

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Moura, Christian Davi Borges de.

Static Semantic Merge - Uma Ferramenta para Integração de Análise
Estática ao Processo de Merge / Christian Davi Borges de Moura. - Recife,
2022.

8 : il., tab.

Orientador(a): Paulo Borba

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de
Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado,
2022.

1. Engenharia de Software. 2. Git. 3. Análise Estática. 4. Conflitos
Semânticos. I. Borba, Paulo. (Orientação). II. Título.

000 CDD (22.ed.)

Static Semantic Merge - Uma Ferramenta para Integração de Análise Estática ao Processo de Merge

Christian Davi Borges de Moura

Orientador: Paulo Borba

Centro de Informática (CIn), Universidade Federal de Pernambuco

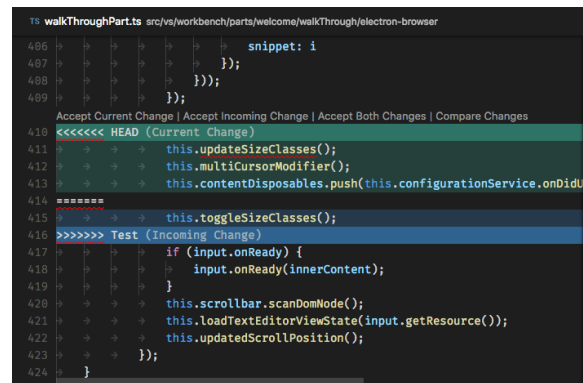
I. INTRODUÇÃO

Durante o desenvolvimento de um software, é comum que times trabalhem paralelamente em um mesmo projeto por diversos fatores, como divisão de tarefas e maior progresso no desenvolvimento [3].

Desenvolvedores trabalham em suas tarefas pessoais separadamente até o momento em que se faz necessário a junção dessas mudanças em uma versão principal. Para isso é geralmente utilizada uma ferramenta de controle de versão para a gestão desse processo, como o *Git*. A junção dessas mudanças é dada pelo comando de *git merge*, e nesse momento podem ocorrer conflitos textuais ou de ordem superior que acarretam problemas momentâneos ou futuros para os desenvolvedores [3].

Conflitos textuais ocorrem quando desenvolvedores alteram a mesma linha ou linhas consecutivas em um mesmo arquivo, ou quando um desenvolvedor edita um arquivo e outro desenvolvedor exclui esse mesmo arquivo [9], esses conflitos são reportados pelo *Git* imediatamente, fazendo com que o desenvolvedor seja obrigado a decidir (Fig. 1) quais mudanças vão ter prioridade e quais serão descartadas. Esse tipo de conflito gera problemas imediatos ao desenvolvedor, pois o faz parar o desenvolvimento para lidar com os conflitos.

No entanto podem haver conflitos de ordem maior, que não são textuais e portanto não são detectáveis pelas ferramentas de controle de versão atuais, que geram problemas futuros aos desenvolvedores, podendo levar a introdução de *bugs* no código e influenciar negativamente na qualidade do produto final. Os chamados conflitos semânticos ocorrem quando desenvolvedores fazem mudanças que não causam conflito textual mas causam



```
13 walkThroughPart.ts src/vs/workbench/parts/welcome/walkThrough/electron-browser
486 |         snippet: i
487 |         });
488 |     });
489 | });
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
416 <<<<<< HEAD (Current Change)
417 |         this.updateSizeClasses();
418 |         this.multiCursorModifier();
419 |         this.contentDisposables.push(this.configurationService.onDidU
420 |         =====
421 |         this.toggleSizeClasses();
422 | >>>>>> Test (Incoming Change)
423 |         if (input.onReady) {
424 |             input.onReady(innerContent);
425 |         }
426 |         this.scrollbar.scanDomNode();
427 |         this.loadTextEditorViewState(input.getResource());
428 |         this.updatedScrollPosition();
429 |     });
430 | }
```

Fig. 1: Exemplo de Conflito textual.

interferência não intencional entre as mudanças [4]. Uma interferência ocorre quando a integração das mudanças não preserva o comportamento individual [6] ou seja, muda o comportamento esperado das mudanças que foram integradas.

A. Detectando Conflitos Semânticos Através de Análise Estática

Existem atualmente algoritmos de análise estática para a detecção de conflitos semânticos, esses algoritmos analisam o código fonte do projeto no momento da integração (*merge*) procurando interferências entre as mudanças das duas *branches* pais.

Uma análise é estática quando a análise feita em cima de um software é feita sem executá-lo de fato, a ideia é computar aproximações do comportamento que pode vir a acontecer em tempo de execução. [8]

Alguns exemplos desses algoritmos são os de *Intraprocedural def-use*, que identifica conflitos que ocorrem quando uma variável definida na *branch A* é usada na *branch B*, ou o algoritmo de *Tainted analysis* que identifica conflitos de *def-use* transitivos,

quando uma variável definida na *branch* A é usada para definir uma variável que é usada na *branch* B, ou então um algoritmo de análise de *Overriding Assignment*, que consiste em detectar alterações (adições e modificações) na *branch* A que semanticamente envolvem uma operação de escrita em um elemento de estado que também está associado a uma alteração de B e não há interferência de base [9].

B. Conflict-Static-Analysis

O *conflict-static-analysis*¹ é um projeto que implementa os algoritmos citados acima dentre outros para analisar projetos Java. Devido o fato do projeto ter como foco a implementação das análises para experimentos científicos e ser implementado para lidar apenas com um cenário por vez, o mesmo não é facilmente acoplável ao ciclo de desenvolvimento de um software, pois é necessária uma coleta de informações prévias para a execução de cada cenário de integração, como um arquivo com a lista das mudanças a serem analisadas e o caminho para o arquivo compilado do projeto analisado. O SSM utilizará as implementações das análises do projeto *conflict-static-analysis* para procurar por interferências em um cenário de integração.

C. Mining Framework

O SSM utiliza também o *Mining Framework*², que é um *framework* para minerar e analisar repositórios *git*. O *Mining Framework* tem o foco em analisar *commits* de *merge* e dispõe de algumas funcionalidades para isto, como pré-processamento dos projetos que vão ser analisados (fazendo *fork*), filtragem dos *commits* do projeto que são de fato de *merge*, coleta de dados de cada *commit* de *merge* como *hashes* de *commits* ou a numeração das linhas modificadas em cada *parent*, interface para execução de ferramenta de análise de *commits* de *merge*, como o *conflict-static-analysis*, e pós processamento dos dados coletados, como como agregação e sumarização de dados e geração de resultados. O SSM utiliza principalmente as funcionalidades de coleta, pós processamento dos dados e a interface para executar o *conflict-static-analysis*.

¹<https://github.com/spgroup/conflict-static-analysis>

²<https://github.com/spgroup/miningframework>

```
1 class Text {
2
3     public String text;
4     public int fixes;
5     public int comments;
6
7     void generateReport() {
8         this.countDuplicatedWhitespaces();
9         this.countComments();
10        this.countDuplicatedWords();
11    }
12 }
```

Fig. 2: Merge de mudanças (*branch* A em verde e *branch* B em azul) que causam um conflito semântico.

D. Static Semantic Merge (SSM)

Com isso, esse trabalho propõe o *Static Semantic Merge (SSM)*³, uma ferramenta que pode ser acoplada ao processo de desenvolvimento de software visando detectar conflitos semânticos em cenários de *merge*, utilizando a implementação das análises do *conflict-static-analysis*. Também utilizamos o *Mining Framework* para coletar as informações de quais métodos e linhas foram modificados pelos desenvolvedores e para servir de interface de comunicação com o *conflict-static-analysis*. Criando assim mais uma camada de proteção a *bugs*.

II. EXEMPLO MOTIVADOR

Para demonstrar o efeito prático de um conflito semântico em uma integração de código segue o exemplo da classe *Text* na Fig. 2, em que as linhas 7, em verde, e a 10, em azul, foram alteradas na *branch* A e B respectivamente. O código na figura não tem conflito textual, já que não há mudança na mesma linha nem em linhas consecutivas pelas duas partes. Não há também nenhum erro de sintaxe, o que permite o código compilar e ser executado normalmente.

Observando melhor as alterações de cada um dos desenvolvedores, assumo que o desenvolvedor da *branch* A quando adicionou o método *countDuplicatedWhitespaces()* tinha a intenção de usar a variável *fixes* para guardar a quantidade de espaços em branco duplicados no texto, assumo também que o desenvolvedor da *branch* B quando adicionou o método *countDuplicatedWords()* tinha a intenção

³<https://github.com/cdbm/static-semantic-merge>

```

1  class TextTestSuite {
2
3      public void countFixesTest() throws Throwable {
4
5          Text t = new Text();
6          t.text = "the_the_dog_dog";
7          t.generateReport();
8          assertTrue(2, t.fixes);
9
10     }
11 }
12 }

```

Fig. 3: Caso de teste que explicita o conflito semântico da Fig. 2

de usar a mesma variável *fixes* para guardar a quantidade de palavras duplicadas no texto, fazendo com que haja uma interferência entre as contribuições.

Para entender melhor porque há interferência nesse caso, considere o caso de teste ilustrado na Fig. 3, em que o texto de referência é "the the dog dog". A ideia aqui é comparar a atribuição de *fixes* no final da execução nas *branches* A, B e de *merge*. Quando o caso teste é executado na *branch* A, o método `countDuplicatedWhiteSpaces()` atribui 1 para *fixes*, quando o caso teste é executado na *branch* B o método `countDuplicatedWords()` atribui 2 para *fixes*, e quando o teste é executado na *branch* de *merge* o resultado atribuído para *fixes* também é 2, logo a intenção do desenvolvedor da *branch* A não foi preservada, o que gera uma interferência. Os desenvolvedores criaram suas funcionalidades de acordo e elas separadamente funcionam perfeitamente, porém quando integradas, a mudança da *branch* B causa mudança de comportamento nas mudanças da *branch* A. Interferências deste tipo podem estar inseridas em cenários com diversas outras mudanças e podem passar despercebidas a primeira vista, causando *bugs* e retrabalho.

O projeto *conflict-static-analysis* é útil para achar esse tipo de interferência [9], porém sua configuração e necessidade de informações externas diminui sua usabilidade em um ambiente de desenvolvimento, daí vem a proposta de criar o SSM para integrar essas análises ao processo de `git merge` e tornar possível a inserção no processo de desenvolvimento.

III. ESTRUTURA E UTILIZAÇÃO DO SSM

A ideia principal do SSM é ser acoplado ao processo de *merge* através do *hook* de *post-merge* para ser executado logo após um *merge* sem conflito textual ser feito, como mostra a Fig.4.

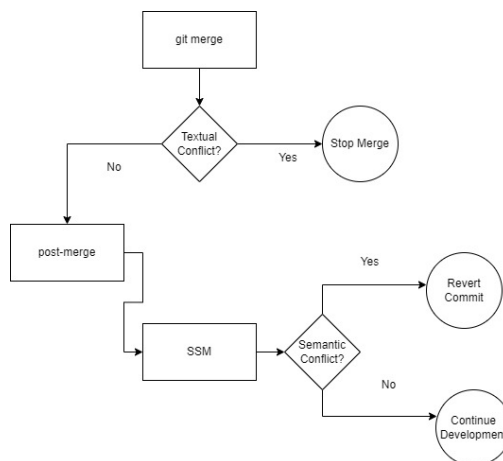


Fig. 4: Representação do funcionamento do SSM.

A. Git Hooks

Segundo a documentação do *Git* [1] *hooks* são *scripts* personalizados que são disparados quando certas ações ocorrem. Existem dois tipos de *hooks* os chamados *client-side* e *server-side*, *hooks client-side* são disparados por ações como fazer *commits* ou *merges*, e *hooks server-side* são disparados por operações de rede como por exemplo receber *commits* de um *push*.

Os *hooks* de um repositório *git* ficam armazenados no subdiretório *hooks*, que quase sempre se encontra em `./git/hooks`, essa pasta contém alguns *hooks* de exemplo e outros que são usados por padrão. Os *hooks* que são inseridos no comando `git init` são *scripts shell*, porém qualquer *script* executável devidamente nomeado funciona perfeitamente, sendo *Python*, *Ruby* ou qualquer outra linguagem.

Um exemplo de *hook client-side* é o *pre-commit* que é executado logo antes de um *commit* ser feito, esse *hook* é usado pelo *git* para checar se está tudo em ordem para o *commit* ser feito como por exemplo se há espaços a direita na mensagem de *commit*, mas pode ser personalizado para por exemplo chamar um *linter* verificar se as convenções de estilo de código estão sendo seguidas.

Um exemplo de *hook server-side* é o de *post-receive* que é executado em um servidor *git* como o *github* depois do processo de *git push* ser executado. Esse *hook* pode ser usado por exemplo para enviar *e-mails* para uma lista específica ou então atualizar um sistema de rastreamento de *tickets*.

Um outro exemplo de *hook client-side* é o

```

1 #!/bin/sh
2 head=$(git rev-parse HEAD)
3 parents=$(git log --pretty=%P -n 1 $head)
4 base=$(git merge-base $parents)
5 mergerPath="PATH_TO_SSM_FOLDER"
6 gradlePath="PATH_TO_GRADLE_BIN"
7 mavenPath="PATH_TO_MAVEN_BIN"
8 java -jar ${mergerPath}/static-semantic-merge-1.0-
  SNAPSHOT.jar $head $parents $base $mergerPath
  $gradlePath $mavenPath

```

Fig. 5: Script de post-merge.

post-merge, que é o utilizado nesse trabalho, que é executado logo após um *merge* textual bem sucedido. Esse *hook* não tem comportamento padrão no *git* mas pode ser usado para restaurar dados ou arquivos que o *git* não está rastreando, ou como no caso desse trabalho usar uma ferramenta para procurar conflitos semânticos na integração.

B. Hook de Post-Merge

Como pode-se ver na Fig. 5, o *script* de post-merge é um *script shell* encarregado de chamar o projeto do SSM logo após o *merge* textual ser bem sucedido. O SSM espera como entrada os hashes dos *commits* de base, left (*branch A*), right (*branch B*) e *merge*, esse último sendo a head no momento após o *merge*. É necessário também passar também o path em que o projeto do SSM se encontra, além do path do maven bin e gradle bin instalados na máquina. A execução então passa para dentro do *jar* do SSM.

A primeira linha do *script* determina que ele é um arquivo *shell*, as linhas 2 a 4 servem para coletar os hashes dos *commits* que são necessários. A linha 2 coleta o hash do *commit* atual, ou seja a head, o comando `git rev-parse` é um comando interno do *git* que tem várias funções, quando é usado com a flag `-git-dir` retorna o caminho relativo do diretório `.git`.

A linha 3 coleta o *hash* dos *parents*, ou seja, o *hash* do último *commit* da *branch A* e *B* (*left* e *right*). `git log` mostra os *logs* dos *commits*, a flag `-pretty=%P` faz com que só os *hashes* dos *parents* de cada *commit* sejam mostrados, a flag `-n 1 $head` faz com que apenas os *hashes* dos *parents* do *commit head* sejam mostrados, ou seja `$parents` guarda uma *string* com dois *hashes* separados por um espaço, sendo o primeiro o hash do parent na *branch A* e o segundo o parent na *branch B*.

A linha 4 coleta o *hash* do melhor ancestral comum aos dois *parents*, ou seja, o momento em

```

1 Integer[] ary = new Integer[ index( str, ch ) + str
  .length( ) ];
2
3 Integer[] ary = new Integer[ index( str, ch ) + str.
  length() ];

```

Fig. 6: Exemplo de código que o DiffJ considera equivalente

que as duas *branches A* e *B* compartilham a mesma base. o comando `git merge-base` é responsável por achar esse *hash*, o parâmetro `$parents` diz para o comando quais *hashes* que se deve procurar o ancestral comum.

As linhas 5 a 7 são para coletar o *path* de onde estão o projeto do SSM que foi baixado pelo usuário e os *bins* do *gradle* e do *maven*, nesse momento é necessário o usuário preencher manualmente esses *paths*. A linha 5 guarda o caminho para o SSM na máquina do usuário, esse caminho é necessário para copiar dependências, como o *DiffJ*, para o projeto que está sendo feita a análise. As linhas 6 e 7 guardam o caminho dos *bins* tanto do *gradle* quanto do *maven*, os *bins* são necessários para gerar a *build* do projeto que está sendo analisado, o usuário precisa preencher manualmente esses caminhos.

A linha 8 é responsável por de fato executar o *jar* do SSM passando os parâmetros coletados usando o comando `java -jar` que executa um *jar* em um caminho especificado.

C. Geração da Build

Dentro do SSM a primeira ação tomada é tentar gerar uma *build* válida do projeto que será analisado, a *build* gerada será utilizada pelo *conflict-static-analysis* que necessita das classes compiladas. Uma *build* é um arquivo pronto para uso com o código do projeto compilado para ser executado por uma máquina. O SSM verifica se o projeto utiliza o *Maven* ou *Gradle* como sistema de compilação e então chama o comando correspondente para gerar a *build*. Essa verificação juntamente com o fato do *conflict-static-analysis* esperar classes java como entrar limita a atuação do SSM a projetos *Java*. Como é necessário uma *build* válida para as análises serem executadas, se a *build* falhar a execução para e o *merge* é revertido.

D. Coletando Linhas Alteradas

Para os casos em que a *build* foi gerada com sucesso, é o momento de coletar as linhas que foram alteradas pelos desenvolvedores dentro de um mesmo método, para isso o SSM usa o *Mining Framework* que tem essa funcionalidade implementada usando o *DiffJ*⁴ como dependência. O *DiffJ* é um programa que compara arquivos java baseando-se apenas no código dos arquivos, sem considerar formatação, comentários ou espaços em branco. Por exemplo na Fig. 6 o *DiffJ* consideraria as linhas 1 e 3 como equivalentes, isso é importante para que apenas casos que realmente tenham mudanças entre versões dentro do mesmo arquivo e mesmo método sejam considerados. Se não houver mudanças entre versões dentro de um mesmo método as análises não vão ser chamadas e a execução considerará que não houve conflito.

E. Executando as análises

Com todas as informações necessárias para as análises o momento é de chamá-las, para isso usamos o *Mining Framework* como interface para chamar o *conflict-static-analysis* para rodar as análises. O *conflict-static-analysis* então vai usar as análises implementadas para procurar conflitos semânticos no cenário de *merge* montado até aqui. Cada análise executada deixa um *log* no *console* e escreve seu resultado num arquivo CSV que será usado posteriormente para determinar se houve ou não conflito semântico no *merge*, a Tabela 1 mostra um exemplo do arquivo CSV que é gerado. A Fig. 7 mostra um exemplo de *log* de uma análise que reportou conflitos.

F. Reportando Conflitos

Com o fim da execução das análises é gerado um CSV como mostrado na Tabela 1, o SSM agora lê esse CSV checando se houve alguma interferência, um valor *TRUE* significa que aquela dada análise reportou um conflito. Se alguma das análises usadas reportar *TRUE* o SSM considera que houve interferência no *merge* e faz a reversão do mesmo.

Tabela 1: CSV mostrando os resultados das análises

class	method	commit	OA Intra	OA Inter
Parser	"parse(Page,String)"	6fdb8...	FALSE	TRUE

⁴<https://github.com/jpace/diffj>

IV. METODOLOGIA

Para comprovar a usabilidade e a viabilidade da ferramenta SSM fizemos um estudo empírico em que o SMM foi executado em diferentes cenários de *merge* em diferentes projetos *Java*. O SSM era inserido nos projetos através da inserção do *hook* de *post-merge* e cada execução avaliou tanto a capacidade de detecção e reversão do *merge* quanto a confiabilidade em relação ao resultado que era esperado. A seguir estão os detalhes da configuração e execução do estudo

A. Amostra

Foram coletados 18 cenários de *merge* do repositório *mergedataset*⁵ para fazer parte do estudo, essa amostra constitui um subconjunto de uma amostra utilizada em estudos anteriores [2] [5]. A utilização de um subconjunto foi necessária pois nem todos os cenários conseguiram gerar uma *build* por questões de dependências ou acesso as mesmas. Todos os cenários utilizados contém mudanças entre duas *branches* em um mesmo método, os projetos selecionados utilizam Java pelo fato de tanto o *mining framework* quanto o *conflict-static-analysis* esperarem projetos Java. A seguir temos a Tabela 2 mostrando quais foram os projetos utilizados e quantos cenários de cada projeto foram usados.

Tabela 2: Cenários de merge coletados por projeto

Nome do Projeto	Cenários
Crawler4j	1
ElasticSearch	4
Jsoup	4
Moshi	1
OkHttp	1
Retrofit	3
RxJava	1
SimianArmy	2
Swagger-maven-plugin	1

B. Execução

Para a execução do estudo cada projeto foi clonado do *GitHub* e em seu diretório foi colocado o *hook* de *post-merge*. Para o estudo o *script* de *post-merge* teve que ser chamado manualmente pois os projetos já se encontravam com o *merge* feito. Alguns cenário apresentaram problemas de *build* gerados

⁵<https://github.com/spgroup/mergedataset>

```

Running right left NonCommutativeConflictDetectionAlgorithm{name = 0A Inter}
Using jar at C:\Users\DAVID\Documents\repositorios\rest-avisos\.files\project\35b8876283fad7ec62871c5cc689641c003b67dc\original-without-dep
endencies\merge\build.jar
out 12, 2022 5:07:04 PM br.unb.cic.analysis.ioa.InterproceduralOverrideAssignment internalTransform
INFORMAÇÕES: CONFLICTS: [source(com.cdbm.restavisos.RestAvisosApplication, main, 15, x = "456", [Stacktrace{sootClass=RestAvisosApplication,
sootMethod=void main(java.lang.String[]), lineNumber=15}]) => sink(com.cdbm.restavisos.RestAvisosApplication, main, 12, x = "123", [Stacktr
ace{sootClass=RestAvisosApplication, sootMethod=void main(java.lang.String[]), lineNumber=12}])]
Runtime: 0.033s
Analysis results
-----
Number of conflicts: 1
Results exported to out.txt
-----

```

Fig. 7: Exemplo de log de análise que reportou conflitos.

por dependências indisponíveis e foram descartados. A seguir estão os resultados do estudo.

V. RESULTADOS

Como podemos ver na Tabela 3, dos 18 cenários executados, 5 retornaram resultado positivo, resultando na reversão do *commit* de merge, e 13 retornaram negativo, resultando na continuação do processo de desenvolvimento. No entanto dos 5 resultados positivos registrados somente um era esperado interferência, resultando em 4 falsos positivos.

Da mesma forma, dos 13 resultados negativos registrados 6 continham interferências que não foram identificadas, gerando falsos negativos. A tabela 4 mostra a matriz de confusão dos resultados obtidos, mostrando tanto os falsos positivos e negativos quanto os verdadeiros positivos e negativos encontrados.

Tabela 3: Resultado obtido e esperado por cenário

Commit	Projeto	Resultado	Eperado
345ad9...	SimianArmy	FALSE	FALSE
a40a41...	RxJava	TRUE	FALSE
c39c19...	SimianArmy	TRUE	FALSE
3764b3...	elasticsearch	FALSE	FALSE
3764b3...	elasticsearch	FALSE	FALSE
3764b3...	elasticsearch	FALSE	FALSE
a44e18...	jsoup	FALSE	TRUE
3f7d2c7...	jsoup	FALSE	TRUE
a8b698...	jsoup	FALSE	TRUE
71f622...	retrofit	FALSE	TRUE
afb82c...	moshi	FALSE	TRUE
6fdb8f...	crawler4j	TRUE	TRUE
59cb6...	elasticsearch	FALSE	FALSE
35166...	okhttp	FALSE	FALSE
e825a...	swagger-maven-plugin	FALSE	TRUE
fee47...	jsoup	FALSE	FALSE
2b6c7...	retrofit	TRUE	FALSE
2b6c7...	retrofit	TRUE	FALSE

Tabela 4: matriz de confusão dos resultados

		Expected		Total
		Positive	Negative	
Predicted	Positive	1	4	5
	Negative	6	7	13
Total		7	11	18

VI. AMEAÇAS A VALIDADE

Esse trabalho foca em apresentar a ferramenta SSM que tem como propósito ser introduzida no processo de desenvolvimento para detectar interferências e conflitos semânticos no momento de integração de duas *branches*. Assim como fazer manualmente, o processo de coleta de informações para realizar a análise e análise do código em si depende de muitas variáveis para ocorrer perfeitamente.

Muitas vezes a integração pode resultar numa versão em que não é possível gerar a *build*, seja porque ainda não foi feita a configuração do processo de geração de *build* para aquele projeto, ou ainda não esteja sendo usado uma ferramenta de automação de compilação como *Maven* e *Gradle*, ou então a que está sendo usada não é suportada pela ferramenta como é o caso do *Apache Ant*.

Há também a situação do tamanho usado da amostra, o que dificulta o processo de procura e identificação de padrões, sendo mais difícil entender o comportamento geral da ferramenta.

Há também as limitações geradas pelas dependências do SSM como o *Mining Framework* e o *conflict-static-analysis* que faz com que apenas projetos java sejam esperados para o funcionamento do SSM. Apesar do SSM ter um processo de uso descomplicado quando se utiliza projetos Java,

para fazer com que o SSM fosse capaz de atender projetos de outras linguagens haveria a necessidade de se mudar todo o processo de geração de *builds*, coleta de informações e até de análises, visto que todos esses processos são ligados a linguagem Java.

VII. TRABALHOS RELACIONADOS

de Oliveira [9] em seu trabalho propõe a implementação de análises estáticas de substituição de atribuição para detecção de conflitos semânticos, ele define [9] (2022, p.17) que uma substituição de atribuição acontece quando:

as alterações em um dos ramos podem semanticamente envolver uma operação de escrita para um elemento de estado que também está associado a uma operação de escrita envolvida nas alterações feitas pelo outro ramo e não há operação de escrita da base entre eles.

A implementação de de Oliveira é utilizada dentro do *conflict-static-analysis* como uma das análises a serem executadas.

Sarma et al. [10] apresenta o Palantir, uma ferramenta que detecta mudanças de desenvolvedores em um mesmo arquivo, o que sugere a diminuição de conflitos. No entanto a ferramenta Palantir não se propõe a de fato analisar as mudanças para procurar interferência e tampouco as tenta impedir de acontecer.

Cabral Júnior [7] propõe o SAM, que tem o propósito de ser uma extensão do *git merge* e de identificar conflitos semânticos através da criação e execução de casos de teste gerados pela ferramenta SMAT [5] para a identificação de mudança de comportamento, podendo assim informar o usuário sobre conflitos semânticos. A ferramenta de Cabral Júnior tem objetivo fluxo de trabalho parecidos com o SSM, as diferenças principais é que invés de usar algoritmos de análise estática para encontrar interferências, são criados casos de testes para avaliar a mudança de comportamento, e invés de reverter o *merge* a ferramenta apenas informa a existência ou não de uma interferência.

VIII. TRABALHOS FUTUROS

Para as próximas versões da ferramenta nós planejamos fazer com que seja possível o usuário escolher se prefere que a ferramenta reverta o *commit* de

merge ou apenas informe os conflitos que houveram no caso de haver uma inferência. Planejamos também adicionar suporte a outras ferramentas de automação de compilação como o *Apache Ant*, visando aumentar os projetos que podem se utilizar do SSM.

IX. CONCLUSÃO

Durante o desenvolvimento de um software é comum que diferentes desenvolvedores modifiquem arquivos dependentes ou o mesmo arquivo, para isso é esperado que haja uma paralelização através de *branches* em ferramentas de versionamento de código, visando a integração posteriormente quando as mudanças forem consideradas satisfatórias. Nessa integração é possível ocorrer tanto conflitos textuais e semânticos, conflitos textuais são reportados pelas ferramentas de versionamento de código e geralmente tomam alguns minutos para serem resolvidas, já conflitos semânticos não são reportados e podem passar despercebidos por olhos não treinados, o que pode gerar *bugs* no futuro, fazendo com que seja necessário alguém com conhecimento das regras de negócio e do processo de desenvolvimento daquele software em específico para poder corrigir esses conflitos.

Existem algoritmos de análise estática que detectam essas interferências em uma integração, essa análise leva em conta a semântica do código e quais funções, variáveis e classes estão sendo usadas, invés de apenas fazer comparações textuais.

Buscando melhorar o processo de identificação e resolução dessas interferências e visando aumentar a qualidade do código produzindo para um software esse trabalho propôs a criação da ferramenta Static Semantic Merge (SSM), que acopla implementações de algoritmos de análise estática ao processo de desenvolvimento de um software através do *hook* de *post-merge* do *Git*, analisando e reportando interferências e conflitos semânticos que podem ocorrer em integrações.

Foi realizado um estudo empírico em cima de cenários de *merge* reais que produziam ou não conflitos semânticos a fim de demonstrar a capacidade da ferramenta de reportar e impedir esses conflitos, assim como identificar que eles não existem, através do estudo foi mostrada a viabilidade da ferramenta para a incorporação de detecção de conflitos semânticos no fluxo de desenvolvimento de um software.

REFERENCES

- [1] 8.3 customizing git - git hooks. <https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks>. Accessed: 2022-10-12.
- [2] Roberto Souto Barros Filho. Using information flow to estimate interference between developers same-method contributions, 2017.
- [3] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, 39(10):1358–1375, 2013.
- [4] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):59:1–59:27, 2017.
- [5] Léuson Da Silva, Paulo Borba, Thorsten Berger, and João Moisakis. Detecting semantic conflicts via automated behavior change detection. *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020.
- [6] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, pages 345–387, 1989.
- [7] Aldiberg Gomes Cabral Junior. Sam - ferramenta de merge semântico baseada em análises comportamentais através de testes unitários, 2022.
- [8] Hankin C. Nielson F., Nielson H. R. Principles of program analysis. [S.l.]: Springer, 2005.
- [9] Matheus Barbosa de OLIVEIRA. Detecção de conflitos semânticos via análise estática de substituição de atribuição, 2022.
- [10] Anita Sarma, David F Redmiles, and Andre Van Der Hoek. Palantir: Early detection of development conflicts arising from parallel code changes. *IEEE Transactions on Software Engineering*, 38(4):889–908, 2012.