UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MATHEUS BARBOSA DE OLIVEIRA

DETECÇÃO DE CONFLITOS SEMÂNTICOS VIA ANÁLISE ESTÁTICA DE SUBSTITUIÇÃO DE ATRIBUIÇÃO

Recife

2022

MATHEUS BARBOSA DE OLIVEIRA

DETECÇÃO DE CONFLITOS SEMÂNTICOS VIA ANÁLISE ESTÁTICA DE SUBSTITUIÇÃO DE ATRIBUIÇÃO

Trabalho apresentado ao Programa de mestrado acadêmico do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da computação

Área de Concentração: Engenharia de software e linguagens de programação

Orientador (a): Paulo Henrique Monteiro Borba

Coorientador (a): Rodrigo Bonifacio de Almeida

Recife

2022

Catalogação na fonte
Bibliotecária Nataly Soares Leite Moro, CRB4-1722

O48d Oliveira, Matheus Barbosa de
Detecção de conflitos semânticos via análise estática de substituição de atribuição / Matheus Barbosa de Oliveira. – 2022.
82 f.: il., fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2022.
Inclui referências e apêndices.

1. Engenharia de software e linguagens de programação. 2. Substituir atribuição. 3. Conflitos de integração de código. 4. Desenvolvimento colaborativo. I. Borba, Paulo Henrique Monteiro (orientador). II. Título

005.1 CDD (23. ed.) UFPE - CCEN 2022 – 54

Matheus Barbosa de Oliveira

“Detecção de conflitos semânticos via análise estática de substituição de atribuição”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação

Aprovado em: 23/02/2022.

BANCA EXAMINADORA

_____

Prof. Dr. Paulo Henrique Monteiro Borba
Centro de Informática / UFPE
(**Orientado**r)

_____

Prof. Dr. Márcio de Medeiros Ribeiro
Instituto de Computação / UFAL

_____

Prof. Dr. Eduardo Figueiredo
Departamento de Ciência da Computação / UFMG

A toda comunidade cientifica da ciência da computação e especialmente da engenharia de software.

AGRADECIMENTOS

A minha família, em especial a minha mãe Maria Celeide, meu pai Lourival Oliveira e meu irmão Darlan Barbosa, aos meus amigos, em especial aos meus colegas de mestrado e doutorado Ivonildo Gomes e Galileu Santos e a minha namorada Clarissa Mendes pelo apoio nos momentos em que necessitei. Ao professor Paulo Borba pela inquestionável competência com a qual orienta seus alunos e pelas oportunidades oferecidas. Aos membros do SPG pelas experiências e conhecimentos compartilhados, bem como os bons e divertidos momentos vividos. Ao meu co-orientador Rodrigo Bonifácio pela fundamental colaboração neste trabalho. Aos membros de minha banca Eduardo Figueiredo e Márcio Ribeiro pela paciência, disponibilidade e comentários que certamente contribuirão para este e trabalhos futuros. Finalmente, agradeço à FACEPE por financiar minha pesquisa e ao Centro de Informática da Universidade Federal de Pernambuco por todo recurso que me ofereceu.

RESUMO

O processo de desenvolvimento de *software* atual, exceto em casos especiais, é feito de forma colaborativa. Na medida que novos requisitos são levantados, novas tarefas são definidas e alocadas a desenvolvedores diferentes. Os desenvolvedores, no que lhes concerne, adicionam suas modificações em repositórios ou versões separadas e isoladas do código, e posteriormente essas modificações precisam ser integradas em um repositório ou versão central. Esse processo de integração de código é propenso a erros, especialmente se as alterações em diferentes ramos entrarem em conflito. Alguns desses conflitos são mais simples e podem ser detectados pelas ferramentas atuais de controle de versão como o Git, no entanto, ainda necessitam de intervenção humana para resolvê-los, o que afeta a produtividade da equipe. Mas esse não é o único problema, existem também os conflitos semânticos que requerem a compreensão do comportamento do *software*, que está além dos recursos da maioria das ferramentas de mesclagem existentes. Isso faz com que esses conflitos dificilmente sejam percebidos por revisões ou detectados em testes e chegam até o usuário final como defeito no *software*. Esse tipo de conflito ocorre quando, no código integrado, as mudanças introduzidas pela versão de um dos desenvolvedores interferem de forma inesperada com as mudanças introduzidas pela versão de outro desenvolvedor, fazendo com o que um contrato pretendido por uma das versões deixe de ser cumprido. Sendo assim, se fazem necessárias ferramentas que possam detectar conflitos desse tipo no processo de integração, de modo a evitar *bugs* e facilitar a resolução dos mesmos. Nesse sentido, esse trabalho propõe uma análise de substituição de atribuição (*Override an Assignment (OA)*), que visa detectar interferências entre as alterações introduzidas por dois desenvolvedores diferentes, onde caminhos de gravação, sem atribuições intermediárias, para um alvo comum indicam interferência. Também foi realizada a implementação e avaliação de duas abordagens (*interprocedural* e *intraprocedural*) para a análise proposta. Para avaliar as implementações da análise proposta foi usado um conjunto de 78 cenários de integração de código, em que ambas versões integradas modificaram o mesmo método. Esses cenários foram extraídos de projetos open-source Java, minerados por uma ferramenta de mineração de cenários de integração do Github. Os resultados mostram que a análise proposta se mostrou capaz de detectar cenários com substituições de atribuições e com interferência localmente observável entre as contribuições, no entanto, teve uma quantidade considerável de falsos negativos, o que indica que ela não é suficiente para detectar cenários com interferência de forma confiável.

No entanto, a análise proposta poderia ser combinada com outras análises para compor uma ferramenta mais robusta para detecção de conflitos de integração semânticos.

Palavras-chaves: substituir atribuição; conflitos de integração de código; desenvolvimento colaborativo.

ABSTRACT

The current software development process, except in special cases, is done collaboratively. As new requirements are raised, new tasks are defined and allocated to different developers. Developers, as far as they are concerned, add their modifications to separate and isolated repositories or versions of the code, and later these modifications need to be integrated into a central repository or version. This code integration process is error prone, especially if changes in different branches conflict. Some of these conflicts are simpler and can be detected by current version control tools like Git, however they still require human intervention to resolve them, which affects team productivity. But that's not the only problem, there are also semantic conflicts that require understanding the behavior of software, which is beyond the capabilities of most existing merge tools. These conflicts are hardly noticed by reviews or detected in tests, reaching the end-user as a software defect. This type of conflict occurs when, in the integrated code, changes introduced by one developer's version unexpectedly interfere with changes introduced by another developer's version, causing a contract intended by one of the versions to be unfulfilled. Therefore, tools are needed that can detect conflicts of this type in the integration process, in order to avoid bugs and facilitate their resolution. In this sense, this work proposes an analysis of assignment substitution (*Override an Assignment (OA)*), which aims to detect interference between changes introduced by two different developers, where recording paths, without intermediate assignments, to a target common indicate interference. The implementation and evaluation of two approaches (interprocedural and intraprocedural) for the proposed analysis was also carried out. To evaluate the implementations of the proposed analysis, a set of 78 code integration scenarios was used, in which both integrated versions modified the same method. These scenarios were extracted from open-source Java projects, mined by Github's integration scenario mining tool. The results show that the proposed analysis was able to detect scenarios with assignment substitutions and with locally observable interference between the contributions, however, it had a considerable amount of false negatives, which indicates that it is not sufficient to detect scenarios with interference. reliably. However, the proposed analysis could be combined with other analyzes to compose a more robust tool for detecting semantic integration conflicts.

Keywords: overriding assigment; code integration conflicts; collaborative development.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O processo de desenvolvimento de *software* atual, exceto em casos especiais, é feito de forma colaborativa. Na medida que novos requisitos são levantados, novas tarefas são definidas e alocadas a desenvolvedores diferentes. Como, por exemplo, o fluxo de desenvolvimento para uma nova funcionalidade (*feature*) ou para a correção de problemas (*bug*) em um *software* normalmente começa com o desenvolvedor realizando uma cópia do ramo (*branch*) principal do projeto para um novo ramo de desenvolvimento individual, onde será registrado uma sequência cronológica de mudanças de código (*commits*). Em um ambiente colaborativo, é comum que vários desenvolvedores estejam trabalhando em seus ramos individuais simultaneamente, esse processo permite que desenvolvedores trabalhem em suas respectivas tarefas de forma isolada. Apesar de tais benefícios, à medida que as tarefas vão sendo finalizadas, decide-se realizar o *Merge*, de modo a permitir que as alterações (*commits*) sejam incorporadas ao ramo principal. No entanto, na prática, o processo de integração de alterações de ramos múltiplos pode ser difícil e propenso a erros, especialmente se as alterações em diferentes ramos entrarem em conflito (BIRD; ZIMMERMANN, 2012).

Os conflitos acontecem quando diferentes desenvolvedores podem ter alterado a mesma linha ou linhas consecutivas de código físico no mesmo arquivo (*merge conflict*), removido ou renomeado os métodos essenciais (*build conflict*), feito alterações que causam falhas no teste (*test conflict*) ou em casos mais extremos onde o código consegue ser integrado sem erros, não levam a erros de compilação ou de testes, mas levam a falhas durante a execução regular, sendo revelado com um defeito no *software* (*production conflict*).

Nesse sentido, os conflitos de *Merge* e *Build* ocorrem no nível sintático, são mais comuns e mais fáceis de se resolver, mas ainda necessitam de intervenção humana e de um esforço substancial para entendimento e resolução do problema afetando a produtividade e a qualidade. (ESTLER et al., 2014)

Já os conflitos de testes (*test conflict*) e conflitos de produção (*production conflict*) ocorrem no nível semântico. Horwitz et al (HORWITZ; PRINS; REPS, 1989a) especificaram formalmente os conflitos semânticos como sendo: duas contribuições advindas de versões *Left* e *Right* para um programa *Base*(versões envolvidas em um *three-way merge*, onde *Base* é o ancestral comum mais recente às duas versões *Left* e *Right*), originam um conflito semântico se as especificações que as versões se propõem a cumprir em isolado não são satisfeitas na versão

integrada *Merge*. Os conflitos semânticos só podem ser identificados e corrigidos com esforço significativo e conhecimento das mudanças a serem integradas, por isso não são detectados pelas ferramentas de controle de versões atuais, como o Git.

Os conflitos tendem a ocorrer principalmente em ramos individuais mais distantes cronologicamente do ramo principal, pois mais alterações no código serão incorporadas nesse tempo e o desenvolvedor precisará "relembrar" das alterações realizadas (BRUN et al., 2011). O problema torna-se ainda mais significativo à medida que conceitos como *Continuous Delivery (CD)* são colocados em prática, impulsionados principalmente pela indústria e pela necessidade de realizar entregas de novas versões mais rapidamente e com a premissa de não perder a qualidade (ADAMS; MCINTOSH, 2016). Nesse sentido, as alterações dos desenvolvedores devem ser incorporadas ao ramo principal, onde são compilados e testados pelo sistema de Integração Contínua *(CI - Continuous integration)*. Os conflitos que ocorrem nesse fluxo oneram o tempo e a produtividade da equipe de desenvolvimento.

Uma solução parcial seria alertar os desenvolvedores no momento em que estão codificando que as suas alterações podem entrar em conflito com as mudanças de outros desenvolvedores, antes mesmo que elas sejam fundidas com o ramo principal. Com esse intuito, algumas ferramentas vêm sendo propostas (HATTORI; LANZA, 2010; BRUN et al., 2011; NORDIO et al., 2011; KASI; SARMA, 2013). O uso desse tipo de abordagem é extremamente rápida e amplamente utilizado na indústria, porém são limitadas, atendendo apenas a detecção e resolução de conflitos de caráter textual. Outro ponto negativo é a alta possibilidade de disparar falsos positivos que tiram o foco do desenvolvedor. Por isso, podem ser tão prejudiciais quanto os próprios conflitos.

Outra possível solução seria utilizar ferramentas de *Merge* semiestruturadas como o s3m[1]. Esse tipo de tratamento possui vantagens em relação ao método não estruturado, pois analisam parte do contexto do código considerando a sintática e semântica, evitando notificar alguns falsos positivos de conflitos. Existem estudos que fazem a comparação entre às duas abordagens e indicam melhores resultados para o *Merge* semiestruturada (CAVALCANTI; BORBA; ACCIOLY, 2017). Mas é fato que ainda não há evidências suficientes comprovando a efetividade da adoção desse tipo de ferramenta no desenvolvimento colaborativo.

Já com relação ao nível de conflitos semânticos, algumas abordagens diferentes já foram utilizadas, como, por exemplo, geração de testes (SILVA et al., 2020) e uso de análises estáticas

[1] Ferramenta de *Merge* semiestruturada para aplicativos java. https://github.com/guilhermejccavalcanti/jFSTMerge

(FILHO, 2017; BINKLEY; HORWITZ; REPS, 1995a; HORWITZ; PRINS; REPS, 1989a), mas esses são baseados em grafos complexos (*System Dependence Graphs* (HORWITZ; PRINS; REPS, 1989a)) e à medida que as bases de código aumentam têm a performance bastante degradada, o que torna essas soluções difíceis de serem aplicadas em bases de código reais de mais de 50 KLOC (FILHO, 2017). Diante disso, surge a necessidade de explorar simplificações dos algoritmos originalmente propostos, de modo a verificar a sua capacidade de detectar os conflitos semânticos com menor custo computacional.

A simplificação proposta por nosso trabalho visa detectar interferências[2] entre as alterações introduzidas por dois desenvolvedores diferentes através de uma análise de substituição de atribuição (*Override an Assignment (OA)*). Consideramos que pode haver OA quando as alterações (adições e modificações) em um dos ramos podem semanticamente (ou seja, sua execução) envolver uma operação de escrita para um elemento de estado que também está associado a uma operação de escrita envolvida nas alterações (acréscimos e modificações) feitas pelo outro ramo e não há operação de escrita da base entre eles.

As implementações da análise proposta (*interprocedural* e *interprocedural*) foram avaliadas usando um conjunto de 78 cenários de integração de código em que ambas versões *Left* e *Right* modificaram o mesmo método. Esses cenários foram extraídos de projetos *open-source* Java, minerados pelo *miningframework*[3]. Alguns desses cenários foram utilizados em outros estudos relacionados (FILHO, 2017; SILVA et al., 2020; SOUZA; REDMILES; DOURISH, 2003; ACCIOLY; BORBA; CAVALCANTI, 2018).

Para cada um dos cenários utilizados foi realizada uma validação manual estruturada para definir o *ground truth* para OA e interferência localmente observável (LOI). Os resultados da execução das análises foram comparados com o *ground truth* para verificar a capacidade das análises em detectar interferências de OA e conflitos semânticos de modo geral. Por fim, também foram coletadas métricas de precisão, revocação e acurácia usadas para comparar as duas abordagens.

Os resultados apontam que a análise proposta se mostrou capaz de detectar cenários com substituições de atribuições e com interferência localmente observável entre as contribuições, no entanto, teve uma quantidade considerável de falsos negativos. Para OA, esses

[2] Para detectarmos conflitos, precisaríamos saber a intenção dos desenvolvedores, pois um conflito nada mais é do que uma interferência não intencional entre mudanças integradas dos desenvolvedores. Desta forma, buscamos detectar uma interferência que é quando o comportamento das mudanças integradas não preserva a intenção das mudanças individuais. Esse trabalho usa o conceito de interferência como aproximação para conflito por não ser possível inferir as intenções dos desenvolvedores.

[3] Um framework para mineração de projetos git

falsos negativos estão associados a limitações da análise, e acontecem, por exemplo, quando as modificações estão em atributos e não existe um método de entrada. Para detecção de interferências, muitos dos falsos negativos e falsos positivos são esperados, pois, nem todo OA implica necessariamente na existência de interferência localmente observável e vise-versa.

Quando comparamos às duas abordagens da análise, seja para detecção de OA ou LOI, temos uma vantagem para a análise *interprocedural* que teve resultados melhores com relação a todas as métricas utilizadas. No entanto, a abordagem *intarprocedural* leva vantagem no quesito de custo computacional e tempo de execução. Esse resultado também era esperado devido a características e natureza das duas abordagens.

O restante deste trabalho está estruturado da seguinte forma: o Capítulo 2 demonstra um caso motivador para o problema de detecção de conflitos semânticos, demonstrando porque acontece um conflito nesse caso e demonstrando como a análise proposta tenta detectar esse tipo de conflito semântico. O Capítulo 3 descreve em detalhes a análise proposta, especificando o algoritmo e descrevendo detalhes das implementações. O Capítulo 4 descreve as questões de pesquisa, metodologia e avaliação da análise proposta e das abordagens implementadas. Esse capítulo também detalha as métricas utilizadas na avaliação, a infraestrutura utilizada para execução da análise e como foi obtido o *ground truth* dos cenários utilizados, além dos resultados obtidos e das ameaças a validade. O capítulo 5 traz os trabalhos relacionados. Por fim, o Capítulo 6 conclui o trabalho descrevendo os possíveis trabalhos futuros.

2 CASO MOTIVADOR

Para exemplificar melhor o conceito de conflito de integração semântico e como a análise proposta consegue detectar OA, considere o seguinte cenário[1] de exemplo. Na imagem da Figura 1 conseguimos observar a classe `Text` que contém dois atributos, sendo uma *string* que referencia o texto representado pelos objetos dessa classe e dois atributos do tipo inteiro que correspondem a quantidade de correções aplicadas ao texto do primeiro atributo, como remoção de espaços e palavras duplicadas e a quantidade de comentários no texto. A classe `Text` também conta com o método `generateReport()` responsável por gerar o relatório da quantidade de `fixes` necessárias e comentários existentes.

Na imagem da Figura 1 é exibido o arquivo resultado da integração das mudanças em amarelo (Linha 8 foi adicionada pela versão *Left*) e as mudanças em verde (Linha 10 foi adicionada pela versão *Right*). As outras linhas de código originam de uma versão *Base* o ancestral comum mais recente de ambas versões *Left* e *Right*[2]. Nesse caso, podemos observar que as mudanças adicionadas por *Left* e *Right* não estão na mesma linha e nem em linhas consecutivas, assim as alterações que podem ser integradas com segurança em um nível textual e nenhum conflito seria reportado pelas ferramentas de *Merge* atuais.

Figura 1 – Caso de exemplo de conflito integração semântico

```
1     class Text {

3         public String text;
          public int fixes;
5         public int comments;

7         void generateReport() {
    +         countDuplicatedWhitespaces();
9             countComments();
    +         countDuplicatedWords();
11        }
      }
```

Fonte: Elaborada pelo autor (2022)

Observando um pouco melhor as alterações de cada um dos desenvolvedores, temos que o desenvolvedor, *Left* ao adicionar a chamada do método `countDuplicatedWhitespaces()` pretende realizar a contagem dos espaços em branco em um determinado texto e ao final,

[1] Assumindo um cenário de *merge* formado por *commits Base, Left, Right e Merge*, esse último contendo o arquivo apresentado na figura.

[2] Para simplificar, assumimos um único ancestral comum (mais recente). As situações de *Merge* cruzado no git, pode haver mais de uma.

sobrescreve o atributo `fixes` com o valor da contagem. Já o desenvolvedor, *Right* adicionou a chamada ao método `countDuplicatedWords()` que sobrescreve `fixes` com o valor da contagem das palavras duplicadas no determinado texto.

Entre as alterações de *Left* e *Right* temos um comando que vem de *Base*. A chamada ao método `countComments()` realiza a contagem dos comentários no atributo `text`, mas diferente das demais chamadas de métodos, altera o atributo `comments`.

Em seguida, iremos demonstrar que apesar de a integração ter gerado um código sintaticamente válido e livre de conflitos textuais, a mudança feita por *Right* interfere com a mudança feita por *Left*, já que a execução da chamada a `countDuplicateWords()` altera o valor do atributo `fixes` que também é alterado pela chamada a `countDuplicateWhitespace()`. Considerando que a intenção ou especificação da tarefa de *Left* era exatamente armazenar em `fixes` a quantidade de espaços duplos, e que isso não acontece ao final da execução de `generateReport()`, quando `fixes` estará armazenando a número de palavras duplicadas, temos também um conflito semântico[3]. Para entender melhor porque há interferência nesse caso, considere o caso de teste ilustrado na Figura 2.

Figura 2 – Caso de teste que demonstra o conflito semântico na Figura 1

```
    class TextTestSuite {
2
        public void countFixesTest() throws Throwable {
4           Text t = new Text();
            t.text = "the the      dog dog";
6           t.generateReport();
            assertTrue(2, t.fixes);
8       }
    }
```

Fonte: Elaborada pelo autor (2022)

O caso de teste executa o método `t.generateReport()` utilizando como *String* de entrada *"the the dog dog"*. Se executarmos esse teste apenas na versão de *Left* (Figura 3) o resultado seria 1, pois existe apenas um espaço em branco no texto (entre as palavras *the* e *dog*). Já executando apenas no ramo de *Right* (Figura 4) o resultado seria 2, pois "the the" e "dog dog" somam duas palavras duplicadas. Agora pensando apenas no cenário de *merge* (Figura 1) o resultado seria 2, pois o último método a ser chamado é o `countDuplicatedWords()` adicionado por *Right* que sobrescreve `fixes` com o valor da quantidade de palavras duplicadas no texto.

[3] Mesmo que a implementação fosse a outra, os resultados (de OA, conflito, interferência) poderiam ser os mesmos, assumindo especificações óbvias para o caso e o não conhecimento mútuo das tarefas de *Left* e *Right*

Figura 3 – Versão *Left* do caso de exemplo da Figura 1

```
1     class Text {

3         public String text;
          public int fixes;
5         public int comments;

7         void generateReport() {
    +         countDuplicatedWhitespaces();
9             countComments();
          }
11    }
```

Fonte: Elaborada pelo autor (2022)

Figura 4 – Versão *Right* do caso de exemplo da Figura 1

```
1     class Text {

3         public String text;
          public int fixes;
5         public int comments;

7         void generateReport() {
              countComments();
9     +       countDuplicatedWords();
          }
11    }
```

Fonte: Elaborada pelo autor (2022)

Em isolado, ambos desenvolvedores implementaram suas tarefas corretamente, cumprindo os contratos que pretendiam. No entanto, devido a um detalhe da implementação adicionada pela versão *Right*, o código integrado não cumpre o contrato pretendido pelo desenvolvedor de *Left* (`fixes` deveria ser igual a 1). Perceba que a intenção de um dos desenvolvedores (*Left*) não é preservada no *merge* ocorrendo assim uma interferência.

Como as alterações adicionadas pelos desenvolvedores não estão na mesma linha ou em linhas consecutivas, esse caso não seria detectado pelas ferramentas de *Merge* atuais. Casos como esse são muitas vezes difíceis e caros de se detectar e resolver. Na verdade, a menos que um projeto adote um código cuidadoso, práticas de revisão e tenha conjuntos de testes fortes, espera-se que a maioria dos conflitos semânticos escapem para os usuários. Mesmo com tais práticas, conflitos semânticos que escapam, ainda são esperados. Se a detecção não for imediata após integração, pode ser ainda mais difícil corrigir conflitos semânticos, pois a resolução envolve a reconciliação da incompatibilidade semântica comportamental. Em nosso exemplo, teríamos que investigar se o defeito está nas implementações individuais de *Left* e *Right* ou em como um deles interfere no outro. Isto exigiria uma investigação não superficial

que quebrasse o limites de abstração estabelecidos pelas declarações dos métodos chamados em `generateReport()`.

Para evitar esses problemas mencionados, bastaria a análise de substituição de atribuição proposta. Ela conseguiria detectar essa interferência, pois a análise de OA busca por alterações onde desenvolvedores diferentes escrevem na mesma variável (`fixes`) sem atribuições intermediárias.

De fato, `countDuplicatedWhitespaces()` e `countDuplicatedWords()` escrevem em `fixes`, enquanto `countComments()` escreve em outra variável(`comments`). Dessa forma, nossa análise detectaria a interferência e daria o seguinte resultado:

Figura 5 – Exemplo simplificado de um alerta reportado pela análise proposta

```
1 [right(Text, countDuplicatedWords, 8, this.<Text: int fixes> = $stack6)
  => left(Text, countDuplicatedWhitespaces, 10, this.<Text: int fixes> = $stack5)]
```

Fonte: Elaborada pelo autor (2022)

Na Figura 5, temos um exemplo simplificado de um alerta reportado pela análise proposta. O alerta é exibido em formato de *array*, pois pode haver mais de uma interferência detectada por cenário. Cada elemento do *array* conta com os detalhes da modificação de cada um dos desenvolvedores. Os detalhes contém a classe, o método, o número da linha e a instrução em que a substituição de atribuição aconteceu. Nesse caso podemos interpretar como: *Right*, na classe `Text`, no método `countDuplicatedWords`, na linha 10, ao atribuir um valor a variável `int fixes`, interfere em *Left*, na classe `Text`, no método `countDuplicatedWhitespaces`, na linha 8, pois *Left* já havia alterado o valor da variável `int fixes`[4] e não existe uma atribuição intermediária de *Base* na variável `int fixes`.

Nesse cenário específico, apenas o algoritmo *interprocedural* conseguiria esse feito. Isso se dá pelo fato da interferência ocorrer em métodos executados pelo método de entrada que são desconsiderados pelo algoritmo OA *intraprocedural*. Já na imagem da Figura 6, o algorítmico *intraprocedural* seria suficiente e a análise detectaria interferência, pois a variável local `fixes` é alterada pelos dois desenvolvedores no método de entrada.

[4] A notação $stack é utilizada pelo *Soot Framework* para denotar variáveis temporárias no código *Jimple*.

Figura 6 – Caso de exemplo de conflito integração semântico detectado por algoritmo intraprocedural.

```
    class Text {
2
        public String text;
4       public int comments;

6       void generateReport() {
+           fixes = countDuplicatedWhitespaces();
8           countComments();
+           fixes = countDuplicatedWords();
10      }
    }
```

Fonte: Elaborada pelo autor (2022)

3 ANÁLISE DE SUBSTITUIÇÃO DE ATRIBUIÇÃO (OA)

3.1 VISÃO GERAL

A análise proposta, essencialmente, verifica se a execução das mudanças feitas por um desenvolvedor (digamos, *Left*) pode substituir uma Atribuição (OA) da execução das mudanças feitas pelo outro desenvolvedor (digamos, *Right*), ou vice-versa.

Consideramos que pode haver OA quando as alterações (adições e modificações) em um dos ramos podem semanticamente (ou seja, sua execução) envolver uma operação de escrita para um elemento de estado[1] que também está associado a uma operação de escrita envolvida nas alterações (acréscimos e modificações) feitas pelo outro ramo e não há operação de escrita entre eles.

Como vimos na seção anterior, o exemplo da Figura 1 contém uma interferência, visto que *Right* altera o valor da variável `fixes` que já havia sido alterada por *Left* e não existem outras atribuições intermediárias na mesma variável. Como essa interferência não foi planejada, podemos afirmar que se trata de um conflito semântico.

Figura 7 – Caso de exemplo sem interferência

```
1     class Text {
3         public String text;
          public int fixes;
5
          void generateReport() {
7   +         countDuplicatedWhitespaces();
              countComments(); // Sobrescreve atributo fixes
9   +         countDuplicatedWords();
          }
11    }
```

Fonte: Elaborada pelo autor (2022)

Já na imagem da Figura 7, a chamada ao método `countComments()` também vai sobrescrever a variável `fixes`. Com isso, o conflito semântico entre *Right* e *Left* deixa de existir. Resumidamente, *Right* não estará sobrescrevendo uma variável alterada por *Left* e sim por *Base*, o que não pode ocasionar conflitos semânticos.

Seguindo o mesmo raciocínio, as referências a atributos de um objeto como `o.a` e `o.b`

[1] Variável local, parâmetro, campo estático ou de instância, arquivo, fluxo, expressão na instrução de retorno, exceção levantada, etc. Incluindo o estado temporário também. Por exemplo, no código ilustrado na Figura 1 tem como elementos de estado os atributos `text`, `fixes` e `comments`

Figura 8 – Caso de exemplo com objetos

```
1     void m() { // SEM INTERFERENCIA
          Object o = new Object();
3   +     o.a = "exemplo A";
          ...
5   +     o.b = "exemplo B";
      }
7     void m() { // COM INTERFERENCIA
          Object o = new Object();
9   +     o = new Object();
          ...
11  +     o.a = "exemplo";
      }
```

Fonte: Elaborada pelo autor (2022)

são considerados diferentes elementos de estado e podem ser escritas pelas ramificações sem levar a OA. Por outro lado, o também é um elemento de estado diferente, mas não pode ser alterado junto com `o.a` ou `o.b` sem causar interferência.(Figura 8).

Figura 9 – Caso de exemplo com arrays

```
      void m() { // SEM INTERFERENCIA
2         String[] s = new String[];
  +       s[0] = "0";
4         ...
  +       s[1] = "1";
6     }
      void m() { // COM INTERFERENCIA
8         String[] s = new String[];
  +       s = = new String[];
10        ...
  +       s[0] = "0";
12    }
```

Fonte: Elaborada pelo autor (2022)

Algo parecido acontece com *arrays*, onde cada posição corresponde a um elemento de estado diferente. Dessa forma `a[0]` e `a[1]` podem ser escritos por versões de desenvolvedores diferentes e isso não leva a OA. No entanto, a também é um elemento de estado diferente, mas não pode ser alterado junto com `a[0]` e `a[1]` sem causar interferência. (Figura 9).

Outro detalhe importante é com relação à rigorosidade com que a análise detecta interferência. A análise proposta de Overriding Assignment é considerada conversadora. Um exemplo disso pode ser observado na imagem da Figura 10 onde a atribuição do desenvolvedor *Right* está dentro de um comando condicional `if`. Por se tratar de uma análise estática, em casos como esse a análise não conseguiria inferir se a condição do `if` seria verdadeira ou false, assim a análise irá verificar as duas possibilidades o que resultará em interferência para esse exemplo.

Figura 10 – Exemplo de análise com condicional

```
    class Text {
2
        public String text;
4       public int fixes;
        public int comments;
6
        void generateReport() {
8   +       countDuplicatedWhitespaces();
            countComments();
10          if(fixes == 0){
    +           countDuplicatedWords();
12          }
        }
14  }
```

Fonte: Elaborada pelo autor (2022)

3.2 ESPECIFICAÇÃO DO ALGORITMO

De modo geral, a análise proposta consiste em percorrer todos os comandos de um programa a partir de um método de entrada. O método de entrada é o método onde ocorreu a primeira modificação feita pelo desenvolvedor *Left*. A partir disso, para cada um dos comandos é executada uma série de verificações. Primeiramente, a análise verifica se o comando foi adicionado por *Left* ou *Right*, caso positivo, ocorre uma segunda verificação para averiguar se o comando é uma atribuição. Se o comando for uma atribuição, ele é adicionado ao que chamamos abstração[2]. Uma abstração é, de forma simplificada, uma lista que contém o comando e a variável que teve o seu valor alterado na atribuição. Em nossa análise mantemos duas abstrações diferentes, uma com as atribuições de *Left* e outra com as atribuições de *Right*. Além de adicionar as atribuições em suas respectivas abstrações, a análise também vai verificar se a variável envolvida na atribuição está presente dentro da outra abstração. Se estiver, é um indício de interferência. Nesse caso, a análise irá criar um alerta e manter em uma lista até o fim da execução. Se o comando foi adicionado por *Left* ou *Right*, mas não é uma atribuição, verificamos se é uma chamada de método. Nesses casos, é onde está a diferença dos algoritmos *intraprocedural* e *interprocedural*. No *intraprocedural* as chamadas de métodos são ignoradas e a análise segue para o próximo comando sem modificar a abstração. Na abordagem *interprocedural* ao se deparar com uma chamada de método, a análise irá acessá-lo e realizará a análise de todo o seu corpo, podendo modificar a abstração, e só ao final retornará e seguirá para o próximo comando. Se o comando foi adicionado por *Left* ou *Right*, mas não

[2] Abstração é um termo normalmente usado quando se utiliza o Soot Framework. Outro termo comumente encontrado na literatura é lattice.

é uma atribuição e nem uma chamada de método, a análise seguirá para o próximo comando. Caso o comando não tenha sido adicionado por *Left* ou *Right*, logicamente esse comando é de *Base*. Para esses casos, a análise também vai verificar se o comando é uma atribuição e se a variável envolvida nessa atribuição está contida em uma das duas abstrações. Caso estiver, essa atribuição será removida da abstração. Quando essa atribuição de *Base* for removida ou se o comando atual não for uma atribuição, a análise seguirá para o próximo comando. Ao fim da verificação em todos os comandos é retornada a lista de interferências detectadas pela análise. Um pseudocódigo do algoritmo pode ser observado na imagem da Figura 11.

Figura 11 – Pseudocódigo da especificação do algoritmo

```
algoritimo "overriding assignment"
var
  entry_point, abstraction_right abstraction_left, conflics

procedimento checkConflict(command, abstraction);
inicio
  se abstraction contem command então
    conflics <- command
fim

procedimento traverse(entry_point);
inicio
para command em entry_point então
  se command isTagged entao
    se command isAssignment então
      se isRight entao
        checkConflict(command, abstraction_left)
        abstraction_right <- command
      senão se isLeft então
        checkConflict(command, abstraction_right)
        abstraction_left <- command
      senão
        conflics <- command
        abstraction_left <- command
        abstraction_right <- command
      fimse
    senão se isMethodCall então
      traverse(command) // Apenas Interprocedural
    fimse
  senão
    se command isAssignment então
      kill() // Remove da Abstracão
    fimse
  fimse
fimfor
escreva(conflics)
fim

inicio // Programa Principal
  traverse(entry_point)
fim
```

Fonte: Elaborada pelo autor (2022)

Nesse trabalho, implementamos duas abordagens diferentes (*intraprocedural* e *interprocedural*) para a análise de substituição de atribuição proposta. Uma análise *intraprocedual* é

menos custosa, mas não consegue detectar interferências quando ocorrem em métodos executados a partir do método de entrada, ou acabam detectando interferências quando, na verdade, não existem. Isso acontece pelo fato de desconsiderar acessar outros métodos, como podemos ver no exemplo da Figura 12

Figura 12 – Exemplo de Interprocedural e Intraprocedural

```
1     class Example {
3         private int x;
5         void m() {
+             x = 0;
7             base();
+             x = 2;
9         }
11        void base() {
              x = 1;
13        }
      }
```

Fonte: Elaborada pelo autor (2022)

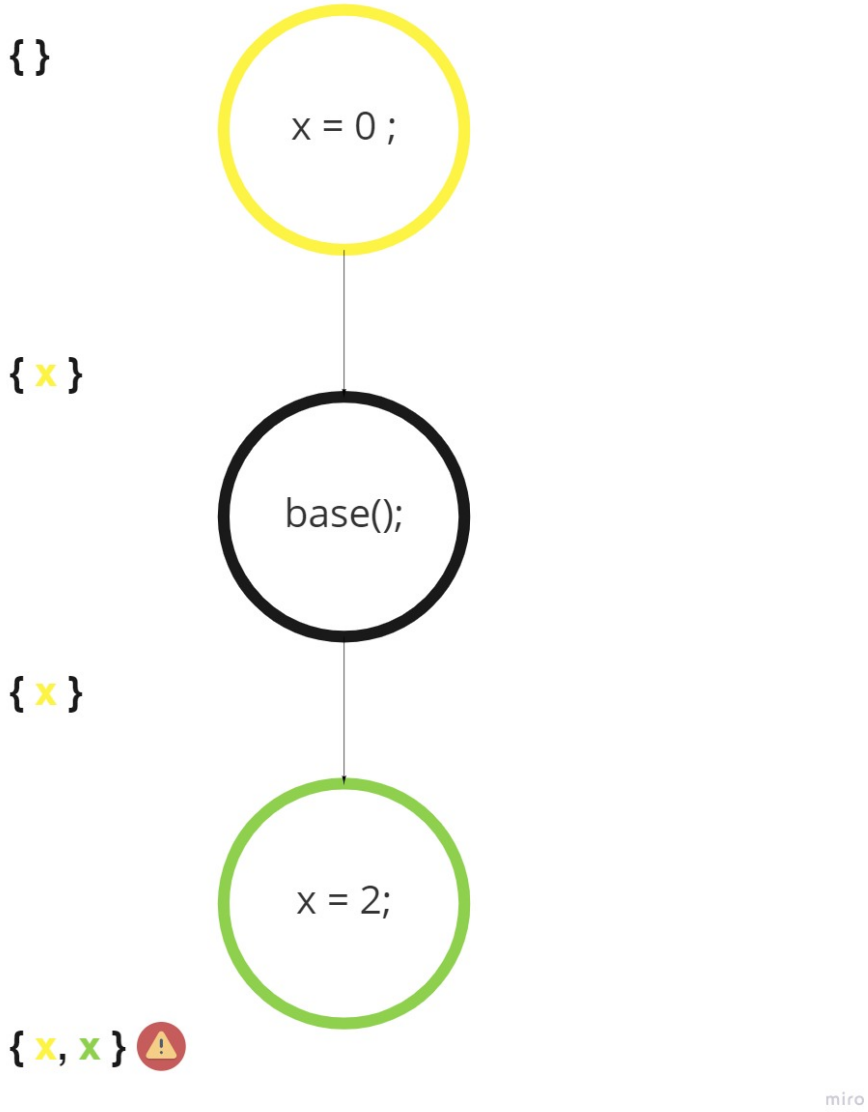
Partindo do ponto em que a análise tem como método de entrada `m()`, a abordagem *intraprocedural* iria detectar incorretamente uma interferência entre a alteração de *Right* na linha 8 (verde) para a alteração de *Left* na linha 6 (amarelo), pois ao analisar a chamada ao método `base()` na linha 7, ela simplesmente desconsidera o comando, não altera a abstração da análise e segue adiante para o próximo comando. A chamada ao método `base()` realiza uma atribuição intermediária a *Left* e *Right* no atributo x. Por sua vez, a abordagem *interprocedural* quando analisa uma instrução de chamada de método, recupera o corpo do método que está sendo chamado e o analisa, eventualmente alterando a abstração da análise, para só depois voltar para analisar a instrução seguinte. No exemplo, ela analisaria o corpo de `base()` e retiraria o x da abstração e não detectaria interferência para esse cenário.

3.2.1 Análise intraprocedural

A abordagem *intraprocedural* é mais barata com relação aos custos computacionais porque ao analisar um comando de chamada de método, ela simplesmente o desconsidera, não alterando a abstração da análise e seguindo adiante para o próximo comando. Justamente por essa característica, esse tipo de abordagem acaba deixando de detectar algumas categorias de interferência e no caso de OA, acaba por vezes detectando interferências que não deveriam

ser detectadas. Como vimos no exemplo da Figura 12.

Figura 13 – Fluxo de execução *intraprocedural* para exemplo da Figura 12



Fonte: Elaborada pelo autor (2022)

Para entendermos melhor como acontece toda a execução da análise de substituição de atribuição *intraprocedural*, podemos observar o exemplo simplificado da Figura 13. Nesse exemplo, o círculo amarelo representa a linha 6 que foi adicionada por *Left*, o círculo preto representa um comando de *Base* e seguindo o mesmo padrão, o círculo verde representa a linha 8 adicionada por *Right*. O símbolo `{}` representa a abstração da análise.

A abstração inicialmente começa vazia, mas logo após a análise do comando `x = 0`, ela irá receber a variável x que recebeu uma atribuição de *Left*. Seguindo para o próximo comando (`base()`), por se tratar de uma chamada de método, o algoritmo *intraprocedural* irá ignorar

e seguirá para o próximo comando sem alterar a abstração. Por último, será feito a análise do comando `x = 2` e novamente o x será adicionado a abstração como uma atribuição de *Right*. No entanto, a variável x já existe na abstração, marcada como *Left*, nesse caso uma interferência de *Right* para *Left* seria reportada pela análise.

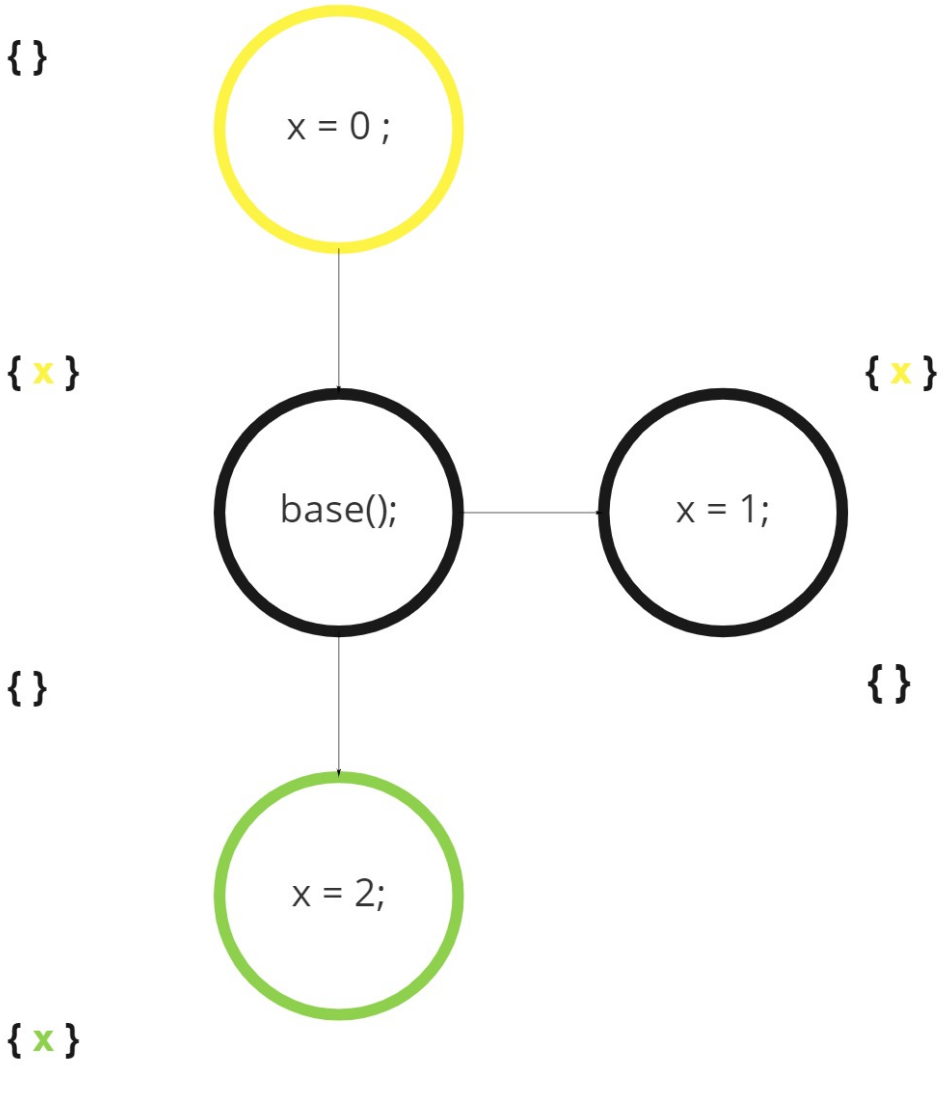
Se analisarmos rapidamente a Figura 12 vamos observar que essa interferência não deveria existir, pois x também foi atribuído pelo comando `x = 1` no método `base()`. No entanto, pela característica da análise *intraprocedutral*, as chamadas de método que ocorrem no método de entrada são ignoradas. Para obter um resultado mais correto para esse exemplo seria necessário usar um algoritmo *interprocedural*.

3.2.2 Análise interprocedural

Diferente da *intraprocedural*, a abordagem *interprocedural* é mais robusta e, consequentemente, mais cara. Quando a análise *interprocedural* recebe um comando de chamada de método, ela recupera o corpo do método que está sendo chamado e o analisa, eventualmente alterando a abstração da análise, para só depois voltar para analisar o comando seguinte à chamada de método.

Para entendermos melhor como acontece todo o fluxo da análise de substituição de atribuição *interprocedutal*, podemos observar o exemplo simplificado da Figura 14. Como já sabemos, a abstração inicialmente começa vazia, ao analisar o comando `x = 0`, adicionado pelo desenvolvedor *Left* a abstração vai ser alterada e passará a conter a variável x. Seguindo a análise, chegou a vez do comando `base()`, ao se deparar com um comando de chamada de método, a análise irá fazer uma cópia da abstração e analisará o corpo do método que está sendo chamado. Dentro do corpo de `base()` temos o comando `x = 1`, como esse comando não foi adicionado nem por *Left*, nem por *Right* a análise irá verificar se já existe uma variável x na abstração e fará a remoção. De volta à linha inicial, já com a abstração vazia, a análise seguirá para o próximo comando `x = 2` e adicionará x novamente na abstração, mas dessa vez como um comando de *Right*. Ao fim da execução, a abstração contará apenas com o x adicionado por *Right* e não irá detectar nenhuma interferência.

Figura 14 – Fluxo de execução *interprocedutal* para exemplo da Figura 12



Fonte: Elaborada pelo autor (2022)

3.3 IMPLEMENTAÇÃO

As abordagens da análise proposta foram implementadas utilizando a linguagem de programação *Java* e o *framework* de análise estática *Soot Framework*[3]. As implementações utilizam a *API* do *Soot Framework* para executar o algoritmo descrito anteriormente para os comandos do programa de entrada. A análise pode ser encontrada no repositório conflict-static-analysis[4] disponível no Github, que reúne as implementações utilizadas para esse e outros trabalhos.

[3] http://soot-oss.github.io/soot/

[4] Este projeto visa implementar uma biblioteca de análises estáticas para detectar conflitos de mesclagem semântica. https://github.com/spgroup/conflict-static-analysis

As análises recebem como entrada o código compilado da versão integrada empacotado em um arquivo *.JAR* e um arquivo CSV, como ilustrado na Figura 15, em que cada linha representa uma linha de código modificada, contendo as informações do nome da classe modificada, o número da linha modificada para a versão integrada e se essa linha deve ser considerada *Left* ou *Right*.

Figura 15 – Exemplo adaptado do arquivo CSV de entrada para o caso motivador da Figura 1

```
Text,left,8
Text,right,10
```

Fonte: Elaborada pelo autor (2022)

O arquivo CSV com as linhas modificadas por *Left* e *Right* é gerado pela ferramenta miningframework que compara as versões do programa usando a ferramenta de comparação sintática de código *Java diffj* para obter as linhas adicionadas ou modificadas por cada desenvolvedor sem considerar mudanças como de espaços em branco e comentários.

Para esse trabalho, as linhas de código deletadas não são coletadas, pois a análise de substituição de atribuição proposta não as considera. As linhas removidas são consideradas para análise manual de interferência, mas não utilizadas pela análise proposta. Apesar de funcionar para a maioria dos casos, a abordagem de anotação de linhas tem limitações, o que pode fazer com que a análise perca interferências. Um dos casos onde essa estratégia falha é quando instruções divididas em várias linhas são modificadas, ocasionando falha em sua marcação, pois a implementação considera a linha em que a instrução começou para comparação com a lista de números de linha de entrada.

As análises retornam como saída uma lista com os pares de instruções *Left* e *Right* em que foram encontrados interferências na substituição de atribuições. (Figura 2.5)

A implementação foi testada com um conjunto de casos de teste, e posteriormente foi testada utilizando o conjunto de dados para avaliação da solução. Nessa fase foram detectados *bugs* e falhas de performance, consertados para a execução final.

4 AVALIAÇÃO

As seções a seguir descrevem as questões de pesquisa, metodologia utilizada para avaliar a análise proposta, os resultados obtidos e às ameaças a validade.

4.1 QUESTÕES DE PESQUISA

Considerando o que foi discutido nas seções anteriores, foram propostas três perguntas de pesquisa que uma vez respondidas ajudam na compreensão da utilidade da análise proposta.

Q1. A análise proposta é capaz de detectar cenários com substituições de atribuições e com interferência localmente observável entre as contribuições?

Q2. Qual a precisão, revocação e acurácia de OA para detectar conflitos semânticos?

Q3. Qual abordagem implementada tem o melhor desempenho na detecção de conflitos semânticos?

4.2 METODOLOGIA

Para avaliar a capacidade da análise proposta de detectar substituição de atribuição entre as contribuições de diferentes versões, foi utilizado um conjunto de dados com 78 cenários de integração de projetos *open-source Java* extraídos do *Github*. Cada um desses cenários representa um método ou campo de uma classe modificado em ambas versões *Left* e *Right*. Esse tipo de cenário em específico foi escolhido, pois, são mais suscetíveis a ocorrência de interferência, e podem mais facilmente serem analisados manualmente para confirmar a ocorrência de interferência. No entanto, é importante perceber que esses não são os únicos casos em que podem acontecer conflitos semânticos. Alguns desses cenários foram aproveitados de outros trabalhos relacionados, sendo 5 de Da Silva et. al. (SILVA et al., 2020), 19 de Barros Filho (FILHO, 2017), 26 de De Souza et. al. (SOUZA; REDMILES; DOURISH, 2003) e 7 de Cavalcanti et. al. (CAVALCANTI et al., 2019). Os outros 21 cenários foram minerados a partir de uma lista de projetos *open-source* populares utilizando o miningframework[1]. Sete desses cenários precisam de adaptações realistas.

A ferramenta miningframework foi utilizada para coletar as linhas modificadas ou adicionadas por cada uma das versões *Left* e *Right* em cada um dos *commits* de integração. A

[1] Um framework para mineração de projetos git. https://github.com/spgroup/miningframework

ferramenta também foi utilizada para gerar os arquivos *JAR* com o código compilado para cada um dos *commits* de integração, no entanto, para os 61,5% dos casos, os arquivos *JAR* não puderam ser gerados automaticamente, eles foram gerados manualmente. Todos os arquivos utilizados para avaliação, assim como as instruções para gerar os arquivos *JAR* que foram gerados manualmente podem ser encontrados no projeto mergedataset[2]. Um resumo da amostra utilizada pode ser observada na Tabela 4.1. A tabela completa pode ser observada no Apêndice B ou na tabela disponível no Apêndice Online [3]

Tabela 1 – Resumo da amostra utilizada do mergedataset.

Descrição	Quantidade
Total de Projetos	35
Total de Cenários	78
Cenários Reais	71
Cenários Realistas	7
Cenários com Alterações em Métodos	75
Cenários com Alterações em Atributos	3

Fonte: Elaborada pelo autor (2022)

Devido mapeamento das linhas do código-fonte, extraídos a partir do *bytecode*, que pode ser impreciso, alguns cenários precisaram passar por alterações realistas. Esse problema ocorre principalmente nos casos em que há chamadas de método encadeadas (como no exemplo da Figura 16 usando o padrão de projetos *builder*. O compilador divide as chamadas de método em instruções separadas, mas pode acontecer que as linhas sejam marcadas incorretamente. Isso está ligado ao modo como a associação do número da linha é armazenada no arquivo de classe (*.class*) e no histórico do compilador *javac*. A tabela de números de linha contém apenas entradas associando números de linha a um local de código que marca seu início. Portanto, todas as instruções após esse local são consideradas como pertencentes à mesma linha até o próximo local que foi explicitamente mencionado na tabela. Como as informações detalhadas ocupam espaço e a especificação não exige uma precisão específica para a tabela de números de linha, os fornecedores do compilador tomaram decisões diferentes sobre quais detalhes incluir. No passado, ou seja, até o Java 7, o *javac* apenas gerava entradas da tabela de números de linha para o início das declarações. A partir do Java 8, o *javac* gera entradas adicionais para invocações de método em uma expressão que abrange várias linhas. Já o compilador do Eclipse

[2] Este repositório agrupa um conjunto de cenários de mesclagem com conflitos semânticos conhecidos, que foram coletados de estudos relacionados. https://github.com/spgroup/mergedataset

[3] https://url.gratis/5WmiEJ

não possui o histórico do *javac*, mas foi projetado para produzir entradas de número de linha para cada uma das expressões, no entanto, esta maior precisão também resulta em arquivos de classe um pouco maiores. Como solução, realizamos simples refatorações de extração de variáveis (*Extract Variable*) no código-fonte do projeto (Figura 17) para que o comportamento original do programa fosse preservado, mas que o *bytecode* tivesse o mapeamento esperado das linhas. Esse tratamento foi aplicado em três cenários.

Figura 16 – Parte do código do cenário real do commit f3d6309 do projeto elasticsearch.

```
Settings settings = Settings.settingsBuilder()
    .loadFromStream(json, getClass().getResourceAsStream(json))
    .put("path.home", createHome())
    .build();
```

Fonte: Elaborada pelo autor (2022)

Figura 17 – Adaptação realizada para o cenário ilustrado na Figura 16.

```
Settings.Builder settingsBuilder = Settings.settingsBuilder();
Settings.Builder loadFromStream = settingsBuilder.loadFromStream(json,
    getClass().getResourceAsStream(json));
Settings.Builder put = loadFromStream.put("path.home", createHome());
Settings settings = put.build();
```

Fonte: Elaborada pelo autor (2022)

Outra adaptação necessária, está ilutada na Figura 18 sendo aplicada em quatro cenários. Consiste basicamente adicionar um método `callRealisticRun` que será utilizado com método de entrada para execução da análise e tem a função de instanciar os parâmetros do método que teve modificações de *Left* e *Right*, essa estratégia foi usada apenas pela abordagem *intraprocedural* e foi necessária, pois a análise não reconhece parâmetros de tipos não primitivos que não foram instanciados previamente no código.

Figura 18 – Adaptação realizada no cenário do commit d896886 do projeto elasticsearch.

```
    private void callRealisticRun() throws IOException {
        DocWriteResponse docWriteResponse = new DocWriteResponse() {};
        docWriteResponse.writeTo(new BytesStreamOutput());
    }
```

Fonte: Elaborada pelo autor (2022)

Esses cenários modificados foram registrados na pasta *realistc/oa* dentro de cada cenário no mergedataset com a descrição das mudanças.

O processo de coleta dos dados utilizados está ilustrado pela Figura 19. A ferramenta de mineração recebe como entrada um arquivo CSV com uma lista de repositórios no *Github* e cria *forks* dos repositórios para configurar uma ferramenta de CI (Integração Contínua). Logo após o *miningframework* cria o cenário contendo os arquivos das versões *Base*, *Left* e *Right* que serão utilizados para coletar as linhas modificadas por cada um dos desenvolvedores. Em seguida, a ferramenta executa o processo descrito na caixa tracejada para cada um dos *commits* de *merge* envolvido no *three-way merge*[4] dos projetos passados como entrada.

Figura 19 – Fluxo de coleta dos dados sobre os cenários de integração

Fonte: Elaborada pelo autor (2022)

[4] Um *three-way merge* acontece quando dois conjuntos de alterações em um arquivo base são integrados à medida que são aplicados, em vez de aplicar um e, em seguida, integrar o resultado com o outro.

Para cada *commit*, a ferramenta primeiro checa se existem métodos ou atributos de classe modificados por ambas as versões *Left* e *Right*, caso não existam, o *commit* é ignorado. Caso existam, a ferramenta continua o processo de coleta de dados. Primeiramente, coleta os números das linhas das modificações de *Left* e *Right*, faz o replay do *merge* e depois verifica a existência de conflitos de integração textuais e registra em uma planilha, e por fim, envia o cenário para geração dos arquivos *JAR* em uma ferramenta de CI. A ferramenta gera um arquivo de configuração para a plataforma de CI *Github Actions* baseado no modelo ilustrado na imagem da Figura 20) [5], que usa os sistemas de *build Maven* ou *Gradle* para gerar os arquivos JAR.

Figura 20 – Template utilizado pela classe GithubActionsPlatform no miningframework para gerar arquivo de configuração para a plataforma de CI.

```
name: Java Build
on: [push]
jobs:
    build:
        runs-on: ubuntu-latest

        steps:
            - uses: actions/checkout@v2
            - name: Set up JDK ${JAVA_VERSION}
              uses: actions/setup-java@v1
              with:
                java-version: ${JAVA_VERSION}
            - name: Build
              run: ${buildCommand}
            - name: Generate tar
              run: |
                mkdir MiningBuild
                find . -name '*.jar' -exec cp {} . \\;
                tar -zcvf result.tar.gz *.jar
            - name: Create release
              id: create_release
              uses: actions/create-release@v1
              env:
                GITHUB_TOKEN: \${{ secrets.GITHUB_TOKEN }}
              with:
                tag_name: \${{ github.ref }}
                release_name: fetchjar-${identifier}
                draft: false
            - name: Upload jar
              id: upload-release-jar
              uses: actions/upload-release-asset@v1
              env:
                GITHUB_TOKEN: \${{ secrets.GITHUB_TOKEN }}
              with:
                upload_url: \${{ steps.create_release.outputs.upload_url }}
                asset_path: ./result.tar.gz
                asset_name: result.tar.gz
                asset_content_type: application/gzip
```

Fonte: Elaborada pelo autor (2022)

[5] Um exemplo de arquivo gerado para um projeto real (jsoup) pode ser visto em: https://gist.github.com/barbosamaatheus/073dc8caa24e3e775e2483629c476856.

Após executar o processo descrito anteriormente, para cada um dos *commits* de *merge*, a ferramenta baixa os arquivos *JAR* gerados na plataforma de CI e gera os arquivos *CSV* de entrada (Figura 21).

Figura 21 – Exemplo do arquivo CSV de entrada gerado pelo miningframework

```
project;merge commit;className;method;left modifications;has_build;left deletions;right
    modifications;right deletions;realistic case path
```

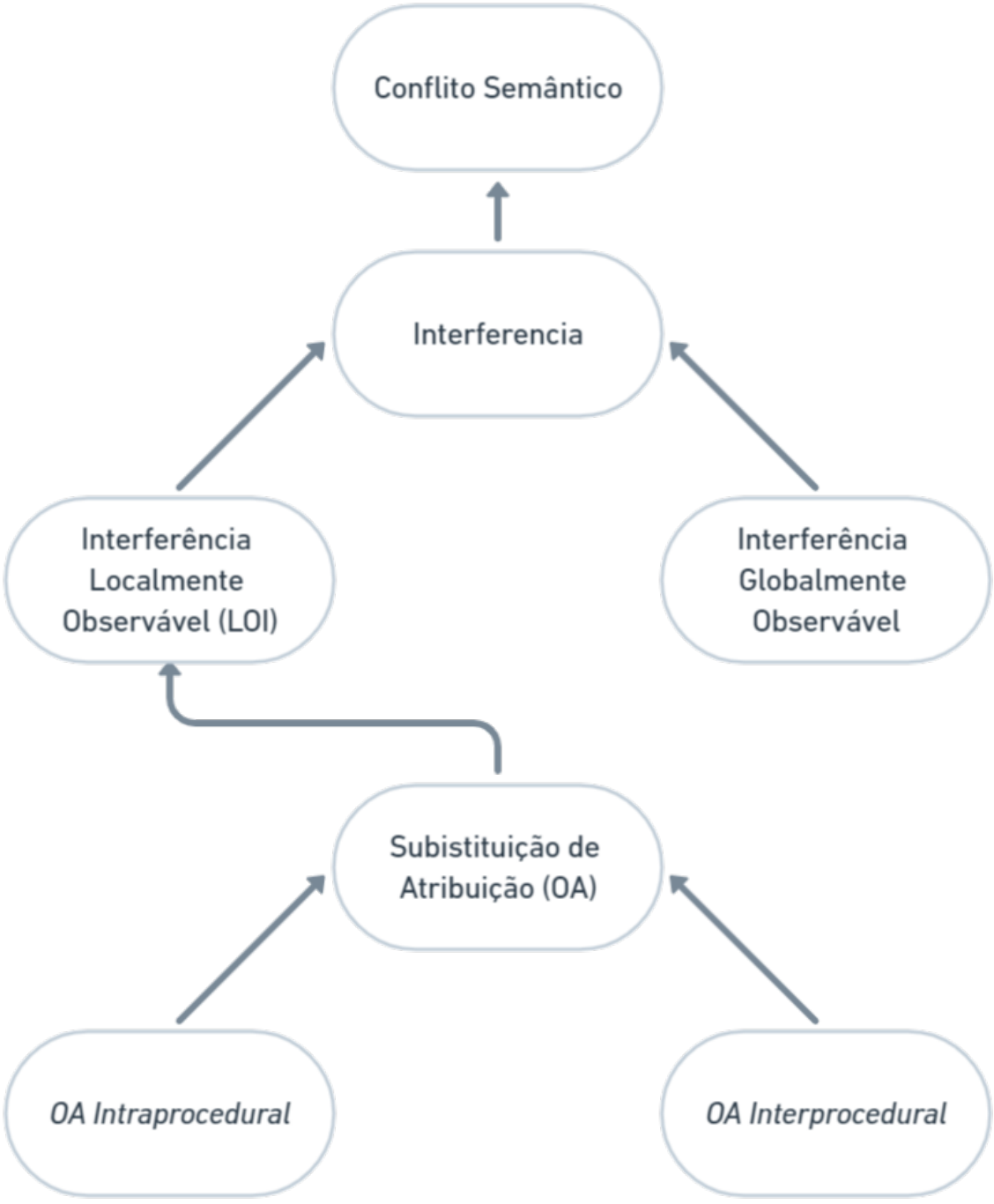
Fonte: Elaborada pelo autor (2022)

Com os dados necessários para os 78 cenários, o miningframework também foi utilizado para executar as abordagens *intraprocedural* e *interprocedural* em cada um dos cenários. Para fins de avaliação, o resultado de uma análise é considerado positivo para um cenário específico se a execução retornar uma ou mais interferências.

Os cenários do conjunto de dados também foram analisados manualmente, utilizando o protocolo disponível no Apêndice A, de modo a verificar a existência de OA e interferência localmente observável entre as modificações de *Left* e *Right*, e assim estabelecer um *Ground Truth* para avaliar a análise em questão. Para definição do *Ground Truth* de OA e interferência localmente observável (LOI), foi usado um processo estruturado de checagem dupla, onde cada cenário foi atribuído a dois colaboradores que realizaram a análise de forma individual registrando justificativas para suas decisões. A análise individual dos dois colaboradores foi comparada e caso o resultado convergisse, o mesmo seria adotado com *Ground Truth* do cenário. Em caso de divergência, os colaboradores se reuniam e discutiam o cenário, buscando chegar a um consenso. Nos casos em que o consenso não foi atingido, o cenário era apresentado a outro colaborador que tomava a decisão.

A definição de interferência localmente observável é inspirada na definição dada por Horwitz et al. (HORWITZ; PRINS; REPS, 1989a), que definiu um conflito de integração semântico como um cenário em que as contribuições das versões interferem entre si. Para ajudar a compreender melhor as definições que utilizamos nesse trabalho, podemos observar a imagem ilustrada na Figura 22. Como dito anteriormente, implementamos duas abordagens para detecção de OA em cenários de integração de código. OA *intraprocedural* e OA *interprocedural* são implementações que tentam ao máximo, dentro de suas características, uma aproximação para a definição da análise de substituição de atribuição OA. No que lhe concerne, OA é um tipo de interferência que busca identificar atribuições de dois desenvolvedores na mesma variável. Nesse trabalho utilizamos as interferências localmente observáveis, pois é o tipo de interferência que

Figura 22 – Representação da hierarquia da até o conflito semântico

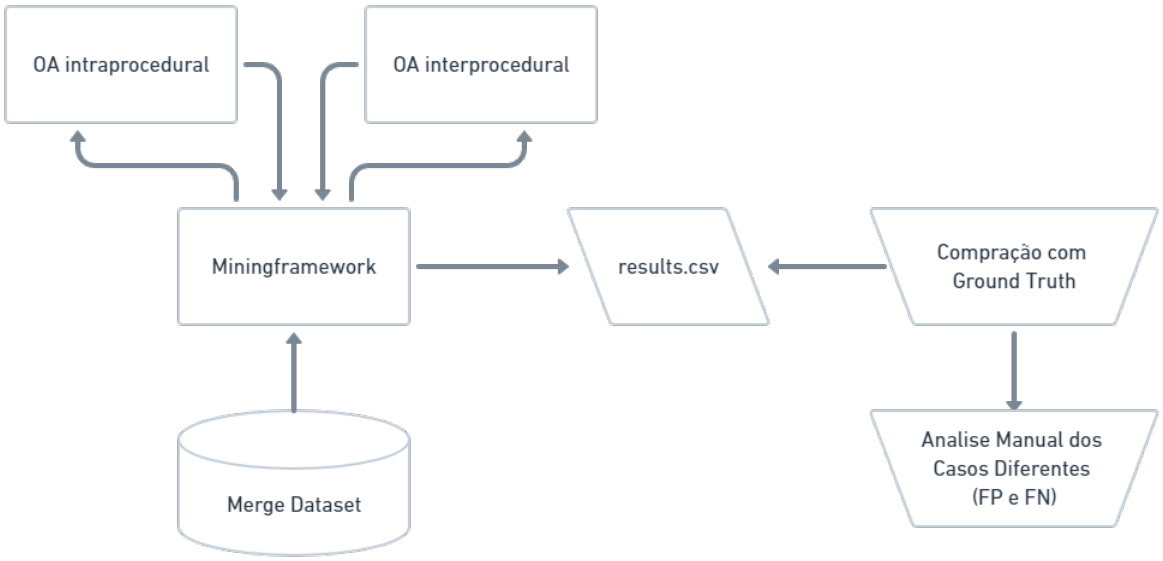


Fonte: Elaborada pelo autor (2022)

facilita a avaliação manual. Por fim, temos o conceito de interferência como aproximação para conflito, pois um conflito nada mais é do que interferência não intencional entre mudanças integradas dos desenvolvedores.

Uma visão completa do fluxo da execução pode ser observado na imagem da Figura 23.

Figura 23 – Fluxo da execução da avaliação das análises.



Fonte: Elaborada pelo autor (2022)

Avaliar a capacidade da análise de detectar OA é importante para validar a sua implementação, para isso, comparamos o resultado da ferramenta com *Ground Truth* de OA. Já avaliar a capacidade de detectar casos de interferência localmente observável é importante para validar capacidade de a análise compor uma ferramenta para detecção de conflitos de integração semânticos mais robusta, para isso, comparamos o resultado da ferramenta com *Ground Truth* de LOI.

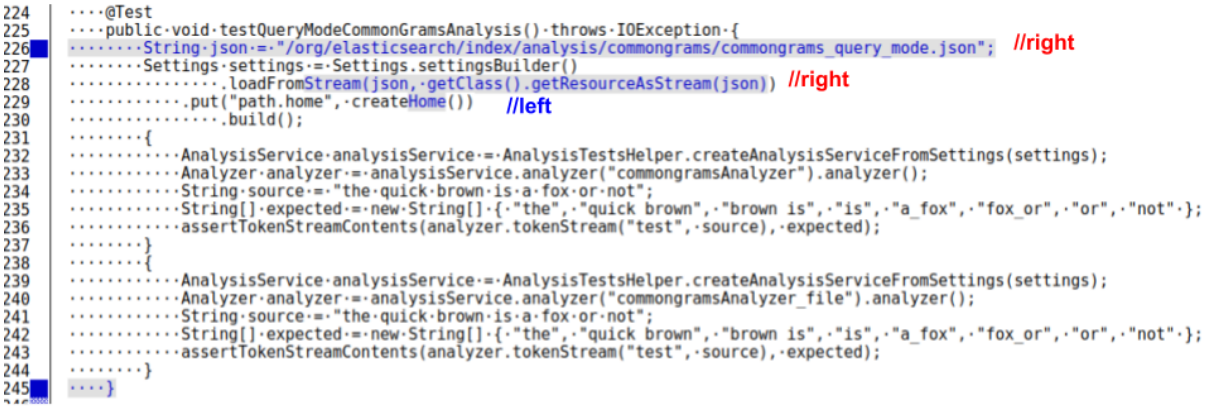
É importante ressaltar que esses dois problemas são independentes, ou seja, existem casos em que existe OA, mas não existe LOI e casos em que existe LOI, mas não existe OA.

Um caso em que existe OA, mas não existe LOI pode ser observado na imagem da Figura 24. Não há interferência localmente observável, pois *Right* fez uma refatoração (*extract variable*). Existe OA, pois os dois desenvolvedores atribuem a variável `setings`.

Já para os casos em que não existe OA, mas existe LOI, temos o exemplo ilustrado na Figura 25. Existe interferência localmente observável, pois *Right* está usando uma variável (`oplogDb`) alterada por *Left*. Essa categoria de interferência chamamos *Direct Flow*. Não existe OA, pois os desenvolvedores não sobrescrevem a mesma variável em nenhum momento.

Por fim, foi realizada uma análise manual dos casos diferentes (Falsos Positivos e Falsos Negativos) onde foi possível identificar os motivos pelos quais a análise errou. Isso, assim como a otimização da implementação e a avaliação da métrica de tempo, não são o objetivo principal deste trabalho que é detectar evidências da utilidade da análise de OA. Desta forma,

Figura 24 – Arquivo de *merge* do cenário f3d6309 como exemplo de OA sem LOI



Fonte: Elaborada pelo autor (2022)

Figura 25 – Arquivo de *merge* do cenário 3d4f995 como exemplo de LOI sem OA

Fonte: Elaborada pelo autor (2022)

não aprofundamos esses itens nesse trabalho, mas pretendemos realizar em trabalhos futuros, assim como a análise dos casos positivos, onde verificaremos se os alertas gerados pela análise correspondem às interferências detectadas na análise manual.

Neste trabalho, a análise manual dos casos diferentes (Falsos Positivos e Falsos Negativos) foi feita apenas para a abordagem *interprocedural*.

4.2.1 Métricas

Para entender melhor cada métrica, primeiro é necessário entender alguns conceitos. Uma matriz de confusão é uma tabela que indica os erros e acertos de um modelo ou ferramenta, comparando com o resultado esperado (*ground truth*). A imagem abaixo demonstra um exemplo de uma matriz de confusão.

Figura 26 – Matriz de Confusão

		Detectada	
		Sim	**Não**
Real	**Sim**	Verdadeiro Positivo (VP)	Falso Negativo (FN)
	Não	Falso Positivo (FP)	Verdadeiro Negativo (VN)

Fonte: Elaborada pelo autor (2022)

- Verdadeiros Positivos (VP): classificação correta da classe Positivo;
- Falsos Positivos (FP): erro em que a ferramenta resultou em Positivo quando o valor do *ground truth* para o cenário era Negativo (Erro Tipo I);
- Falsos Negativos (FN): erro em que a ferramenta resultou em Negativo quando o valor do *ground truth* para o cenário era Positivo (Erro Tipo II);
- Verdadeiros Negativos (VN): classificação correta da classe Negativo.

Sendo assim, a partir da contagem de todos esses termos e da obtenção da matriz de confusão, é possível calcular métricas de avaliação para a classificação. Em nosso trabalho, utilizamos as métricas de precisão, revocação e acurácia e tempo.

4.2.1.1 Precisão

A métrica de precisão indica quantas classificações positivas feitas pela ferramenta estão corretas, ou seja, quantos dos casos que as análises indicaram haver OA e quantos dos casos que as análises indicaram haver LOI realmente tinham. Essa métrica é importante, pois uma

análise de baixa precisão pode indicar muitos "alarmes falsos", o que pode fazer com o que os desenvolvedores tenham que verificar muitos casos manualmente sem necessidade. Para calcular a precisão usamos a seguinte equação:

$$\frac{VP}{VP + FP}$$

4.2.1.2 Revocação

A métrica de revocação indica, dentre todas as situações com o valor de Positivo esperado, quantas estão corretas. Nesse contexto, seria a razão dos casos em que existe OA e dos casos em que existe LOI e a ferramenta detecta. Essa métrica é importante, pois uma análise que tem essa proporção muito baixa, não detectaria cenários com OA e LOI que poderiam indicar conflitos semânticos (FILHO, 2017), introduzindo *bugs* ao sistema. Para calcular a revocação usamos a seguinte equação:

$$\frac{VP}{VP + FN}$$

4.2.1.3 Acurácia

A métrica de acurácia indica uma performance geral da ferramenta. Dentre todas as classificações, quantas a ferramenta classificou corretamente. Essa métrica, no entanto, não é muito representativa para esse trabalho, pois os cenários negativos são predominantes na amostra, o que distorce essa métrica. Ela foi incluída para demonstrar essa característica do problema. Para calcular a acurácia usamos a seguinte equação:

$$\frac{VP + VN}{VP + VN + FP + FN}$$

4.2.1.4 Tempo

A métrica de tempo é um indicativo importante relacionado ao custo computacional da ferramenta. Foi considerado o tempo de execução dos cenários e calculado uma média de tempo por execução. Como a análise da métrica de tempo não é um dos focos principais deste trabalho, o cálculo levou em consideração apenas uma execução e não contém o rigor

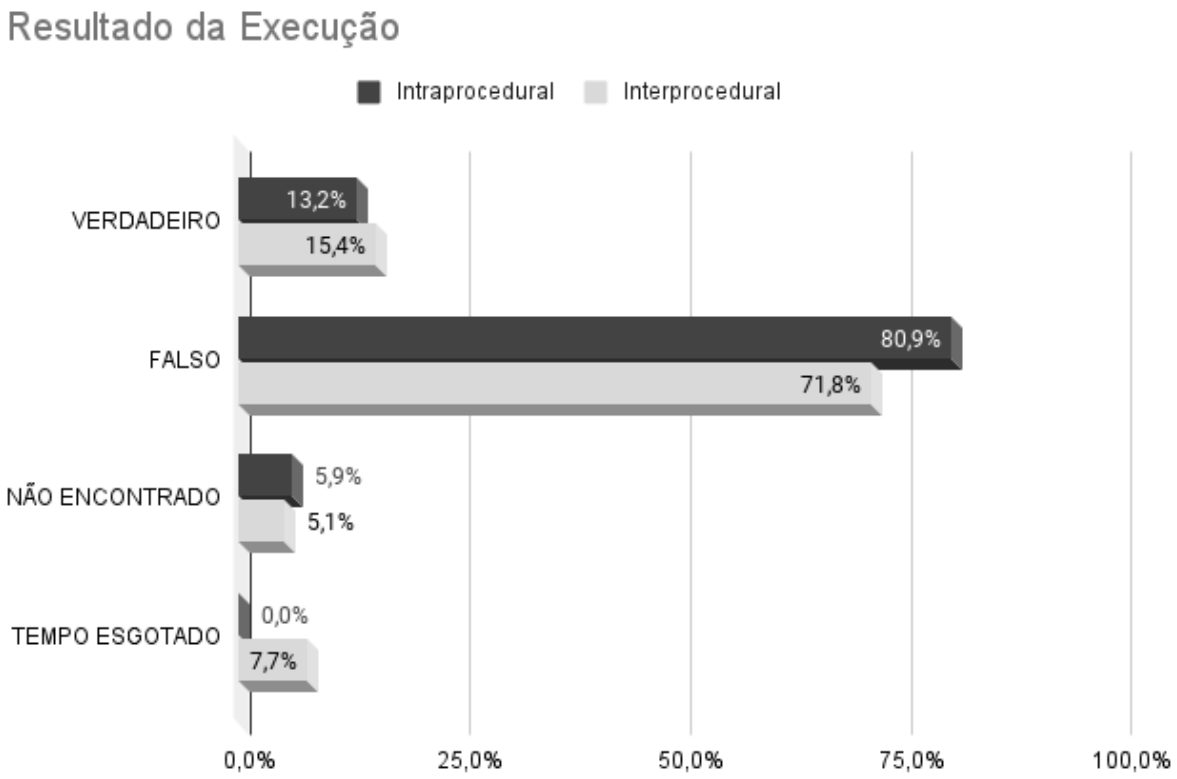
metodológico necessário. No entrando queremos mostrar com essa métrica que a abordagem *interprocedural* por sua natureza é mais pesada do que uma abordagem *intraprocedural*, algo que já é conhecido na literatura e esperado neste trabalho.

4.3 RESULTADOS

4.3.1 Resultados Reportados nas Execuções da Análise

Na imagem da Figura 27 podemos visualizar o resultado da execução da análise para o conjunto de dados de avaliação.

Figura 27 – Resultados da execução da análise para detecção de OA no conjunto de dados.



Fonte: Elaborada pelo autor (2022)

Observando primeiro os resultados da abordagem *intraprocedural* que contou com um total de 68 cenários de 28 projetos, existem 4 casos (5,9%) que não foram encontrados pela análise, isso acontece quando a análise não consegue encontrar o arquivo JAR que contenha a classe e o método alterados no cenário ou quando existem apenas modificações de um desenvolvedor. Os 4 casos se enquadram na segunda alternativa e apresentam apenas remoções de linhas

feitas pelo desenvolvedor *Right*. Para fins de comparação com o *ground truth* esses casos foram desconsiderados.

Podemos observar também que em outros 13,2%, equivalente a 9 casos, a análise reportou o *status* de verdadeiro, ou seja, a análise sugere haver alguma substituição de atribuição entre as contribuições dos desenvolvedores para esses cenários. Aqui não estamos preocupados em contabilizar a quantidade de interferências reportadas no cenário e sim se existe ou não OA entre as contribuições dos desenvolvedores para o caso avaliado. Por fim, temos os demais 55 casos ou 80,9% em que a análise reportou falso, ou seja, a análise não conseguiu detectar nenhuma substituição de atribuição entre as contribuições dos desenvolvedores para esse conjunto de cenários.

Ainda na imagem da Figura 27 que ilustra os resultados da execução da análise de substituição de atribuição para o conjunto de dados de avaliação. A abordagem *interprocedural* contou com um total de 78 cenários de 29 projetos e também reportou os mesmos 4 casos (5,1%) como não encontrado. Além disso, apresentou 6 casos (7,7%) em que a análise foi interrompida por exceder o limite de tempo estipulado para cada cenário. O *timeout* pode acontecer, pois, devido à característica da abordagem *interprocedural* que acessa e analisa o corpo de todos os métodos chamados nós próprios métodos analisados, o que exige um tempo maior de execução por isso é definido um tempo limite que quando é atingido, para a execução da análise e retorna um status de *timeout*. Esses casos em que a execução foi interrompida por *timeout* e os casos em que o JAR não foi encontrado, foram desconsiderados para fins de comparação com o *ground truth* .

A abordagem *interprocedural* também reportou 12 casos (15,4%) como positivos, ou seja, a análise sugere haver alguma substituição de atribuição entre as contribuições dos desenvolvedores para esses cenários. Aqui não estamos preocupados em contabilizar a quantidade de interferências reportadas no cenário e sim se existe ou não OA entre as contribuições dos desenvolvedores para o caso avaliado. Os demais casos, cerca de 71,8% (56 casos), receberam o *status* de falso, pois a análise não conseguiu detectar nenhuma substituição de atribuição entre as contribuições dos desenvolvedores para esse conjunto de cenários.

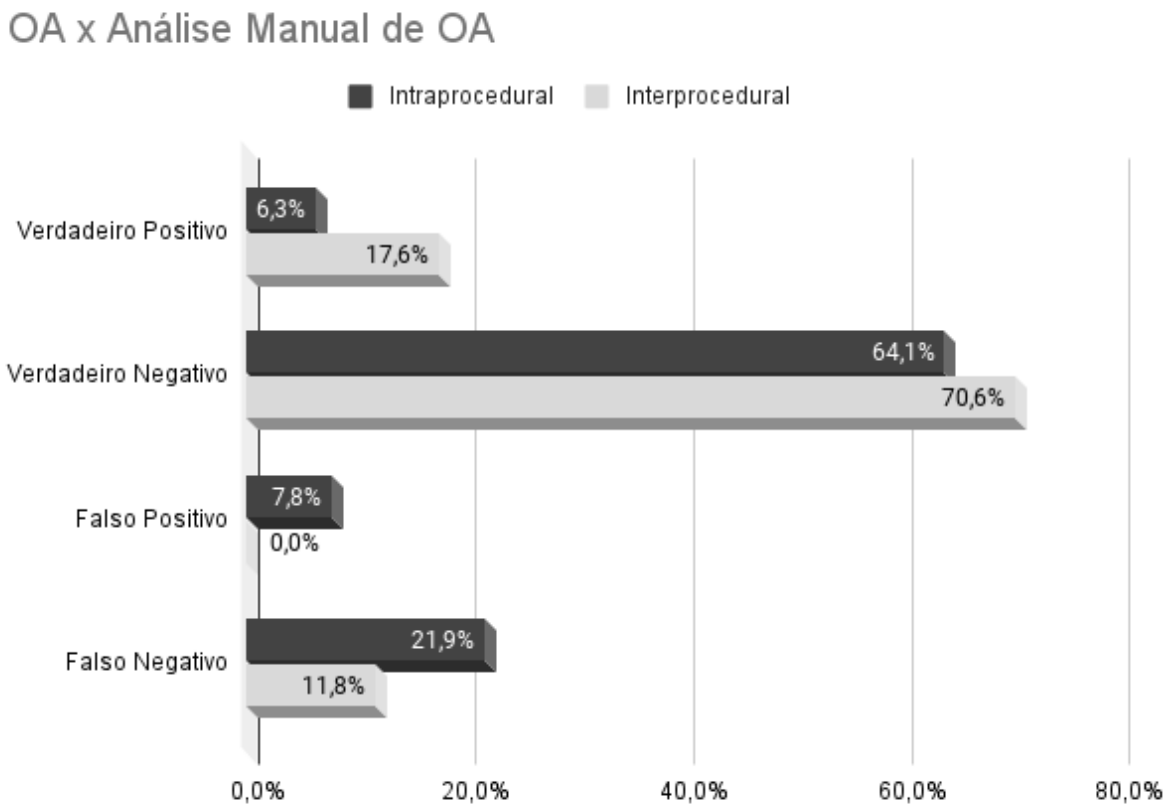
A seguir, demonstramos a validade dos resultados da análise proposta em relação a dois aspectos: o da detecção de substituição de atribuição e o da detecção de interferências localmente observáveis entre as modificações introduzidas. Os casos considerados como substituição de atribuição ocorrem quando as contribuições das versões *Left* e *Right* atribuem valor à mesma variável sem outras atribuições intermediárias de *Base*. Os casos considerados com interferên-

cia são os que foram detectados alguma categoria de interferência, não necessariamente OA, entre as mudanças. Interferência localmente observável pode ser provada por um caso de teste que passa em uma das versões integradas, e falha na versão integrada. É importante ressaltar que esses dois problemas são independentes, ou seja, pode existir OA e não interferência e vice-versa, como explicado no final na seção anterior.

4.3.2 Análise Manual de Overriding Assigment

Os resultados apresentados nessa seção foram obtidos comparando os resultados da análise proposta com o *Ground Truth* para OA. Essa comparação visa avaliar a capacidade da análise proposta para detecção da existência de substituição de atribuições (OA) entre as contribuições de dois desenvolvedores em um cenário de integração de código.

Figura 28 – Resultados da análise em comparação com o *Ground Truth* de OA.



Fonte: Elaborada pelo autor (2022)

Os resultados da abordagem *intrarprocedural* contaram com 64 cenários considerados para a comparação com a *ground truth* de OA, e estão ilustrados na Figura 28. Dos 18 casos em

que o *ground truth* determinou a existência de OA como podemos observar no Apêndice B ou na tabela disponível no Apêndice Online [6], a ferramenta acertou apenas 4, com isso a taxa de verdadeiros positivos ficou em 6,3%, enquanto a de falsos negativos ficou em 21,9%. Por outro lado, o *ground truth* contabiliza 46 casos como falsos, ou seja, não deve existir OA. Desses a ferramenta reporta corretamente 41, atingindo 64,1% de verdadeiros negativos. No entanto, isso indica que a análise reporta mais falsos positivos (7,8%) do que verdadeiros positivos (6,3%). Isso pode ser ruim, pois pode levar os desenvolvedores a perder tempo tentando identificar problemas que não existem de fato.

Ainda na imagem da Figura 28, podemos observar o resultado da comparação da análise *interprocedura* com o *ground truth* de OA. Nesse caso, estamos lidando com 68 cenários, em que 20 foram classificados como verdadeiros e 48 como falsos pelo *ground truth*. Dentre os classificados como verdadeiros, a análise de OA detectou 12, atingindo 17,6% de verdadeiros positivos e 70.6% de verdadeiros negativos. Dos cenários classificados como falsos a análise detectou todos, ou seja, não gerou falsos positivos. Indicando que a análise teve poucos erros, nenhum falso positivo e 8 falsos negativos (11,8%). Isso é ótimo, pois, segundo esses resultados, a análise pode ser usada sem reportar nenhum alarme falso para seus usuários, ainda que não consiga detectar todos os casos de OA.

Um sumário com a descrição e a quantidade de erros da análise *interprocedural* proposta pode ser encontrado na Tabela 4.2.

Tabela 2 – Descrições dos cenários em que a análise *interprocedural* proposta errou na detecção de OA

Descrição do Cenário (em relação à causa do erro)	Quantidade	Tipo de erro
String com mais de uma linha é alterada em linhas diferentes	1	FN
Alterações apenas na inicialização de atributos que possuem mais de uma linha	3	FN
Callgraph não consegue encontrar possíveis classes de destino, pois faltam dependências	3	FN
Conflito de *merge* ocorreu e o integrador preservou a mudança de apenas um dos desenvolvedores	1	FN

Fonte: Elaborada pelo autor (2022)

O primeiro caso da tabela pode ser observado na imagem ilustrada na Figura 29. Para esse caso temos que *Left* (azul) e *Right* (roxo) alteraram a *String* que vai ser usado no retorno

[6] https://url.gratis/5WmiEJ

da função. O *ground truth* classifica isso como OA, no entanto, ao nível de execução da análise implementada, o *Soot framework* vai gerar variáveis temporárias para cada uma das linhas, dessa forma a análise não consegue associar as mudanças feitas pelos desenvolvedores a mesma variável. Isso é uma limitação da implementação que pretendemos resolver em trabalhos futuros.

Figura 29 – Cenário Storm ad2be67 no método toString() como exemplo para o falso negativo.

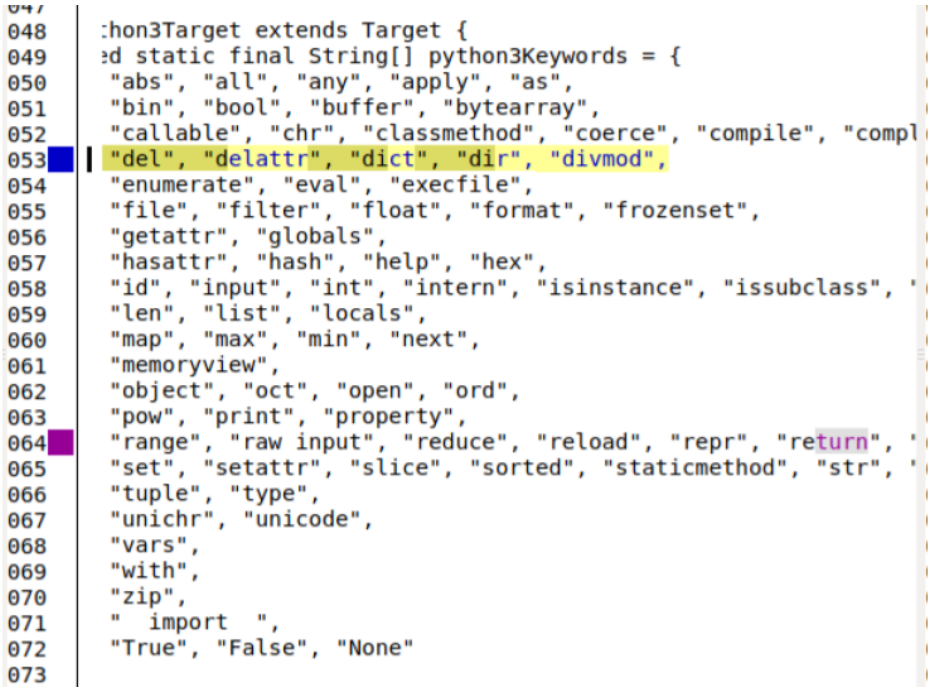
Fonte: Elaborada pelo autor (2022)

Já para o caso em que as alterações são apenas na inicialização de atributos que possuem mais de uma linha, temos o exemplo ilustrado na imagem da Figura 30. Nesse caso temos o mesmo problema do exemplo anterior, no entanto, ao resolver esse problema ainda não seria possível detectar OA nesse cenário. Isso porque as alterações ocorrem na inicialização de atributos e não passam por nenhum método. Sendo assim, não temos um método de entrada que atualmente é obrigatório para iniciar a execução. Uma solução para isso seria uma ferramenta que adicionasse automaticamente um método de entrada para casos como esse ou remover a dependência de um método de entrada para a execução da análise, passando a especificar o ponto de início da execução de outra forma.

Os casos em que o *Call graph*[7] não consegue encontrar possíveis classes de destino, ocorre pela falta de algumas dependências do arquivo JAR utilizado pela análise. Isso pode acon-

[7] Um *Call graph* permite ao usuário visualizar a relação entre as sub-rotinas Pais e Filhos em seu programa. Em essência, cada nó no *Call graph* representa uma função (f) e cada aresta representa a função (g) sendo chamada por (f). Um nó pode ser considerado como a "função pai"e cada nó como a "função filho".

Figura 30 – Cenário Antlr4 69ff266 como exemplo para o falso negativo.

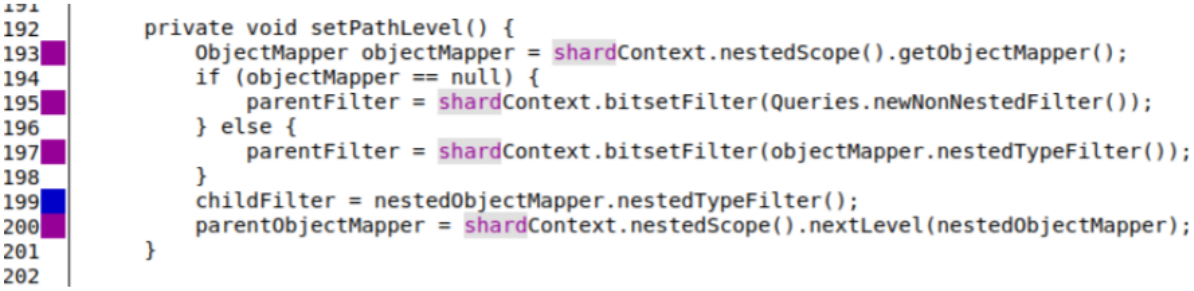


Fonte: Elaborada pelo autor (2022)

tecer por algum problema na geração do arquivo JAR ou algum problema nas versões das dependências. Pretendemos tratar casos como esses em trabalhos futuros.

Para os casos em que ocorreram conflitos de *merge* e o integrador preservou a mudança de apenas um dos desenvolvedores, podemos observar o exemplo ilustrado na Figura 31

Figura 31 – Cenário Elasticsearch 59cb67c como exemplo para o falso negativo.



Fonte: Elaborada pelo autor (2022)

Right (roxo) muda o tipo de `childFilter` para `Filter` e atribui um novo valor usando o comando `nestedObjectMapper.nestedTypeFilter()`. Já o desenvolvedor *Left* (azul), não muda o tipo de `childFilter`, mas muda o valor atribuído usando o novo atributo adicionado `shardContext`. Dessa forma, um conflito de *merge* ocorre e após a interferência do integrador, o resultado foi exatamente como *Right* implementou: `childFilter =`

`nestedObjectMapper.nestedTypeFilter()`. Nesse nível, não acontece OA. Mas aí pela orientação de ver se juntando as mudanças dos *branchs*, sem considerar o que o *integrador* fez, poderia ter duas atribuições ao mesmo atributo `childFilter`. Por isso, OA. A análise não consegue detectar esses casos, pois analisa apenas o *merge*.

A partir dos resultados das análises foram calculadas as métricas representadas pela Tabela 4.3.

Para a métrica de precisão, a análise *interprocedural* obteve o valor 1,0. Isso indica que a abordagem indica casos como positivos quando eles realmente existem. Já a análise *intraprocedural* proposta tem uma precisão de 0,44, o que indica que alguns dos casos indicados como positivos foram classificados corretamente, mas também que grande parte dos casos indicados como positivos foram classificados incorretamente.

Para a métrica de revocação, a análise *intraprocedural* obteve um resultado ruim (0,22%), o que indica que bem mais da metade dos casos positivos não são detectados por elas. Já a análise *interprocedural* proposta tem um valor de 0,6 para a métrica de revocação, o que indica que a análise detecta a maioria dos casos positivos, fazendo com que ela seja mais confiável para detectar OA.

A análise *interprocedural*, por sua característica e natureza, tem uma acurácia melhor por ser uma análise menos sensível e detectar mais casos positivos que a análise *intraprocedural*, isso fica comprovado no resultado onde a análise *interprocedural* teve uma acurácia de 0,88, um pouco maior do que a *intraprocedural* que recebeu 0,7.

Tabela 4 – Métricas para a detecção de OA para as análises testadas

	Intraprocedural	Interprocedural
Precisão	0,44	1.0
Revocação	0,22	0,6
Acurácia	0,7	0,88

Fonte: Elaborada pelo autor (2022)

Para todas as métricas, a análise *interprocedural* se saiu melhor, principalmente na comparação com a métrica de precisão. Isso é esperado, pois, a *interprocedural* percorre e analisa uma quantidade maior de código-fonte ao acessar as chamadas de método, o que faz com que ela consiga sinalizar uma quantidade maior possíveis casos de interferência. No entanto, essa abordagem tem como desvantagem a performance. O tempo médio da execução por cenário da abordagem *interprocedural* é 25,8 segundos mais lenta, utilizando a máquina descrita na

seção 4.2.4.

A análise proposta se mostrou capaz de detectar cenários com substituições de atribuições entre as contribuições, no entanto, reportou falsos positivos e falsos negativos. O número de falsos positivos foi baixo e isso indica que a análise reporta poucos "alarmes falsos", o que gera um custo pequeno para o desenvolvedor. Os falsos negativos estão associados a limitações da análise, e acontecem, por exemplo, quando as modificações introduzidas pelos desenvolvedores estão em inicializações de atributos com múltiplas linhas e não existe um método de entrada. Pretendemos aplicar correções para problemas como esse em trabalhos futuros. Com isso concluímos que a análise pode ser usada visando a detecção de OA, mas não se deve confiar apenas nela em situações que a detecção de erros no sistema são mais críticos.

4.3.3 Análise Manual de Interferência Localmente Observável

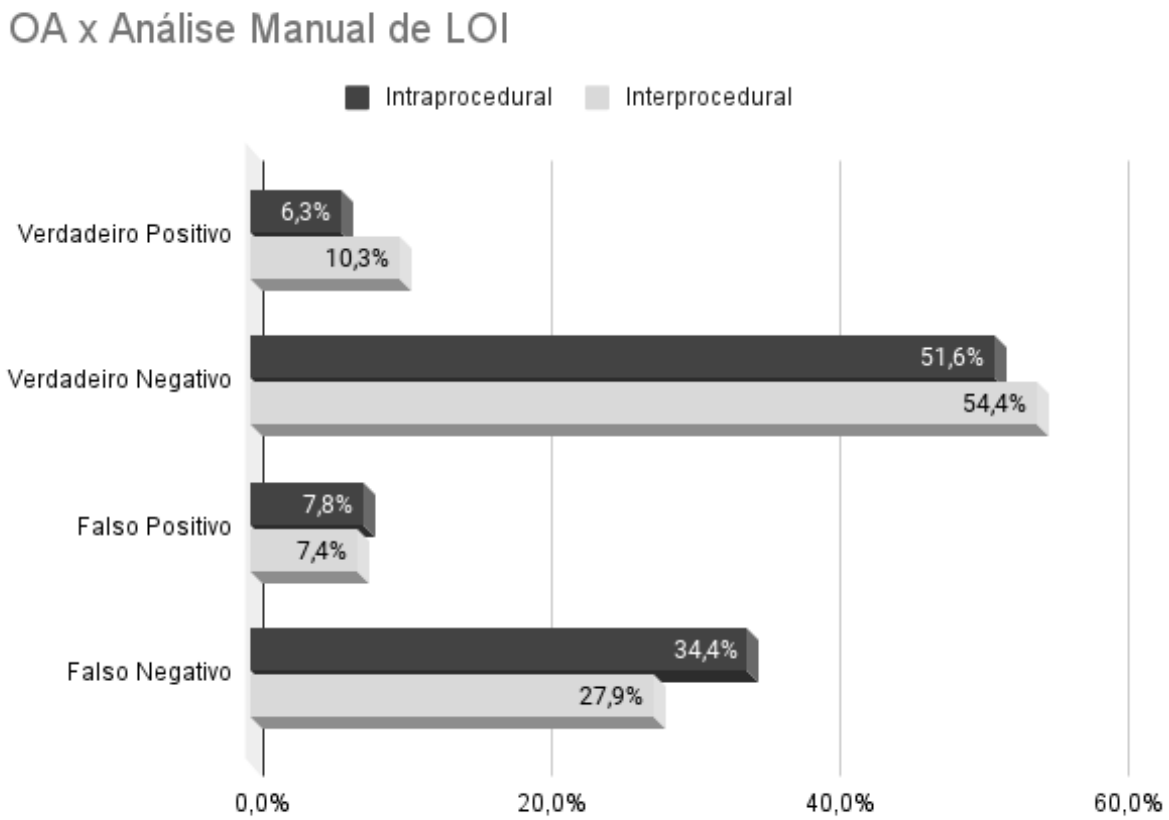
Os resultados apresentados nessa seção foram obtidos comparando os resultados da análise proposta com o *Ground Truth* para interferência localmente observável (LOI). Essa comparação pretende avaliar o algoritmo para detecção de interferências entre as contribuições das versões integradas, portanto, para detecção de conflitos semânticos. Para entender um pouco melhor a definição de LOI, veja a explicação abaixo da Figura 22

Os resultados ilustrados na Figura 32 mostram os resultados da comparação dos 64 cenários considerados na execução *intraprocedural* com o *ground truth* de LOI. Podemos observar que a quantidade de verdadeiros positivos e falsos positivos foi a mesma com relação ao *ground truth* de OA, enquanto a quantidade de verdadeiros negativos foi menor e de falsos negativos, maior. Isso é esperado, pois, nem todos os cenários com *ground truth* de interferência positivos contêm interferências de substituição de atribuição detectados pela análise.

Analisando apenas a comparação com o *ground truth* de LOI, temos que dos 26 casos com interferência, houve um acerto da ferramenta de 15,3%, ou seja, 4 casos. Isso gera uma taxa alta de 34,4% de falsos negativos. Dos casos com não contendo interferência a ferramenta acertou 33 (86,8%).

Ainda nos resultados ilustrados na Figura 32, conseguimos observar os resultados da comparação dos 68 cenários considerados na execução *interprocedural* com o *ground truth* de LOI. Ao comparar esses resultados com relação ao *ground truth* de OA, observamos uma redução na quantidade de verdadeiros positivos e verdadeiros negativos, o que indica que a análise acertou menos ao gerar alerta sobre interferências. Esse resultado é esperado, pois, a análise

Figura 32 – Resultados da análise em comparação com o *Ground Truth* de LOI.



Fonte: Elaborada pelo autor (2022)

tem foco na detecção apenas de OA e não gera alerta sobre outras categorias de interferências.

Tivemos também um número maior de falsos negativos, no entanto, isso é esperado, pois, nem todos os cenários com *ground truth* de interferência positivos contêm interferências de substituição de atribuição detectados pela análise. Para essa execução também tivemos um número maior de falsos positivos, o que também é esperado visto que o fato de existir uma substituição de atribuição, não necessariamente precisa gerar uma interferência localmente observável.

Analisando apenas a comparação com o *ground truth* de LOI, temos que dos 26 casos com interferência, houve um acerto de 26,9%, ou seja, 7 casos. Isso gera uma taxa de 27,9% de falsos negativos. Dos casos com não contendo interferência a ferramenta acertou 37 (88,1%). Esses números são levemente superiores em relação à abordagem *intraprocedural*. Podemos ter uma melhor noção olhando para os resultados das métricas apresentados na Tabela 4.5.

Um sumário com a descrição e a quantidade de erros da análise *interprocedural* proposta pode ser encontrado na Tabela 4.4. Os dois últimos itens da tabela já são esperados, pois, a

análise foi projetada para detectar OA e como já dito anteriormente a existência de OA não implica necessariamente na existência de interferência localmente observável e vice-versa.

Tabela 5 – Descrições dos cenários em que a análise *interprocedural* proposta errou na detecção de LOI.

Descrição do Cenário (em relação à causa do erro)	Quantidade	Tipo de erro
String com mais de uma linha é alterada em linhas diferentes	1	FN
Alterações apenas na inicialização de atributos que possuem mais de uma linha	2	FN
Callgraph não consegue encontrar possíveis classes de destino, pois faltam dependências	2	FN
Ground truth de OA é falso, mas *Ground truth* de LOI é verdadeiro	14	FN
Ground truth de OA é verdadeiro, mas *Ground truth* de LOI é falso	5	FP

Fonte: Elaborada pelo autor (2022)

A explicação para os quatro primeiros itens descritos na tabela pode ser observada na seção anterior, logo abaixo da Tabela 4.2. O princípio é o mesmo, pois para esses casos existia OA e LOI segundo o *ground truth*. Já os casos em que o *ground truth* diz que não existe OA, mas existe LOI ou o *ground truth* diz que existe OA, mas não existe LOI, são esperados, pois, como explicamos ao final da seção de metodologia, OA não acarreta necessariamente em LOI e vice-versa.

A partir dos resultados das análises foram calculadas as métricas representadas pela Tabela 4.5.

Para a métrica de precisão, a análise *interprocedural* obteve o valor de 0,6, enquanto a análise *intraprocedural* proposta tem uma precisão de 0,4. Os valores são bem próximos e indicam que alguns dos casos indicados como positivos foram classificados corretamente, mas também que grande parte dos casos indicados como positivos foram classificados incorretamente.

Para a métrica de revocação, a análise *interprocedural* obteve o valor de 0,3, enquanto a análise *intraprocedural* obteve 0,2. Novamente os valores são muito próximos e indicam que a maioria dos casos positivos não são detectados pelas abordagens.

A análise *interprocedural*, por sua característica e natureza, tem uma acurácia melhor por ser uma análise menos sensível e detectar mais casos positivos que a análise *intraprocedural*, isso fica comprovado no resultado onde a análise *interprocedural* teve uma acurácia de 0,64, enquanto a *intraprocedural* recebeu 0,57.

Tabela 7 – Métricas para a detecção de LOI para as análises testadas

	Intraprocedural	Interprocedural
Precisão	0,4	0,6
Revocação	0,2	0,3
Acurácia	0,57	0,64

Fonte: Elaborada pelo autor (2022)

Comparando os resultados das duas execuções, podemos observar resultados semelhantes com uma leve vantagem para a abordagem *interprocedural*. Isso é esperado, pois a *interprocedural* percorre e analisa uma quantidade maior de código-fonte, o que faz com que ela consiga sinalizar mais possíveis casos de interferência. No entanto, pelo mesmo motivo, o custo de execução *interprocedural* é maior, como podemos ver na seção 4.2.4. Isso faz com que a abordagem *intraprocedural* seja ainda mais relevante.

Em geral, podemos concluir que a análise proposta consegue detectar interferência entre as modificações em um cenário de integração gastando poucos recursos computacionais, o que pode indicar que ela é uma boa ferramenta para detecção de conflitos de integração semânticos. No entanto, existem outros tipos de cenário de interferência que análises OA não é capaz de detectar, o que indica que somente a análise proposta não é suficiente para ser utilizada sozinha, como uma ferramenta confiável para detecção de conflitos semânticos. A análise proposta poderia ser combinada com outras análises que buscam detectar outros tipos de interferência para criar uma ferramenta mais robusta. Análises de OA podem errar na detecção de interferência e com a análise proposta não é diferente. Nesse sentido, uma ferramenta utilizando ela pode ser muito sensível e apresentar falsos positivos e falsos negativos. No entanto, a análise poderia ser refinada para reduzir essa quantidade de erros.

4.3.4 Detalhes da execução

Às duas abordagens da análise foram executadas utilizando uma máquina virtual com sistema operacional Linux Ubuntu 20.04.2 LTS 64-bit, 4 *Gigabyte* de memória RAM, processador Intel® Core™ i5-9300H CPU @ 2.40GHz $\times$ 4

A execução *intraprocedural* foi configurada com um *timeout* de 240 segundos. Na Tabela 4.6, temos os dados da estatística básica sobre o tempo da execução em segundos.

A execução *interprocedural* foi configurada com um *timeout* de 240 segundos e um limite

de profundidade no acesso aos métodos de 5. Na Tabela 4.6, temos os dados da estatística básica sobre o tempo da execução em segundos.

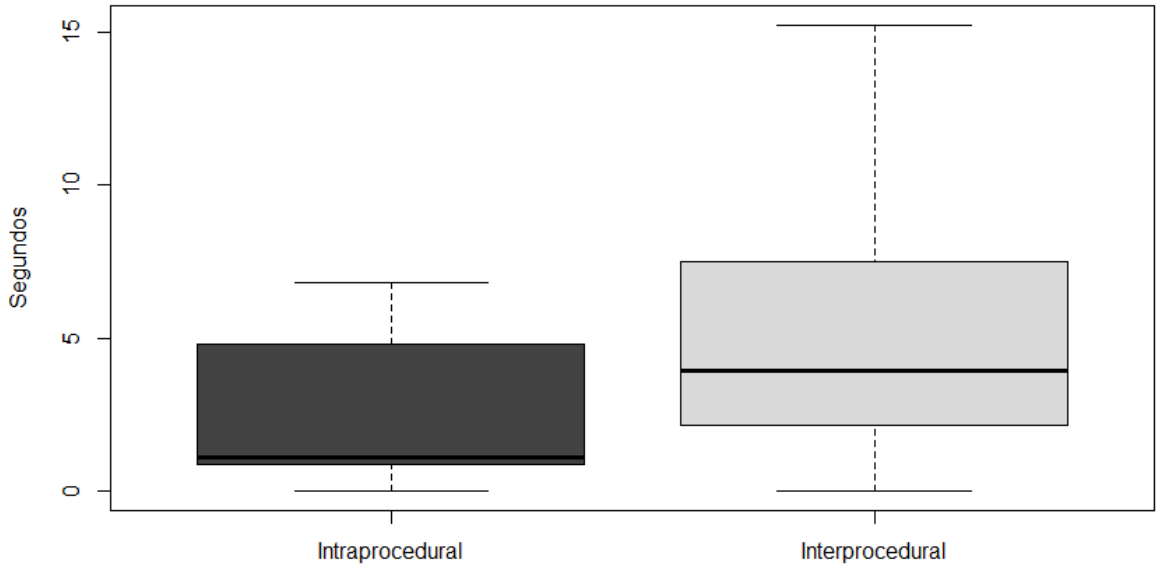
Tabela 8 – Dados estatísticos da execução.

	Média	Mediana	Desvio Padrão	Variância
Intraprocedural	2,32	1,09	2,42	5865,25
Interprocedural	28,12	3,94	68,81	4735053

Fonte: Elaborada pelo autor (2022)

Para facilitar a visualização, na Figura 33, ilustramos o *bloxplot* com os dados temporais da execução *intraprocedural* e *interprocedural* em segundos. Para construir o *bloxplot* nós removemos os *ouliers* para melhorar a visualização.

Figura 33 – Boxplot da execução sem outliers



Fonte: Elaborada pelo autor (2022)

Analisando os resultados podemos perceber que a abordagem *interprocedural* é consideravelmente mais lenta em relação à média de tempo por cenário do que a abordagem *intraprocedural*. Isso é esperado, pois, como a *intraprocedural* tem como característica, ignorar as chamadas de método no método de entrada, ela percorre um caminho mais curto durante a execução. O contrário ocorre na execução *interprocedural* que permite acessar e analisar todo

o corpo de todas as chamadas de método até que o seu limite de profundidade, definido por parâmetro, seja atingido.

4.4 AMEAÇAS A VALIDADE

Neste capítulo, discutimos as limitações, bem como as ameaças à validade do nosso estudo.

4.4.1 Definição de *ground truth*

Nosso estudo se restringe à ocorrência de interferência local. Portanto, é possível que nosso exemplo tenha cenários de mesclagem que pai alterações de commit não interferem umas com as outras localmente, mas interferem globalmente. O contrário também pode ocorrer. Então o número de falsos negativos e falsos positivos em relação a uma noção de interferência pode ser diferente do nosso relatório de resultados.

Além disso, ao definir o *ground truth*, conhecer os resultados da análise proposta antes da análise manual pode ter influenciado o veredicto. Por exemplo, sabendo que a ferramenta não foi bem sucedida para um determinado cenário traz o risco de impedir uma análise manual mais aprofundada. Para reduzir essa ameaça, envolvemos dois autores na definição de *ground truth*, e exigiu-se que eles fornecessem uma explicação de por que não há interferência; isso muitas vezes requer a compreensão do mudanças em detalhes para detectar refatorações, elementos de estado alterados, e como eles impactam uns aos outros. O risco é significativamente reduzido para os casos em que as ferramentas foram bem sucedidas, ameaça pode ser minimizada analisando a interferência revelando ameaça, executando-a e verificando manualmente.

4.4.2 Ferramenta pode ter problemas

O objetivo principal deste trabalho é verificar a utilidade da análise proposta em detectar interferências entre as contribuições. Com isso, não investimos na otimização das implementações da análise, o que pode fazer com que a implementação atual aprensente erros. Alguns

problemas conhecidos já foram mencionados, mas podem existir outros problemas que ainda não foram detectados.

4.4.3 Preparação da amostra

Alguns dos cenários do *dataset* utilizado precisaram de adaptações realistas. Essas alterações foram necessárias devido ao mapeamento das linhas do código-fonte, extraídos a partir do *bytecode*, que pode ser impreciso e a análise não reconhece parâmetros de tipos não primitivos que não foram instanciados previamente no código. Essas alterações foram realizadas dentro do código fonte de sete cenários buscando manter o máximo das características originais. No entanto essas alterações foram feitas de forma manual e não foram executados testes para validar que o comportamento original foi mantido, no entanto deixamos o alerta que esses casos são tratados como cenários realistas e não reais de fato.

5 TRABALHOS RELACIONADOS

Neste capítulo descrevemos alguns dos estudos anteriores que usamos como base de evidência para nosso estudo e trabalhos relacionados.

5.1 IMPACTO DOS CONFLITOS

Antes de mais nada, é necessário entender o que os pesquisadores já investigavam sobre os conflitos e como eles afetam os desenvolvedores e a produtividade, de forma geral.

O trabalho de Perry et al. (PERRY; SIY; VOTTA, 2001) conduziu um estudo de caso observacional para analisar o impacto de mudanças paralelas em um sistema de *software* industrial de grande escala. Eles relataram que enquanto 90% dos arquivos foram integrados sem problemas houve um alto grau de alterações paralelas - envolvendo conflitos de *Merge* entre duas e dezesseis alterações.

Em contraste, Zimmermann (ZIMMERMANN, 2007) avaliou o número de conflitos de *Merge* usando uma métrica diferente, pois os autores replicaram integrações de arquivos de 4 projetos hospedados no CVS - não cenários de *Merge*.

Outros estudos não mediram quantitativamente o custo da resolução de conflitos, mas de acordo com observações experimentais relatam que a resolução de conflitos por merge não é tão trivial, pode levar um tempo considerável e é uma atividade propensa a erros. Por exemplo, Sarma et al. (SARMA; REDMILES; HOEK, 2012) relatam que os desenvolvedores geralmente se apressam para finalizar suas tarefas antes dos outros, para não precisarem lidar com conflitos ao enviar alterações para um repositório compartilhado. Além disso, Bird e Zimmermann (BIRD; ZIMMERMANN, 2012) relatam que uma causa frequente de erros de integração são conflitos de *Merge* que não foram resolvidos corretamente.

Na mesma linha, McKee et al. (MCKEE et al., 2017) realizam uma série de entrevistas e pesquisas para descobrir o que os desenvolvedores pensam sobre conflitos de *Merge*. Eles relatam que, se os desenvolvedores acharem os conflitos muito complexos, ou se não tiverem um bom entendimento da área conflitante do código, podem se sentir compelidos a mudar suas estratégias de resolução, revertendo alterações conflitantes e, em alguns casos, retardando as tarefas de resolução de conflitos.

Como alternativa, Estler et al. (ESTLER et al., 2014) investiga o impacto das informações

de conscientização em um ambiente de desenvolvimento de *software* distribuído globalmente. Entre suas descobertas, eles concluíram que a falta de informações de conscientização tem um impacto maior no desempenho do desenvolvedor do que os conflitos reais de *Merge*.

Com o objetivo de investigar quais fatores tornam os conflitos de mesclagem mais demorados para serem resolvidos na prática, Vale et al. (VALE et al., 2021), realizam um estudo em duas fases. Primeiramente, analisaram 66 projetos contendo cerca de 81 mil cenários de *Merge*, buscando investigar quais variáveis influenciam no tempo para realizar o *Merge*. Foi descoberto que dentre outras variáveis, o número de linhas de código, desenvolvedores envolvidos e a complexidade do código conflitante influenciam no tempo de resolução de conflitos de *Merge*. Em seguida, foi realizada uma pesquisa com 140 desenvolvedores de projetos temáticos com o objetivo de validar os resultados da primeira fase do estudo. Os resultados apontam que *commits* pequenos tornam a resolução de conflitos de *Merge* mais rápida e dependência inerente entre código conflitante e não conflitante é um dos principais fatores que influenciam no tempo de resolução de conflitos. Os pesquisadores ainda elaboraram uma taxonomia de quatro tipos de desafios na resolução de conflitos de *Merge*.

5.2 PREDIÇÃO/PREVENÇÃO DE CONFLITO

LeBenich et al. (LEENICH et al., 2018) conduziu um estudo empírico examinando como diferentes fatores (tamanho das alterações, número de arquivos alterados e local das alterações) podem acarretar um número maior de conflitos de *Merge*. No entanto, nenhum dos fatores analisados em seu estudo teve a capacidade de prever a frequência de conflitos de *Merge*.

Em continuação a esse trabalho, Dias et al. (DIAS; BORBA; BARRETO, 2020) investigaram sete fatores relacionados à modularidade, tamanho e tempo de contribuição dos desenvolvedores. Para isso, reproduziram e analisaram 73.504 cenários de *Merge* em repositórios GitHub de projetos Ruby e Python MVC. Como resultado, encontraram evidências de que a probabilidade de ocorrência de conflito de *Merge* aumenta significativamente quando as contribuições a serem integradas não são modulares no sentido de que envolvem arquivos da mesma fatia MVC (arquivos de modelo, visão e controlador relacionados). Também foi descoberto que contribuições maiores envolvendo mais desenvolvedores, *commits* e arquivos alterados estão mais provavelmente associadas a conflitos de *Merge*. Em relação aos fatores temporais, foi observado que que contribuições desenvolvidas em períodos mais longos de tempo estão mais provavelmente associadas a conflitos. Nenhum fator avaliado mostra poder preditivo tanto em

relação ao número de conflitos de *Merge* quanto ao número de arquivos com conflitos.

Já Accioly et al. (ACCIOLY; BORBA; CAVALCANTI, 2018) conduziram um estudo para avaliar e classificar padrões conflitantes em projetos *open source* de linguagem Java para determinar quais padrões na estrutura do código levam a conflitos. Uma de suas descobertas foi que a maioria dos conflitos obtidos após a execução de uma ferramenta de *Merge* semiestruturada ocorreu quando desenvolvedores modificaram a mesma região de um mesmo bloco de código (um método, por exemplo).

Na linha de como a estrutura de ramificação pode afetar a qualidade, Shihab et al. (SHIHAB; BIRD; ZIMMERMANN, 2012) apresenta um estudo empírico avaliando e quantificando a relação entre a qualidade do *software* e vários aspectos da estrutura de ramificação utilizada em projetos de *software*. Eles relatam que a estratégia de ramificação realmente tem um impacto na qualidade do *software* e que esse desalinhamento da ramificação e da estrutura organizacional está associado a maiores taxas de falhas pós-lançamento.

Por fim, em relação às técnicas de *merge* de *software*, Mens (MENS, 2002) descreve uma visão abrangente do campo e sugere direções para pesquisas futuras. Entre elas, ele afirma ser necessária uma classificação detalhada independente da linguagem, das mudanças que podem ocorrer no *software* e dos conflitos correspondentes.

5.3 FERRAMENTAS DE *MERGE*

Com o intuito de progredir no processo de *merge* e reduzir esforços da integração, foram criadas diversas ferramentas. Westfechtel (WESTFECHTEL, 1991) e Buffenbarger (BUFFENBARGER, 1993) foram pioneiros em propor soluções para realizar *merge* usando estruturas de arquivos. Posteriormente, outros pesquisadores implementaram soluções baseadas em construções específicas de linguagens de programação, como *Java* (APIWATTANAPONG; ORSO; HARROLD, 2007) e *C++* (GRASS., 1992).

5.3.1 Ferramentas de *Merge* Textual

Em um estudo dedicado ao diff3, Khanna et al. (KHANNA; KUNAL; PIERCE, 2007) desenvolve uma investigação formal da ferramenta, descrevendo através de formulações e provas teóricas propriedades do diff3 que auxiliam a entender seu funcionamento. Em especial, os autores mostram que ainda que as mudanças feitas nos arquivos que serão integrados pela

ferramenta sejam feitas localmente em regiões distintas do código, não há garantia de que o diff3 conseguirá integrar sem conflitos, mesmo intuitivamente parecendo o contrário.

Diversos outros trabalhos existem na academia dedicados a investigar e aprofundar os conhecimentos em diferentes categorias de abordagens e ferramentas de *merge*. Entre eles, Clementino (CLEMENTINO; BORBA; CAVALCANTI, 2021), propõe uma nova ferramenta de *merge* textual que visa simular a abordagem estruturada, recorrendo aos separadores sintáticos de uma dada linguagem. Neste estudo, ele analisa a performance da ferramenta proposta com dois diferentes conjuntos de separadores para a linguagem Java. Embora o impacto na resolução de conflitos não tenha sido muito expressivo, foi o esforço inicial que preparou terreno para trabalhos nessa linha.

5.3.2 Ferramentas de Notificando de *Merge*

Pesquisadores analisaram outras maneiras de detectar ou prevenir conflitos, minimizando assim o impacto na produtividade. Sarma et al. (SARMA; REDMILES; HOEK, 2012) apresenta *Palantir*, uma ferramenta projetada para reduzir conflitos, notificando os desenvolvedores sobre mudanças paralelas no mesmo artefato. Com esse intuito, existem também outras ferramentas como: *Syde* (HATTORI; LANZA, 2010), *CloudStudio* (NORDIO; MEYER; ESTLER, 2011) e *Cassandra* (KASI; SARMA, 2013). Também temos *Bellevue* (GUZZI et al., 2015), uma extensão IDE que torna as alterações confirmadas sempre visíveis e o histórico de código acessível no espaço de trabalho do desenvolvedor.

Alternativamente, quando a tarefa está pronta para ser integrada, o TIPMerge (COSTA et al., 2016) possui o algoritmo que recomenda o desenvolvedor mais adequado para realizar a integração. Isso é feito baseado em diversas métricas, como as experiências anteriores do desenvolvedor com o projeto, sua consciência envolvida e também a dependência existente entre os arquivos modificados.

5.3.3 Ferramenta de *Merge* Semiestruturado

Visando reduzir o alto custo de criação de ferramentas estruturadas, a abordagem semiestruturada foi proposta. Por exemplo, Apel et al. (APEL et al., 2011) propuseram e avaliaram a ferramenta semiestruturada FSTMerge. Alguns estudos (APEL et al., 2011; CAVALCANTI; ACCIOLY; BORBA, 2015; CAVALCANTI; BORBA; ACCIOLY, 2017), utilizando o FSTMerge, mostraram

que a abordagem semiestruturada consegue reduzir o número de falsos conflitos gerados pela não estruturada, porém, essa redução não ocorreu para todos os projetos e cenários analisados. Descobertas semelhantes, mas em menor grau, são observadas por Trindade et al. (TAVARES et al., 2019) ao investigarem o *merge* semiestruturado em projetos Javascript. Apel et al. (APEL; LEßENICH; LENGAUER, 2012) também propuseram uma ferramenta estruturada, chamada JDime, capaz de ajustar o processo de *merge* em tempo real, alternando entre o estruturado e não estruturado, a depender da ocorrência de conflitos. Seguindo a mesma linha, Zhu et al. (ZHU; HE; YU, 2019) propuseram a AutoMerge, uma ferramenta baseada no JDime. Eles conseguiram reduzir o número de falsos conflitos, comparado ao JDime original. Além disso, o estudo realizado por Cavalcanti et al. (CAVALCANTI et al., 2019) comparou a abordagem semiestruturada com a estruturada e constatou que apesar da semiestruturada tender a gerar mais falsos conflitos, a estruturada gera uma quantidade maior de falsos negativos.

Cavalcanti et al. (CAVALCANTI; BORBA; ACCIOLY, 2017), apresenta uma comparação entre os métodos de merge textual ou não estruturada e semiestruturada.Foram reproduzidos 30.000 *merges* de 50 *open source* identificando conflitos relatados incorretamente por uma abordagem, mas não pela outra (falsos positivos), e conflitos relatados corretamente por uma abordagem, mas perdidos pela outra (falsos negativos). Nos resultados e análises complementares indicam melhores resultados para a merge semiestruturada. Também foi implementada uma ferramenta de *Merge* semiestruturada que combina as duas abordagens para reduzir os falsos positivos e os falsos negativos do *Merge* semiestruturada. Encontraram-se evidências de que a ferramenta, quando comparada ao *Merge* não estruturada na amostra estudada, reduz o número de conflitos relatados pela metade, não possui falsos positivos adicionais e tem pelo menos 8% menos falsos negativos.

5.3.4 Ferramenta de *Merge* Semântico

5.3.4.1 Ferramenta de Merge *Semântico Baseada em Testes*

Brun et al.(BRUN et al., 2013) sugerem uma análise especulativa para detecção e prevenção de conflitos. Para isso, analisaram três projetos *Java*. Porém, precisam confiar nos testes de projeto que muitas vezes não são suficientes para detectar interferências. Eles criam *commits* de *merge* localmente, e se o processo de compilação falhar devido a um teste com falha, eles consideram a cena de *Merge* como tendo um conflito semântico. Outro detalhe importante é

que testes com falha não são executados nos *commits Left*, *Right* e *Base* do cenário de *Merge*, o que pode levar a falsos positivos, pois testes com falha podem ser causados por mudanças que acontecem em apenas um ramo.

Nguyen et al (NGUYEN et al., 2015) apresentam o Semex, uma ferramenta para detectar qual combinação de mudanças integradas causa um conflito de teste baseado em uma técnica chamada execução consciente da variabilidade (NGUYEN; KäSTNER; NGUYEN, 2014). Primeiro, a ferramenta separa as alterações feitas por cada commit *Left* e *Right* no cenário de *Merge* e codifica cada uma delas usando condicionais em seu entorno (instruções if), para integrar todas essas alterações em um único programa. A Semex usa a execução com reconhecimento de variabilidade para detectar conflitos semânticos, executando testes de projetos existentes, se disponíveis neste único programa, explorando todas as combinações possíveis das alterações codificadas. A ferramenta reconhece quais combinações de confirmações levam à falha de teste e relata o conjunto de confirmações que, se integradas, causariam um conflito de teste. Relatar um conflito baseado exclusivamente na falha de um teste no código integrado nem sempre implica em conflito ou interferência. Se o teste falhar em um dos commits *Left* ou *Right* também, a falha no *Merge* pode simplesmente indicar a herança de um defeito. Se o teste passar na *Base* e em um commit *Left* ou *Right*, a falha no código integrado pode ser simplesmente causada por uma mudança de comportamento não interferente de outro commit *Left* ou *Right*. Por fim, a Semex é avaliada preliminarmente usando exemplos *toy* PHP, com casos de testes criados manualmente, especialmente para revelar conflitos, pois o foco principal é avaliar o potencial da execução consciente da variabilidade, com o intuito de identificar combinações de ramificações conflitantes, e não o potencial para detecção de conflitos.

Da Silva *et al* (SILVA et al., 2020) propuseram uma abordagem baseada em geração de testes automatizados para detecção de interferência em cenários de integração. Com conflito de 38 cenários testados, a abordagem conseguiu detectar alguns conflitos e não apresentou nenhum falso positivo, o que pode indicar que a abordagem proposta seria um ferramenta útil para detecção de conflitos semânticos. Já a nossa abordagem, baseada em análise estática, busca utilizar uma abordagem diferente para detecção dos conflitos, e propõe uma técnica capaz de detectar mais casos positivos, no entanto, também apresenta uma alta taxa de falsos negativos para os 78 cenários da amostra testados. Uma combinação das duas abordagens poderia ser usada em trabalhos futuros, buscando reduzir o número de erros, principalmente os falsos negativos.

5.3.4.2 *Ferramenta de* Merge *Semântico Baseada em Análise Estática*

Horwitz et al. (HORWITZ; PRINS; REPS, 1989a) propuseram uma definição do problema de interferência na integração de código e seu algoritmo de detecção. O algoritmo usa um gráfico para detectar o controle e o fluxo de dados entre as contribuições das versões a serem integradas. No entanto, este algoritmo não é implementado e tem limitações: ele assume uma linguagem de programação simples onde as expressões são apenas escalares, variáveis, constantes e apenas instruções de atribuição, condicionais e *loops* de repetição *while*. Em contraste com este trabalho, que propõe uma análise e a implementa para a linguagem de programação *Java*, a avaliando em projetos *open-source* reais.

Barros Filho (FILHO, 2017) também propôs a utilização de análises com o objetivo principal de entender se o Information Flow Control (IFC), uma técnica de segurança utilizada para descobrir vazamentos em software, pode ser utilizada para indicar a presença de conflitos semânticos dinâmicos entre as contribuições dos desenvolvedores em cenários de merge. A análise implementada usava o *framework JOANA (Java Object-sensitive Analysis)* para executar uma análise de *Information Control Flow*. Essa análise se mostrou capaz de detectar casos de interferência, porém com uma alta taxa de falsos positivos. Contudo, a análise implementada utilizava um grafo complexo para representar as dependências entre unidades do código, o que faz com que a análise consuma muitos recursos computacionais. Em nosso trabalho apresentamos uma análise mais simplificada que busca detectar apenas conflitos de OA. Avaliamos a análise proposta utilizando uma amostra maior do que o dobro da amostra de Barros Filho e obtivemos um número de falsos positivos consideravelmente menor.

Por fim, ferramentas baseadas em estratégias semânticas também foram propostas (BINKLEY; HORWITZ; REPS, 1995b; HORWITZ; PRINS; REPS, 1989b; JACKSON; LADD, 1994; YANG; HORWITZ; REPS, 1992), mas por muito tempo têm-se demonstrado impraticáveis. Em um esforço recente para viabilizar ferramentas semânticas, Sousa et. al. (SOUSA; DILLIG; LAHIRI, 2018) propuseram SafeMerge, uma ferramenta semântica que verifica se o *merge* de programas não introduz novos comportamentos indesejados. Eles descobriram que a ferramenta proposta pode identificar problemas comportamentais em integrações problemáticas gerados por ferramentas não estruturadas. Ainda para reduzir conflitos, Rocha et al. (ROCHA; BORBA; SANTOS, 2019) propuseram uma ferramenta que, no contexto de BDD (desenvolvimento orientado por comportamento), analisa estaticamente testes para inferir quais arquivos serão modificados pelos desenvolvedores, evitando a execução paralela de tarefas que potencialmente levarão a

conflitos de código.

6 CONCLUSÕES

O desenvolvimento colaborativo de *software* traz alguns desafios quando os desenvolvedores fazem suas tarefas, separadamente e simultaneamente. Quando diferentes contribuições são integradas, podem surgir conflitos prejudicando a produtividade. Os conflitos podem surgir em diferentes fases de desenvolvimento: durante o *merge*, quando diferentes contribuições são integradas, após a integração, ao construir o resultado da integração, durante a execução dos testes ou em estágio de produção quando ocorre um comportamento inesperado do *software*.

Nesse estudo apresentamos uma proposta de análise estática de verificação de substituição de atribuições entre contribuições de dois desenvolvedores, de modo a detectar conflitos semânticos. Foi implementado duas abordagens para a análise proposta sendo uma *intraprocedural*(Ignora as chamadas de método dentro do método de entrada) e outra *interprocedural* (Acessa todas as chamadas de métodos até um certo limite de "profundidade"pré estabelecido).

A análise proposta se mostrou capaz de detectar cenários com substituições de atribuições e com interferência localmente observável entre as contribuições, no entanto, teve uma quantidade considerável de falsos negativos. Para OA, esses falsos negativos estão associados a limitações da análise, e acontecem, por exemplo, quando as modificações estão em atributos e não existe um método de entrada. Para detecção de interferências, muitos dos falsos negativos e falsos positivos são esperados, pois, nem todo OA implica necessariamente na existência de interferência localmente observável e vise-versa.

Quanto à capacidade de detectar cenários com interferência entre as modificações, apesar da análise proposta conseguir detectar alguns dos casos positivos, ela também deixou de detectar uma quantidade considerável, o que indica que ela não é suficiente para detectar cenários com interferência de forma confiável. No entanto, foi demonstrado que grande parte desses cenários são casos em que não foi encontrado OA pela análise manual, o que pode indicar que esses cenários não são detectáveis por análises de substituição de atribuição, portanto, a análise proposta poderia ser combinada com outras análises para compor uma ferramenta mais robusta para detecção de conflitos de integração semânticos.

Quando comparamos às duas abordagens da análise, seja para detecção de OA ou LOI, temos uma vantagem para a análise *interprocedural* que teve resultados melhores com relação a todas as métricas utilizadas. No entanto, a abordagem *intraprocedural* leva vantagem no

quesito de custo computacional e tempo de execução. Esse resultado também era esperado devido a características e natureza das duas abordagens. O tempo de execução total para todos os cenários analisados da abordagem *intraprocedural* foi de cerca de 4 minutos enquanto o da *interprocedural* foi de 46 minutos. Entendemos que em um cenário onde custo computacional e tempo é algo restrito a abordagem *intraprocedural* é mais interessante.

Por fim, analisando os resultados podemos perceber que a abordagem *interprocedural* é consideravelmente mais lenta em relação à média de tempo por cenário do que a abordagem *intraprocedural*. Isso é esperado, pois, como a *intraprocedural* tem como característica, ignorar as chamadas de método no método de entrada, ela percorre um caminho mais curto durante a execução. O contrário ocorre na execução *interprocedural* que acessar e analisa todo o corpo de todas as chamadas de método até que o seu limite de profundidade, definido por parâmetro, seja atingido.

Em seguida, iremos descrever possíveis trabalhos futuros na área, explorando as limitações deste trabalho.

6.1 TRABALHOS FUTUROS

Como trabalho futuro pretende-se implementar a resolução de algumas limitações conhecidas para a análise.

Umas dessas limitações acontece quando as alterações adicionadas pelos desenvolvedores são na inicialização de atributos que possuem mais de uma linha, dessa forma não existe um método de entrada. Uma solução para isso seria uma ferramenta que adicionasse automaticamente um método de entrada para casos como esse ou remover a dependência de um método de entrada para a execução da análise, passando a especificar o ponto de início da execução de outra forma. Para o caso em que os comandos alterados pelos desenvolvedores tem mais de uma linha pode ser usado um mapeamento das variáveis temporárias criadas pelo *Soot Framework* para determinar que todas estão ligadas a uma única variável resultante.

Buscando diminuir ainda mais a quantidade de falsos negativos apresentados pela análise nos resultados desses estudos, pretende-se realizar um esforço para regerar os arquivos JAR, utilizando o compilador do eclipse, pois ele tem uma precisão maior no mapeamento das linhas, certificando-se que todas as dependências estão incluídas no JAR final de cada cenário.

Pretende-se também, adicionar um tratamento para os casos em que ocorreu conflito de *merge* e o integrador preservou a mudança de apenas um dos desenvolvedores.

Outra funcionalidade que deve ser incorporada a implementação atual é uma análise de propagação de constantes, isso traria a capacidade de verificar os valores atribuídos nas variáveis, pois se o valor atribuído por *Left* em uma variável for o mesmo valor atribuído por *Right* nessa mesma variável, não existiu interferência entre as contribuições dos desenvolvedores, no entanto, atualmente a implementação detecta esses casos como OA.

Como foi descrito no Capítulo 3, a abordagem atual para marcação de linhas tem limitações que podem fazer com que a análise proposta classifique cenários de integração incorretamente. Nesse sentido, seria interessante explorar outras abordagens para detecção e marcação de código modificado. Além disso, a análise proposta não considera linhas de código removidas, que também podem causar interferências, nesse sentido é importante estudar formas de incluir esse tipo de modificação na análise proposta.

No Capítulo 4, informamos que não houve tempo hábil para montagem do sumário de erros da execução da análise *intraprocedural*. Isso será realizado em trabalhos futuros juntamente com uma análise mais detalhada dos acertos das duas abordagens, visando identificar se os alertas de interferências reportadas pelas execuções *intraprocedural* e *intraprocedural* correspondem às interferências detectadas durante a análise manual.

Por fim, pretende-se realizar estudos utilizando a análise proposta em combinação com outras ferramentas de detecção de conflitos semânticos relacionadas. Tanto outras análises estáticas como ferramentas baseadas em testes. Visando diminuir a quantidade de erros, principalmente dos falsos negativos.

REFERÊNCIAS

ACCIOLY, P.; BORBA, P.; CAVALCANTI, G. Understanding semi-structured merge conflict characteristics in open-source java projects (journal-first abstract). In: _____. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018. p. 955. ISBN 9781450359375. Disponível em: <https://doi.org/10.1145/3238147.3241983>.

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

APEL, S.; LEßENICH, O.; LENGAUER, C. Structured merge with auto-tuning: balancing precision and performance. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2012. p. 120–129.

APEL, S.; LIEBIG, J.; BRANDL, B.; LENGAUER, C.; KäSTNER, C. Semistructured merge: Rethinking merge in revision control systems. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2011. (ESEC/FSE '11), p. 190–200. ISBN 9781450304436. Disponível em: <https://doi.org/10.1145/2025113.2025141>.

APIWATTANAPONG, T.; ORSO, A.; HARROLD, M. J. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, Kluwer Academic Publishers, USA, v. 14, n. 1, p. 3–36, mar 2007. ISSN 0928-8910. Disponível em: <https://doi.org/10.1007/s10515-006-0002-0>.

BINKLEY, D.; HORWITZ, S.; REPS, T. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, v. 4, p. 3–35, 1995.

BINKLEY, D.; HORWITZ, S.; REPS, T. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, v. 4, n. 1, p. 3–35, jan 1995. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/201055.201056>.

BIRD, C.; ZIMMERMANN, T. Assessing the value of branches with what-if analysis. In: *SIGSOFT FSE*. [S.l.: s.n.], 2012.

BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Proactive detection of collaboration conflicts. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2011. (ESEC/FSE '11), p. 168–178. ISBN 9781450304436. Disponível em: <https://doi.org/10.1145/2025113.2025139>.

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

BUFFENBARGER, J. Syntactic software merging. In: *Software Configuration Management. Springer*. [S.l.: s.n.], 1993. p. 153–172.

CAVALCANTI, G.; ACCIOLY, P.; BORBA, P. Assessing semistructured merge in version control systems: A replicated experiment. In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2015. p. 1–10.

CAVALCANTI, G.; BORBA, P.; ACCIOLY, P. Evaluating and improving semistructured merge. *Proc. ACM Program. Lang.*, Association for Computing Machinery, New York, NY, USA, v. 1, n. OOPSLA, oct 2017. Disponível em: <https://doi.org/10.1145/3133883>.

CAVALCANTI, G.; BORBA, P.; SEIBT, G.; APEL, S. The impact of structure on software merging: Semistructured versus structured merge. In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2019. (ASE '19), p. 1002–1013. ISBN 9781728125084. Disponível em: <https://doi.org/10.1109/ASE.2019.00097>.

CLEMENTINO, J.; BORBA, P.; CAVALCANTI, G. Textual merge based on language-specific syntactic separators. In: _____. *Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021. p. 243–252. ISBN 9781450390613. Disponível em: <https://doi.org/10.1145/3474624.3474646>.

COSTA, C.; FIGUEIREDO, J.; MURTA, L.; SARMA, A. Tipmerge: Recommending experts for integrating changes across branches. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016. (FSE 2016), p. 523–534. ISBN 9781450342186. Disponível em: <https://doi.org/10.1145/2950290.2950339>.

DIAS, K.; BORBA, P.; BARRETO, M. Understanding predictive factors for merge conflicts. *Information and Software Technology*, v. 121, p. 106256, 05 2020.

ESTLER, H. C.; NORDIO, M.; FURIA, C. A.; MEYER, B. Awareness and merge conflicts in distributed software development. In: *2014 IEEE 9th International Conference on Global Software Engineering*. [S.l.: s.n.], 2014. p. 26–35.

FILHO, R. de B. Using information flow to estimate interference between same-method contributions. *Master's thesis, Universidade Federal de Pernambuco*, 2017.

GRASS., J. E. Cdiff: A syntax directed differencer for c++ programs. In: *Proceedings of the USENIX C++ Conference. USENIX Association*. [S.l.: s.n.], 1992.

GUZZI, A.; BACCHELLI, A.; RICHE, Y.; DEURSEN, A. van. Supporting developers' coordination in the ide. In: *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work amp; Social Computing*. New York, NY, USA: Association for Computing Machinery, 2015. (CSCW '15), p. 518–532. ISBN 9781450329224. Disponível em: <https://doi.org/10.1145/2675133.2675177>.

HATTORI, L.; LANZA, M. Syde: A tool for collaborative software development. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. New York, NY, USA: Association for Computing Machinery, 2010. (ICSE '10), p. 235–238. ISBN 9781605587196. Disponível em: <https://doi.org/10.1145/1810295.1810339>.

HORWITZ, S.; PRINS, J.; REPS, T. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, v. 11, p. 345–387, 1989.

HORWITZ, S.; PRINS, J.; REPS, T. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 11, n. 3, p. 345–387, jul 1989. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/65979.65980>.

JACKSON; LADD. Semantic diff: a tool for summarizing the effects of modifications. In: *Proceedings 1994 International Conference on Software Maintenance*. [S.l.: s.n.], 1994. p. 243–252.

KASI, B. K.; SARMA, A. Cassandra: Proactive conflict minimization through optimized task scheduling. In: *Proceedings of the 2013 International Conference on Software Engineering*. [S.l.]: IEEE Press, 2013. (ICSE '13), p. 732–741. ISBN 9781467330763.

KHANNA, S.; KUNAL, K.; PIERCE, B. C. A formal investigation of diff3. In: *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*. Berlin, Heidelberg: Springer-Verlag, 2007. (FSTTCS'07), p. 485–496. ISBN 3540770496.

LEENICH, O.; SIEGMUND, J.; APEL, S.; KäSTNER, C.; HUNSEN, C. Indicators for merge conflicts in the wild: Survey and empirical study. *Automated Software Engg.*, Kluwer Academic Publishers, USA, v. 25, n. 2, p. 279–313, jun 2018. ISSN 0928-8910. Disponível em: <https://doi.org/10.1007/s10515-017-0227-0>.

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

MENS, T. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, v. 28, n. 5, p. 449–462, 2002.

NGUYEN, H. V.; KäSTNER, C.; NGUYEN, T. N. Exploring variability-aware execution for testing plugin-based web applications. In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 907–918. ISBN 9781450327565. Disponível em: <https://doi.org/10.1145/2568225.2568300>.

NGUYEN, H. V.; NGUYEN, M. H.; DANG, S. C.; KäSTNER, C.; NGUYEN, T. N. Detecting semantic merge conflicts with variability-aware execution. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 926–929. ISBN 9781450336758. Disponível em: <https://doi.org/10.1145/2786805.2803208>.

NORDIO, M.; ESTLER, H.; FURIA, C.; MEYER, B. Collaborative software development on the web. *Computing Research Repository - CORR*, 05 2011.

NORDIO, M.; MEYER, B.; ESTLER, H. Collaborative software development on the web. *CoRR*, abs/1105.0768, 2011. Disponível em: <http://arxiv.org/abs/1105.0768>.

PERRY, D. E.; SIY, H. P.; VOTTA, L. G. Parallel changes in large-scale software development: An observational case study. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, v. 10, n. 3, p. 308–337, jul 2001. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/383876.383878>.

ROCHA, T.; BORBA, P.; SANTOS, J. P. Using acceptance tests to predict files changed by programming tasks. *Journal of Systems and Software*, v. 154, p. 176–195, 2019. Disponível em: <https://app.dimensions.ai/details/publication/pub.1113835899>.

SARMA, A.; REDMILES, D. F.; HOEK, A. van der. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, v. 38, n. 4, p. 889–908, 2012.

SHIHAB, E.; BIRD, C.; ZIMMERMANN, T. The effect of branching strategies on software quality. In: *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2012. p. 301–310.

SILVA, L. D.; BORBA, P.; MAHMOOD, W.; BERGER, T.; MOISAKIS, J. Detecting semantic conflicts via automated behavior change detection. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2020. p. 174–184.

SOUSA, M.; DILLIG, I.; LAHIRI, S. K. Verified three-way program merge. *Proc. ACM Program. Lang.*, Association for Computing Machinery, New York, NY, USA, v. 2, n. OOPSLA, oct 2018. Disponível em: <https://doi.org/10.1145/3276535>.

SOUZA, C. R. B. de; REDMILES, D.; DOURISH, P. "breaking the code", moving between private and public work in collaborative software development. In: *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work*. New York, NY, USA: Association for Computing Machinery, 2003. (GROUP '03), p. 105–114. ISBN 1581136935. Disponível em: <https://doi.org/10.1145/958160.958177>.

TAVARES, A. T.; BORBA, P.; CAVALCANTI, G.; SOARES, S. Semistructured merge in javascript systems. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2019. p. 1014–1025.

VALE, G.; HUNSEN, C.; FIGUEIREDO, E.; APEL, S. Challenges of resolving merge conflicts: A mining and survey study. *IEEE Transactions on Software Engineering*, p. 1–1, 2021.

WESTFECHTEL, B. Structure-oriented merging of revisions of software documents. In: *Proceedings of the 3rd International Workshop on Software Configuration Management*. New York, NY, USA: Association for Computing Machinery, 1991. (SCM '91), p. 68–79. ISBN 0897914295. Disponível em: <https://doi.org/10.1145/111062.111071>.

YANG, W.; HORWITZ, S.; REPS, T. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Trans. Softw. Eng. Methodol.*, Association for Computing Machinery, New York, NY, USA, v. 1, n. 3, p. 310–354, jul 1992. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/131736.131756>.

ZHU, F.; HE, F.; YU, Q. Enhancing precision of structured merge by proper tree matching. In: *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE Press, 2019. (ICSE '19), p. 286–287. Disponível em: <https://doi.org/10.1109/ICSE-Companion.2019.00117>.

ZIMMERMANN, T. Mining workspace updates in cvs. In: *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. [S.l.: s.n.], 2007. p. 11–11.

APÊNDICE A – PROTOCOLO PARA DEFINIÇÃO DE GROUND TRUTH

PROTOCOLO DE ANÁLISE MANUAL PARA DEFINIÇÃO DE *GROUND TRUTH*

Para definição do *ground truth* deve ser realizada uma checagem dupla:

- Para cada cenários uma dupla é alocada para realizar a análise manual seguindo as instruções abaixo.
- A análise manual deve ser feita de forma individual pelos colaboradores alocados no cenário.
- A análise individual dos dois colaboradores comparada
    - Se convergirem, o resultado deve ser adotado com *ground truth* do cenário;
    - Se divergir, os colaboradores devem se reunir e discutir o cenário juntos buscando chegar a um consenso.
        - Se não for possível chegar a um consenso, o cenário deve ser apresentado a outro colaborador que tomará a decisão.

INSTRUÇÕES PARA REALIZAÇÃO DA ANÁLISE MANUAL PARA DEFINIÇÃO DE *GROUND TRUTH*

O conjunto de dados consiste em um número de assuntos ou casos a serem analisados. Cada assunto é um par formado por um cenário de *merge* (tupla com *base*, *left*, *right* e *merge commits*) e uma declaração de método ou campo (sim, não focamos apenas em métodos) que foi alterado independentemente por dois *branches* (ou dois desenvolvedores). Um cenário que contém apenas uma única declaração alterada por duas ramificações leva a um único assunto (e, consequentemente, a uma única linha na planilha do conjunto de dados). Um cenário s que contém n declarações alteradas por duas ramificações leva a n assuntos: (s,*declaration* 1), …, (s,*declaration* n).

Para cada assunto(s,d) no conjunto de dados, a análise manual deve seguir estas etapas:

1. Verifique se você tem a versão mais recente do conjunto de dados (https://github.com/spgroup/mergedataset) em sua máquina.
2. Use o kdiff (http://kdiff3.sourceforge.net) para entender as mudanças feitas na base nos ramos esquerdo e direito.
    a. Basta carregar os *commits base*, esquerdo e direito de s no kdiff;
    b. Procure a declaração d alterada em ambas as ramificações;
    c. Concentre a análise nesta declaração;
    d. Consideramos o ramo esquerdo como a sequência (caminho) de *commits* do *commit* esquerdo até a base; da mesma forma para o ramo direito.
3. Faça uma ou mais capturas de tela que revelem as alterações na declaração sendo o foco da etapa anterior e como elas são integradas ao *commit* de mesclagem.

a. Adicione as capturas de tela a esta apresentação (https://docs.google.com/presentation/d/1_IhJulqfsj4OF8pGrTEEtbVwhtP9SFXldDDPtzYYyhg/edit?usp=sharing);
b. Considerando a ordem em que o cenário de mesclagem aparece na planilha do conjunto de dados, adicione um *slide* com o *hash* de confirmação de mesclagem;
c. Logo em seguida, adicione um ou mais *slides* com as capturas de tela do diff entre esquerda-base-direita (esta ordem é importante);
d. Depois disso, adicione uma captura de tela do *commit* de mesclagem com as anotações "// *left*" e "// *right*" no final das linhas alteradas pelas ramificações esquerda e direita, respectivamente;
    i. Se uma linha for alterada pela esquerda e pela direita, adicione "// *left and right*";
e. O foco das capturas de tela deve ser revelar as alterações na declaração, mas se a declaração for curta o suficiente, as capturas de tela devem mostrar toda a declaração
f. Se as alterações à esquerda e à direita levarem a um conflito de mesclagem
    i. Adicione uma nota ("Conflito de mesclagem") ao *slide*;
    ii. Se os arquivos no *commit* de mesclagem ainda tiverem os marcadores de conflito (ou seja, os conflitos não foram resolvidos), tente criar um novo *commit* que remova os marcadores e incorpore as alterações da esquerda e da direita de maneira lógica;
        1. Se isso não for possível, o assunto deve ser descartado.
        2. Caso contrário, continue com os arquivos mesclados no novo *commit* de mesclagem que você criou para corrigir os conflitos e adicione sua captura de tela à apresentação como no item "b" (a apresentação terá capturas de tela para a mesclagem original e fixa)
g. Se as alterações nas ramificações esquerda e direita não são incorporadas no *commit* de mesclagem (algumas das alterações podem ter sido removidas ou adaptadas pelo integrador por vários motivos: resolver um conflito de mesclagem, resolver um conflito de compilação, teste ou produção detectado, remover redundância, etc.), ou são incorporados, mas o *commit* de mesclagem tem alterações adicionais feitas pelo integrador (ou seja, algumas alterações não vêm dos ramos esquerdo e direito)
    i. Adicione uma observação ("Mesclar alterado pelo integrador") ao slide;
    ii. Reproduza a mesclagem (`git checkout left-hash; git merge right-hash`) e verifique se o resultado é diferente do que está armazenado no repositório original;
    iii. A análise manual deve prosseguir considerando ambos os *commits* de *merge* (o original no repositório e o repetido), com *screenshots* anotados para ambos (use "// *integrator*" se uma linha foi alterada pelo

integrador; se uma linha foi alterada pelo esquerda, direita e o integrador, adicione os três nomes ao comentário);

1. A análise do original nos ajudará a entender se há interferência localmente observável causada pelas mudanças feitas pela esquerda, direita e pelo integrador (possivelmente uma interferência que não foi causada pelas mudanças na esquerda e direita, mas pelas mudanças feitas pelo integrador);
2. A análise do repetido nos ajudará a entender se há interferência observável localmente causada pelas alterações feitas pela esquerda e pela direita (certamente uma interferência original causada pelas alterações da esquerda e da direita);

4. Resuma as alterações feitas no ramo esquerdo conforme observado no kdiff
    a. O foco está no que o kdiff mostra, não em todas as alterações feitas por *commits* na ramificação esquerda;
    b. Portanto, se um *commit* inicial no *branch* adiciona, por exemplo, uma atribuição à variável x (digamos x=1), e um *commit* posterior remove isso, não veremos isso na Etapa 1 acima; o resumo também não deve mencionar essa mudança, pois o foco está no resultado;
    c. O resumo deve ser expresso em termos de:
        i. As ações realizadas: adicionados, removidos, alterados;
        ii. As mudanças sintáticas feitas nos elementos do programa: classe, campo, construtor, declarações de método, blocos de inicialização, instruções, expressões, argumentos e parâmetros, condições `if` e `while`, `then branch`, `else branch`, corpo do método, corpo do loop, anotações, comentários;
        iii. Como nos estilos a seguir: "*Left* remove a referência ao campo `maxRetries`, na expressão *string* que é retornada pelo método `toString`"; "*Left* altera a condição de uma instrução *if* no método `outerHtmlHead`. A ramificação then do if chama o método `preserveWhitespace`, que agora só é chamado se o nó do parâmetro não for nulo e for uma instância de *Element*"
    d. Para decidir o que deve constar no resumo, pense em um exemplo "*toy*" que reproduza a essência das diferenças no assunto original. O resumo deve então descrever tal essência, utilizando o vocabulário do item acima.
5. Resuma as alterações feitas no ramo direito conforme observado no kdiff (semelhante ao item anterior);
6. Agora, analisando o *commit* do *merge*, verifique se a execução das alterações à esquerda pode substituir uma atribuição (OA) da execução das alterações à direita, ou vice-versa.
    a. Consideramos que pode haver tal fluxo quando as alterações (adições e modificações) em uma das ramificações podem semanticamente (ou seja, sua execução) envolver uma operação de escrita em um elemento de estado que também está associado a uma operação de escrita envolvida no alterações

(adições e modificações) feitas pela outra filial e não há operação de escrita da Base entre elas;
   b. Referências de campo como `o.a` e `o.b` são consideradas elementos de estado diferentes e podem ser escritas pelas ramificações sem levar ao OA. Por outro lado, `o` também é um elemento de estado diferente, mas não pode ser alterado junto com `o.a` ou `o.b` sem causar interferência; mas eles (`o` e `o.a`) podem ser escritos por ambos os ramos sem levar ao OA;
   c. Para *arrays* cada posição corresponde a um elemento de estado diferente. Dessa forma `a[0]` e `a[1]` podem ser escritos por versões de desenvolvedores diferentes e isso não leva a OA. No entanto, `a` também é um elemento de estado diferente, mas não pode ser alterado jundo com `a[0]` e `a[1]` sem causar interferência;
   d. Se uma das ramificações apenas remover o código, podemos saber facilmente que não há OA.
   e. Se *overriding* depender da execução de um `if`, considerar como OA;
   f. Inicializações de um mesmo *array* ou *var*, ou alterações no mesmo *return* em várias linhas, considerar como OA;
   g. Para estabelecer se existe OA, não pense na implementação atual do OA, basta seguir a definição conceitual de OA como nos itens anteriores;
7. Por fim, analisando o *commit* do *merge*, verifique se há interferência localmente observável. Não há necessidade de responder a todas as perguntas abaixo; responder apenas um é suficiente.
   a. As alterações em um dos ramos interferem (afetam negativamente) as alterações no outro ramo?
      i. Existe um elemento de estado x que *left* ou *right* calcula um valor diferente de base e *merge*?
      ii. *left*, *right* e base calculam o mesmo valor para x, mas *merge* calcula um valor diferente?
   b. O comportamento das alterações integradas (no *merge commit*) preserva o comportamento pretendido das alterações individuais (nos ramos esquerdo e direito)?
      i. Os novos efeitos da esquerda (direita) em relação à base são preservados na mesclagem, que não possui novos efeitos além dos da esquerda e da direita?
   c. Você consegue pensar em uma especificação (par pré-pós-condição) que é satisfeita pela declaração à esquerda (ou à direita), mas não é satisfeita pela declaração na mesclagem?
      i. A especificação deve referir-se apenas aos elementos de estado que estão transitivamente envolvidos nas mudanças à esquerda (ou à direita).
   d. Você consegue pensar em um teste que passaria na esquerda (ou direita), mas falharia na base e mesclaria?

   i. A asserção de teste deve se referir apenas aos elementos de estado (que podem envolver exceções, conforme explicado anteriormente) que estão transitivamente envolvidos nas mudanças à esquerda (ou à direita).

e. Você consegue pensar em um teste que falharia na esquerda (ou direita), mas passa na base e mescla?
   i. A asserção de teste deve se referir apenas aos elementos de estado que estão transitivamente envolvidos nas mudanças à esquerda (ou à direita).

f. Referências de campo como `o.a` e `o.b` são consideradas elementos de estado diferentes e podem ser alteradas pelas ramificações sem interferência; `o` também é um elemento de estado diferente, mas não pode ser alterado com `o.a` ou `o.b` sem causar interferência. O mesmo se aplica para *a[0]* e *a[1]* (em vez de `o.a` e `o.b)` e `a` (em vez de `o`). O raciocínio subjacente é que `o.a` pode ser alterado sem alterar ou afetar `o` valor armazenado em `o.b`, enquanto quando alteramos `o`, afetamos automaticamente o valor armazenado em `o.a`. Contrastando, ler `o`, como em `this.m(o)`, não significa necessariamente ler `o.a`, `o.b` e todos os campos de `o`. Elementos de tipos primitivos, *strings*, arquivos e fluxos são considerados atômicos; ou seja, alterar diferentes bits do mesmo inteiro está afetando o mesmo elemento de estado portanto, alterando diferentes partes do mesmo arquivo de texto ou gravando no mesmo fluxo;

g. Se as alterações em pelo menos uma das ramificações afetarem uma assinatura de classe (renomeação da classe ou de seus membros; adição, remoção ou alterações de visibilidade de membros da classe), às duas questões anteriores relacionadas ao teste devem assumir que as assinaturas podem ser feitas da mesma forma. Mesmo em todos os *commits* sem afetar o comportamento.

h. Se as alterações em uma das ramificações são simplesmente alterações de espaço em branco ou alterações nos comentários, claramente não há interferência.

i. Se as alterações em uma das ramificações são simplesmente refatorações estruturais (extrair campo, método, classe, renomeação, etc.) podemos dizer não haver interferência.
   i. Alterando "`if (x==null) return; y=x; if (x!=null) y=x.n();`" em "`if (x==null) return; y=x; y=x.n();`" é uma refatoração, mas não é estrutural, portanto, pode levar a interferências. Portanto, não podemos dizer não haver interferência sem analisar as mudanças no outro ramo.
   ii. Alterar a 'interface' de uma classe `C` não é uma refatoração de `C`. Por exemplo, alterar o método `m` de `C` de "`void m(){this.x=1;}`" para "`int m(){this.x=1; return this.x;}`" não é uma refatoração de `C`, pois isso quebra os clientes de `C`. Mas essa mudança

na declaração de `m`, junto com as mudanças que corrigem todos os clientes de m, é uma refatoração de todo o sistema se os clientes são devidamente fixados de uma forma que preserva o comportamento. Portanto, se uma classe `D` que chama m como em "`c.m();r=c.x`" é alterada para ter "`r=c.m()`", podemos dizer que `D` foi refatorado (mesmo que `C` não tenha sido). Se `D` for a classe em análise, podemos considerar que ela foi refatorada estruturalmente, portanto, não há interferência observável localmente.

j. Se uma declaração de campo (ao invés de um método) foi alterada em ambas as ramificações, a análise fica mais fácil: há interferência apenas se ambas as ramificações alterarem a parte de inicialização da declaração com valores diferentes. É equivalente às duas ramificações alterando a mesma instrução de atribuição no mesmo corpo do método.
    i. Nesses casos, não há necessidade de preencher a coluna OA, pois nem consideramos que a análise poderia ser chamada em tais situações (o *framework* de mineração não invoca a análise para esses cenários)
k. Em muitas, mas não em todas as situações, a possibilidade de uma OA não implicar em interferência localmente observável por vários motivos:
    i. a possibilidade de OA nunca se realizar devido, por exemplo, a uma condição que nunca é verdadeira;
    ii. os OA detectados já existiam na base, e apareceram em nossa análise porque as mudanças nas linhas envolvidas na análise tinham apenas mudanças de espaço em branco ou refatorações estruturais;
    iii. o OA detectado não existia na base, mas não tem efeito prejudicial (por exemplo, quando as mudanças em esquerda e direita atribuem o mesmo valor à mesma variável; temos OA, mas nenhuma interferência real).
    iv. Por isso temos que responder uma das perguntas acima, ao invés de simplesmente checar se a análise foi positiva nos itens anteriores desta lista.
l. Em muitas, mas não em todas as situações, a falta de DF, CF ou OA implica na falta de interferência localmente observável. Isso ocorre especialmente porque as análises podem não ser sólidas (podem levar a falsos negativos, como quando o código usa reflexão, por exemplo), e porque a interferência pode estar associada a outros tipos de análise que não implementamos (como uma análise que verifica se as alterações de uma ramificação afetam o fluxo de controle das alterações da outra ramificação).
m. Se uma das ramificações apenas remover o código, podemos saber facilmente que não há OA, mas ainda pode haver interferência localmente observável.

APÊNDICE B – TABELA SIMPLIFICADA DOS RESULTADOS DA ANALISE

Project	Merge Commit	Class Name	Method or field declaration changed by the two merged branches	Original Sample	Lines in the merge commit with changes made by left	Lines in the merge commit with changes made by right	Manual build?	Locally Observable Interference	Manual analysis of OA	OA Inter	OA Intra
antlr4	69ff2669eec265e25721dbc27cb00f6c381d0b41	org.antlr.v4.codegen.target.Python2Target	python2Keywords	Leuson	[64]	[53]	Yes	TRUE	TRUE	FALSE	FALSE
antlr4	69ff2669eec265e25721dbc27cb00f6c381d0b41	org.antlr.v4.codegen.target.Python3Target	python3Keywords	Leuson	[64]	[53]	Yes	TRUE	TRUE	FALSE	FALSE
druid	05168808c278c080c59c19e858d9471b316cd1f5	com.metamx.druid.loading.S3SegmentPusher	push(File, DataSegment)	Roberto	[66, 118, 139, 110]	[105, 125]	No	FALSE	FALSE	FALSE	FALSE
netty	193acdb36cd3da9bfc62dd69c4208dff3f0a2b1b	org.jboss.netty.handler.codec.frame.LengthFieldBasedFrameDecoder	decode (ChannelHandlerContext, Channel, ChannelBuffer)	Roberto	[358, 360]	[322, 374]	Yes	TRUE	FALSE	FALSE	FALSE
OpenTripPlanner	4c506dce43775704919d084f0acfba86d251bf4a	org.opentripplanner.routing.spt.MultiShortestPathTree	dominates (State, State)	Roberto	[144, 146, 147, 148, 149, 152, 141]	[123]	Yes	TRUE	FALSE	timeout	FALSE
webbit	74d2d2b87704d003acacb34e4ca8fb5f897b938f	org.webbitserver.netty.WebSocketClient	adjustPipelineToWebSocket (ChannelHandlerContext, MessageEvent, ChannelHandler, ChannelHandler)	Mining	[260]	[262]	Yes	FALSE	FALSE	FALSE	FALSE
resty-gwt	867b917c43c32acbdcac55767e7f04334006c866	org.fusesource.restygwt.rebind.DirectRestServiceInterfaceClassCreator	getMethodCallback(JMethod)	Guilherme	[83, 84, 85]	[86, 87, 88, 89, 90, 91]	Yes	TRUE	FALSE	FALSE	FALSE
storm	ad2be678831b3b060229fd936e3908110162b7ac	org.apache.storm.kafka.spout.KafkaSpoutConfig	toString()	Leuson	[]	[515, 516, 521, 522]	Yes	TRUE	TRUE	FALSE	FALSE
storm	bd1f5c54752f67b484a83c2666733123423 4d3a3	org.apache.storm.kafka.spout.KafkaSpout	emitTupleIfNotEmitted (ConsumerRecord<K,V>)	Leuson	[305, 306, 307, 312]	[316, 319]	Yes	FALSE	FALSE	FALSE	FALSE
swagger-maven-plugin	e825a7fdc6ef688f1253b93d2cb236e710acfc56	com.github.kongchen.swagger.docgen.reader.AbstractReader	hasValidAnnotations (List<Annotation>)	Guilherme	[342, 343]	[342]	Yes	TRUE	TRUE	TRUE	TRUE
jsoup	a8b6982de98ff76ef254031d7152fff57f6bf941	org.jsoup.helper.HttpConnection	execute (Connection.Request, Response)	Roberto	[584, 585, 586]	[544, 547]	No	TRUE	FALSE	FALSE	FALSE
jsoup	fee4762322f85a1109edd75ccb67f38cf5008c80	org.jsoup.helper.HttpConnection	createConnection(Connection.Request)	Guilherme	[609, 610, 611, 612, 613, 614]	[617]	No	FALSE	FALSE	FALSE	FALSE
jsoup	3f7d2c71dbbbb289c684f339874eed8ac2747fa0	org.jsoup.helper.HttpConnection	execute (Connection.Request, Response)	Roberto	[456]	[483, 484, 485]	No	TRUE	FALSE	FALSE	FALSE
jsoup	a44e18aa3c1fcd25a68a5965f9490d8f7d026509	org.jsoup.nodes.TextNode	outerHtmlHead (StringBuilder, int, Document.OutputSettings)	Roberto	[94]	[98]	Yes	TRUE	FALSE	FALSE	FALSE
retrofit	2b6c719c6645f8e48dca6d0047c752069d321bc4	retrofit.RestAdapter	logAndReplaceResponse (String, Response, long)	DeSouza	[438]	[415]	No	FALSE	TRUE	FALSE	TRUE
retrofit	2b6c719c6645f8e48dca6d0047c752069d321bc4	retrofit.RestAdapter	logAndReplaceRequest (Request)	DeSouza	[398]	[369]	No	TRUE	TRUE	FALSE	TRUE
retrofit	71f622ce51031b152a0be6ad5facfb27a654bf5a	retrofit.RequestBuilder	build()	Roberto	[88]	[96, 97, 100, 95]	No	TRUE	FALSE	FALSE	FALSE
activiti	50d8e43eb5917c63abfbcdec1e68e510943f325a	org.activiti.engine.impl.persistence.entity.DeploymentEntityManager	deleteDeployment(String, boolean)	Roberto	[107]	[114, 115, 116, 117]	No	TRUE	FALSE	FALSE	FALSE
activiti	bf46684ba62f5883673ea8fb0a14aecfe0aedea2	org.activiti.engine.impl.bpmn.behavior.UserTaskActivityBehavior	execute (ActivityExecution)	Roberto	[112, 113, 114, 115, 116, 117, 110, 111]	[118, 125, 126, 127]	No	TRUE	TRUE	TRUE	FALSE
okhttp	1151c9853ccc3c9c3211c613b9b845b925f8c6a6	com.squareup.okhttp.internal.bytes.GzipSource	consumeHeader(Deadline)	Roberto	[112]	[138]	No	FALSE	FALSE	FALSE	FALSE
okhttp	35166168529bd27281685e56a0a122eff44460e9	com.squareup.okhttp.OkHttpClient	copyWithDefaults()	DeSouza	[277, 293]	[287]	No	FALSE	TRUE	TRUE	FALSE
HikariCP	1bca94af9ec625f21d1b58ff10efb5be71ab87a6	com.zaxxer.hikari.HikariConfig	validate()	Roberto	[577, 578, 579, 580, 581, 582]	[]	No	TRUE	FALSE	FALSE	FALSE
pushy	58901c846e4f0874977c5aabbc34bcb4de3670e0	com.relayrides.pushy.apns.PushManager	handleConnectionClosure (ApnsConnection<T>)	Guilherme	[520, 510]	[515, 500, 502, 503]	No	FALSE	FALSE	FALSE	FALSE

Project	Merge Commit	Class Name	Method or field declaration changed by the two merged branches	Original Sample	Lines in the merge commit with changes made by left	Lines in the merge commit with changes made by right	Manual build?	Locally Observable Interference	Manual analysis of OA	OA Inter	OA Intra
java-faker	ca42fcfaf45cae1754c58e02e1d5d2a58ec03561	com.github.javafaker.Faker	Faker(Locale, Random)	Guilherme	[69, 59]	[70]	No	FALSE	FALSE	FALSE	FALSE
swagger-core	e7fea7c4889dbdd6a4e0b2d059c8aa0f126ab1c2	io.swagger.jaxrs2.OperationParser	getSchemaFromAnnotation(io.swagger.oas.annotations.media.Schema)	Guilherme	[240, 241, 234, 235, 236, 237, 238, 239]	[244, 187, 188]	No	FALSE	FALSE	FALSE	error
cucumber-jvm	4505c156b6267c1b760deec570ddbfe047b42aa9	cuke4duke.internal.java.JavaLanguage	load(String)	Roberto	[38, 40]	[36]	No	FALSE	FALSE	FALSE	FALSE
SimianArmy	345ad9513aafff397050d613fa87ad06ddffe99d	com.netflix.simianarmy.basic.janitor.BasicJanitorMonkeyContext	getInstanceJanitor()	Roberto	[227, 228, 229, 230]	[232, 233, 234, 235, 237, 238]	Yes	FALSE	FALSE	FALSE	FALSE
libgdx	da27e2dae56be0a159e82231e5c3a5b83b099063	com.badlogic.gdx.backends.lwjgl3.Lwjgl3Application	newWindow(ApplicationListener, Lwjgl3WindowConfiguration)	DeSouza	[318]	[325]	No	FALSE	FALSE	FALSE	FALSE
libgdx	da27e2dae56be0a159e82231e5c3a5b83b099063	com.badlogic.gdx.backends.lwjgl3.Lwjgl3ApplicationConfiguration	copy(Lwjgl3ApplicationConfiguration)	DeSouza	[96, 97, 98, 99]	[107]	No	FALSE	FALSE	FALSE	FALSE
RxJava	a40a4130edcdadc2340da5787ca30353018173f2	rx.internal.operators.OperatorMulticast	connect(Action1<? superSubscription>)	Roberto	[94]	[128, 129, 131, 117, 118, 119, 120, 121, 122, 124, 125]	No	TRUE	TRUE	TRUE	FALSE
RxJava	1c47b0cb26e6d971eefa42ebb3479099236b0125	rx.concurrency.TestScheduler	triggerActions(long)	DeSouza	[100]	[96, 97, 95]	Yes	FALSE	FALSE	FALSE	TRUE
elasticsearch-river-mongodb	3d4f99516ba3177f7d88a1f600923138a8b77cc8	org.elasticsearch.river.mongodb.Slurper	assignCollections()	Roberto	[288, 289, 290, 322, 291, 292, 293, 294, 295, 297, 298, 286, 287]	[315, 331, 302]	Yes	TRUE	FALSE	FALSE	TRUE
hector	0588608e7a2bdf974c985ff546207104f672bf6c	me.prettyprint.cassandra.connection.client.HSaslThriftClient	open()	Roberto	[96, 97, 94, 111, 95]	[122]	Yes	TRUE	TRUE	TRUE	TRUE
titan	387c16ea05ef9fa312f37139228d2bbf61455ff4	com.thinkaurelius.titan.graphdb.database.serialize.SerializerInitialization	initialize(Serializer)	Roberto	[52, 53, 54, 55, 57]	[31]	Yes	FALSE	FALSE	FALSE	FALSE
MPAndroidChart	af114d180da6ec5633d32c701ff5467f7629fcf3	com.xxmassdeveloper.mpchartexample.fragments.SimpleFragment	generateScatterData(int, float, int)	DeSouza	[80, 91]	[73]		FALSE	FALSE	FALSE	FALSE
MPAndroidChart	9297923f09460d0484cd713b0abbeaea9b888ec0	com.xxmassdeveloper.mpchartexample.LineChartActivity2	onCreate(Bundle)	DeSouza	[126]	[117]	Yes	FALSE	FALSE	FALSE	FALSE
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.common.settings.IndexScopedSettings	BUILT_IN_INDEX_SETTINGS	Mining	[119, 120]	[121]	Yes	TRUE	TRUE	FALSE	error
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.action.index.TransportIndexAction	executeIndexRequestOnPrimary(IndexRequest, IndexShard, MappingUpdatedAction)	Mining	[196]	[201]	Yes	FALSE	FALSE	FALSE	error
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.action.support.replication.ReplicationOperationTests	testReplication()	Mining	[106, 108, 125, 126]	[116, 103]	Yes	FALSE	FALSE	FALSE	error
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.index.IndexSettings	IndexSettings(IndexMetaData, Settings, Predicate<String>, IndexScopedSettings)	Mining	[238]	[246, 269]	Yes	FALSE	FALSE	timeout	error
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.action.bulk.TransportShardBulkAction	update(IndexMetaData, BulkShardRequest, IndexShard, long[], VersionType[], Translog.Location, int, BulkItemRequest)	Mining	[268, 255]	[244, 249, 266, 254]	Yes	FALSE	FALSE	FALSE	error

Project	Merge Commit	Class Name	Method or field declaration changed by the two merged branches	Original Sample	Lines in the merge commit with changes made by left	Lines in the merge commit with changes made by right	Manual build?	Locally Observable Interference	Manual analysis of OA	OA Inter	OA Intra
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.action.support.replication.TransportReplicationAction	doRun()	Mining	[455, 458, 460]	[457, 459]	Yes	TRUE	FALSE	FALSE	FALSE
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.action.support.replication.ReplicationOperation	execute()	Mining	[98, 109]	[113, 98, 99, 101, 119, 108, 110]	Yes	FALSE	FALSE	FALSE	TRUE
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.index.engine.InternalEngineTests	testRecoverFromForeignTranslog()	Mining	[2091]	[2124]	Yes	FALSE	FALSE	FALSE	error
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.index.engine.InternalEngine	InternalEngine(EngineConfig)	Mining	[134]	[161, 162, 163, 164]	Yes	FALSE	FALSE	FALSE	FALSE
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.action.DocWriteResponse	writeTo(StreamOutput)	Mining	[140]	[141]	Yes	TRUE	TRUE	TRUE	FALSE
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.action.DocWriteResponse	readFrom(StreamInput)	DeSouza	[129]	[130]	Yes	FALSE	FALSE	FALSE	FALSE
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.action.DocWriteResponse	toXContent(XContentBuilder, Params)	DeSouza	[163, 164, 165, 166]	[160]	Yes	TRUE	TRUE	TRUE	FALSE
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.cluster.metadata.MetaDataCreateIndexService	createIndex(CreateIndexClusterStateUpdateRequest, ActionListener<ClusterStateUpdateResponse>)	Mining	[315]	[304, 305, 309, 310, 312, 346, 300]	Yes	FALSE	FALSE	timeout	error
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.action.update.TransportUpdateAction	shardOperation(UpdateRequest, ActionListener<UpdateResponse>, int)	Mining	[224, 185, 254]	[192, 249, 250]	Yes	FALSE	TRUE	TRUE	error
elasticsearch	d896886973660785aac45275ddb110c1a6babc57	org.elasticsearch.action.delete.TransportDeleteAction	executeDeleteRequestOnPrimary(DeleteRequest, IndexShard)	Mining	[136]	[140]	Yes	FALSE	FALSE	FALSE	error
elasticsearch	3764b3ff800c94293aba0bb0fa18c7df80a764f7	org.elasticsearch.rest.action.cat.RestNodesAction	getTableWithHeader(RestRequest)	DeSouza	[]	[153, 154, 156, 157, 158, 159]	Yes	FALSE	FALSE	timeout	FALSE
elasticsearch	3764b3ff800c94293aba0bb0fa18c7df80a764f7	org.elasticsearch.rest.action.cat.RestIndicesAction	getTableWithHeader(RestRequest)	DeSouza	[]	[144, 147, 150, 153, 138, 141]	Yes	FALSE	FALSE	timeout	FALSE
elasticsearch	3764b3ff800c94293aba0bb0fa18c7df80a764f7	org.elasticsearch.rest.action.cat.RestShardsAction	getTableWithHeader(RestRequest)	DeSouza	[]	[112, 113]	Yes	FALSE	FALSE	timeout	FALSE
elasticsearch	0404db65e3497452886173957729c8e82cfd4a03	org.elasticsearch.test.ESSingleNodeTestCase	newNode()	DeSouza	[171]	[173, 174]	Yes	FALSE	FALSE	FALSE	FALSE
elasticsearch	59cb67c7bd0ab6311115b20954e013412b676b29	org.elasticsearch.index.query.support.NestedInnerQueryParseSupport	setPathLevel()	DeSouza	[193, 195, 197, 200]	[199]	Yes	FALSE	TRUE	FALSE	FALSE
elasticsearch	59cb67c7bd0ab6311115b20954e013412b676b29	org.elasticsearch.index.query.MultiMatchQueryBuilder	doXContent(XContentBuilder, Params)	Mining	[306, 367]	[]	Yes	TRUE	FALSE	FALSE	FALSE
elasticsearch	f3d63095dbcc985e24162fbac4ee0d6914dc757d	org.elasticsearch.index.analysis.synonyms.SynonymsAnalysisTest	testSynonymsAnalysis()	DeSouza	[65, 66, 67, 68, 69, 70, 71, 76]	[73, 75]	Yes	FALSE	FALSE	FALSE	TRUE
elasticsearch	f3d63095dbcc985e24162fbac4ee0d6914dc757d	org.elasticsearch.index.analysis.commongrams.CommonGramsTokenFilterFactoryTests	testQueryModeCommonGramsAnalysis()	Mining	[229]	[226, 228]	Yes	FALSE	TRUE	TRUE	FALSE
elasticsearch	f3d63095dbcc985e24162fbac4ee0d6914dc757d	org.elasticsearch.index.analysis.commongrams.CommonGramsTokenFilterFactoryTests	testCommonGramsAnalysis()	Mining	[144]	[141, 143]	Yes	FALSE	TRUE	TRUE	FALSE

Project	Merge Commit	Class Name	Method or field declaration changed by the two merged branches	Original Sample	Lines in the merge commit with changes made by left	Lines in the merge commit with changes made by right	Manual build?	Locally Observable Interference	Manual analysis of OA	OA Inter	OA Intra
elasticsearch	f3d63095dbcc985e24162fbac4ee0d6914dc757d	org.elasticsearch.index.analysis.KuromojiAnalysisTests	createAnalysisService()	Mining	[196, 197, 198, 199, 200, 201, 202, 206]	[204, 207]	Yes	FALSE	FALSE	FALSE	FALSE
elasticsearch	36884807b3cc9d660db4da062275c7fdbec8ba67	org.elasticsearch.index.query.SimpleIndexQueryParserTests	setup()	DeSouza	[217]	[208]	Yes	FALSE	FALSE	FALSE	FALSE
fitnesse	4d9ba9d221d879507440feb084fa7521b95111ec	fitnesse.testsystems.slim.tables.SlimTableFactoryTest	setUp()	Roberto	[32, 31]	[38]	Yes	TRUE	FALSE	FALSE	FALSE
fitnesse	4d9ba9d221d879507440feb084fa7521b95111ec	fitnesse.testsystems.slim.tables.SlimTableFactory	SlimTableFactory()	Roberto	[24, 25]	[31]	Yes	TRUE	FALSE	FALSE	FALSE
cloud-slang	20bac30d9bd76569aa6a4fa1e8261c1a9b5e6f76	io.cloudslang.lang.api.SlangImpl	getAllEventTypes()	Leuson	[116]	[123, 124, 125]	Yes	TRUE	TRUE	TRUE	FALSE
spring-boot	6664ce19d6f2388ebd6cf54763f54fdddd226b9a	org.springframework.boot.jta.atomikos.AtomikosProperties	asProperties()	DeSouza	[323, 324, 325, 326, 327, 328, 317]	[]	No	TRUE	FALSE	not-found	not-found
spring-boot	6664ce19d6f2388ebd6cf54763f54fdddd226b9a	org.springframework.boot.jta.atomikos.AtomikosPropertiesTests	testProperties()	Mining	[49, 68, 55, 56, 57, 58, 74, 75, 76, 77]	[]	No	TRUE	FALSE	not-found	not-found
spring-boot	6664ce19d6f2388ebd6cf54763f54fdddd226b9a	org.springframework.boot.jta.atomikos.AtomikosPropertiesTests	testDefaultProperties()	Mining	[89, 93, 94, 95, 96, 98]	[]	No	TRUE	FALSE	not-found	not-found
spring-boot	958a0a45f164601d01cb706c19f22ed3e25eff56	org.springframework.boot.autoconfigure.mongo.MongoProperties	builder (MongoClientOptions)	DeSouza	[272]	[261, 262, 263, 264, 265, 266, 267, 268, 271]	No	FALSE	FALSE	FALSE	FALSE
spring-boot	af20dc6cc45c032573413c401f9f73aa75371744	org.springframework.boot.loader.archive.ExplodedArchiveTests	getUrl()	Mining	[]	[]	Yes	FALSE	FALSE	FALSE	FALSE
spring-boot	af20dc6cc45c032573413c401f9f73aa75371744	org.springframework.boot.loader.archive.ExplodedArchive	getUrl()	DeSouza	[]	[]	Yes	FALSE	FALSE	FALSE	FALSE
spring-boot	c93ea54ea3e08eaa2a17640d5d2b3e60264c1a9c	org.springframework.boot.autoconfigure.cache.RedisCacheConfiguration	cacheManager (RedisTemplate<Object, Object>)	DeSouza	[55]	[60]	No	FALSE	FALSE	not-found	not-found
spring-boot	ea8107b6a53fa60b5f23b33e1b6d2e88bb60133c	org.springframework.boot.context.embedded.undertow.UndertowEmbeddedServletContainerFactory	createDeploymentManager (ServletContextInitializer)	DeSouza	[357, 358, 359, 360, 345, 361, 362]	[364]	Yes	FALSE	FALSE	FALSE	TRUE
spring-boot	3444ebbc05b99a164474c14d6a6784f749514428	org.springframework.boot.context.web.SpringBootServletInitializer	createRootApplicationContext (ServletContext)	DeSouza	[99]	[121, 123]	No	FALSE	FALSE	FALSE	FALSE
spring-boot	074771ec125dd407af0282b92960e9e9e3377e84	org.springframework.boot.context.web.SpringBootServletInitializer	createRootApplicationContext (ServletContext)	DeSouza	[87]	[88]	No	FALSE	TRUE	TRUE	FALSE
spring-boot	fdd3f12ee0f92ac18844c08bf71df39feebb6673	org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer	logAutoConfigurationReport (boolean)	DeSouza	[114]	[110]	No	TRUE	TRUE	FALSE	FALSE
spring-boot	2d4e68a9777601bfb8309c94d8b74bc21be80ad1	org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServletContainerFactory	customizeConnector (Connector)	DeSouza	[240, 241, 242, 243, 244, 246, 247, 248, 249]	[228]	No	FALSE	FALSE	FALSE	FALSE

Project	Merge Commit	Class Name	Method or field declaration changed by the two merged branches	Original Sample	Lines in the merge commit with changes made by left	Lines in the merge commit with changes made by right	Manual build?	Locally Observable Interference	Manual analysis of OA	OA Inter	OA Intra
quickml	bae968d9a85c0501ba8842a6f88e7fc2c6b78693	quickdt.experiments.OutOfTimeCrossValidatorRunner	getRandomForestBuilder(int, int)	Guilherme	[]	[42, 43]	No	FALSE	FALSE	FALSE	FALSE