Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Uma análise pessimista de fluxo de dados para detecção de conflitos de integração semânticos

Rafael Mota Alves

Trabalho de Graduação

Recife
16 de Dezembro de 2021

Universidade Federal de Pernambuco
Centro de Informática

Rafael Mota Alves

Uma análise pessimista de fluxo de dados para detecção de conflitos de integração semânticos

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Paulo Henrique Monteiro Borba*

Recife
16 de Dezembro de 2021

Resumo

Em times de desenvolvimento de software, tipicamente desenvolvedores adicionam suas modificações em repositórios ou versões separadas do código, e posteriormente essas modificações precisam ser integradas em um repositório ou versão central. Esse processo permite que desenvolvedores trabalhem em suas respectivas tarefas em isolado. No entanto, nesse processo de integração podem ocorrer conflitos entre as versão integradas, esses conflitos podem ou não ser detectados pela ferramentas de integração atuais e caso não sejam, podem introduzir *bugs* ao sistema.

Um dos tipos de conflito de integração que não é detectado pelas ferramentas atuais de integração é o semântico. Esse tipo de conflito ocorre quando, no código integrado, as mudanças introduzidas por uma versão interferem de forma inesperada com as mudanças introduzidas pela outra em tempo de execução fazendo com o que um contrato pretendido por uma das mudanças deixe de ser cumprido. Esse tipo de conflito é difícil de detectar e resolver, pois demanda contexto sobre as intenções de ambas versões.

Esse trabalho propõe o uso de uma análise de fluxo de dados pessimista para detecção de conflitos de integração semânticos. Essa análise considera a possibilidade de que métodos executados modifiquem e leiam o estado da aplicação sem estender a execução para métodos além do método de entrada.

A implementação da análise proposta foi avaliada para detecção de cenários com fluxo de dados ou interferência entre as modificações usando um conjunto de 72 cenários de integração de código em que ambas versões sendo integradas modificaram o mesmo método. Esse cenários foram extraídos de projetos *open-source Java*, que foram minerados por uma ferramenta de mineração de cenários de integração do *Github*. Os resultados obtidos foram comparados com implementações de análises de fluxo de dados tradicionais.

Os resultados mostram que a análise é capaz de detectar tanto cenários com fluxos de dados quanto com interferência entre as contribuições. No entanto, indica também uma grande quantidade de falsos positivos. A análise detectou corretamente mais casos com fluxos de dados que as implementações tradicionais (33,3% invés de 10% e 16,7%), mas teve uma taxa maior de falsos positivos (20% invés de 0% para ambas análises tradicionais). Apesar de detectar cenários em que houve interferência, a análise teve uma taxa considerável de falsos negativos (16,1%), o que indica que ela não é suficiente sozinha para detectar cenários com interferência mas que pode ser combinada com outras análises para conseguir um bom resultado.

Palavras-chave: Fluxo de Dados, Conflitos de Integração, Semântico, Análise Estática

Abstract

In software development teams, developers typically implement their changes in separate repositories or versions of the code and subsequently these changes need to be merged in a central repository or version. This process allows developers to work in their tasks separately, but when integrating, the versions changes can conflict with each other, some of those conflicts can be detected with regular code merge tools prompting the developers to solve them and some of them can't, possibly introducing bugs to the merged program.

One of the types of code integration conflicts that can't be detected with the traditional integration tools is the semantic type. This kind of conflict happens when the changes introduced by one of the versions interferes in an unexpected way with the changes introduced by the other version in runtime causing the merged code to not have the intended behaviour. This kind of conflict is very hard to detect and fix because the resolving developer needs context of the intentions of the changes in both integrated versions.

This work evaluates the use of a pessimistic Data-flow analysis for semantic conflicts detection. The proposed analysis is a pessimistic version of the intraprocedural Data-flow analysis, that considers the possibility that invoked methods read and change the state of the application without checking the methods implementation.

The implementation of the proposed analysis was evaluated for detection of merge scenarios with Data-flows and interference between the integrated changes using a merge scenario dataset with 72 scenarios that both integrating versions changed the same method body. These scenarios were extracted from open source Java projects from Github. The results obtained for the proposed implementation were compared with traditional Data-flow analysis implementations.

The obtained results show that the proposed analysis is capable of detecting merge scenarios with Data-flows and interference, but also indicates the analysis has a big ratio of false positives for Data-flow and interference. The analysis correctly detected more positive cases than the traditional implementations (33,3% instead of 10% and 16,7%), but also had a bigger false positive ratio (20% instead of 0% for both traditional implementations). Although the analysis detects most of the cases that occurred interference, it had an expressive false negative ratio (16,1%), which indicates that the proposed implementation is not sufficient to detect merge scenarios with interference alone, but it is also demonstrated that the proposed analysis could be combined with other types of analysis for composing a effective semantic conflict detection tool.

Keywords: Data-flow, Merge Conflicts, Semantic, Static Analysis

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

Em times de desenvolvimento de software, tipicamente desenvolvedores adicionam suas modificações em repositórios ou versões separadas do código, e posteriormente essas modificações precisam ser integradas em um repositório ou versão central. Esse processo permite que desenvolvedores trabalhem em suas respectivas tarefas em isolado. No entanto, o processo de integração ocasiona custos: seja para resolver conflitos textuais de merge, que são indicados pelas ferramentas de integração atuais, ou para detectar e resolver *bugs* introduzidos por conflitos que não são detectados pelas ferramentas convencionais.

Um dos tipos de conflito de integração que não é detectado pelas ferramentas atuais de integração é o semântico. Esse tipo de conflito ocorre quando, no código integrado, as mudanças introduzidas por uma versão interferem de forma inesperada com as mudanças introduzidas pela outra em tempo de execução. Horwitz et al [5] especificaram formalmente esse tipo de conflito da seguinte forma: 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*. Esse tipo de conflito é difícil de detectar e resolver, pois é necessário o contexto da mudança de ambas versões para entender as intenções de cada desenvolvedor e determinar se elas interferem entre si, e resolver essa interferência [1].

Com conflitos semânticos em mente, se fazem necessárias ferramentas que possam detectar conflitos desse tipo no processo de integração, a fim de evitar bugs e facilitar a resolução dos mesmos. Nesse sentido, abordagens baseadas em geração de testes [7] e em algoritmos de análise estática [2][5][4] foram propostas.

Algoritmos de análise estática para detecção de conflitos semânticos já foram propostos [2][5][4], mas esses são baseados em grafos complexos (*System Dependence Graphs* [5]) e a medida que as bases de código aumentam tem a performance bastante degradada, o que torna essas soluções difíceis de ser aplicadas em bases de código reais de mais de 50 KLOC [4]. Diante disso, surge a necessidade de explorar simplificações dos algoritmos originalmente propostos, a fim de verificar a sua capacidade de detectar os conflitos semânticos com menor custo computacional.

Uma das possíveis simplificações dos algoritmos propostos é a análise de fluxo de dados, que pode ser usada para detectar para quais partes de um programa um valor atribuído a uma

[1] A diferença entre um cenário com interferência e um com conflito é a intenção do desenvolvedor, caso uma mudança interfira com o resultado de outra de forma planejada, haverá interferência mas não conflito, esse trabalho foca no conceito de interferência como aproximação para conflito por não ser possível inferir as intenções dos desenvolvedores.

variável pode ser propagado. Esse tipo de algoritmo é utilizado para implementar otimizações em compiladores e em análises de segurança de código. Para o problema de detecção de conflitos semânticos, nossa hipótese é que um algoritmo de fluxo de dados poderia ser utilizado para detectar fluxos entre o código introduzido pelas versões *Left* e *Right*, e vice-versa, a fim de detectar possíveis interferências entre as modificações [4]. A ideia básica, por exemplo, é que se uma variável cujo valor é modificado pelas alterações de código feitas por *Left* , e uma alteração de *Right* faz com que esse valor seja lido, há forte risco de interferência. A hipótese é que a alteração de *Right* foi feita assumindo o valor original da variável, não o novo valor definido por *Left*, que é o que efetivamente será lido pelo código modificado por *Right* na versão integrada do código. Dessa forma, há chances de que a mudança feita por *Left* interfira no comportamento da mudança feita por *Right*, fazendo com que o contrato assumido pelo desenvolvedor de *Right* não seja satisfeito na versão do código que integra as mudanças de *Left* e *Right*.

Estudos iniciais com esse tipo de análise para detecção de conflitos são promissores. Mesmo as análises intraprocedurais que consideram apenas o método alvo para a análise, são capazes de detectar conflitos com uma boa performance, mas são incapazes de detectar conflitos causados por fluxos de dados interprocedurais, o que faz com que esse tipo de análise apresente mais falsos negativos. Já as análises interprocedurais, que consideram todo o caminho de execução a partir do método alvo, são mais complexas, por requererem a execução em vários métodos e a inferência de implementações em casos em que o código usa abstrações como interfaces, o que faz com elas tenham uma performance pior em bases de código maiores.

Com a motivação de explorar o potencial de melhor performance de uma abordagem intraprocedural, mas com reduzido número de falsos negativos, este trabalho propõe uma versão pessimista da análise de fluxo de dados intraprocedural, que considera a possibilidade de que métodos invocados modifiquem e leiam o estado do objeto alvo da chamada sem estender a execução para métodos além do método de entrada. Ror exemplo, em um caso em que *Left* adiciona uma chamada para um método em um objeto *o* que tem potencial de modificar um campo *c*, e *Right* adiciona uma chamada para outro método que calcula um valor baseado em *c* no objeto *o* (o.c) e retorna esse valor, a análise proposta detectaria esse fluxo de dados sem a necessidade de checar a implementação dos métodos chamados. No entanto, em um cenário em que *Left* adiciona uma chamada para um método que modifica um campo *c2*, e *Right* adiciona a mesma chamada para o método que apenas lê o campo *c*, a análise proposta detectará um fluxo incorretamente, caracterizando um falso positivo. A análise proposta busca assim detectar dependências que seriam detectadas apenas por fluxos interprocedurais, com o desempenho de um algoritmo intraprocedural.

A implementação da análise proposta foi avaliada usando um conjunto de 72 cenários de integração de código em que ambas versões *Left* e *Right* modificaram o mesmo método. Esse cenários foram extraídos de projetos *open-source Java*, que foram minerados pela ferramenta de mineração de cenários de integração do *Github* miningframework. Alguns desses cenários foram utilizados em outros estudos relacionados [4] [7].

Os resultados da execução da análise foram comparados com informações de existência de fluxo de dados e interferência entre as contribuições das versões *Left* e *Right* obtidos a partir de uma análise manual dos cenários. O primeiro *ground truth*, de fluxo de dados, é útil para anali-

sar o potencial da nossa análise para detectar a existência de fluxos de dados reais. O segundo *ground truth*, de interferência, é útil para analisar o potencial da nossa análise com relação ao objetivo maior de detectar conflitos semânticos, já que fluxo de dados entre contribuições integradas é apenas um entre vários indícios de interferência e conflito semântico. Além disso, a capacidade da análise proposta de detectar fluxos de dados foi comparada com a capacidade de duas implementações tradicionais do algoritmo de fluxo de dados: uma intraprocedural otimista e outra interprocedural.

Os resultados mostram que, a análise pessimista proposta é capaz de detectar corretamente mais cenários com fluxos de dados entre as contribuições de diferentes versões do que as implementações tradicionais (33,3% invés de 10% e 16,7%); esses cenários são casos de fluxos interprocedurais em que as análises interprocedurais tipicamente falham: os que tratam de abstrações como interfaces, em que é necessária a inferência das implementações dos métodos invocados. No entanto, por sua natureza pessimista, a análise proposta também tem uma taxa de falsos positivos maior (20% invés de 0% para ambas análises tradicionais). Esses falsos positivos em geral são casos em que os desenvolvedores adicionaram chamadas para métodos com os mesmos objetos alvo mas essas chamadas não ocasionam fluxos de dados. Isso ocorre quando os métodos chamados não modificam ou não leem campos, ou modificam e leem campos diferentes. No entanto, a análise teve um único falso negativo, o que indica que a análise "perde" poucos casos positivos.

Em comparação com os dados de interferência obtidos a partir de uma análise manual, a análise proposta se mostrou capaz de detectar cenários com interferência, com uma taxa de 22,6% de positivos verdadeiros. No entanto, também ficou claro que apenas a análise proposta não é suficiente para detectar conflitos semânticos, podendo ser combinada com outras análises para obter uma taxa de acerto melhor. A análise também apresentou um taxa considerável de falsos positivos, esses casos estão associados aos falsos positivos para fluxo de dados causados pela natureza pessimista da análise proposta e a casos em que as modificações de uma ou ambas versões foram refatorações, em que pelo o comportamento não mudar, não ocorre interferência entre as mudanças.

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 as análise proposta, especificando o algoritmo e descrevendo detalhes da implementação, o Capítulo 4 descreve como análise proposta foi avaliada e os resultados obtidos e o Capítulo 5 conclui o trabalho descrevendo trabalhos relacionados e possíveis trabalhos futuros.

CAPÍTULO 2

Caso Motivador

Para ilustrar a noção de conflito de integração semântico e demonstrar como a análise proposta tenta detectar esse tipo de conflito, utilizaremos um cenário de integração proposto por Silva *et all* [7]. A classe *Text* ilustrada na Figura 2.1, é resultado da integração das mudanças em verde (Linha 6 foi adicionada, por um versão *Left*) e as mudanças em vermelho (Linha 8 foi adicionada, por um versão *Right*). As outras linhas de código originam de uma versão *Base*, da qual ambas versões *Left* e *Right* originaram. Perceba que, nesse caso, as mudanças estão separadas por uma linha de código e o código integrado é sintaticamente válido, o que indica que ferramentas de integração tradicionais não indicariam conflito. Em seguida, iremos demonstrar que apesar de não ser detectado, ocorreu um conflito de integração semântico nesse cenário de integração.

```
1     class Text {
2
3         public String text;
4
5         void cleanText() {
6   +         this.normalizeWhitespace();
7             this.removeComments();
8   +         this.removeDuplicatedWords();
9         }
10    }
```

Figura 2.1 Caso de exemplo de conflito integração semântico

A modificação introduzida pela versão *Left* tinha como objeto garantir que a *string* contida na classe *Text* não tivesse caracteres de "espaço" consecutivos duplicados. Para isso, o desenvolvedor estendeu a implementação do método *cleanText* adicionando uma chamada para o método *normalizeWhitespace* que modifica o campo *text*. Já a intenção da modificação introduzida pela versão *Right* foi garantir que não haveriam palavras consecutivas duplicadas. Para isso o desenvolvedor, estendeu a implementação do método *cleanText* adicionando uma chamada para o método *removeDuplicateWords* que também modifica o campo *text* removendo as palavras duplicadas sem remover os espaços entre elas.

Em isolado, ambos desenvolvedores implementaram suas tarefas corretamente, cumprindo os contratos que pretendiam. No entanto, por causa de um detalhe da implementação adicionada pela versão *Right*, o código integrado não cumpre o contrato pretendido pelo desenvolvedor de *Left* (a *string* resultante não deve ter caracteres de espaço repetidos). A quebra deste

contrato pode ser demonstrada por um caso de teste que passa em *Left*, mas falha na versão integrada. Um caso de teste que demonstra esse conflito, que está ilustrado na Figura 2.2, executa o método *cleanText* com a *string* de entrada “the the dog” e verifica se a *string* resultante tem espaços duplicados. Quando executado na versão *Left* a *string* resultante é igual à de entrada que não tem espaços duplicados e o teste passa, já quando executado na versão integrada a *string* resultante seria: "the dog", que não tem palavras duplicadas mas tem espaços duplicados o que faz o teste falhar.

```
1   class TextTestSuite {
2
3       public void test1() throws Throwable {
4           Text t = new Text();
5           t.text = "the the dog";
6           t.cleanText();
7           assertTrue(t.noDuplicateWhiteSpace());
8       }
9   }
```

Figura 2.2 Caso de teste que demonstra o conflito semântico na Figura 2.1

A análise proposta busca detectar esse tipo de conflito encontrando o fluxo de dados entre as mudanças de *Left* e as de *Right*. Nesse caso em especifico, há um fluxo de dados desse tipo pois ambos desenvolvedores adicionaram chamadas a métodos que leem e modificam o campo *text*, e portanto a execução do método *normalizeWhitespace*, adicionado por *Left* modificaria o campo *text* e a execução do método *removeDuplicateWords* leria o valor modificado, o que caracterizaria um fluxo de dados novo entre as mudanças de *Left* e *Right*, indicando o conflito. No entanto, vale salientar que só acontece interferência e portanto um conflito nesse caso pois o fluxo de dados detectado não existe nas versões *Left* e *Right*, pois caso existisse seria um comportamento esperado e portanto não caracterizaria conflito.

Um algoritmo de fluxo de dados intraprocedural tradicional não seria capaz de detectar esse conflito, pelo fato dos fluxos ocorrerem em métodos executados pelo método de entrada. Já um algoritmo interprocedural, poderia detectar um conflito nesse caso específico, mas seria necessário gerar grafos de execução e verificar a implementação de todos os métodos executados a partir do método de entrada o que pode ser custoso para sistemas grandes e quando o fluxo só pode ser detectado após vários níveis de chamadas de método. Nesse caso, temos a profundidade miníma, 1, nas duas chamadas. A análise proposta busca detectar o conflito a partir de uma abordagem mais pessimista, assumindo que os métodos executados leem e modificam todos os campos do objeto alvo. Neste caso, a assunção feita pela análise é correta, o que faz com que a análise detecte corretamente um conflito semântico.

CAPÍTULO 3

Análise pessimista de fluxo de dados

As seções a seguir definem a análise proposta e descrevem a sua implementação:

3.1 Visão Global

A análise proposta segue uma abordagem "pessimista" para detecção de fluxos de dados, ela considera a possibilidade de que ocorra a escrita e a leitura dos campos para os objetos em que métodos são executado, sem verificar as suas implementações.

Tem como base uma análise convencional de fluxos de dados intraprocedural, em que o caminho de execução de um método de entrada é percorrido, buscando sequências de *def* (instruções que fazem atribuições de elementos de estado do programa como: variáveis locais, campos de objetos e posições de *arrays*) e *use* (instruções que leem elementos de estado do programa) em que a instrução *def* está marcada como *source* e a instrução *use* está marcada como *sink*, sendo as instruções marcadas como *source* os pontos de origem, e as instruções marcadas como *sink* os de destino. Também considera fluxos de dados indiretos, ou seja, se um valor marcado como modificado por uma instrução *source* for utilizado por uma instrução não *source* para definir um valor, esse valor também será marcado como modificado por *source*, e caso uma instrução *sink* leia esse valor, será detectado um fluxo.

Além do comportamento supracitado, a análise tem um comportamento especial ao encontrar instruções de invocação de método, ela considera que houve tanto um *def* quanto um *use* dos campos do objeto alvo. A partir dessa estratégia vem o comportamento "pessimista" da análise, pois ela não checa a implementação do método executado, mas considera que houve tanto a mudança quanto a leitura dos campos a fim de detectar fluxos interprocedurais. Apesar de também estarem sujeitos a terem seus campos modificados ou lidos, os objetos passados como parâmetro para chamadas de método não tem o mesmo comportamento. Esse comportamento não foi implementado como uma tentativa de reduzir falsos positivos, e por acreditarmos não ser um caso comum em programas *Java*.

Para esse trabalho, buscamos encontrar fluxos entre as instruções introduzidas por uma versão para instruções introduzidas pela outra, portanto para um cenário de integração de duas versões *Left* e *Right* a análise é executada duas vezes: uma com a versão *Left* como *source* e a versão *Right* como *sink* e outra com a versão *Left* como *sink* e a versão *Right* como *source*.

3.2 Especificação do Algoritmo

O código ilustrado na Figura 3.1 define a estrutura alto nível da implementação do algoritmo. Onde define uma versão simplificada do algoritmo e pressupõe métodos já implementados para fins didáticos. O fluxo principal do algoritmo é definido pelo método *execute*, que recebe uma lista de instruções na ordem em que elas são executadas para o método de entrada. Para cada instrução, o algoritmo executa uma série de procedimentos verificando e modificando uma abstração, que mantém os elementos de estado do programa (variáveis locais, campos, posições em *arrays*) que foram modificados por instruções marcadas como *source* direta ou indiretamente e verificando se a instrução atual é *sink* e utiliza algum dos elementos de estado marcados.

O primeiro procedimento a ser executado é o representado pelo método *detectConflicts*, que checa se a instrução atual é *sink* e se algum dos elementos de estado usados pela instrução está marcado na abstração, e caso esteja retorna verdadeiro. Adicionalmente, se a instrução for uma instrução de invocação de método, checa se o objeto alvo tem algum campo marcado, caso sim também retornará verdadeiro, essa parte é responsável pelo comportamento pessimista que considera que uma chamada de método lê todos os campos do objeto alvo.

Caso não seja encontrado um fluxo de dados, a análise continuará e executará a etapa implementada pelo método de *kill*. Essa etapa é responsável por desmarcar elementos de estado que foram adicionados a abstração, mas que foram sobrescritos por outras instruções, essa etapa é necessária para que esses valores não seja usados para detectar fluxos de dados posteriormente. O algoritmo checa quais foram os valores definidos pela instrução e os remove da abstração.

Por fim, é executado o procedimento *gen*, que adiciona os valores modificados por instruções *source* à abstração para que ela seja checada em instruções posteriores. O procedimento checa primeiro se a instrução está marcada como *source* ou se ela usa algum valor marcado na abstração (parte responsável por detectar fluxos indiretos), em seguida todos os valores definidos pela instrução são adicionados à abstração. Adicionalmente, caso a instrução seja uma instrução de invocação de método, o procedimento marcará todos os campos para o objeto alvo do método, essa parte é responsável pelo comportamento pessimista que considera que uma chamada de método modifica todos os campos do objeto alvo.

Em seguida, iremos demonstrar a execução do algoritmo descrito para o caso motivador ilustrado na Figura 2.1. Para isso, iremos considerar as linhas contribuídas por *Left* (em verde) como *source* e as linhas contribuídas por *Right* (em vermelho) como *sink*.

A primeira instrução a ser executada seria a da linha 6 (contribuição de *Left*), que está marcada como *source*. Para a execução do método *detectConflicts* ela seria ignorada, pois não é *sink*. No método *kill*, não removeria nenhum valor da abstração, pois a instrução não é uma instrução de atribuição. Já para a execução do método *gen*, por ser uma instrução *source* e uma instrução de invocação de método o método *markFields* seria executado, marcando todos os campos do objeto *this* (o objeto em que o método *cleanText* está definido), e consequentemente marcando o campo *text*.

Em seguida, o loop executará para a instrução da linha 7 (faz parte da versão *Base*), que não está marcado nem como *source* nem como *sink*. Essa instrução será ignorada pelos métodos *detectConflicts* e *kill* por não ser nem *source* nem *sink* e não ser uma instrução de atribuição. Já na

execução do método *gen*, os campos do objeto *this* serão adicionados na abstração novamente, pois consideramos que a instrução utiliza os campos já marcados do objeto *this*.

Por fim, o loop executará para a instrução da linha 8 (contribuição de *Right*), que está marcada como *sink*. Por ser uma instrução *sink* e de invocação de método, o método *detectConflicts* irá verificar se o objeto *this* tem campos marcados, e como eles foram marcados, irá retornar verdadeiro para a existência de fluxos de dados.

3.3 Marcação dos Campos

Ao encontrar uma instrução de invocação de método, a análise marca todos os campos do objeto alvo com modificados na abstração. No entanto, por ser intraprocedural e pessimista, a análise não checa a implementação da classe do objeto alvo, e considera de forma "cega" que o objeto tem campos marcados sem especificar quais.

Essa abordagem evita que tenhamos que listar todos os campos de longas estruturas hierárquicas para classes que herdam de outras classes e também retira a necessidade de inferir implementações concretas para subclasses e interfaces. Ambas as características citadas acima fazem com que a análise tenha uma performance melhor que outras análises que precisam considerar esses casos e reduzem a possibilidade de falsos negativos causados por erros na inferência de implementações.

No entanto, essa abordagem pode aumentar a taxa de falsos positivos pois considerará que os campos foram modificados até em casos em que o objeto em questão não tem campos declarados.

3.4 Implementação

As análises tradicionais e a proposta foram implementadas utilizando a linguagem de programação *Java* e o framework de análise estática *Soot Framework*. As implementações utilizam a *API* do *Soot Framework* para executar o algoritmo descrito anteriormente para as instruções do programa de entrada. Fazendo parte do repositório conflict-static-analysis, que reúne as implementações utilizadas para esse e outros trabalhos.

As análises recebem como entrada o código compilado da versão integrada empacotado em um arquivo *JAR* e um arquivo CSV, ilustrado na Figura 3.2, em que cada linha representa uma linha de código modificada, cada linha tem as informações do nome da classe modificada, o número da linha modificada para a versão integrada e se essa linha deve ser considerada *source* ou *sink*.

O arquivo CSV com as linhas modificadas por *Left* e *Right* é gerado pela ferramenta mining-framework 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 fluxo de dados 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 fluxos. Um dos casos onde essa estratégia falha é quando instruções que estão dividas 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 *source* e *sink* em que foram encontrados fluxos de dados entre.

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

```
1  public class PessimisticAnalysis {
2     public boolean execute(List<Statement> statements) {
3        for (Statement statement : statements) {
4           if (detectConflicts(statement) {
5              return true;
6           }
7           kill(statement);
8           gen(statement);
9        }
10       return false;
11    }
12
13    public boolean detectConflicts(Statement statement) {
14       if (statement.isSink()) {
15          for (Value value : statement.getUses()) {
16             if (isMarked(value) {
17                return true;
18             }
19          }
20          if (statement.isInvoke() &&
               hasMarkedFields(statement.target)) {
21             return true;
22          }
23       }
24       return false;
25    }
26
27    public void gen(Statement statement) {
28       if (statement.isSource() || usesMarkedValue(statement)) {
29          for (Value value : statement.getDefs()) {
30             mark(value, statement);
31          }
32          if (statement.isInvoke()) {
33             markFields(statement.target, statement);
34          }
35       }
36    }
37
38    public void kill(Statement statement) {
39       for (Value value : statement.getDefs()) {
40          unmark(value, statement);
41       }
42    }
43 }
```

Figura 3.1 Versão simplificada da análise implementada usando Java

```
1 Text,source,6
2 Text,sink,8
```

Figura 3.2 Exemplo do arquivo CSV de entrada para o caso motivador em que a análise encontraria fluxos de dados

CAPÍTULO 4

Avaliação

As seções a seguir descrevem a metodologia utilizada para avaliar a análise proposta e apresentam os resultados obtidos.

4.1 Metodologia

Para avaliar a capacidade da análise proposta de detectar fluxos de dados e interferências entre as contribuições de diferentes versões, foi utilizado um conjunto de dados com 72 cenários de integração reais de projetos *open-source* Java extraídos do *Github*. Cada um desses cenários representa um método ou campo de uma classe em que ambas versões *Left* e *Right* introduziram mudanças. Esse tipo de cenário em específico foi escolhido pois acreditamos que aumentaria as chances de detectar casos de interferência, além disso é possível detectar interferências com mais facilidade, tanto por um algoritmo quanto manualmente. 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 [7][4], e outros foram minerados a partir de uma lista de projetos *open-source* populares.

A ferramenta miningframework foi utilizada para coletar quais linhas foram modificadas ou adicionadas por cada uma versões *Left* e *Right* em cada um dos *commits* de integração. A ferramenta também foi utilizada para gerar os arquivos *JAR* com o código compilado para cada um dos *commits* de integração. Alguns dos casos em que os arquivos *JAR* não puderam ser gerados automaticamente, eles foram gerados manualmente. Todos os arquivos utilizados para avaliação, junto com as instruções para gerar os arquivos *JAR* que foram gerados manualmente podem ser encontrados no projeto mergedataset.

Devido as limitações da estratégia de marcação de linhas, alguns cenários tiveram código ou marcação de linhas ajustados manualmente de forma que a marcação de linhas funcionasse corretamente e que a essência do cenários de integração fosse mantida. Esses cenários modificados também foram registrados no mergedataset junto com a descrição das mudanças e o motivo delas.

O processo de coleta dos dados utilizados está descrito pela Figura 4.1. A ferramenta de mineração recebe como entrada um arquivo CSV com uma lista de repositórios no *Github* e cria *forks* dos repositórios para que possa configurar uma ferramenta de CI (Integração contínua). Em seguida, a ferramenta executa o processo descrito dentro da caixa tracejada para cada um dos *commits* de *merge* dos projetos passados como entrada.

Para cada *commit*, a ferramenta primeiro checa se existem métodos que foram 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, primeiro coletando os números das linhas das modificações de *Left* e *Right*, depois verificando a existência de conflitos de integração textuais e registrando em uma planilha e por fim, enviando 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* que usa os sistemas de *build Maven* ou *Gradle* para gerar os arquivos JAR.

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.

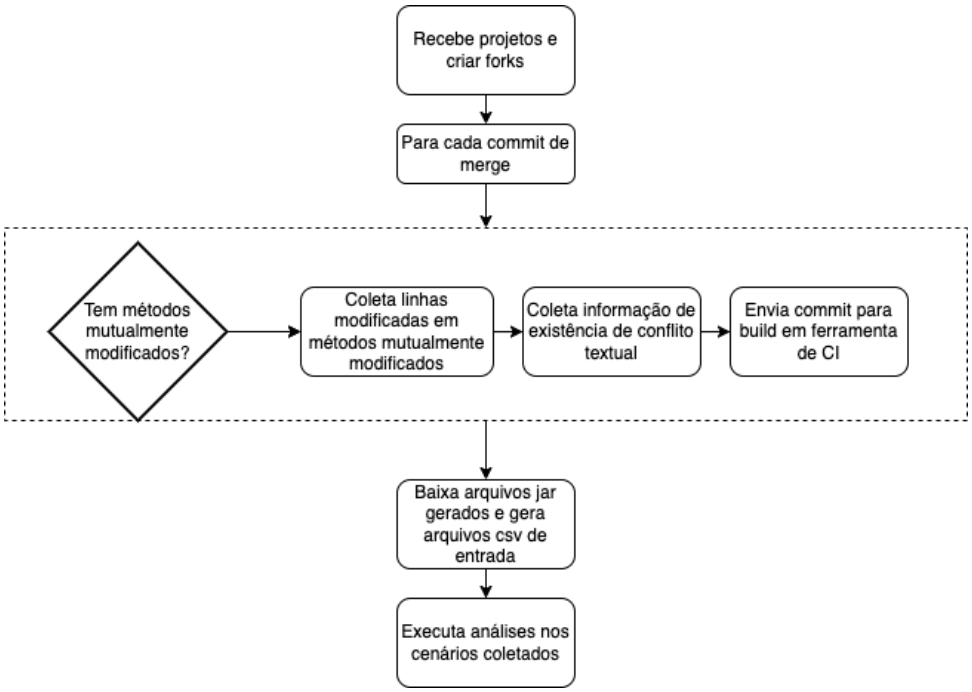

Figura 4.1 Fluxo de coleta dos dados sobre os cenários de integração

Com os dados necessários para os 72 cenários, o miningframework também foi utilizado para executar as análises intraprocedural e interprocedural e a análise proposta em cada um dos cenários. Cada análise avaliada foi executada duas vezes para cada cenário, uma execução com as mudanças de *Left* como *source* e as mudanças de *Right* como *sink* e outra com as mudanças de *Left* como *sink* e as mudanças de *Right* como *source*. Para fins de avaliação, o resultado de uma análise é considerado positivo para um cenário específico se uma, ou ambas, execuções retornarem um ou mais fluxos de dados.

Os cenários do conjunto de dados também foram analisados manualmente a fim de verificar a existência de fluxos de dados e interferências entre as modificações de *Left* e *Right* e assim estabelecer um *Ground Truth* para avaliar a análise proposta para detecção de cenários com fluxos de dados e interferência entre as contribuições. Para ambos, foi usado um processo estruturado e foram registradas justificativas para cada um dos cenários.

Avaliar a capacidade da análise de detectar fluxo de dados é importante para validar a sua implementação e comparar essa capacidade com a de outras análises de fluxo de dados tradi-

cionais. Já avaliar a capacidade de detectar casos de interferência vem da definição dada por Horwitz et al [5], que definiu um conflito de integração semântico como um cenário em que as contribuições das versões interferem entre si, portanto avaliar a capacidade da análise detectar interferência, é avaliar a capacidade da análise de compor uma ferramenta para detecção de conflitos de integração semânticos.

4.1.1 Métricas

A partir dos resultados da análises avaliadas e do *Ground Truth* estabelecido pela análise manual para os cenários dos conjunto de dados foram calculadas as taxas de acerto e erro e as métricas de desempenho para cada uma das análises em comparação. Os casos em que por algum motivo a execução da análise retornou algum erro, não foram considerados para comparação. Além das métricas calculadas, os cenários que foram falsos positivos ou negativos, para fluxo de dados ou interferência, foram analisados manualmente a fim de identificar porque a análise proposta "erra" para esses cenários.

A seguir descrevemos as métricas calculadas e como elas se aplicam para o contexto deste trabalho.

4.1.1.1 Precision

A métrica de *precision* indica a proporção dos casos classificados como relevantes que realmente são, ou seja, quantos dos casos que as análises indicaram que havia fluxo de dados ou interferência realmente tinham. Essa métrica é importante pois uma análise que tem essa proporção baixa pode indicar muitos "alarmes falsos", o que pode fazer com o que os desenvolvedores tenham que verificar muitos casos manualmente.

4.1.1.2 Recall

A métrica de *recall* indica a proporção de casos que são positivos e a ferramenta detecta. Nesse contexto, seria a razão dos casos em que existe fluxo de dados ou interferência e a ferramenta detecta. Essa métrica é importante pois uma análise que tem essa proporção muito baixa, não detectaria cenários que tem fluxos de dados ou interferências que poderiam indicar conflitos semânticos [4], introduzindo *bugs* ao sistema.

4.1.1.3 Acurácia

A métrica de acurácia indica a proporção de casos classificados 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 a favor de análises menos sensíveis a fluxos de dados como as análises tradicionais, ela foi incluída para demonstrar essa característica do problema.

4.2 Resultados

A Figura 4.2 ilustra os resultados da análise proposta para o conjunto de dados de avaliação. Cerca de 14% dos cenários resultaram em um erro de execução, esses erros estão associados ao *Soot Framework* e são causados por incompatibilidades entre as versões do *Java*, nesses casos, o código do cenário utiliza dependências ou funcionalidades da linguagem que a versão utilizada do *framework* não suporta, o que faz com que não seja possível gerar a lista de instruções que será utilizada por entrada pela análise proposta. Esses 10 casos em que a execução falhou foram desconsiderados para fins de comparação.

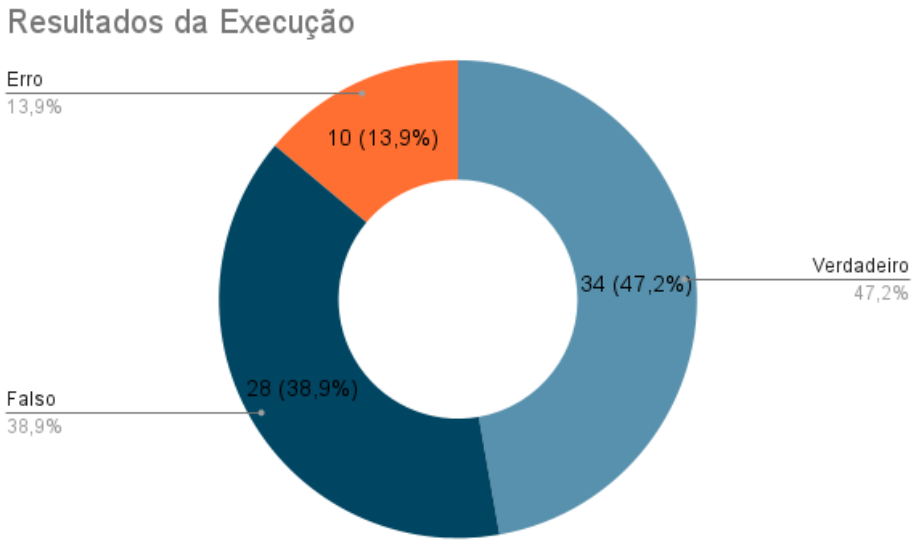

Figura 4.2 Resultados da execução da análise pessimista para detecção de fluxos de dados no conjunto de dados

A seguir, iremos demonstrar os resultados da análise proposta para dois problemas: o da detecção de fluxos de dados e o da detecção de interferências entre as modificações introduzidas. Os casos considerados como com fluxo de dados são os casos em que foi detectado um ou mais fluxos de dados entre as contribuições das versões *Left* e *Right* pela a análise manual. Os casos considerados como com interferência são os casos em que foi detectado algum tipo de interferência entre as mudanças, provado 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 fluxo de dados e não interferência e vice-versa.

4.2.1 Fluxo de Dados

Os resultados ilustrados na Figura 4.3 foram obtidos comparando os resultados da análise proposta com o *Ground Truth* para fluxo de dados, essa comparação tem como objetivo avaliar a capacidade da análise proposta para detecção da existência de fluxos de dados entre as contribuições de dois desenvolvedores em um cenário de integração de código.

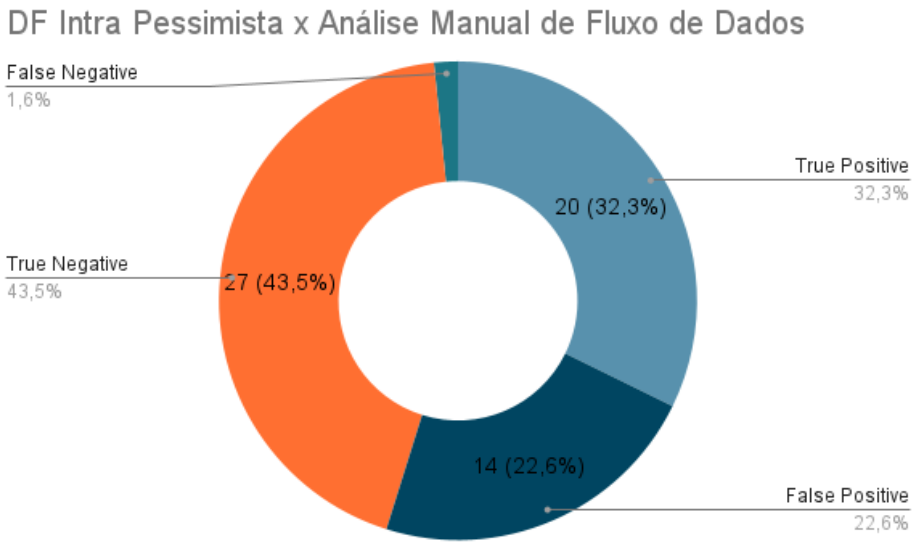

Figura 4.3 Resultados da análise pessimista em comparação com o *Ground Truth* de fluxo de dados

Dos 62 cenários considerados para comparação com a análise manual de fluxo de dados, mais da metade (54,9%) foram indicados como positivos, o que indica que a análise é bastante sensível a possíveis fluxos de dados. A análise detecta a grande maioria dos casos positivos, tendo apenas 1 cenário como falso negativo, o que indica que análise "perde" poucos casos positivos. No entanto, ela também indica uma grande quantidade de falsos positivos.

```
    if (taskDefinition.getFormKeyExpression() != null) {
        final Object formKey = (String) taskDefinition.getFormKeyExpression().getValue(execution);
        if (formKey != null) {
                if (formKey instanceof String) {
                        task.setFormKey((String) formKey);
                } else {
                        throw new ActivitiIllegalArgumentException("FormKey expression does not resolve to a string: " +
                taskDefinition.getFormKeyExpression().getExpressionText());
                }
        }
    }

    handleAssignments(task, execution);

    // All properties set, now firing 'create' events
    if (Context.getProcessEngineConfiguration().getEventDispatcher().isEnabled()) {
      Context.getProcessEngineConfiguration().getEventDispatcher().dispatchEvent(
        ActivitiEventBuilder.createEntityEvent(ActivitiEventType.TASK_CREATED, task));
    }

    task.fireEvent(TaskListener.EVENTNAME_CREATE);
  }
```

Figura 4.4 Cenário Activiti bf46684 como exemplo para o falso negativo de fluxo dados.

O único falso negativo indicado nos resultados está ilustrado pela Figura 4.4, neste cenário as contribuições da versão *Left* (código destacado em roxo) modificam o campo *formKey* no objeto *task* através da chamada do método *setFormKey* e as modificações da versão *Right* (código destacado em azul) passam esse objeto *task* como parâmetro em uma chamada do método *createEntityEvent* que lê o campo modificado por *Left*. Como citado anteriormente, esse tipo

de cenário não foi considerado para implementação da análise.

Já os cenários em que a análise indicou falsos positivos foram mais predominantes, com 22,6% de falsos positivos para fluxos de dados. Esses casos foram em sua maioria cenários em que a assunção pessimista da análise de que os métodos executados nos cenários leem e modificam todos os campos dos objetos estava errada. Sendo a maioria, casos em que *Left* e *Right* apenas adicionam métodos que seguem a convenção *getter* (métodos cuja a única responsabilidade é expor campos privados para leitura) da linguagem *Java*. Um sumário das causas para falsos positivos pode ser encontrado na Tabela 4.1.

Descrição do Cenário	Quantidade
Um desenvolvedor executa um método que modifica campos e outro executa um método que não lê esses campos	4
Ambos desenvolvedores executam métodos que apenas leem campos do objeto	7
Um desenvolvedor muda a assinatura do método e o outro usa um dos parâmetros	1
Ambos os desenvolvedores executam métodos que apenas modificam campos diferentes	2

Tabela 4.1 Descrições dos cenários em que a análise proposta indicou falsos positivos para fluxo de dados

Um exemplo do tipo de falso positivo mais predominante está ilustrado na Figura 4.5, nesse cenário ambas versões introduziram chamadas ao método *getId* para o objeto *processDefinition*, esse método funciona como um *getter* para o campo *id* e portanto apenas lê esse campo.

```
if (timerStartJobs != null && timerStartJobs.size() > 0) {

  long nrOfVersions = new ProcessDefinitionQueryImpl(Context.getCommandContext())
    .processDefinitionKey(processDefinition.getKey())
    .count();

  long nrOfProcessDefinitionsWithSameKey = 0;
  for (ProcessDefinition p : processDefinitions) {
    if (!p.getId().equals(processDefinition.getId()) && p.getKey().equals(processDefinition.getKey())) {
      nrOfProcessDefinitionsWithSameKey++;
    }
  }

  if (nrOfVersions - nrOfProcessDefinitionsWithSameKey <= 1) {
    for (Job job : timerStartJobs) {
      if (Context.getProcessEngineConfiguration().getEventDispatcher().isEnabled()) {
        Context.getProcessEngineConfiguration().getEventDispatcher().dispatchEvent(
          ActivitiEventBuilder.createEntityEvent(ActivitiEventType.JOB_CANCELED, job, null, null, processDefinition.getId()));
      }

      ((JobEntity)job).delete();
    }
```

Figura 4.5 Cenário Activiti 50d8e43 como exemplo para o falso positivo de fluxo dados.

Ao analisar os cenários descritos na Tabela 4.1 é possível sugerir estratégias para reduzir a quantidade de falsos positivos indicados pela análise. Uma forma de reduzir a quantidade de falsos positivos, seria detectar métodos que não fazem nenhuma modificação ao estado do objeto inferindo essa informação a partir nomes dados aos métodos, e considerar que métodos que seguem a convenção de *getters* (tem prefixo "*get*" ou "*is*") não modificam os campos do

objeto. Outra estratégia para aumentar a precisão da análise seria seguir com a estratégia pessimista, mas usando uma abordagem interprocedural de profundidade limitada em que a análise funcionaria como uma análise interprocedural tradicional, mas que após atingisse uma certa profundidade no grafo de chamadas, utilizasse a estratégia pessimista da análise proposta.

4.2.1.1 Comparação com análises de fluxo de dados tradicionais

Para comparação da análise proposta com as implementações tradicionais de análise de fluxo de dados, foi usado um subconjunto dos cenários com 60 casos. Os casos que foram descartados do conjunto inicial foram os casos em que ocorreu algum erro de execução para as implementações tradicionais ou no caso da análise interprocedural, os casos em que o tempo de execução excedeu o máximo estipulado de 120 segundos.

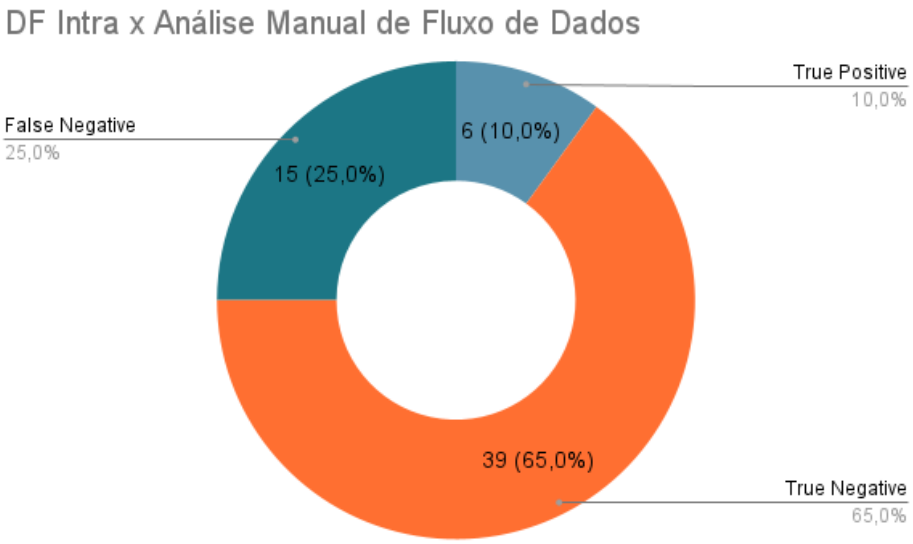

Figura 4.6 Resultados da execução da análise tradicional intraprocedural para detecção de fluxos de dados no conjunto de dados reduzido

Os resultados obtidos para as análises avaliadas no conjunto de dados reduzido, estão ilustrados nas Figuras 4.6, 4.7 e 4.8. Como esperado, a análise interprocedural detecta mais casos positivos do que a análise intraprocedural tradicional (16,7% invés de 10%). A análise proposta detecta mais casos positivos que as análises tradicionais, apresentando uma taxa de falsos negativos de apenas 1,7%, enquanto as análises intraprocedural e interprocedural apresentam taxas de respectivamente, 25% e 18,3%, o que indica que a análise proposta "perde" consideravelmente menos casos. No entanto, quando se tratam de falsos positivos, ambas as análises tradicionais não apresentam nenhum, enquanto a análise proposta apresenta uma taxa de 20% de falsos positivos.

Os cenários que a análise proposta detecta que as análises tradicionais não detectam são: no caso da análise intraprocedural, os casos de fluxos de dados em que acontece como consequência da execução de métodos pelo método de entrada e no caso da análise interprocedu-

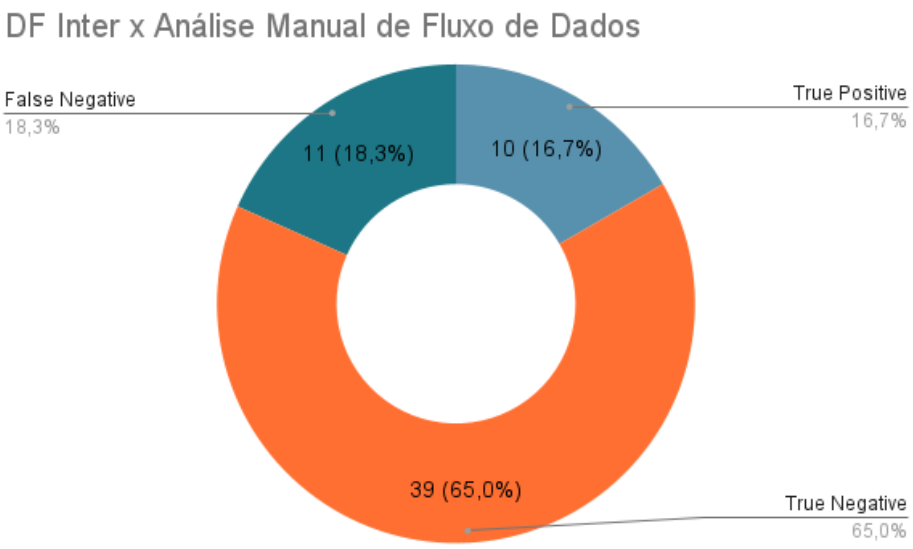

Figura 4.7 Resultados da execução da análise tradicional interprocedural para detecção de fluxos de dados no conjunto de dados reduzido

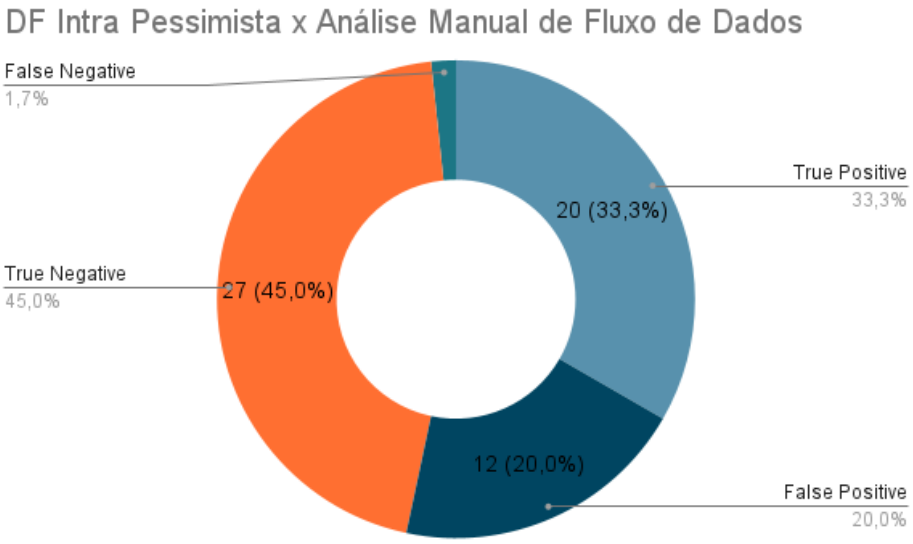

Figura 4.8 Resultados da execução da análise pessimista para detecção de fluxos de dados no conjunto de dados reduzido

ral cenários em que os métodos chamados podem ter várias implementações por mecanismos como herança e interfaces em que a análise não consegue inferir corretamente a implementação correta e em cenários em que o fluxo de dados é complexo e está “escondido” por uma longa cadeia de chamada de métodos.

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

4.2.

Para a métrica de *precision*, ambas as análises tradicionais tem o valor máximo de 1.0, o que indica que as análises só indicam casos como positivo quando eles realmente são, isso acontece pois ambos análises tradicionais não indicam falsos positivoss. Já a análise proposta tem um *precision* de 0.62, o que indica que a maioria 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 *recall*, ambas as análises tradicionais tem valores abaixo de 0.5, o que indica que mais da metade dos casos positivos não são detectados por elas. Já a análise proposta tem um valor de 0.95 para a métrica de *recall*, o que indica que a análise detecta a grande maioria dos casos positivos, fazendo com que ela seja mais confiável para detectar fluxos de dados.

A análise interprocedural tem uma acurácia melhor por ser uma análise menos sensível e detectar mais casos positivos que a análise intraprocedural, no entanto, como é indicada pelas outras métricas, ela "perde" muitos casos positivos. A análise proposta tem uma acurácia um pouco menor que a interprocedural pela sua grande quantidade de falsos positivos, mas supera a análise intraprocedural.

	Intraprocedural	Interprocedural	Pessimista
Precision	1.0	1.0	0.63
Recall	0.29	0.48	0.95
Acurácia	0.75	0.82	0.78

Tabela 4.2 Métricas para a detecção de fluxo de dados para as análises testadas

Ao analisar o tempo de execução, as análises intraprocedural e pessimista obtiveram performances equivalentes, com o tempo de execução se limitando a poucos segundos para cada cenário. Já a análise interprocedural, teve um desempenho pior, demorando minutos para alguns cenários.

Por fim, podemos concluir que a análise proposta é capaz de detectar mais casos positivos para fluxo de dados que as implementações das duas análises tradicionais (33,3% invés de 10% e 16,7%) com o desempenho semelhante ao de uma análise intraprocedural mas em contrapartida ela indica uma quantidade maior de falsos positivos (20%) enquanto as análises tradicionais não indicam nenhum, isso acontece pois as análises tradicionais avaliadas são mais conservadoras, apenas indicando um fluxo de dados quando ele realmente ocorre, sem fazer assunções, o que faz com que as análises não errem indicando positivos que não são, mas que errem perdendo fluxos mais difíceis de detectar.

4.2.2 Interferência Localmente Observável

Os resultados ilustrados na Figura 4.9 foram obtidos comparando os resultados obtidos pela análise proposta com o *Ground Truth* de interferência, essa comparação tem como objetivo avaliar o algoritmo para detecção de interferências entre as contribuições das versões integradas, e portanto, para detecção de conflitos semânticos.

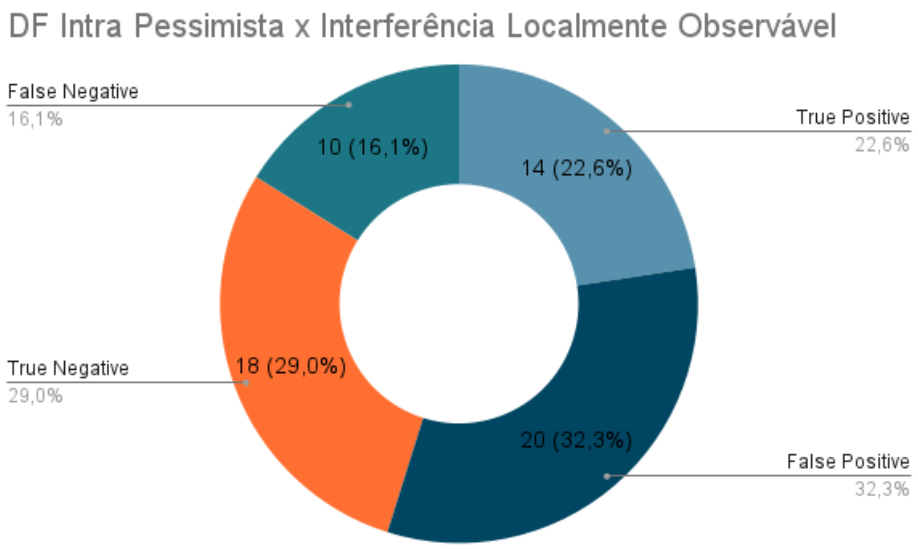

Figura 4.9 Resultados da análise pessimista para detecção de interferência localmente observável

A análise proposta se mostrou capaz de detectar cenários com interferência, com uma taxa de casos positivos detectados de 22,6%, mas também não detectou grande parte dos casos positivos, com um taxa de falsos negativos de 16,1%, o que demonstra que a apenas a análise proposta não é suficiente para a detecção de conflitos semânticos. Além disso, a análise teve 32,3% dos cenários classificados incorretamente como positivos.

Os falsos negativos para a análise proposta são também casos em que a análise manual indicou que não havia fluxo de dados e portanto são casos em que a análise não se propõe a detectar. A maioria desses cenários, são casos que poderiam ser detectados por outras simplificações das análises propostas em outros trabalhos [5][4][2]. Um exemplo seria a análise de *Overriding Assignments*, que busca detectar casos em que as mudanças introduzidas por um desenvolvedor sobrescrevem atribuições das mudanças introduzida pelo outro. Outro exemplo seria a análise de *Control Dependence*, que busca detectar interferências das mudanças de um desenvolvedor no fluxo de controle das mudanças do outro. Um sumário com a quantidade de cenários detectáveis por análises já propostas pode ser encontrado na Tabela 4.3.

Descrição do Cenário	Quantidade
Detectável por uma análise de Overriding Assignments	5
Detectável por uma análise de Control Dependence	2
Não identificado	3

Tabela 4.3 Descrições com quantidade dos cenários que a análise indicou falso positivo para interferência.

Os falsos positivos para interferência localmente observável, podem ser divididos em duas classes: casos que também são falsos positivos para fluxo de dados e casos em que as mudanças de um ou ambos desenvolvedores foram apenas refatorações, nesse caso o fluxo de dados já

existia e portanto não representa interferência entre as mudanças. Um sumário das causas para falsos positivos pode ser encontrado na Tabela 4.3.

Descrição do Cenário	Quantidade
Falsos Positivos para fluxo de dados	11
Casos em que a mudança que causou o fluxo foi uma refatoração	9

Tabela 4.4 Descrições dos cenários que a análise indicou falso positivo para interferência.

Ao analisar os casos descritos, na Tabela 4.4 podemos sugerir formas de reduzir a quantidade de falsos positivos para interferência. Para o caso dos falsos positivos para fluxo de dados, aplicar as sugestões para redução de falsos positivos para fluxo de dados poderiam também reduzir a quantidade de falsos positivos para interferência. Já para o caso em que as mudanças que causaram o fluxos de dados eram refatorações, seria possível usar ferramentas que detectam refatorações para ignorar essas modificações ao executar a análise.

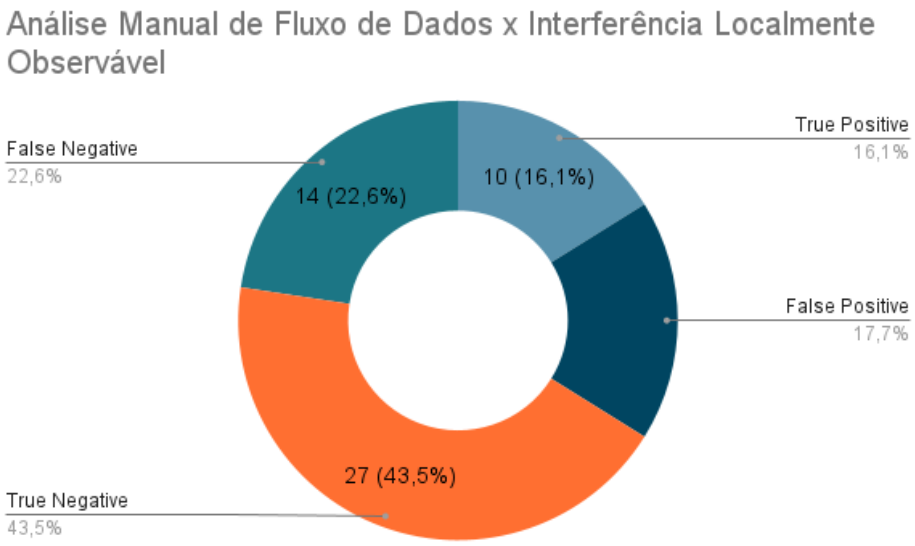

Figura 4.10 Resultados do *Ground truth* de fluxo de dados para detecção de interferência localmente observável

A igura 4.10 ilustra os resultados do *Ground Truth* de fluxo de dados para detecção de cenários com interferência. Esses resultados demonstram a capacidade de uma análise de fluxo de dados "perfeita" para detecção de interferências, e podem ser úteis para avaliar como a análise proposta se aproxima dela para esse problema.

A análise proposta detecta mais casos positivos do que a análise manual de fluxo de dados, o que pode indicar que a análise também é capaz de detectar alguns cenários de interferência em que não há fluxo de dados. Alguns dos falsos positivos para fluxo de dados que apresentam interferência, são cenários que seriam detectados por uma análise de *Overriding Assignments*, isso acontece por causa do comportamento pessimista da análise que também pode funcionar

para alguns casos de *Overriding Assignments* interprocedurais. A análise proposta também apresenta uma quantidade grande de falsos positivos a mais que a análise manual, esses cenários são os mesmos que apresentam falsos positivos para fluxo de dados.

Avaliando as métricas apresentadas na Tabela 4.5, podemos verificar que a análise proposta e o *Ground Truth* de fluxo de dados tem valores próximos para as métricas avaliadas, o que pode indicar que a análise proposta é tão boa quanto uma análise de fluxo de dados "perfeita" para detecção de cenários com interferência.

A métrica de *precision*, para o *Ground Truth* de fluxo de dados teve um resultado um pouco melhor. Isso indica que a análise proposta tem uma proporção um pouco maior de casos classificados como positivos incorretamente. Para a métrica de *recall*, a análise proposta tem um valor superior ao do *Ground Truth* de fluxo de dados, o que indica que a análise proposta detecta mais casos positivos. Isso ocorre por causa de sua natureza pessimista, que faz com que ela detecte outros tipos de interferência detectáveis por outras análises como a de *Overriding Assignments*.

	Manual	Pessimista
Precision	0.48	0.41
Recall	0.41	0.58
Acurácia	0.60	0.52

Tabela 4.5 Métricas para a detecção de interferência para a análise proposta e a análise manual

Em geral, podemos concluir que a análise proposta é capaz de 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 integração semânticos. No entanto, existem outros tipos de cenário de interferência que análises de fluxo de dados não são capazes de detectar, o que indica que somente a análise proposta não é suficiente para 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 fluxo de dados podem indicar também falsos positivos para a detecção de interferência, e no caso da análise proposta esse número é ainda maior. Nesse sentido, uma ferramenta utilizando ela pode ser muito sensível e apresentar muitos "alarmes falsos". No entanto, a análise poderia ser refinada de forma a reduzir essa quantidade de falsos positivos.

CAPÍTULO 5

Conclusão

A análise proposta se mostrou capaz de detectar cenários com fluxo de dados e com interferência entre as contribuições, no entanto, por sua natureza pessimista teve uma quantidade considerável de falsos positivos. Para fluxo de dados, esses falsos positivos estão associados com a natureza pessimista da análise, e acontecem quando as assunções que são base da análise estão incorretas. Para detecção de interferências, alguns dos falsos positivos são os mesmos dos falsos positivos para fluxo de dados e outros são casos em que as mudanças que causariam os fluxos são refatorações, o que indica que o comportamento não mudou e portanto não existe interferência.

Quando comparada com as análises tradicionais, a análise proposta teve um taxa maior de casos positivos detectados porém também com um taxa de falsos positivos consideravelmente maior. Quanto ao custo de recursos computacionais e tempo, a análise proposta teve um desempenho semelhante à análise intraprocedural tradicional executando em poucos segundos, enquanto a análise interprocedural teve um desempenho pior tomando vários minutos para executar em alguns cenários.

Quanto a capacidade de detectar cenários com interferência entre as modificações, apesar da análise proposta ser capaz de detectar 58,3% dos casos positivos, ela também perdeu uma quantidade considerável de casos (42,7% dos casos positivos), o que indica que ela não é suficiente para detectar cenários com interferência de forma confiável. Foi demonstrado que grande parte desses cenários são casos em que não foi encontrado fluxo de dados pela análise manual, o que pode indicar que esses cenários não são detectáveis por uma análise de fluxo de dados, foi demonstrado também que grande parte desses cenários são detectáveis por outras análises e portanto a análise proposta poderia ser combinada com outras análises para compor uma ferramenta mais robusta para detecção de conflitos de integração semânticos.

Em seguida, iremos descrever trabalhos relacionados e como eles se comparam com este trabalho. Iremos também descrever possíveis trabalhos futuros na área, explorando as limitações deste trabalho

5.1 Trabalhos Relacionados

Horwitz *et al* [5] propuseram uma definição para o problema de interferência em integração de código e um algoritmo para a sua detecção. Esse algoritmo utilizava um grafo para detectar fluxos de controle e dados entre as contribuições das versões a serem integradas. No entanto, esse algoritmo não foi implementado e tinha limitações: assumia uma linguagem de programação simples em que a expressões apenas tem escalares, variáveis e constantes e que só existem ins-

truções de atribuição, condicionais e o loop de repetição *while*. Em contraste com esse trabalho, que propõe uma análise e a implementa para a linguagem de programação *Java* a avaliando em projetos *open-source* reais.

De Barros [4] também propôs a utilização de análises de fluxo de dados e de controle para detecção interferências em cenários de integração reais. 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. No entanto, 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. Este trabalho busca implementar uma aproximação da análise proposta por Roberto [4] de forma que tenha resultados semelhantes, mas com uma implementação menos custosa, que possa ser aplicada em bases de código maiores.

De Souza *et al* [7] propuseram uma abordagem baseada em geração de testes automatizados para detecção de interferência em cenários de integração. A abordagem foi capaz de detectar 4 cenários com conflito de 40 cenários testados e não apresentou nenhum falso positivo, o que pode indicar que a abordagem proposta seria um ferramenta útil para detecção de conflitos semânticos. Esse trabalho 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 positivos.

Pesquisadores investigaram outras maneiras de detectar ou prevenir conflitos. Sarma *et al* [6] apresentaram *Palantir*, uma ferramenta que busca reduzir a ocorrência de conflitos notificando os desenvolvedores sobre mudanças paralelas no mesmo artefato. Brun *et al* [3] propuseram incorporar uma análise especulativa para detecção e prevenção de conflitos. Para detectar conflitos semânticos, analisaram três projetos *Java* e usaram os testes do projeto, que muitas vezes não são suficientes para detectar interferências.

5.2 Trabalhos Futuros

Como demonstrado no Capítulo 4, a análise proposta indica uma grande quantidade de falsos positivos tanto para fluxo de dados quanto para interferência. Para trabalhos futuros é importante reduzir o número de falsos positivos. Para os reduzir falsos positivos para existência de fluxo de dados, foram sugeridas estratégias como: inferir se os executados método modificam os campos do objeto utilizando as convenções de nomenclatura *Java*, em que métodos que tem prefixos como “is” ou “get” são métodos de apenas leitura ou transformar a análise proposta em uma análise interprocedural de profundidade limitada para aumentar a precisão da análise mantendo a boa performance. É possível reduzir os falsos positivos para interferência reduzindo os falsos positivos para fluxo de dados, mas também existem os casos em que o falso positivo é causado por mudanças são refatorações, nesse caso é possível utilizar uma ferramenta de detecção de refatorações para desconsiderar essas mudanças para a análise de fluxo de dados.

Houve também uma alta taxa de falsos negativos para interferência, mas esses foram os cenários que também não foram detectados pela análise manual de fluxo de dados, o que indica que são casos que não poderiam ser detectados por uma análise de fluxo de dados. Nesse sentido, seria interessante explorar outras análises que poderiam ser utilizadas em conjunto com

a análise proposta para compor uma ferramenta robusta para detecção de conflitos de integração semânticos. Alguns dos cenários com interferência não detectados poderiam ser detectados por análises já propostas como a de *Overriding Assignments* e a de *Control Dependence*. Nesse sentido, seria interessante avaliar a capacidade dessas análises e de um ferramenta que compõe esses tipos de análises para detecção de casos com interferência.

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 modificação na análise proposta.

Outro possível trabalho futuro seria explorar estratégias de marcação dos campos dos objetos alvos alternativas à descrita na Seção 3.3 e comparar com os resultados obtidos com a estratégia atual. Uma das alternativas seria checar a implementações dos objetos e listar os campos para marca-los e outra alternativa seria marcar todo o objeto alvo invés do seus campos ao encontrar um chamada de método.

Referências Bibliográficas

[1] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Commun. ACM*, 19:137–147, 1976.

[2] David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4:3–35, 1995.

[3] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early detection of collaboration conflicts and risks. 39(10):1358–1375, oct 2013.

[4] Roberto de Barros Filho. Using information flow to estimate interference between same-method contributions. 2017.

[5] Susan Horwitz, Jan Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11:345–387, 1989.

[6] Anita Sarma, David F. Redmiles, and André van der Hoek. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, 38(4):889–908, 2012.

[7] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. Detecting semantic conflicts via automated behavior change detection. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 174–184, 2020.