ferramenta recebe como entrada uma lista de caminhos absolutos para os arquivos que terão suas dependências extraídas e busca por cada uma delas na lista de *relations*. Essa busca é feita comparando o campo *file* de cada objeto com o caminho recebido como entrada. Essa procura resulta nos nomes das dependências encontradas na classe, que são usados, em seguida, numa busca sobre a lista de *definitions*. Retornando, finalmente, o caminho absoluto para as dependências encontradas. Após o fim da extração é feita uma limpeza no retorno, removendo arquivos duplicados.

Voltando com o exemplo da Fig. 5, uma busca por */app/controllers/LessonCompletionsController.rb* retornaria uma lista de documentos com: *app/models/LessionCompletion.rb, app/models/Lesson.rb, app/controllers/ApplicationController.rb*

D. Integração à TAITI

A união da ferramenta com TAITI é feita em dois momentos. No instante que antecede a criação da interface, ou seja, no momento após a atualização do projeto para o local em que o *commit* que mesclou as modificações feitas no decorrer da tarefa foi gerado. É realizado o pré processamento e o processamento do projeto, gerando assim o arquivo *dependencies.json*, que é necessário para a busca e extração das dependências. O segundo momento é logo após a formação dessa interface, que ocorre quando TAITI termina de analisar os testes relacionados à tarefa e gera *TestI*. Passando essa interface como entrada para a extração, que irá adicionar sua saída e, com esse novo resultado, que iremos chamar de *TestIDep*, é feita a análise de similaridade.

V. Estudo

Com o objetivo de investigar se a ferramenta apresentada ajuda TAITI a melhor prever mudanças, foi realizado um estudo com o foco, principalmente, em comparar as diferença de precisão e revocação entre as duas versões de TAITI, com e sem extração de dependências. Essa análise utilizou 84 tarefas de 8 projetos distintos, com o intuito de entender se a adição de novas classes à interface gerada diminui o número de falsos negativos apresentados pela ferramenta e se essa introdução de arquivos aumenta o número de falsos positivos. Também foram feitas análises manuais em mais de 20 tarefas para entender melhor os resultados encontrados, observando o comportamento da ferramenta criada em várias situações diferentes.

Para apresentar os testes e resultados da ferramenta é necessário explicar o método com que os dados utilizados foram extraídos. Essa seção também se baseia na seção 5, *Study setup*, do artigo que explica TAITI[12].

A. Seleção de projetos

Utilizando o GitHub como fonte de projetos foi feita uma busca por projetos Rails que usam Cucumber como ferramenta de implementação de testes. Essas buscas utilizam um script implementado para fazer *querys* no banco de dados do GitHub, usando a API GitHub [16] para Java. Como o GitHub não contém um mecanismo que filtre projetos de acordo com suas ferramentas, o script fez o download de todos os projetos Ruby, checando se algum deles continha as ferramentas desejadas. Dado que projetos Ruby contém um arquivo chamado *gemfile*, que lista todas as dependências do projeto, é simples checar o uso de Rails e Cucumber.

De forma a otimizar a busca e melhorar as chances de encontrar projetos que satisfaçam os requerimentos, apenas projetos criados após 2010 foram considerados, pois Cucumber e BDD eram menos populares. Além do mais, versões mais antigas de Ruby e Rails poderiam ser um problema para os *parsers* utilizados.

Em resumo, 61 projetos projetos que estavam de acordo com as restrições impostas foram encontrados.

B. Extração de tarefas

Após obter projetos relevantes que usassem as ferramentas necessárias foi feita uma nova filtragem, removendo projetos que não continham tarefas que contribuíssem com código de produção e testes Cucumber ao mesmo tempo. Esse filtro foi feito para tentar tornar mais fácil a identificação de testes de aceitação (Que são usados para computar *TestI* e *TestIDep*) e código de produção que implemente os requerimentos da tarefa.

Dado que nem todos os projetos encontrados usam identificadores de tarefas em mensagens de commit, são assumidas as seguintes regras: (*i*) contribuições de tarefas são integradas através de merge commits; (*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*) Uma tarefa só pode ser concluída se houver alguma modificação em seu código; (*iv*) testes adicionados ou alterados em uma contribuição são necessários para validar a tarefa.

Para extrair tarefas de um determinado projeto, é feito um *clone* do repositório git, seguido de uma busca por commits de merge (excluindo os resolvidos através de *fast-fowarding*) usando a API JGit [17]. Logo após, é feita a extração de duas tarefas de cada merge commit, correspondendo a cada uma das contribuições feitas. Como uma forma de filtragem, são selecionadas apenas tarefas que modifiquem arquivos de produção e arquivos de teste Gherkin. Outras filtragens incluem a identificação de cenários de teste alterados, calculando assim a interface, descartando tarefas que produzam um *TestI* ou que não modifiquem controladores. Analogamente, tarefas que possuem testes de aceitação parcialmente implementados, não implementados , ou que contém *steps* com erros de compilação também são descartadas.

O resultado dessa mineração diminui consideravelmente o conjunto de projetos que satisfazem os requerimentos da ferramenta. Oito deles foram selecionados, totalizando 84 tarefas escolhidas de forma aleatória com o propósito de testar o desempenho da ferramenta de extração de dependências.

Mostrando como exemplo uma tarefa tirada do projeto diaspora, que teve início na data de 23/08/2015, terminou em 27/08/2015, que envolveu seis desenvolvedores e dezesseis *commits*. A tarefa envolveu a modificação de dois testes, um deles pode ser visto na Fig. 7. A criação desse novo formato de login ocasionou várias mudanças, uma delas aconteceu no arquivo *app/controllers/home_controller.rb*. Nele foi adicionado um novo método, *force_mobile*, que obriga usuários de telefone a serem redirecionados para tela de login.

VI. Resultados

Os testes foram realizados passando como entrada um projeto e seu conjunto de tarefas, recebendo como saída métricas estatísticas (média, mediana e desvio padrão) sobre revocação e precisão calculados. Para todas as tarefas do projeto, revocação é calculado pela intersecção entre *RealI* e *TestI*, dividida por *RealI*, produzindo a quantidade de arquivos corretamente previstos pela ferramenta, ou seja:

```
7 ■■■■■ features/mobile/home.feature

...  ...  @@ -1,12 +1,19 @@
 1    1   @javascript @mobile
 2    2   Feature: Visit the landing page of the pod
 3    3     In order to find out more about the pod
 4    4     As a user
 5    5     I want to see the landing page
 6    6
 7    7     Scenario: Visit the home page
 8    8       When I am on the root page
 9    9       Then I should see "LOG IN"
10   10       When I toggle the mobile view
11   11       And I go to the root page
12   12       Then I should see "Welcome, friend"
     13  +
     14  +     When I am on the root page
     15  +     Then I should see "Welcome, friend"
     16  +     When I go to the mobile path
     17  +     Then I should see "LOG IN"
     18  +     When I go to the mobile path
     19  +     Then I should see "LOG IN"
```

Fig. 7: Exemplo de teste modificado em commit

$$revocação(t) = \frac{TestI(t) \cap RealI(t)}{RealI(t)}$$

Já a precisão é calculada pela mesma intersecção entre *RealI* e *TestI*, sendo que dessa vez a divisão é por *TestI*, ou seja:

$$precisão(t) = \frac{TestI(t) \cap RealI(t)}{TestI(t)}$$

Cada tarefa foi submetida duas vezes para a ferramenta, uma utilizando TAITI sem modificações e outra aplicando a extração de dependências à interface criada.

A. Análise dos projetos

Ao comparar as métricas resultantes é possível afirmar que o revocação aumenta para todos os projetos analisados, com esse aumento variando entre 3.5% e 16.5%, apresentando em média um ganho de 8%. Esse aumento, porém, vem, na maioria dos casos, em troca de uma perda de precisão, que varia entre 0.2% e 9%, com média de 3.5% perdido.

Observando os resultados mostrados na tabela II, alguns projetos se destacam. Tracks e Diaspora, pintados de vermelho, por terem a maior diminuição de precisão. Odin, pintado de verde, por ter sido o único projeto com aumento de precisão e otwarchive, pintado de amarelo, por parecer um bom resultado mas que após análise se mostrou ruim.

A diminuição da precisão se dá por vários fatores que afetam o resultado de *TestIDep*. Analisando manualmente algumas tarefas de Tracks, Diaspora e otwarchive é possível

TABLE I: Média de precisão e revocação antes e depois da extração de dependências

Projeto	Prec. antes	Prec. depois	Rev. antes	Rev. depois
Diaspora	19.1%	12.1%	26%	34.5%
Odin	25.5%	29.8%	14.6%	31.1%
oneclickorgs	75%	70.6%	43.1%	46.2%
otwarchive	1%	0.8%	67%	82%
rapidFTR	11.7%	10.8%	63.4%	70.3%
tip4commit	52.8%	51.1%	80%	83.2%
tracks	31.6%	22.3%	58.5%	61.7%
whitehall	19%	14.2%	13%	20.2%

TABLE II: Média de variação da precisão e do revocação ao utilizar a extração de dependências

Projeto	Precisão	Recall
Diaspora	-7.03%	+8.40%
Odin	+4.3%	+16.5%
oneclickorgs	-4.3%	+3.1%
otwarchive	-0.2%	+14.51%
rapidFTR	-0.9%	+6.9%
tip4commit	-1.7%	+3.2%
tracks	-9.4%	+3.2%
whitehall	-4.8%	+7.2%

afirmar que entre esses fatores se encontram: A qualidade do resultado da interface, em casos em que *TestI* é muito generalista (muitos falsos positivos), fazendo com que a predição de *TestIDep* também seja, em sua maioria, de falsos positivos. Esse caso aparece em todas as tarefas analisadas de otwarchive, onde em uma tarefa específica foram gerados 86 falsos positivos pela ferramenta de extração para 182 falsos positivos gerados por TAITI, com *RealI* contendo apenas um arquivo modificado. Outras tarefas de tracks e diaspora apresentam o mesmo resultado, porém em menor escala, em diaspora a pior tarefa gerou 104 falsos positivos, no total e em tracks 25 falsos positivos, contando o resultado de TAITI e da ferramenta de extração de dependências. Um outro motivo são os momentos em que *TestI* não consegue prever corretamente os arquivos modificados em *RealI*, porque os testes não cobrem corretamente o funcionamento da *feature* também resultam num *TestIDep* pior. Um exemplo claro disso pode ser visto em algumas tarefas de diaspora, onde é feita uma mudança no teste *conversations.feature* que exercita, em seus passos, classes como *app/models/visibilities.rb, app/models/conversation.rb*, porém nenhuma dessas classes mostradas é alterada nos *commits*. Uma hierarquia de dependências muito complicada também aparenta afetar negativamente a precisão, pois introduz vários falsos positivos, tendo os verdadeiros positivos em algum lugar da hierarquia de dependências da classe retornada. Através da análise desses casos é possível apenas inferir que esses projetos já estão maturados, já possuindo várias funcionalidades implementadas. Um exemplo real acontece na tarefa de id 57, pertencente ao projeto Tracks. Uma *feature, test_show_exposes_deferred_todos*, foi analisada por TAITI, resultando em uma interface que incluía a classe *app/controllers/Contexts_controller.rb*, e outras mais. A extração de dependências, quando aplicada à esse controlador resulta em dois arquivos, *app/controllers/application_controller.rb* e *app/models/Context.rb*. Nenhum desses arquivos foi modificado nos commits relacionados à tarefa, porém, a classe *app/models/Null_context.rb* foi. Essa classe é dependência de Context e, poderia ter sido prevista pela nova ferramenta caso dependências indiretas, ou seja, dependências transitivas dos arquivos de *TestI*, fossem analisadas.

Já o aumento de precisão foi identificado num único projeto da amostra, odin. Esse aumento de 4% não está relacionado diretamente a nenhum dos pontos citados para a diminuição de precisão, pois se estivesse, outros projetos que possuem testes bem escritos e uma hierarquia de dependências com poucas dependências indiretas também gozariam desse aumento. Mais especificamente, 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, inclusive, mais do que odin em ambos, demonstrando que esse valor de cobertura não é o único determinante para o bom aproveitamento de TAITI. A alternativa mais lógica para odin ter um aumento de precisão e tip4commit ter uma diminuição foi identificada ao observar as interfaces gráficas geradas pela biblioteca de extração de dependências. O projeto odin é bem menor do que o projeto tip4commit, tanto em questão do número de classes total, quanto em número de dependências apresentadas. De todos os projetos analisados nesse estudo odin é definitivamente o menor, e com uma margem grande para os outros. Para demonstrar seu tamanho, imagine que, uma ferramenta que retornasse qualquer dependência de forma aleatória, teria uma chance de 1/27 de acertar a dependência exercitada na tarefa. O segundo menor projeto analisado é otwarchive que possui muito mais que o dobro de classes e dependências das de odin. O tamanho do projeto, porém, não é determinante para o aumento da precisão, ele precisa estar acompanhado dos pontos já citados, como alta taxa de cobertura dos testes, uma boa descrição desses testes e uma hierarquia de dependências direta.

Um projeto em que é importante levantar questionamentos é o projeto Otwarchive. Todas as suas tarefas retornam, com a ferramenta de extração de dependências, um revocação maior que 50%. O custo desse revocação alta é justamente a precisão, que no melhor caso chega a apenas 1%. Esse resultado se explica, inicialmente, ao analisar *TestI* e *TestIDep*. Nas tarefas analisadas manualmente é possível notar que o número de falsos positivos produzidos por *TestI* é enorme (O número de arquivos retornados pela interface fica entre 164 e 233, ao analisar entre 2 e 55 *commits* por tarefa. Em outros projetos esse valor só passa de 50 em tarefas com a análise de mais de 100 *commits*), fazendo com que os arquivos adicionados na extração de suas dependências tenham alta probabilidade de também serem falsos positivos. Essa grande quantidade de arquivos na interface aparece porque a maioria dos *commits* usados nas tarefas para o teste desse projeto contém modificações apenas em arquivos de teste ou que descreviam testes (Das tarefas em que foi feita a análise manual, 50% ou mais arquivos eram mudanças em código de teste). Isso Resulta

em mais classes encontradas por TAITI, devido à presença de mais funcionalidades a serem analisadas pela ferramenta, consequentemente aumentando também o número de arquivos retornados em *TestIDep*. Analisando um pouco mais a fundo é possível perceber que a cobertura dos testes para esse projeto é baixa, cobrindo pouco mais da metade do código existente. Além disso, observando alguns *commits* é possível perceber que os testes alterados não exercitam nenhum dos arquivos que foram modificados no desenvolvimento da tarefa. Com isso é provável que um valor ruim de precisão seja devido ao uso incorreto da técnica BDD, vendo da perspectiva de TAITI. Um exemplo concreto desse mau uso de BDD é exatamente a criação da funcionalidade de *tag_wrangling_admin.feature* e a não modificação de seu controlador ou view nos arquivos que estão contidos nessa mesclagem. Essa funcionalidade também exercita algumas outra *views* e controladores, como *app/controllers/user_controller.rb*, *app/models/fandom.rb*, porém nenhuma delas tem código alterado até o momento do merge.

B. Análise a nível de tarefa

Observando os resultados de modo a analisar precisão e revocação por tarefa, ao invés da média dos projetos, antes e depois do uso da ferramenta de extração, é possível notar uma mudança constante no comportamento desses valores. Apesar da maior quantidade de tarefas ter continuado entre 0% e 10% de precisão, ela, que possuía 22 tarefas com precisão abaixo de 10%, passou a possuir 28 tarefas nessa margem, após a integração da nova ferramenta. Tarefas com alta precisão também não sofreram uma mudança significativa, o número de tarefas no intervalo de 60% a 100% de precisão caiu de 12 para 11. Com uma diminuição das tarefas entre 70% e 80% e aumento das entre 60% e 70%. A análise desses valores serve para mostrar 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 de otwarchive, que chega a perder mais de 30% em certas tarefas.

Ao analisar os dados de revocação, é possível perceber uma diminuição significativa, e também constante, da quantidade de tarefas abaixo de 30%, saindo de 25 tarefas nessa margem para 16, com essas tarefas indo para os intervalos entre 40% e 50%. Já o número de tarefas entre 60% e 100% de revocação aumentou, passando de 25 para 29, com o número de tarefas com 100% de revocação passando de 8 para 10.

Os resultados apresentados evidenciam a melhora da revocação, principalmente para tarefas que tem esse valor abaixo de 40%, beneficiando o propósito geral da ferramenta, apesar de diminuir a precisão. Demonstrando que a extração de dependências diminui o número de falsos negativos porém adiciona falsos positivos. Esses resultados apontam também para um aumento ou diminuição constante dos valores analisados. Demonstrando que, em termos gerais, a ferramenta se comporta da mesma maneira para projetos que não apresentam um baixo valor de cobertura ou um número reduzido de dependências.

C. Análise de arquivos em TestIDep

Analisando manualmente os resultados de mais de 20 tarefas é possível separar os arquivos que resultam da interface em três grupos. O primeiro é composto dos arquivos contidos nas pastas *MVC*, ou seja, qualquer arquivo contido nos caminhos *app/controllers/, app/models/* e *app/views/*. O segundo grupo é composto de arquivos de configuração contidos na pasta *config*. Já o último grupo é composto, em sua maioria, por helpers e classes de suporte ao código principal, contidos em *lib/*.

Dessa análise é possível afirmar que o grupo de configurações não beneficia o resultado da ferramenta em nenhum momento, pois nunca aparece em *RealI*. Também é possível afirmar que classes do terceiro tipo são responsáveis por poucos acertos de previsão, ao comparar com o primeiro fragmento. No geral, classes MVC chegam a representar 89% dos arquivos contidos em *TestIDep*, podendo chegar a representar 75% em determinados casos de *RealI*. 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 acabaria melhorando a precisão da ferramenta. E que uma análise de mais casos poderia explicar se o uso apenas de classes contidas na pasta MVC melhoraria a precisão sem impactos grandes na revocação (Esse resultado foi visto como positivo para um *TestI* que contém apenas controladores, como é explicado no estudo de TAITI, mas apenas a análise manual não foi suficiente para dizer se a adição de *views* e *models* também seria positiva).

VII. Ameaças à validade do estudo

Nesta seção são discutidas potenciais ameaças ao estudo realizado neste projeto.

A. Validade interna

Pelo fato de utilizar uma biblioteca que não considera *metaprogramming*, (Recurso de Rails que permite ao desenvolvedor escrever código que gera mais código em tempo de execução), tornando muito difícil a extração de dependências desse código de forma estática, é possível que o extrator não tenha adquirido todas as dependências de um arquivo contido na interface. A ferramenta também pode, devido a limitações descritas na seção IV, retornar menos ou mais dependências do que um arquivo realmente possui. Esse problema é solucionado parcialmente pelo refino dos valores encontrados, utilizando o caminho do arquivo do dependente e a linha em que a classe que gera a dependência aparece para buscar pelos nomes das classes que não estão incluídas na lista gerada pelo processamento da ferramenta, como explicado, também, na seção IV.

B. Validade externa

Para esse estudo foi utilizada uma amostra reduzida de 84 tarefas, considerando apenas projetos Rails hospedados no GitHub que utilizam Cucumber como ferramenta para criação de testes de aceitação. Como visto anteriormente, a ferramenta criada nesse projeto é feita para analisar, especificamente,

projetos Ruby. Uma análise de projetos de outra linguagem necessitaria de uma remodelagem completa do código da ferramenta, porém os resultados exibidos idealmente apresentam um mínimo de compatibilidade com os resultados de outras linguagens estaticamente tipadas.

VIII. Melhorias futuras

É necessário refinar o módulo de processamento da ferramenta para que, quando seja necessário descobrir o nome de um arquivo que não foi encontrado pela biblioteca de extração de dependências, ele consiga extrair esses nomes com maior precisão, tornando o módulo de extração o mais preciso possível (Sem contar com as chamadas a *metaprogramming*). O refino desse módulo utilizaria, provavelmente, um parser que gerasse a árvore sintática da classe (ao invés de expressões regulares) e buscasse pela subárvore que contém os elementos vistos na linha apontada pela ferramenta. Com essa subárvore seria possível isolar o nome da classe com mais facilidade e assim comparar esse nome com os nomes contidos na lista de definições.

É possível também, implementar um filtro que limite o máximo de arquivos a ser retornado pela extração de dependências. Esse limite pode ajudar na diminuição de falsos positivos em tarefas onde *TestI* tenha uma quantidade de resultados que não parecem condizer com o número de features extraídas. Seu uso seria recomendável quando a quantidade de arquivos na interface ultrapassasse um limiar previamente configurado, restringindo a adição de mais classes à *TestIDep*. Um bom exemplo real que serviria para ilustrar o uso da ferramenta seria o projeto tracks, onde a precisão foi 29% menor após a extração de dependências e o aumento na revocação foi de apenas 5%. Um filtro que soubesse reconhecer quais as melhores classes para extração poderia minimizar essa perda de precisão.

Além das melhorias apresentadas anteriormente é possível remover todas as chamadas a arquivos do tipo *conf*, pois, como visto nos resultados não há uma melhoria de revocação na adição dessas classes a *TestIDep*, apenas uma piora na precisão da ferramenta. Talvez seja possível remover algumas chamadas à classes do tipo *lib*, diminuindo ainda mais a quantidade de falsos positivos. Essa melhoria, se aplicada, deve ser feita com cuidado, pois, mesmo representando uma fatia pequena dos arquivos de *TestIDep* há a possibilidade de um aumento no número de falsos negativos em relação à ferramenta contendo classes dessa pasta.

IX. Trabalhos relacionados

Muitos estudos dão suporte ao desenvolvimento colaborativo. Ferramentas como FSTMerge [18] e JDime [19] tentam diminuir o trabalho necessário para realizar a integração de códigos ao automaticamente resolvendo conflitos de mesclagem. Outras ferramentas tentam auxiliar na detecção de conflitos antes de sua ocorrência, de forma a facilitar a resolução desses conflitos, como Palantír [2] e Crystal [20]. Outra ferramentas mais recentes, como JFSTMerge [5], buscam entender e melhorar essas ferramentas através de abordagens híbridas, como a união da mesclagem estruturada e não estruturada.

Com o objetivo de evitar conflitos, da mesma forma que TAITI, Cassandra [21] é uma ferramenta que analisa um conjunto de restrições para recomendar uma ordem otimizada na qual as tarefas de desenvolvimento devem ser executadas. Suas restrições são baseadas nos arquivos que cada tarefa deve modificar, que são descritas pelos desenvolvedores, seus arquivos dependentes, que são identificados a partir de *call-graphs* e a preferência do desenvolvedor. No quesito de interfaces de tarefa, o tópico principal dessa pesquisa, Cassandra depende do desenvolvedor para selecionar quais arquivos devem ser listados como modificáveis pela tarefa, ou não. Como visto no estudo de TAITI e nesse, essa forma de inferir pode ser difícil e trabalhosa, levando a possíveis erros na definição desses arquivos. 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 dessa tarefa. De uma forma, as duas ferramentas podem se complementar para que TAITI fosse usada na detecção de arquivos modificados, enviando seu resultado para Cassandra, que faria sua análise baseada em TAITI, e não num possível conjunto de arquivos incorretos. O uso da ferramenta mostrada nesse projeto auxiliaria ainda mais a predição de TAITI, tornando a união com Cassandra mais proveitosa.

X. Conclusão

Neste projeto, foi demonstrada uma estratégia para melhorar a previsão de interfaces baseadas em testes (TestIDep), assumindo um contexto BDD. Essa estratégia, que utiliza um extrator para coletar dependências em chamadas a definições de classe, módulos e constantes, é relevante pois melhora a cobertura da interface gerada, permitindo uma maior predição dos arquivos modificados numa tarefa e aperfeiçoando a escolha de tarefas para desenvolvimento paralelo, possivelmente diminuindo a ocorrência de conflitos.

Ao criar uma ferramenta para refinar a análise estática de TAITI, foi feito um estudo, que confirma a diminuição de falsos negativos, através de uma análise comparativa utilizando 84 tarefas de 8 projetos distintos. Os resultados desse estudo comprovam que a revocação tem aumento médio de 32.5% quando comparado com TAITI sem o uso da ferramenta, já a precisão apresenta uma diminuição média de 13%. Também é possível confirmar alguns pontos que prejudicam e beneficiam a qualidade de sua extração. Pontos como a qualidade do resultado de *TestI*, que pode mudar de acordo com quão bem feitos foram os testes de aceitação e quão alta é a cobertura desses testes, afetam diretamente a quantidade de falsos positivos gerados pela ferramenta. Outros pontos como o formato da hierarquia de dependências, que influencia, também, na quantidade de falsos positivos (Quando existem muitas dependências indiretas). O tamanho dos projetos.

O estudo também demonstra formas de melhorar a extração e seu uso. É possível melhorar o número de dependências encontradas, através de um refino no resultado da biblioteca de extração, possivelmente diminuindo a quantidade de falsos

negativos. Também seria interessante criar um filtro que limite a adição de classes a *TestI* em casos extremos, como momentos em que o número de classes retornadas esteja próximo do total de classes contidas nas pastas de views, controllers e models. Esse filtro ajudaria no decrescimento do número de falsos positivos, diminuindo a diferença entre a precisão de TAITI com e sem a ferramenta. É possível também remover algumas classes relacionadas a pastas específicas, em que a pesquisa mostrou sua inutilidade, como pastas de configuração de frameworks e de banco de dados.

Por fim, *TestIDep*, assim como *TestI*, representa um conjunto de arquivos relevantes para a execução de uma tarefa de desenvolvimento, porém não é possível esperar que todos os arquivos contidos nesse conjunto sejam modificados por essa tarefa. Os valores de precisão apresentados demonstram isso. Outro ponto importante a se notar é que mudanças no código são não coesas, um desenvolvedor pode modificar um arquivo que não esteja no escopo de sua tarefa e causar conflitos da mesma forma.

REFERENCES

[1] T. Zimmermann, "Mining workspace updates in cvs," *Proceedings of the Fourth International Workshop on Mining Software Repositories MSR'07. IEEE Computer Society*, 2007.

[2] A. Sarma, D. Redmiles, and A. van der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Transactions on Software Engineering*, vol. 38, pp. 889 – 908, 2011.

[3] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *SIGSOFT FSE*, 2012.

[4] B. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," 05 2013, pp. 732–741.

[5] G. Cavalcanti, P. Borba, and P. R. G. Accioly, "Evaluating and improving semistructured merge," *PACMPL*, vol. 1, pp. 59:1–59:27, 2017.

[6] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel changes in large scale software development: An observational case study," in *ICSE*, 1998.

[7] G. Cavalcanti, P. R. G. Accioly, and P. Borba, "Assessing semistructured merge in version control systems: A replicated experiment," *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–10, 2015.

[8] D. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39(10), p. 1358 – 1375, 2013.

[9] R. E. Grinter, "Supporting articulation work using software configuration management systems." *Computer Supported Cooperative Work (CSCW)*, vol. 5, pp. 447–465, 12 1996.

[10] C. R. B. de Souza, D. F. Redmiles, and P. Dourish, ""breaking the code", moving between private and public work in collaborative software development," in *GROUP*, 2003.

[11] L. C. Briand, D. Bianculli, S. Nejati, F. Pastore, and M. Sabetzadeh, "The case for context-driven software engineering research: Generalizability is overrated," *IEEE Software*, vol. 34, pp. 72–75, 2017.

[12] T. Rocha, P. Borba, and J. Santos, "Using acceptance tests to predict files changed by programming tasks," 2018, (unpublished).

[13] J. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications, 2014. [Online]. Available: https://books.google.com.br/books?id=2BGxngEACAAJ

[14] Ruby on Rails, http://rubyonrails.org/, Acessado em: Novembro de 2018.

[15] Rubrowser, https://github.com/emad-elsaid/rubrowser, Acessado em: Novembro de 2018.

[16] Java GitHub API, https://github.com/eclipse/egitgithub/tree/master, Acessado em: Novembro de 2018.

[17] JGit, https://www.eclipse.org/jgit/, Acessado em: Novembro de 2018.

[18] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: rethinking merge in revision control systems," in *SIGSOFT FSE*, 2011.

[19] O. Le, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, pp. 367–397, 2014.

[20] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1358–1375, 2013.

[21] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," *2013 35th International Conference on Software Engineering (ICSE)*, pp. 732–741, 2013.