Semantic conflict detection with overriding assignment analysis

Matheus Barbosa
Centro de Informática
Universidade Federal de Pernambuco
Brazil
mbo2@cin.ufpe.br

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco
Brazil
phmb@cin.ufpe.br

Rodrigo Bonifacio
Universidade de Brasilia
Brazil
rbonifacio@unb.br

Galileu Santos
Centro de Informática
Universidade Federal de Pernambuco
Brazil
gsj@cin.ufpe.br

RESUMO

Developers typically work collaboratively and often need to embed their code into a major version of the system. This process can cause *merge* conflicts, affecting team productivity. Some of these conflicts require understanding *software* behavior (semantic conflicts) and current version control tools are not able to detect that. So here we explore how such conflicts could be automatically detected using static analysis of the integrated code. We propose and implement an assignment overriding analysis, which aims to detect interference between changes introduced by two different developers, where write paths, without intermediate assignments, to a common target indicate interference. To evaluate the implementations of the proposed analysis, a set of 78 code integration scenarios was used. The results show that the proposed analysis is able to detect scenarios with assignment overriding and with locally observable interference between the contributions.

ACM Reference Format:
Matheus Barbosa, Paulo Borba, Rodrigo Bonifacio, and Galileu Santos. 2022. Semantic conflict detection with overriding assignment analysis. In *36th Brazilian Symposium on Software Engineering*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3555228.3555242

1 INTRODUÇÃO

O processo de desenvolvimento de *software* atual, exceto em casos especiais, é feito de forma colaborativa. Em um ambiente colaborativo, é comum que vários desenvolvedores estejam trabalhando em seus ramos individuais (*branches*) simultaneamente, e com frequência incorporem seus códigos a uma versão principal do sistema. 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 [7], afetando a produtividade da equipe [16].

Alguns desses conflitos requerem a compreensão do comportamento do *software* (conflitos semânticos), e não podem ser detectados por ferramentas atuais de *Merge* como o diff3 [21], podendo levar à introdução de bugs no código, influenciando negativamente na qualidade do produto final. Horwitz et al. [18] especificaram formalmente os conflitos semânticos: duas contribuições advindas de versões *Left* e *Right* para um programa *Base*, originam um conflito semântico se as especificações que as versões se propõem a cumprir em isolado não são satisfeitas na versão integrada *Merge*.[1] Na prática, não é possível checar a existência dos conflitos semânticos, pois não temos as especificações e não sabemos a intenção real dos desenvolvedores.

Algumas abordagens diferentes já foram utilizadas, como, por exemplo, geração de testes [28]. Porém, essas tendem a apresentar um número alto de falsos negativos, pois verificam se há interferência a partir de valores específicos na execução do código testado.

Também já foram propostos algoritmos de análise estática para detecção de conflitos semânticos [5, 14, 18], mas esses são baseados em grafos complexos (*System Dependence Graphs* [18]), 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 [14]. 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.

Dessa forma, neste trabalho implementamos e avaliamos um tipo específico de análise estática para detectar interferências[2] entre as alterações introduzidas por dois desenvolvedores diferentes, através de uma análise de substituição de atribuição (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 (adições e modificações) feitas pelo outro ramo, sem operação de escrita da base entre eles.

XXXVI Simpósio Brasileiro de Engenharia de Software, Uberlândia, MG, 3 a 7 de outubro

ACM ISBN 978-1-4503-9735-3/22/10…$15.00
https://doi.org/10.1145/3555228.3555242

[1] Versões envolvidas em um *three-way merge*, onde *Base* é o ancestral comum mais recente às duas versões *Left* e *Right*

[2] Para detectarmos conflitos, precisaríamos saber a intenção dos desenvolvedores, pois assumimos um conflito como sendo 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.

Para melhor entender o compromisso entre acurácia e eficiência de OA, implementamos uma versão *intraprocedural*, que desconsidera chamadas de método no código analisado e outra *interprocedural*, que acessa e analisa o corpo das chamadas de método no código analisado.

As implementações da análise proposta (*intraprocedural* e *interprocedural*) foram avaliadas usando um conjunto de 78 cenários de integração de código. Esses cenários foram extraídos de projetos *open-source* Java.

Para cada um dos cenários utilizados foi realizada uma validação manual estruturada para definir o *ground truth* de 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. Por fim, também foram coletadas métricas de precisão, revocação e acurácia usadas para comparar as implementações *intraprocedural* e *interprocedural*. Adicionalmente, também realizamos uma comparação dos resultados com um *ground truth* de OA, buscando identificar o quão próximo as implementações atuais estão de uma implementação ideal.

Os resultados apontam que as implementações da análise proposta se mostraram capazes de detectar cenários com interferência entre as contribuições, no entanto, teve uma quantidade considerável de falsos negativos. Isso indica que ela não é suficiente para detectar cenários com interferência de forma confiável. Portanto, a análise proposta poderia ser combinada com outras análises para compor uma ferramenta mais robusta para detecção de conflitos de integração semânticos.

2 MOTIVAÇÃO E CONCEITOS BÁSICOS

2.1 Caso Motivador

Para exemplificar melhor o conceito de conflito de integração semântico, considere o cenário de exemplo da Figura 1.[3] Conseguimos observar a classe `Text`, que contém três atributos: uma *string*, que referencia o texto representado pelos objetos dessa classe e dois atributos do tipo inteiro, que correspondem à quantidade de correções aplicadas ao texto (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 Figura 1 é exibido o arquivo resultado da integração das mudanças em amarelo (Linha 6 foi adicionada pela versão *Left*) e as mudanças em verde (Linha 8 foi adicionada pela versão *Right*). As outras linhas de código são de uma versão *Base*, o ancestral comum mais recente de ambas versões *Left* e *Right*[4]. 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 podem ser *textualmente* integradas com segurança e nenhum conflito seria reportado pelas ferramentas de *Merge* atuais.

Observando um pouco melhor as alterações de cada um dos desenvolvedores, assuma que *Left*, ao adicionar a chamada do método `countDuplicatedWhitespaces()`, pretende realizar a contagem

```
1     class Text {
2         public String text;
3         public int fixes;
4         public int comments;
5         void generateReport() {
6   +         countDuplicatedWhitespaces();
7             countComments();
8   +         countDuplicatedWords();
9         }
10    }
```

Figura 1: Caso de exemplo de conflito integração semântico

dos espaços em branco em um determinado texto e, ao final, sobrescreve o atributo `fixes` com o valor da contagem. Já *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`.

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 ao `countDuplicateWords()` altera o valor do atributo `fixes`, que também é alterado pela chamada ao `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 o número de palavras duplicadas, teremos então, um conflito semântico.[5]

Em particular, a mudança feita por *Right* inadvertidamente anula a mudança feita por *Left*. Para entender melhor porque há interferência nesse caso, considere o caso de teste ilustrado na Figura 2, escrito por *Left* para confirmar que sua tarefa foi corretamente implementada.

```
1     public void countFixesTest() throws Throwable {
2         Text t = new Text();
3         t.text = "the the   dog dog";
4         t.generateReport();
5         assertTrue(1, t.fixes);
6     }
```

Figura 2: Caso de teste que demonstra o conflito semântico na Figura 2.1

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* (sem a linha 8 Verde), o teste passará, pois existe apenas um espaço em branco no texto (entre as palavras `the` e `dog`). Já executando apenas no ramo de *Right* (sem a linha 6 amarela) o teste falhará, pois ‘‘`the the`’’ e ‘‘`dog`

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

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

[5] Mesmo que a implementação fosse 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*

dog'' somam duas palavras duplicadas. Agora pensando apenas no cenário de *merge* (Figura 1) o teste também irá falhar, 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.

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.

Casos como esse, são muitas vezes difíceis e caros de se detectar e resolver. Na verdade, exceto em projetos que adotem boas práticas de escrita de código, 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 ainda são esperados. Se a detecção não for imediata após integração, pode ser ainda mais difícil corrigi-los, pois a resolução envolve a reconciliação da incompatibilidade semântica comportamental [28]. 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 os limites da abstração estabelecidos pelas declarações dos métodos chamados em `generateReport()`.

Para evitar esses problemas mencionados, bastaria executar a análise de substituição de atribuição proposta. No caso específico, nossa implementação detecta essa interferência, pois a análise de OA busca por alterações feitas por desenvolvedores diferentes, e que durante a execução, escrevem no mesmo componente do estado (`fixes`), sem atribuições intermediárias. De fato, a atribuição feita por `countDuplicateWhitespace()` em `fixes` é sobrescrita pela atribuição feita por `countDuplicatedWords()` e `countComments()` escreve em `comments`, não interferindo em `fixes`.

```
right no método Text.countDuplicatedWords na linha 8,
    atribui o valor 2 a variável fixes e interfere em left
    no método Text.countDuplicatedWhitespaces na linha 6
    que atribuiu o valor 1 para variável fixes.
```

Figura 3: Exemplo simplificado de um alerta reportado pela análise proposta

Na Figura 3, temos um exemplo simplificado de um alerta reportado pela análise proposta.[6] Os detalhes contêm a classe, o método, o número da linha e a instrução em que a substituição de atribuição aconteceu.

2.2 Conceitos Básicos

A definição de interferência localmente observável é inspirada na definição dada por Horwitz et al. [18], 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 neste trabalho, podemos observar a imagem ilustrada na Figura 4. 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 e atributos. Neste trabalho, utilizamos as interferências localmente observáveis, pois é a categoria de interferência que 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.

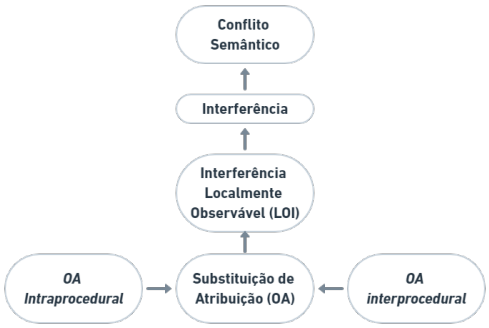

Figura 4: Representação da hierarquia da análise até o conflito semântico

É importante ressaltar que OA e LOI são independentes, ou seja, para que um exista não é necessária a existência do outro.

Um exemplo de caso em que existe OA, mas não existe LOI, pode ser observado na imagem da Figura 5. Não há LOI, pois *Right* fez uma refatoração (*extract variable* na variável `json`). No entanto, existe OA, pois os dois desenvolvedores atribuem a variável `setings`.

```
1  public void testQueryModeCommonGramsAnalysis() {
2      String json = '/org/elasticsearch/...'
3      Settings settings = Settings.settingsBuilder()
4          .loadFrom Stream(json, ...)
5          .put("path.home", create Home ())
6          .build();
7      }
```

Figura 5: Adaptação do Arquivo de *merge* do cenário f3d6309 como exemplo de OA sem LOI

Para os casos em que não existe OA, mas existe LOI, temos o exemplo ilustrado na Figura 6. Existe LOI, pois *Right* está usando uma variável (`oplogDb`) alterada por *Left*. Não existe OA, pois os desenvolvedores não sobrescrevem a mesma variável em nenhum momento.

[6] Este é um exemplo simplificado para fins didáticos. A saída da ferramente pode ser um pouco diferente.

```
...
oplogDb = oplogDb.getMongo().getDB(...);
...
oplogRefsCollection = oplogDb.getCollection(...);
...
```

Figura 6: Adaptação do arquivo de *merge* do cenário 3d4f995 como exemplo de LOI sem OA

3 ANÁLISE DE SUBSTITUIÇÃO DE ATRIBUIÇÃO (OA)

A análise proposta, essencialmente, verifica se a execução das mudanças feitas por um desenvolvedor (digamos, *Left*), pode sobrescrever 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[7], que também estão associados a uma operação de escrita envolvida nas alterações (acréscimos e modificações) feitas por outro ramo e não há operação de escrita entre eles.

Como vimos na seção anterior, o exemplo da Figura 1 contém uma interferência. Como essa interferência não foi planejada, podemos afirmar que se trata de um conflito semântico. Agora, imagine que a chamada ao método `countComments()` também sobrescrevesse a variável `fixes`. Com isso, não existiria interferência entre *Right* e *Left*. Resumidamente, *Right* não estaria sobrescrevendo uma variável alterada por *Left* e sim por *Base*, o que não ocasiona conflitos semânticos.

Seguindo o mesmo raciocínio, as referências a atributos de um objeto como `o.a` e `o.b` são consideradas diferentes elementos de estado e podem ser escritas pelas ramificações sem levar à interferência. Por outro lado, a variável `o` também representa um elemento de estado diferente, mas não pode ser alterado junto com `o.a` ou `o.b` sem causar interferência. (Figura 7).

```
    void m() { // Sem Interferencia
        Object o = new Object();
+       o.a = "exemplo A";
        ...
+       o.b = "exemplo B";
    }
    void m() { // Com Interferencia
        Object o = new Object();
+       o = new Object();
        ...
+       o.a = "exemplo";
    }
```

Figura 7: Caso de exemplo com objetos

[7]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`

Algo similar 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 à interferência. No entanto, a variável `a` corresponde a um elemento de estado diferente, mas não pode ser alterado junto com `a[0]` e `a[1]` sem causar interferência.

Outro detalhe importante é com relação à acurácia com que a análise detecta interferência. A análise proposta é considerada conservadora. Para exemplificar isso, podemos imaginar uma versão da Figura 1, em que a atribuição do desenvolvedor *Right* está no escopo 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 falsa, Dessa forma, a análise irá verificar as duas possibilidades, o que resultaria em interferência para esse exemplo, mesmo que a condição do `if` nunca viesse a ser *true* em execuções do sistema.

3.1 Especificação do Algoritmo

Um pseudocódigo simplificado do algoritmo pode ser observado na imagem a seguir.

Algorithm 1: Algoritmo de substituição de atribuição

input : Um método de entrada *m*
output: Uma lista de interferências

```
1  traverse (m):
2      foreach cmd ∈ m do
3          if isTagged (cmd) then
4              if isAssigment (cmd) then
5                  if isRight (cmd) then
6                      abstractionRight ← cmd
7                  else if isLeft (cmd) then
8                      abstractionLeft ← cmd;
                   /* verifica se existe Interferência
                   e adiciona na lista de conflitos  */
9                  checkConflict (cmd);
10             else if isMethodCall (cmd) then
11                 traverse (cmd);
12         else
               /* remove o comando das listas.      */
13             kill (cmd);
14         end
15     end
16 end
```

De modo geral, o algoritmo da análise proposta consiste em percorrer os comandos de um método de entrada.[8] Quando encontra uma atribuição adicionada por *Left* ou *Right*, guarda essa informação em um conjunto que contém os elementos do estado alterado.[9] Uma interferência é reportada sempre que o elemento de estado já existe no conjunto. Quando encontra uma atribuição de *Base*,

[8]A implementação atual considera como método de entrada o método onde ocorreu a primeira modificação feita pelo desenvolvedor *Left*

[9]Tal conjunto corresponde a uma abstração na infraestrutura de análise de programas *Soot Framework*. Outro termo comumente encontrado na literatura é *lattice*.

tira do conjunto. Para *interprocedural*, adicionalmente atravessa recursivamente os comandos do método chamado quando encontra uma chamada de método.

Neste trabalho, implementamos duas abordagens diferentes: (*intraprocedural* e *interprocedural*) para a análise de substituição de atribuição proposta.

Análise intraprocedural. A implementação *intraprocedural* é mais barata com relação aos custos computacionais, assim, não consegue detectar interferências quando ocorrem em métodos chamados a partir do método de entrada, pois desconsidera comandos de chamada de método, não alterando a abstração da análise e seguindo para o próximo comando.

Análise interprocedural. A implementação *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 próximo comando.

3.2 Implementação

As abordagens da análise proposta foram implementadas utilizando a linguagem de programação Java e a *API* do *Soot Framework*[10] para executar o algoritmo descrito anteriormente.[11]

As análises recebem como entrada o código compilado da versão integrada empacotado em um arquivo *.JAR* e um arquivo *.CSV* gerado pela ferramenta *miningframework*[12], em que cada linha representa uma linha de código modificada[13], contendo as informações do nome da classe modificada e o número da linha modificada para a versão integrada, e se essa linha foi modificada por *Left* ou *Right*.

4 AVALIAÇÃO

As seções a seguir descrevem a metodologia utilizada para compreender as implicações do uso da abordagem proposta na identificação de conflitos semânticos. Além disso, verificamos o quão perto as nossas implementações atuais da análise estão perto de uma implementação ideal de OA. Fazemos também uma breve comparação do desempenho das implementações. Por fim, apresentamos os resultados obtidos e às ameaças a validade.

4.1 Dataset

Para avaliar a capacidade da análise proposta de detectar interferência 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 contém um método ou campo de uma classe modificado em ambas versões Left e Right — isto é, os ramos integrados em uma determinada operação de merge. Esse tipo de cenário em específico foi escolhido, pois são mais suscetíveis a ocorrência de interferência. Alguns desses cenários foram aproveitados de outros trabalhos relacionados, sendo 5 de Da Silva et. al. [28], 19 de Barros Filho [14], 26 de De Souza et. al. [15] e 7 de Cavalcanti et. al. [12]. Os outros 21 cenários foram minerados a partir de uma lista de projetos *open-source* populares utilizando o *miningframework*. A lista com todos os cenários e projetos utilizados pode ser encontrada no Apêndice online [1].

O processo de coleta dos dados utilizados é ilustrado pela Figura 8. A ferramenta de mineração recebe como entrada um arquivo 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*[14] dos projetos passados como entrada.

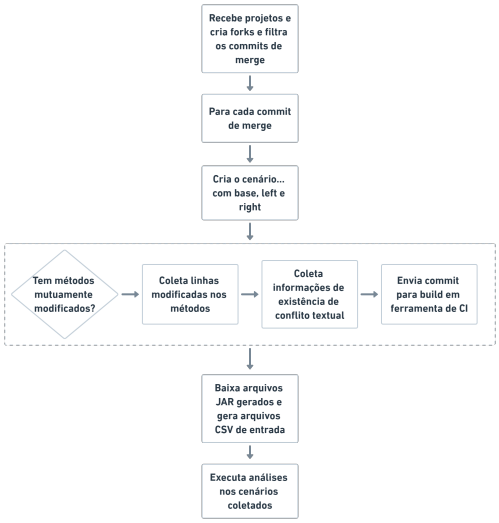

Figura 8: Fluxo de coleta dos dados sobre os cenários de integração

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: 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. 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*[15], que usa os sistemas de *build Maven* ou *Gradle* para gerar os arquivos JAR.

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

[11] As implementações da análise proposta pode ser encontrada no repositório https://github.com/spgroup/conflict-static-analysis. Este projeto, disponível no *Github*, reúne as implementações utilizadas para esse e outros trabalhos e visa implementar uma biblioteca de análises estáticas para detectar conflitos de merge semântico.

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

[13] 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.

[14] 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.

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

O Processo supracitado foi utilizado na maioria dos casos, no entanto, em alguns desses cenários, o mapeamento das linhas do código-fonte extraídos a partir do *bytecode*, pode ser impreciso, por isso precisaram passar por adaptações, assim dizemos que eles são realistas. Esse problema ocorre principalmente nos casos em que há chamadas de método encadeadas que ocupam várias linhas. 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*. Para evitar esses comandos que ocupam várias linhas, quebramos o mesmo em vários comandos que ocupam apenas uma linha, através de simples refatorações de extrações de variáveis no código-fonte do projeto 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.

Para a análise de cenários cujo método de entrada tinha pelo menos um argumento com tipo não primitivo, adicionamos um método que instancia os parâmetros e chama o método de entrada. Essa estratégia foi aplicada em quatro cenários e usada apenas pela implementação *intraprocedural* e foi necessária, pois a análise não reconhece parâmetros de tipos não primitivos que não foram instanciados previamente no código.

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 de entrada para a execução da análise de OA.

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[16]. Um resumo da amostra utilizada pode ser observada no Apêndice Online [1].

4.2 Metodologia

A Figura 9 apresenta o fluxo de trabalho seguido no estudo empírico.

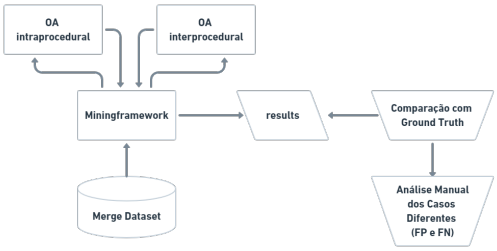

Figura 9: Fluxo da execução da avaliação das análises.

Com os dados necessários para os 78 cenários, o miningframework foi utilizado para executar as implementações *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.

Todos os 78 cenários do conjunto de dados também foram analisados manualmente, utilizando o protocolo disponível no Apêndice Online [1], de modo a verificar a existência de interferências entre as modificações de *Left* e *Right*, e assim estabelecer um *Ground Truth*. Para definição do *Ground Truth*, 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 para juntos chegarem à decisão.

Buscando verificar a capacidade da análise em detectar interferência entre as contribuições ou até mesmo compor uma ferramenta para detecção de conflitos de integração semânticos mais robusta, os resultados apresentados pelas implementações da análise proposta foram comparados com o *Ground Truth* em cada cenário. Adicionalmente, para verificarmos o quão próximo a nossa implementação atual está de uma implementação ideal de OA, comparamos o resultado da ferramenta com *Ground Truth* exclusivo de OA.

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 não produziu o resultado esperado.[17] Isso não é o objetivo principal deste trabalho, assim como a otimização da implementação e a avaliação da métrica de tempo. Desta forma, não aprofundamos esses itens, 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.

4.3 Resultados

4.3.1 Resultados Reportados nas Execuções das Implementações da Analíse. Ao executar a abordagem *intraprocedural* da análise proposta para 78 cenários do *dataset* obtivemos 13,2% de casos reportados como verdadeiros, ou seja, a análise reportou pelo menos uma interferência nesses cenários. Em 80,9% dos casos, a análise não detectou nenhuma interferência. Ainda foram reportados 4 casos (5,9%) que não foram encontrados pela análise, isso ocorreu, pois, esses cenários tinham apenas remoções de linhas feitas pelo desenvolvedor *Right*. Outros 10 casos apresentaram erro na execução. Os casos com erro ou que não foram encontrados foram desconsiderados para fins de comparação com o *ground truth*.

Já a execução *interprocedural* 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[18] estipulado para cada cenário. Esses 10 casos, foram desconsiderados para fins de comparação com o *ground truth*. A execução *interprocedural* também reportou 15.4% dos casos como verdadeiros e 71,8% como falso.

4.3.2 OA x Análise Manual de LOI. Os resultados apresentados nessa subseção foram obtidos comparando os resultados da execução das implementações da análise proposta com o *Ground Truth*

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

[17] Feita apenas para a abordagem *interprocedural*

[18] 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*.

para interferência localmente observável. Essa comparação pretende avaliar o algoritmo para detecção de interferências entre as contribuições das versões integradas, portanto, para detecção de conflitos semânticos.

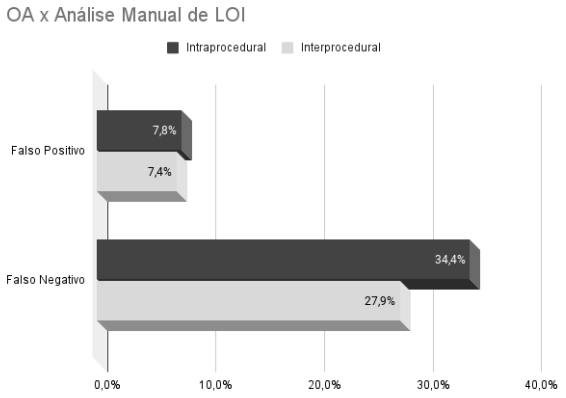

Figura 10: Resultados da análise em comparação com o *Ground Truth* de LOI.

Os resultados ilustrados na Figura 10 mostram a comparação dos 64 cenários considerados na execução *intraprocedural* com o *ground truth* de LOI. Obteve-se uma taxa alta de 34,4% de falsos negativos e 7,8% de falsos positivos. Logicamente, os outros 57,9% foram de acertos da ferramente, sendo 51,6% verdadeiros negativos e 6,3% de verdadeiros positivos.

Ainda na Figura 10, conseguimos observar também os resultados da comparação dos 68 cenários considerados na execução *interprocedural* com o *ground truth* de LOI. Obtendo-se uma taxa de 27,9% de falsos negativos e 7,4% de falsos positivos. os outros 64,7% foram de acertos da ferramente, sendo 54,4% verdadeiros negativos e 10,3% de verdadeiros positivos.

A Tabela 1 apresenta um sumário com a descrição e a quantidade de erros da análise *interprocedural* proposta.

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
Interferência não é detectável com OA	14	FN
Não existe interferência entre as contribuições	5	FP

Tabela 1: Descrições dos cenários em que a análise *interprocedural* proposta errou na detecção de LOI.

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 LOI e vice-versa.

O primeiro caso da tabela pode ser observado na imagem ilustrada na Figura 11. Para esse caso temos que *Left* e *Right* alteraram a *String* que vai ser usado no retorno da função. O *ground truth* classifica isso como OA, no entanto, ao nível de execução da implementação da análise, 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.

```
1   @Override
2   public String toString()() {
3       return "KafkaSpoutConfig{" +
4       ", keys"+ getKeyDeserializer () +
5       ", value=" + getValueDeserializer () +
6       ...
7       
8       ...
9       '}';
10  }
```

Figura 11: Cenário Storm ad2be67 no método toString() como exemplo para o falso negativo.

A partir dos resultados das execuções das implementações da análise foram calculadas as métricas representadas pela Tabela 2.

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. A quantidade de falsos positivos é esperada, pois, existem casos em que existe OA, mas *Ground truth* de LOI é falso, desta forma as implementações vão detectar OA, mas não existe interferência real. Um exemplo disso acontece um dos desenvolvedores fez apenas uma refatoração estrutural no código.

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 priori, esses números podem parecer baixos, no entanto, a principal razão para isso está em nossa amostra que contém diversos tipos de interferência, como, por exemplo, casos em que um desenvolvedor escreve em uma variável que outro desenvolvedor futuramente lê. Esses tipos diferentes de interferência não são detectados pelo algoritmo de OA que busca apenas variáveis atribuídas por dois desenvolvedores diferentes. A combinação da análise de OA com outras análises melhoraria consideravelmente os resultados de revocação. Além disso, a solução dos problemas e limitações conhecidas nas implementações atuais, colocaria os resultados de revocação em seu nível máximo.

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. De modo geral, os resultados para essa métrica são considerados bons, mas podem ser significativamente melhorados aplicando principalmente as soluções supracitadas para melhoria da revocação.

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

Tabela 2: Resultados das Métricas para a detecção de LOI

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 casos adicionais de interferência. No entanto, pelo mesmo motivo, o custo de execução *interprocedural* é maior. 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, 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 com suporte a OA pode ser muito sensível e apresentar falsos positivos e falsos negativos.

4.3.3 OA x Análise Manual de OA. Os resultados apresentados nessa subseção 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. Através dessa comparação, podemos identificar a qualidade das implementações atuais e o quão perto elas estão de uma implementação ideal de OA.

A partir dos resultados obtidos através da execução das implementações da análise foram calculadas as métricas representadas pela Tabela 3.

A análise proposta se mostrou capaz de detectar cenários com substituições de atribuições entre as contribuições, principalmente na sua abordagem *interprocedural*. No entanto, a análise reportou falsos positivos e falsos negativos. O número de falsos positivos foi baixo e isso indica que a análise reporta poucos "alarmes falsos", gerando um custo pequeno para o desenvolvedor, que não vai ser distraído facilmente por problemas que não existem de fato. 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.

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

Tabela 3: Resultados das métricas para a detecção de OA

Com isso concluímos que a análise apresentou excelentes índices, principalmente na métrica de acurácia. (0,7% para *intraprocedural* e 0,88% para *interprocedural*). Isso significa que as implementações atuais precisam de poucos ajustes para se tornarem ideias no sentido de detecção de OA.

4.3.4 Detalhes da execução. Embora o desempenho não fosse uma prioridade para nosso projeto, deixamos aqui os registros dos dados coletados relacionados ao tempo de execução da análise.

As 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 × 4. Ambas tiveram o *timeout* configurado em 240 segundos. A execução *interprocedural* foi configurada com um limite de profundidade no acesso aos métodos de 5.

Com dados de uma execução e tomando a média dos tempos medidos, a execução *intraprocedural* levou 2,32 segundos/por cenário para analisar toda a amostra, com mediana de 1,09 e desvio padrão de 2,42. Já a execução *interprocedural* teve media de 28,12 segundos/por cenário, mediana de 3,94 e desvio padrão de 68,81.

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. Utilizando o referencial dos resultados atuais, indicamos a implementação *interprocedural* pois a mesma obteve resultados melhores na detecção de interferência, mas devido ao seu custo maior, acreditamos que a mesma possa ser utilizada em rotinas que executam à noite, por exemplo. Para contextos onde o tempo é um fator limitante a implementação *intraprocedural* é a mais recomendada.

É importante ressaltar que as implementações utilizadas nesse trabalho são protótipos e não foram otimizadas visando desempenho, uma vez que o objetivo era verificar a capacidade da mesma em detectar interveniências. Com isso, queremos dizer que os resultados de desempenho podem ser substancialmente melhorados.

4.4 Ameaças a validade

Nosso estudo se restringe à ocorrência de interferência local. Portanto, é possível que nossos exemplos tenham cenários de *merge*

que as alterações 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 diferir 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 porque não há interferência.

O tipo de cenário contido no nosso dataset, com mudanças em paralelo no mesmo método ou atributo pode ser mais suscetível a ocorrência de interferência, mas isso não influencia o resultado final principal apresentado no artigo sobre a acurácia e performance de uma ferramenta baseada na análise OA, já que no dataset temos vários casos de interferência, mas também vários casos sem interferência, e nossa avaliação principal não é sobre a frequência com que interferência ocorre, mas sim se análise estática seria viável para detectar interferência. Adotamos esse critério de filtro dos cenários de merge para reduzir um pouco a dificuldade de analisar manualmente cenários reais para confirmar a ocorrência de interferência. Nenhum outro filtro sobre o conteúdo dos métodos e atributos dos cenários foi aplicado.

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 apresente erros. Alguns problemas conhecidos já foram mencionados, mas podem existir outros problemas que ainda não foram detectados. Alguns dos cenários do *dataset* utilizado precisaram de adaptações realistas. Essas alterações foram realizadas no 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

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

Vários pesquisadores já investigaram sobre os conflitos de *merge* e como eles afetam os desenvolvedores e a produtividade de forma geral [7, 16, 21, 22, 25, 27, 31, 35]. Em contraste, aqui focamos na detecção de conflitos semânticos, que normalmente são mais difíceis de se detectar e resolver. Não conhecemos trabalhos que analisam o impacto de conflitos semânticos em produtividade. No entanto, ferramentas como a que discutimos aqui são essenciais para ajudar a entender o impacto de conflitos semânticos em produtividade.

Com o intuito de progredir no processo de detecção de conflitos de *merge* de forma mais precisa e reduzir esforços da integração, foram criadas diversas ferramentas. Westfechtel [32] e Buffenbarger [9] 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* [4] e *C++* [17]. Existem também várias ferramentas de *merge* avançadas que usam a estrutura sintática dos programas integrados [2, 3, 10–13, 30, 34], mas nenhuma delas captura conflitos semânticos dinâmicos como os ilustrados aqui.

Para de fato detectar conflitos semânticos, Da Silva *et al* [28] propuseram uma abordagem baseada em geração de testes unitários automatizados como especificações parciais para detecção de interferência em cenários de integração. Com 38 cenários testados, a abordagem conseguiu detectar 4 casos com conflitos (verdadeiros positivos), 11 casos falsos negativos (28,95%) e nenhum falso positivo. Já em nosso trabalho, utilizamos uma abordagem diferente, baseada em análise estática. Os resultados para os 78 cenários da amostra utilizada apontam um percentual maior de falsos positivos (7,8% para *intraprocedural* e 7,4% para *interprocedural*) e falsos negativos (34,4% para *intraprocedural*). A nossa abordagem foi superior na comparação de falsos negativos *interprocedural* (27,9%). Contudo, argumentamos que a análise proposta neste trabalho é somente uma parte de uma solução maior utilizando analises estáticas, enquanto o trabalho de Da Silva é uma solução completa. A amostra de 38 cenários utilizada no trabalho de testes é um subconjunto da amostra utilizada em nosso trabalho. Nesse sentido, analisamos os cenários detectados por ambas as ferramentas e constatamos serem cenários diferentes. Dessa forma, entendemos que uma combinação das duas abordagens poderia ser usada em trabalhos futuros, buscando aumentar a quantidade de cenários detectados e reduzir o número de erros, principalmente os falsos negativos. Além disso, existem outros trabalhos importantes que seguem a linha de ferramentas baseadas em testes [8, 23, 24], mas estes são baseados em testes do projeto, que muitas vezes não são suficientes para detectar conflitos.

Por fim, ferramentas baseadas em estratégias de análise estática também foram propostas [6, 18–20, 26, 29, 33]. Barros Filho [14] também propôs a utilização de análises com o objetivo principal de entender se o Information Flow Control (IFC) pode ser utilizada para indicar a presença de conflitos semânticos dinâmicos entre as contribuições dos desenvolvedores em cenários de merge. Essa análise se mostrou capaz de detectar casos de interferência, porém com uma taxa de 57.14% 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 cerca de 87% menor.

6 CONCLUSÕES

Neste estudo apresentamos uma proposta de análise estática de substituição de atribuições entre contribuições de dois desenvolvedores, de modo a detectar interferências. Foi implementado duas abordagens para a análise proposta, sendo uma *intraprocedural* e outra *interprocedural*.

Apesar de haver outros tipos de análise que podem detectar interferência, focamos em OA por ela ter mais chances de ser efetiva, já que a nossa expectativa era de que normalmente quando há

esse tipo de sobreposição há também interferência. Além disso, existe a necessidade de entender com precisão os pontos fortes e fracos de cada análise individualmente, de forma a poder sugerir uma combinação de análises que seja mais efetiva para detectar interferência.

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. Portanto, a análise proposta poderia ser combinada com outras para compor uma ferramenta mais robusta para detecção de conflitos de integração semânticos.

Como trabalho futuro, pretende-se implementar a resolução de algumas limitações conhecidas para a análise, além de realizar combinações com outras ferramentas de detecção de conflitos semânticos relacionadas.

DISPONIBILIDADE DE ARTEFATOS

Disponibilizamos o código-fonte das implementações da análise, o *dataset* utilizado na avaliação, uma planilha com os resultados das execuções, um documento com o protocolo utilizado para definição de *Ground Truth* e instruções de replicação do estudo [1].

AGRADECIMENTOS

Agradecemos Rafael Alves, Léuson Da Silva e Vinícius dos Santos pelo suporte na criação do nosso conjunto de dados. Agradecemos também ao INES (Instituto Nacional de Engenharia de Software), FACEPE (IBPG-28-1.3/20 e IBPG-567-1.03/22) e CNPq (309235/2021-9).

REFERÊNCIAS

[1] 2022. Online Appendix. available at: https://spgroup.github.io/papers/semantic-conflicts-SBES2022.html. https://spgroup.github.io/papers/overriding-assignment-sbes2022.html

[2] Sven Apel, Olaf Leßenich, and Christian Lengauer. 2012. Structured merge with auto-tuning: balancing precision and performance. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 120–129. https://doi.org/10.1145/2351676.2351694

[3] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/2025113.2025141

[4] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2007. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. *Automated Software Engg.* 14, 1 (mar 2007), 3–36. https://doi.org/10.1007/s10515-006-0002-0

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

[6] David Binkley, Susan Horwitz, and Thomas Reps. 1995. Program Integration for Languages with Procedure Calls. *ACM Trans. Softw. Eng. Methodol.* 4, 1 (jan 1995), 3–35. https://doi.org/10.1145/201055.201056

[7] Christian Bird and Thomas Zimmermann. 2012. Assessing the value of branches with what-if analysis. In *SIGSOFT FSE*.

[8] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *IEEE Trans. Softw. Eng.* 39, 10 (oct 2013), 1358–1375. https://doi.org/10.1109/TSE.2013.28

[9] Jim Buffenbarger. 1993. Syntactic software merging. In *Software Configuration Management. Springer*. 153–172.

[10] Guilherme Cavalcanti, Paola Accioly, and Paulo Borba. 2015. Assessing Semistructured Merge in Version Control Systems: A Replicated Experiment. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–10. https://doi.org/10.1109/ESEM.2015.7321191

[11] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (oct 2017), 27 pages. https://doi.org/10.1145/3133883

[12] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. 2019. The Impact of Structure on Software Merging: Semistructured versus Structured Merge. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1002–1013. https://doi.org/10.1109/ASE.2019.00097

[13] Jônatas Clementino, Paulo Borba, and Guilherme Cavalcanti. 2021. *Textual Merge Based on Language-Specific Syntactic Separators*. Association for Computing Machinery, New York, NY, USA, 243–252. https://doi.org/10.1145/3474624.3474646

[14] Roberto de Barros Filho. 2017. Using information flow to estimate interference between same-method contributions. *Master's thesis, Universidade Federal de Pernambuco* (2017).

[15] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. 2003. "Breaking the Code", Moving between Private and Public Work in Collaborative Software Development. In *Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work* (Sanibel Island, Florida, USA) *(GROUP '03)*. Association for Computing Machinery, New York, NY, USA, 105–114. https://doi.org/10.1145/958160.958177

[16] H. Christian Estler, Martin Nordio, Carlo A. Furia, and Bertrand Meyer. 2014. Awareness and Merge Conflicts in Distributed Software Development. In *2014 IEEE 9th International Conference on Global Software Engineering*. 26–35. https://doi.org/10.1109/ICGSE.2014.17

[17] Judith E. Grass. 1992. Cdiff: A Syntax Directed Differencer for C++ Programs. In *Proceedings of the USENIX C++ Conference. USENIX Association.*

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

[19] Susan Horwitz, Jan Prins, and Thomas Reps. 1989. Integrating Noninterfering Versions of Programs. *ACM Trans. Program. Lang. Syst.* 11, 3 (jul 1989), 345–387. https://doi.org/10.1145/65979.65980

[20] Jackson and Ladd. 1994. Semantic Diff: a tool for summarizing the effects of modifications. In *Proceedings 1994 International Conference on Software Maintenance*. 243–252. https://doi.org/10.1109/ICSM.1994.336770

[21] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. 2007. A Formal Investigation of Diff3. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science* (New Delhi, India) *(FSTTCS'07)*. Springer-Verlag, Berlin, Heidelberg, 485–496.

[22] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. 2017. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 467–478. https://doi.org/10.1109/ICSME.2017.53

[23] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 907–918. https://doi.org/10.1145/2568225.2568300

[24] Hung Viet Nguyen, My Huu Nguyen, Son Cuu Dang, Christian Kästner, and Tien N. Nguyen. 2015. Detecting Semantic Merge Conflicts with Variability-Aware Execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 926–929. https://doi.org/10.1145/2786805.2803208

[25] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. 2001. Parallel Changes in Large-Scale Software Development: An Observational Case Study. *ACM Trans. Softw. Eng. Methodol.* 10, 3 (jul 2001), 308–337. https://doi.org/10.1145/383876.383878

[26] Thaís Rocha, Paulo Borba, and João Pedro Santos. 2019. Using acceptance tests to predict files changed by programming tasks. *Journal of Systems and Software* 154 (2019), 176–195. https://doi.org/10.1016/j.jss.2019.04.060

[27] Anita Sarma, David F. Redmiles, and André van der Hoek. 2012. Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes. *IEEE Transactions on Software Engineering* 38, 4 (2012), 889–908. https://doi.org/10.1109/TSE.2011.64

[28] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and João Moisakis. 2020. Detecting Semantic Conflicts via Automated Behavior Change Detection. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 174–184. https://doi.org/10.1109/ICSME46990.2020.00026

[29] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified Three-Way Program Merge. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 165 (oct 2018), 29 pages. https://doi.org/10.1145/3276535

[30] Alberto Trindade Tavares, Paulo Borba, Guilherme Cavalcanti, and Sérgio Soares. 2019. Semistructured Merge in JavaScript Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1014–1025. https://doi.org/10.1109/ASE.2019.00098

[31] Gustavo Vale, Claus Hunsen, Eduardo Figueiredo, and Sven Apel. 2021. Challenges of Resolving Merge Conflicts: A Mining and Survey Study. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3130098

[32] Bernhard Westfechtel. 1991. Structure-Oriented Merging of Revisions of Software Documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management* (Trondheim, Norway) *(SCM '91)*. Association for Computing Machinery, New York, NY, USA, 68–79. https://doi.org/10.1145/111062.111071

[33] Wuu Yang, Susan Horwitz, and Thomas Reps. 1992. A Program Integration Algorithm That Accommodates Semantics-Preserving Transformations. *ACM Trans. Softw. Eng. Methodol.* 1, 3 (jul 1992), 310–354. https://doi.org/10.1145/131736.131756

[34] Fengmin Zhu, Fei He, and Qianshan Yu. 2019. Enhancing Precision of Structured Merge by Proper Tree Matching. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 286–287. https://doi.org/10.1109/ICSE-Companion.2019.00117

[35] Thomas Zimmermann. 2007. Mining Workspace Updates in CVS. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. 11–11. https://doi.org/10.1109/MSR.2007.22