UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

Hugo Alves Cardoso

Asserções de objetos completos em testes para detecção de conflitos semânticos, uma análise crítica

Recife
2025

Hugo Alves Cardoso

Asserções de objetos completos em testes para detecção de conflitos semânticos, uma análise crítica

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Engenharia da Computação

Orientador (a): Paulo Henrique Monteiro Borba

Recife

2025

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

Cardoso, Hugo Alves.
Asserções de objetos completos em testes para detecção de conflitos semânticos, uma análise crítica / Hugo Alves Cardoso. - Recife, 2025.
21 p : il., tab.

Orientador(a): Paulo Henrique Monteiro Borba
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2025.
Inclui referências.

1. Conflitos semânticos. 2. Equivalência de Objetos. 3. Asserções de testes. 4. Geraçnao de testes unitários. I. Borba, Paulo Henrique Monteiro. (Orientação). II. Título.

000 CDD (22.ed.)

Hugo Alves Cardoso

Asserções de objetos completos em testes para detecção de conflitos semânticos, uma análise crítica

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Engenharia da Computação.

Aprovado em: 04/04/2025

BANCA EXAMINADORA

_____

Prof. Dr. Paulo Henrique Monteiro Borba (Orientadora)

Universidade Federal de Pernambuco

_____

Prof. Dr. Breno Alexandro Ferreira de Miranda (Examinador Interno)

Universidade Federal de Pernambuco

Asserções de objetos completos em testes para detecção de conflitos semânticos, uma análise crítica

Hugo Alves Cardoso[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 7.851 – 50.732-970 – Recife – PE – Brazil
Orientador: Paulo Borba

hac@cin.ufpe.br

Abstract. *In software development environments, it is common for developers to work in a parallel and collaborative manner, which makes the code integration a process that can deviate the software behavior from the one introduced by developers, i.e. introducing semantic conflicts between versions. Unit tests generator tools have been used to detect such conflicts, but this approach still generates a high rate of false negatives. To address this issue a new method to generate test assertions has been proposed, where a hashcode for complex objects created in the tests prefix is generated and used in the test assertions to detect the objects state change between the code versions. This study compared the new assertion approach with SAM, a semantic merge tool consolidated in the literature, over real code integration scenários in open source projects, over the capabilities of detecting these conflicts. It was found that not only the proposed approach was not capable of detecting more semantic conflicts than SAM's, it generated false positive results not observed by the last one. The results indicate that this assertion generation approach is very sensitive to code changes that do indeed change the complex objects state but do not introduce a semantic conflict, thus the new approach can lead to the accusation of false positives in many textual integration scenarios.*

Keywords: Semantic Conflicts, object equivalence, test assertions, unit test generation

Resumo. *Em ambientes de desenvolvimento de software, é comum que desenvolvedores trabalhem de maneira paralela e colaborativa, o que faz com que a integração de código possa afastar o comportamento do software daquele introduzido pelos desenvolvedores, i.e. introduzindo conflitos semânticos entre as versões. Ferramentas geradoras de testes unitários têm sido utilizadas na detecção desses conflitos, mas essa abordagem ainda gera uma alta taxa de falsos negativos. Para solucionar esse problema um novo método de geração de asserções nos testes foi proposto, onde um hashcode para objetos complexos criados no prefixo do teste é gerado e utilizado nas asserções dos testes para detectar a mudança de estado do objeto entre as versões de código. Esse estudo comparou a nova abordagem com SAM, uma ferramenta de merge semântico consolidada na literatura, sobre cenários reais de integração de código em projetos open source, sobre a capacidade de detecção desses conflitos. Os resultados indicam que essa abordagem de geração de assertivas é muito sensível às mudanças no código que, de fato, alteram o estado do objeto complexo mas não introduzem conflito semântico, portanto a nova abordagem pode levar a acusação de falsos positivos em muitos cenários de integração textual.*

1. Introdução

Em ambientes de desenvolvimento de software modernos, desenvolvedores trabalham de maneira paralela e colaborativa, onde tarefas são divididas entre si e cada colaborador é responsável por aplicar as modificações e integrar suas contribuições a um repositório central. Durante o processo de integração, comumente, são verificadas a existência de conflitos textuais e de *build*. Essas verificações, entretanto, não garantem que as contribuições dos diferentes desenvolvedores não conflitem entre si à nível de comportamento.

Conflitos semânticos ocorrem quando dois desenvolvedores modificam componentes de um sistema que possuem dependência entre si, por exemplo um método que se utiliza de outro método para a realização de suas funções, sejam eles dentro da mesma classe ou de classes distintas. Essas modificações conflitam quando há a mudança do comportamento do componente, modificação que não foi levada em consideração por outros desenvolvedores e que tinham em mente outras características ao implementarem suas contribuições. Para exemplificar esse tipo de conflito, imaginemos um sistema bancário que possui um método denominado “debitar conta”, o desenvolvedor “A” adiciona uma verificação à esse método, garantindo que a conta após ser debitada não possui saldo negativo, já o desenvolvedor B realiza modificações no método “realizar pagamento” que utiliza o método anterior para realizar a transação, entretanto, o desenvolvedor “B” não levou em consideração as modificações introduzidas por A, o que pode levar a um comportamento não esperado para os casos de transações em que a conta debitada terá saldo final negativo.

A verificação da ocorrência de conflitos semânticos em versões de código é um problema indecidível, ou seja, não existe algoritmo capaz de determinar com precisão a existência ou não desse tipo de conflito, dessa forma, soluções baseadas em heurísticas foram propostas na literatura para endereçar esse problema. Uma dessas soluções que se mostram promissoras em seus resultados é a utilização de ferramentas baseadas em geradoras de testes unitários, como o SAM (Da Silva et al., 2024), na detecção de conflitos semânticos. A partir da geração de testes estes são executados entre as diferentes versões do código, os resultados dessas execuções são anotados e posteriormente utilizados para serem verificados por heurísticas de existência de conflitos semânticos. Um exemplo de conflito detectado segundo as heurísticas estabelecidas por Da Silva et al. (2020) é o caso em que um teste gerado, ao ser executado nas quatros versões do código, passa em todas a não ser a versão final, aquela gerada após integração textual, esse resultado indica que um comportamento que estava presente nas demais versões anteriores não se preservou após o *merge* textual, indicando conflito.

Apesar de promissoras, essas ferramentas detectoras de conflitos semânticos baseadas em geradoras de testes unitários ainda apresentam limitações que repercutem numa baixo *recall* quando utilizadas na detecção de casos de conflitos semânticos em cenários reais de integração de código. No estudo realizado (Da Silva et al., 2024) SAM foi capaz de detectar 9 cenários de conflitos dentro 28 ocorrências em cenários de integração de projetos *open source*. Uma das razões apontadas por ele, que pode justificar baixo *recall* encontrado, se deve ao fato de que as geradoras de testes, como o *Evosuite* (Fraser & Arcuri, 2011), *Randoop* (Pacheco et al., 2007) e suas versões modificadas, *Differential EvoSuite* (Shamshiri, 2015) e *Randoop Clean* (Da SIlva et al., 2024) , não são capazes de gerar *asserts* que explorem características complexas do objeto que estão avaliando,como analisar o conteúdo de posições específicas de um *array*, estas ferramentas, no entanto, geram *asserts* simples, que verificam se o tamanho de um *array* é maior que zero ou se o conteúdo de uma variável é não nulo.

Numa tentativa de superar a limitação da geração de asserts simples por parte das geradoras de testes unitários, outro estudo (Castanho, 2021) propôs uma modificação no Evosuite, o EvoSuiteR. Essa nova implementação é capaz de gerar *asserts* que verificam o objeto como um todo, através de *hashcode* gerado para objetos complexos instanciados no prefixo do teste. Por objetos complexos ele teve em mente objetos de tipo não primitivo e não enumerável, dessa forma, esses objetos teriam associados a eles um *hashcode* capaz de incorporar os valores dos seus atributos e, recursivamente, dos objetos referenciados, propondo, assim, solucionar a incapacidade dos testes gerarem *asserts* relevantes capazes de analisar minuciosidades de um objeto no prefixo do teste. Em seu trabalho (Castanho, 2021) executou a sua ferramenta detectora de conflitos semânticos, *UNSETTLE,* sobre uma base de dados compreendendo 19 casos de *merge textual* em projetos *opensource*, como resultado, sua ferramenta foi capaz de detectar 6 casos de conflitos dentre os 19 analisados (≈ 32%).

Para entender as vantagens e desvantagens da implementação proposta, o presente trabalho realizou um estudo comparativo entre as diferentes geradoras de testes utilizadas por Da Silva et al. (2024) no SAM e o *EvoSuiteR* proposta por Castanho (2021). Configurando o SAM para utilizar somente o *EvoSuiteR* como geradora de testes, a replicação foi realizada sobre a mesma base de dados utilizada por Da Silva et al. (2024), o *mergedataset*, visto que essa traz cenários de projetos reais e com grande complexidade. Após a execução foram observados os resultados de detecção de conflitos segregados por geradora de teste, e finalmente confrontamos os resultados encontrados utilizando o *EvoSuiteR* e os resultados obtidos por Da SIlva et al. (2024). Pudemos observar que na nossa execução não somente não fomos capazes de detectar novos casos de conflitos semânticos que não foram detectados pelo SAM, como também observamos um caso de falso positivo causado pela utilização do *EvoSuiteR*. Evidenciando que asserts utilizando *hashcodes* e que verificam objetos por completo podem induzir a casos de falso positivos em cenários de adição ou remoção de atributos de uma mesma classe por desenvolvedores distintos, cenários esse comum em casos de integração de código.

Este trabalho foi organizado da seguinte forma: Na Seção 2 exploramos o conceito de conflitos semânticos, trazendo exemplos de cenários onde esses conflitos ocorrem, e mostrando a aplicação do SAM na detecção desses conflitos. Na Seção 3 explicamos as heurísticas introduzidas por Da Silva et al. (2024) que utilizaremos para basear os resultados dos nossos experimentos. Na seção 4 abordamos as mudanças introduzidas pelo *EvoSuiteR*. Na seção 5 trazemos a metodologia adotada no estudo, na seção 6 trazemos os resultados obtidos e traçamos uma comparação com o que foi observado por Da Silva et al. (2024). Na seção 7 elaboramos uma avaliação crítica dos resultados encontrados, explorando as limitações enfrentadas por ferramentas como o *EvoSuiteR*. Na seção 8 listamos alguns trabalhos relacionados à detecção de de conflitos semânticos em cenários de *merge* textual, traçando comparativos com o presente estudo e por fim, na seção 9 concluímos o trabalho, indicando possíveis linhas de investigação para trabalhos futuros.

2. Motivação

Para exemplificar o conceito de conflito semântico, considere o exemplo elaborado por Da Silva et al. (2024) o qual é baseado em um projeto *open source*.

Dessa forma, observamos a classe *Text*, que possui um método denominado *cleanText*, esse método tem como objetivo a remoção de palavras duplicadas, comentários e normalizar espaços em branco em uma *string*, para isto, o método faz chamada a três métodos da classe Text: *removeDuplicateWords*, *removeComments* e *normalizeWhitespace*. Entretanto, o método *removeDuplicatedWords* também faz chamada ao método

normalizeWithespace. A partir desse cenário, dois desenvolvedores, *Left* e *Right,* pretendem remover a duplicidade da chamada ao método *normalizeWhitespace*.

O desenvolvedor *Left* remove a chamada do método duplicado no corpo do método *cleanText*, já o desenvolvedor *Right* remove a chamada do método duplicado no corpo do método *removeDuplicatedWords*, ambos os desenvolvedores enviam suas contribuições para o repositório central e as contribuições são textualmente integradas, visto que não há conflitos textuais entre as versões. Entretanto, nesse novo cenário, não há mais uma chamada ao método *normalizeWhitespace* o que ocasionaria a não remoção de espaços múltiplos em uma *string* na chamada do método *cleanText*. Dessa forma percebe-se o surgimento de conflitos semânticos em cenários de integração textual. Na Figura 1 podemos ver o cenário de integração textual referente ao exemplo descrito, e na Figura 2 podemos ver um teste que evidencia a mudança de comportamento entre as versões, nesse teste temos a criação de uma variável denominada *t* da classe *Text* e que recebe como valor para seu atributo *text* a string “the the dog”, ao executarmos esse teste nas diferentes versões de código, *Base*, *Left*, *Right* e *Merge*, vemos que esse teste passará em todas as versões com exceção de *Merge*, e que, segundo as heurísticas de conflito semântico consideradas nesse estudo e que serão aprofundadas na seção 3, evidenciam um cenário de desaparecimento de comportamento, e portanto um conflito semântico.

```
1  class Text {
2
3    public String text;
4
5    void cleanText() {
6      this.removeDuplicatedWords();
7      this.removeComments();
8      this.normalizeWhitespace();
9    }
10
11   void removeDuplicatedWords(){
12     ...
13     this.normalizeWhitespace();
14   }
15
16   void normalizeWhitespace(){
17     ...
18   }
19 }
```

Figura 1. Cenário de integração onde o desenvolvedor *Left* remove o trecho destacado em vermelho e o desenvolvedor *Right* remove o trecho destacado em verde (Da silva et al., 2024)

```
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. Teste que, quando executando nas quatro versões do código, e segundo nossas heurísticas de conflito semântico, verifica o conflito nesse cenário de integração textual (Da Silva et al., 2024)

Como elaborado por Da silva et al. (2024), as geradoras de teste utilizadas por ele em sua ferramenta, SAM, demonstraram não produzir *asserts* significativos, contrastando com os *asserts* da figura 2. Essas ferramentas usualmente trazem *asserts* mais simples,

verificando características não relevantes de objetos criados no corpo do teste, como por exemplo, se esse objeto possui valor nulo ou não. Dada essa limitação, verificou-se uma baixa taxa de recall alcançada por sua ferramenta, que está diretamente relacionada à incapacidade das geradoras de testes trazerem *asserts* relevantes no corpo dos seus testes.

Como uma tentativa de superar essas limitações, Castanho (2021) propõe uma nova forma de elaboração de *asserts,* em que os testes são gerados de forma que tenham a capacidade de incorporar neles informações sobre o estado dos atributos do objeto em determinada versão do código. Essas modificações foram implementadas na sua nova geradora de testes, o *EvoSuiteR,* na qual traz um método que percorre recursivamente os atributos de um objeto instanciado no teste, gerando assim, um valor único para este objeto em determinada versão do código.

3. Heurísticas de detecção de conflitos semânticos

Para verificarmos se em um cenário de execução de testes entre diferentes versões de código há ou não conflitos semânticos, nos baseamos em critérios heurísticos para realizar essa determinação. Com base em Da Silva et al. (2020) utilizamos dois conjuntos de critérios heurísticos para determinar a existência de conflitos, chamamos eles de heurística 1 e heurística 2.

A primeira heurística, heurística 1, verifica se os comportamentos adicionados por um dos desenvolvedores, *Left* ou *Right*, são preservados na versão final, *Merge*. Dessa forma, um teste que passa em *Left* ou *Right* e falha em *Merge* indica a existência de um conflito semântico no cenário de integração textual segundo nossa primeira heurística. A Figura 3 ilustra essa heurística em um cenário de integração textual. Verifica-se que o teste falha em *Base* nos dois casos, mas passa nas respectivas versões dos desenvolvedores, *Left* ou *Right*, indicando que o comportamento implementado pelos desenvolvedores de fato não estava presente em *Base*, já ao executar esse teste em *Merge* e verificar que o teste falha, pode-se determinar que o comportamento do desenvolvedor não se preservou na versão final, portanto evidenciando um conflito semântico.

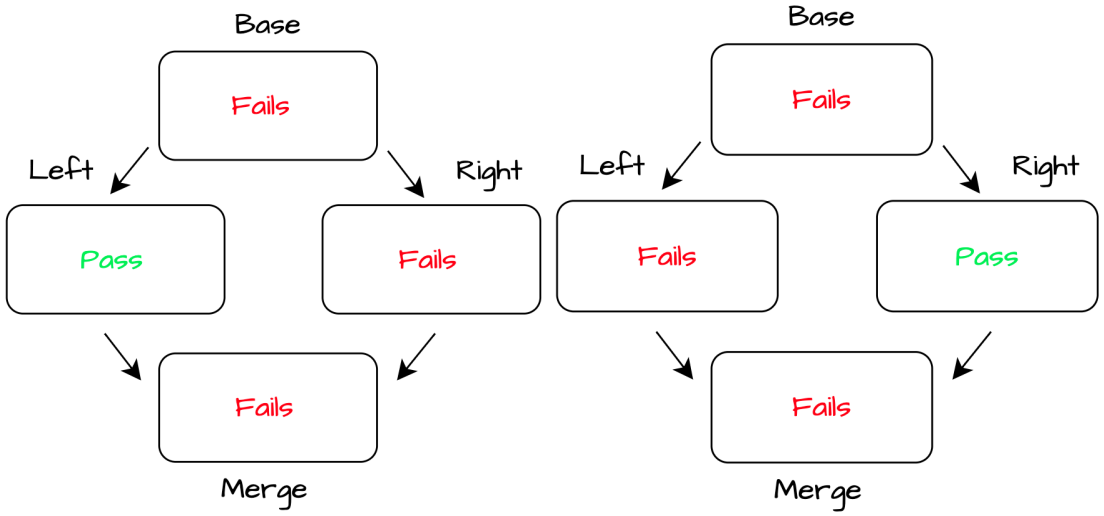

Figura 3. Ilustração da primeira heurística de conflito semântico para o caso do desenvolvedor *Left*, à esquerda, e para o caso do desenvolvedor *Right* à direita

A segunda heurística, heurística 2, verifica se há algum comportamento emergente ou desaparecimento de comportamento na integração. Para o caso de comportamento emergente referimos a um caso de comportamento que não estava presente em nenhuma versão anterior à *Merge* (*Base, Left e Right*) mas que após a integração textual esse

comportamento tornou-se evidente. Para o caso de desaparecimento de comportamento verifica-se a negação lógica, ou seja, um comportamento que estava presente nas versões anteriores a *Merge* (*Base, Left e Right*) mas que não se verifica na versão final. A Figura 4 ilustra os casos de desaparecimento e surgimento de comportamentos, respectivamente. Abordando o caso de conflito semântico elaborado na seção 2, vemos que esse se enquadra no critério da segunda heurística, mais especificamente para o caso de desaparecimento de comportamento, isso é entendido pelo fato de que, na versão final do exemplo, o comportamento observado nas versões anteriores a *Merge* não se verifica nessa versão final do código.

Castanho (2021) em seu estudo, traz heurísticas de conflitos semânticos similares às de Da Silva et al. (2020), entretanto traz como diferença a consideração da possibilidade de que um teste gerado pode, eventualmente, não ser executável em uma das versões, isto é, o teste pode não ser compilável para uma dada versão do código. Portanto, Castanho (2021) formula duas heurísticas, uma para detecção de comportamento emergente e outra para detecção de comportamento perdido. Em ambos os casos ele considera a possibilidade de que o teste executado pode ter como resultado em uma das versões um erro, e que mesmo assim pode ser considerado para a determinação da existência ou não de conflito. Segundo ele, isto se deve ao fato de que métodos implementados por um desenvolvedor podem não estar presentes em outras versões do código, assim gerando os erros considerados. Na Figura 5 podemos ver os exemplos de heurística mencionados para desaparecimento de comportamento e comportamento emergente respectivamente.

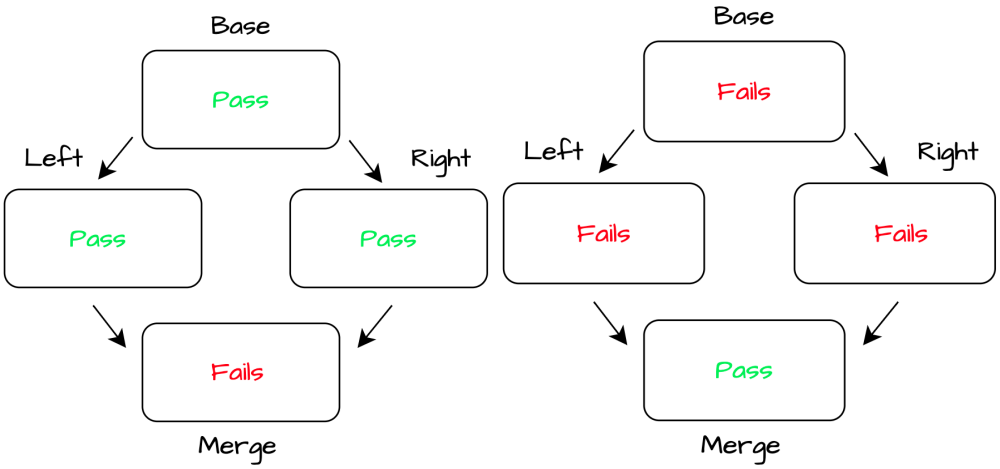

Figura 4. Ilustração da segunda heurística para o caso de desaparecimento de comportamento, à esquerda, e surgimento de comportamento, à direita

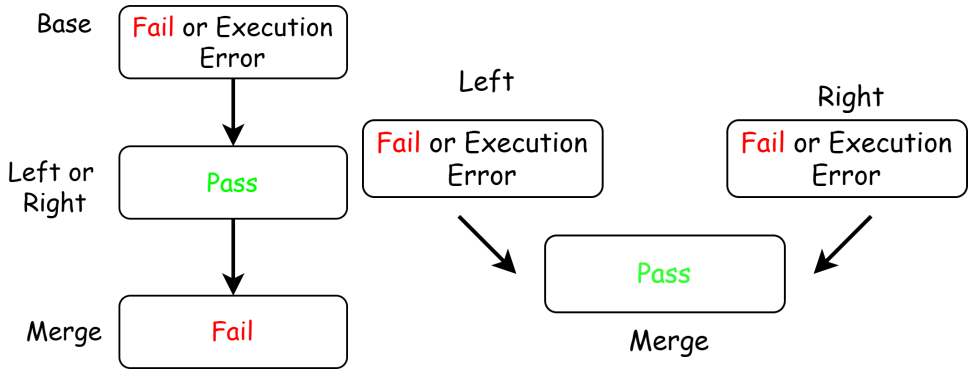

Figura 5. Ilustração das heurísticas de desaparecimento de comportamento, à esquerda, e surgimento de comportamento, à direita, segundo (Castanho 2021), nota-se que nessa heurística, para o caso de surgimento de comportamento, o teste não é gerado em *Base*

Para este estudo, levamos em consideração a heurística descrita por Da Silva et al. (2024), por compartilhar de sua consideração de que levar em conta execuções de testes que resultaram em erro podem introduzir casos falso positivos nos resultados (Da SIlva et al., 2024), de forma que considerar esse erros pode contribuir para um viés de favorecimento à ferramenta proposta por Castanho (2021).

4. EvoSuiteR

Um dos pontos apontados por Da Silva et al. (2020) como razões para uma baixa taxa de *recall* alcançada pelo SAM era devido às limitações inerentes às geradoras de testes unitários. Segundo ele, algumas das limitações dessas geradoras eram referentes à incapacidade de criar objetos complexos e gerar, através de chamadas de métodos, um cenário que evidencie o conflito semântico. Outro ponto levantado por ele referia-se a simplicidade dos *asserts* gerados por elas, que comumente não eram capazes de explorar detalhes dos objetos, como por exemplo, verificar o conteúdo em posições específicas de um array ou verificar os valores de atributos de um objeto, no entanto, se verificou que estas se atêm a verificações simplistas como, por exemplo, verificar se o conteúdo de uma variável é não nulo, ou verificar se o comprimento de um *array* é maior que um valor, esse tipo de abordagem se evidenciou não serem capazes de detectar os conflitos. Como uma tentativa para melhorar os *asserts* gerados pelas geradoras de testes unitários, uma adaptação do *EvoSuite*, o *EvoSuiteR* foi proposta por Castanho (2021) com o objetivo de modificar a forma como os *asserts* dos testes são gerados, de forma que tenham a capacidade de incorporar neles informações sobre o estado dos atributos do objeto em determinada versão do código.

O *EvoSuiteR* trouxe duas modificações importantes de se notar para a geração de prefixos e *asserts* em testes. Sobre as modificações relacionadas à geração de prefixos, a ferramenta passou a utilizar três versões de código (*Left, Right e Merge*), onde, durante a geração de prefixos, os prefixos candidatos eram executados nas três versões do código e seus resultados eram comparados, de modo que a função objetivo utilizada na geração de prefixos, que também foi modificada, avaliava a diferença de estado de todos os objetos do teste nas três versões de código, mantendo como prefixos candidatos aqueles que eram capazes de gerar a maior diferença de estado entre as três versões mencionadas. Esse comportamento da geração de prefixos tem em mente a tentativa de encontrar testes que evidenciam conflitos semânticos relacionados com a segunda heurística, em especial aqueles que evidenciam comportamento emergente, como mencionado por Castanho (2021).

Traçando um comparativo entre essa nova versão e as versões do *EvoSuite* utilizadas por Da silva et al. (2024) no SAM, o *EvoSuite* (Fraser & Arcuri, 2011) e o *Differential EvoSuite* (Shamshiri, 2015), observamos que o primeiro utiliza somente uma versão do código para gerar os testes, isso pode implicar em testes gerados que não executam em uma das outras versões do código, além dessa versão trazer a geração de asserts de uma forma simples, que não favorece a detecção dos conflitos semânticos. O *Differential EvoSuite* utiliza duas versões de código para a geração dos asserts, fazendo regressão entre as versões, tentando identificar diferenças entre as versões, de maneira semelhante ao *EvoSuiteR*, e explorando essas diferenças por meio dos *asserts*, novamente, apesar da versão possuir um mecanismo de busca de prefixos mais elaborada ainda traz consigo uma formulação de *asserts* mais simples.

Dada a necessidade de compreender um objeto instanciado no prefixo do teste, e nessa compreensão observar suas minúcias, como seus atributos e seus valores, bem como os atributos de seus atributos e seus valores, quando o caso, (Castanho 2021) propõe um

novo método para interpretar estes objetos complexos, o *allFieldsMethod*, nesse método objetos complexos são dados como parâmetro e nele seus atributos são percorridos para serem avaliados, caso os atributos sejam de tipos não primitivos o método é chamado recursivamente, ao final da execução o *allFieldsMethod* retorna um *hashcode* para esse objeto, compreendendo nesse valor toda a informação observado sobre o seu estado. Dessa forma, objetos complexos teriam associados a eles um valor único que levaria em consideração todo o seu estado, conseguindo capturar toda a complexidade do objeto em um valor de fácil comparação quando pensamos na aplicação deste teste em outras versões do código. Um exemplo de teste resultante da execução do *EvoSuiteR* sobre o cenário de *merge* do projeto *Antlr4* pode ser observado abaixo na Figura 6, nesse exemplo o teste cria em seu prefixo um objeto do tipo *Python2Target* denominado *python2Target0*, após a declaração, o teste faz uma chamada ao método que gera a inconsistência entre as versões, o *getBadWords*, mais adiante no teste, o objeto *python2Target* é passado como parâmetro à chamada ao método *allFieldsMethod*, e o valor resultante da execução desse método, o *hashcode*, é armazenado na variável *long2*, essa variável é então utilizada no *assert* seguinte, comparando o valor esperado ao valor obtido. O valor esperado, para o caso do *EvoSuiteR* é sempre o valor obtido para a versão *Merge*, isto é, o teste gerado por ela levando em consideração as versões *Left*, *Right*, eMerge sempre passará em *Merge*, visto que os valores esperados estabelecidos nos *asserts* levam em consideração essa versão do código.

```
1 @Test(timeout = 4000)
2 public void test0() throws Throwable {
3   CodeGenerator codeGenerator0 = null;
4   Python2Target python2Target0 = newPython2Target(codeGenerator0);
5   Set<String> set0 = python2Target0.getBadWords();
6   ...
7   long long2 = AllFieldsCalculator.allFieldsMethod(python2Target0);
8   assertEquals(932380841235L, long2);
9   ...
10 }
```

Figura 6. Teste resultante da execução do *EvoSuiteR* sobre o cenário de *merge* do projeto *Antlr4*, nesse caso o objeto *python2Target0* possui como valor de *hashcode* associado elaborado pelo *allFieldsMethod* o número "932380841235"

Traçando um paralelo com o comportamento do *EvoSuite*, percebemos que este traria em seus testes gerados asserts que não incorporam toda essa complexidade, podendo trazer comparações simples como verificar se o objeto possui ou não valor nulo como explicado anteriormente. E quando pensamos na utilização de resultados de testes como componentes heurísticos para a detecção de conflitos semânticos, a utilização de uma versão do *EvoSuite* que traga consigo maior complexidade na avaliação de objetos sugere ser uma opção mais indicada à essa tarefa.

5. Metodologia

Dada a nova ferramenta proposta por Castanho (2021), o *EvoSuiteR*, onde nela se incorpora a capacidade de formular *asserts* que verificam minúcias de objetos complexos criados no corpo do teste, juntamente com as observações feitas por Da Silva et al. (2024) onde em seu estudo verificou o desempenho do SAM sobre uma base de dados maior e mais robusta do que aquela utilizada por Castanho (2021) surgiu a seguinte pergunta motivadora: "Configurar o SAM para utilizar o *EvoSuiteR* resultaria na identificação de casos de conflito semânticos negligenciados pelos demais geradoras de testes unitários utilizadas por Da Silva et al. (2024) em seu estudo?"

Castanho (2021) em seu estudo trouxe uma análise em uma amostra de 14 cenários de *merge* onde em todos eles possuíam conflitos semânticos, apesar de ser uma amostra com cenários de projetos reais, seria interessante submeter o *EvoSuiteR* sobre a amostra coletada por Da Silva et al. (2024), onde nela foram coletados 85 cenários de *merge* textual, contendo 28 casos de conflito semântico. Essa base de dados além de contar mais exemplos de conflitos, contém 57 cenários sem conflitos, importantes para a determinação de casos falsos positivos eventualmente observáveis pela ferramenta, assim, estendendo as análises feitas por Castanho (2021) em uma base de dados mais ampla.

Em sua ferramenta, Da Silva et al. (2024) utiliza quatro ferramentas geradoras de testes unitários para compor o SAM, são elas, *EvoSuite, Modified EvoSuite*, *Randoop*, e *Randoop Clean*, nenhuma dessa ferramentas traz consigo a proposta feita no *EvoSuiteR*. Dessa forma, executamos o SAM sobre o *mergedataset*, base de dados utilizada por Da Silva et al. (2024) e que compreende mais de 80 cenários de integração textual, onde nesses cenários existem 28 casos de conflitos semânticos, todos esses cenários compreendem cenários de projetos *open source* complexos, o que traz uma alta qualidade à base de dados. Além disso, aproveitamos do fato do SMA possuir a capacidade de configurar, isto é, adicionar ou remover geradoras de testes, para isto, basta inserimos ao SAM o arquivo com o código compilado de uma geradora de testes e configurar o SAM para prover adequadamente a execução dessas ferramentas através da interface de linha de comando (CLI).

Em seu estudo, Da Silva et al. (2024) aplicou duas modificações nos códigos fontes dos cenários de *merge* antes de gerar os arquivos compilados, transformado e serializado. A primeira, substitui modificadores de acesso não públicos por público, sendo aplicada a classes, métodos, construtores e declarações. Ao tornar todos os elementos públicos, mais elementos podem ser chamados e acessados pelos testes gerados, possivelmente aumentando as chances de detectar conflitos. Já a modificação de serialização fornece às geradoras de testes unitários representações dos objetos complexos através de grafos desses objetos, tentando assim, facilitar a criação desses objetos no corpo dos testes gerados.

Na execução do SAM, incorporamos à ele somente a nova ferramenta proposta por Castanho (2021), o *EvoSuiteR*, isto foi feito pela característica *pseudo-randômica* das geradoras de testes, que dependem do valor da semente utilizada para a execução dos seus algoritmos na elaboração dos prefixos do teste. Tendo isto em mente, a replicação do estudo realizado por Da Silva et al. (2024) utilizando outros valores de semente poderiam ter um viés de “sorte” ou “azar” na obtenção dos novos resultados, dessa forma, consideramos os resultados apontados por ele em seu estudo como os resultados a serem confrontados por nosso experimento.

Após a execução do SAM os resultados foram analisados, onde pode-se observar quais cenários de *merge* o SAM sob nova configuração foi capaz de detectar a existência de conflito semântico. Após a verificação dos cenários capturados pelo SAM na nossa execução, fizemos um comparativo com os resultados obtidos por Da Silva et al. (2024). Como no caso de Da Silva et al. (2024) sua configuração do SAM possuía 4 ferramentas, o que trouxe a sua execução uma maior capacidade de detecção de cenários de conflito, tivemos o cuidado de gerar comparações por ferramentas geradoras de testes unitários individualmente, de forma que a comparação não se tornasse injusta.

Outro ponto importante de se destacar é que Castanho (2021) em sua ferramenta detectora de conflitos semânticos, o UNSETTLE, além além da nova implementação do *EvoSuite*, o *EvoSuiteR*, trouxe uma implementação de uma DSL previa a execução da ferramenta geradora de testes unitário, essa DSL tem como objetivo verificar árvores sintáticas das versões *Base*, *Left* e *Right* e determinar qual o método alvo do teste e quais

métodos devem ser chamados até a chamada do método alvo . Destacamos que no nosso estudo não aplicamos a DSL implementada por Castanho (2021) visto que passamos diretamente o método alvo para o SAM e deixamos a construção do corpo do teste como atribuição da geradora de testes, de forma semelhante à metodologia adotada por Da Silva et al.(2024).

Na Figura 7 abaixo podemos ver a representação gráfica da metodologia adotada. Para a execução usamos a *seed* “1”para o *EvoSuiteR*, utilizando um limite de tempo de geração de testes de 300 segundos.

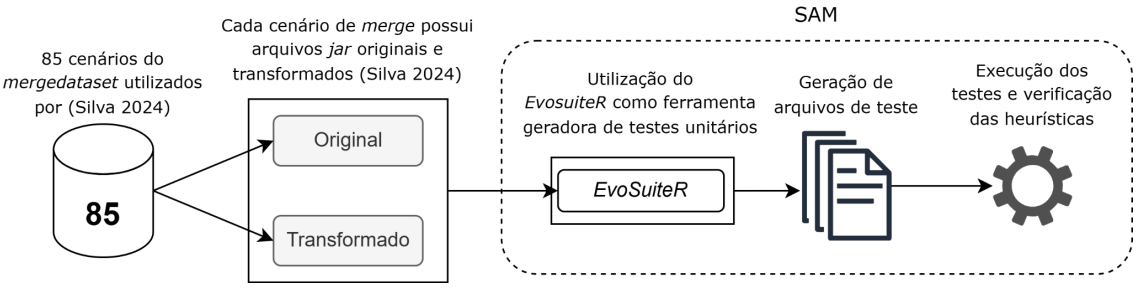

Figura 7. Representação gráfica da metodologia adotada

Na base de dados *mergedataset*, Da Silva et al. (2024) utiliza três tipos de versão de código compilado, original, transformado e serializado. No caso das versões originais, diz respeito a nenhuma modificação realizada ao código fonte andes dor arquivos *jar* serem gerados, já os arquivos transformados possuíam algumas modificações em seu código fonte, como modificação da visibilidade de atributos e métodos antes da compilação do código fonte, os arquivos serializados, por sua vez pretendem fornecer as ferramentas geradoras de testes grafos do objeto para facilitar a criação de objetos complexos no prefixo do teste. Essas modificações foram realizadas, segundo Da Silva et al. (2024) com o objetivo de melhorar a cobertura dos testes e sua capacidade de gerar as infecções segundo o modelo RIP (*Reach Infect Propagate*).

6. Resultados

Após a configuração do SAM com o *EvoSuiteR*, executamos a ferramenta sobre a base de dados utilizada por Da Silva et al. (2024) e verificamos os casos de conflito semântico detectados por nosso experimento. Para traçar comparativos, verificamos individualmente os resultados obtidos por Da Silva et al. (2024) para cada ferramenta, para que o comparativo seja justo com a execução do SAM em nosso caso.

Ao observar na Tabela 1 os resultados obtidos por nossa execução, verificamos que o SAM utilizando o *EvoSuiteR* como ferramenta geradora de testes unitários foi capaz de verificar apenas três cenários de conflitos semântico dentre os 28 existentes. Comparando os resultados obtidos pelo *EvoSuiteR* com aqueles obtidos pelo *EvoSuite* na execução de Da Silva et al. (2024), vemos que o *EvoSuite* foi capaz de detectar 6 casos de conflito semântico dentre os 28. A Tabela 1 traz a comparação entre os resultados obtidos entre essas duas ferramentas. Destacados em verde são os casos de conflitos detectados pelo *EvoSuiteR* que não foram detectados pelo *EvoSuite*, em azul estão os casos detectados pelo *EvoSuite* que não foi detectado pelo *EvoSuiteR*.

Tabela 1. Casos de conflitos semânticos detectados pelo *EvoSuiteR* e *EvoSuite*

EvoSuiteR			*EvoSuite*		
Projeto	Classe	JAR	Projeto	Classe	JAR

Antlr4	Python2Target	O & T	Antlr4	Python2Target	O & T
Antlr4	Python3Target	O & T	Antlr4	Python3Target	O & T
Fitnesse	SlimTableFactory	O & T	HikariCP	HikariPool	O & S
			Spring-Boot	AtomikosProperties	O & T
			Spring-Boot[1]	AtomikosPropertiesTests	O & T
			Spring-Boot[2]	AtomikosPropertiesTests	O & T

Na coluna JAR, indicamos o tipo de arquivo *jar* que a ferramenta utilizou para gerar o teste que identifica o conflito semântico. A letra "O" indica arquivo *jar* original, "T" indica arquivo transformado, e "S" serializado. O projeto Spring-Boot[1] corresponde ao método testProperties(), já o projeto Spring-Boot[2] corresponde ao método testDefaultProperties().

Pelo fato de trazer uma nova forma de gerar *asserts*, o *EvoSuiteR* trouxe um caso novo de conflito semântico que foi negligenciado pelo *EvoSuite*. Entretanto, o EvoSuite conseguiu compreender mais casos de conflito do que a nova versão. A menor detecção de casos de conflito semânticos pelo *EvoSuiteR* pode ser explicado também pela possível não execução de seus testes nas outras versões do código, relembrando que na nossa heurística descartamos as análises caso um teste possua um erro ao ser executado em uma das versões. Outro ponto que pode justificar a baixa detecção de casos de conflito pode ser o "azar" da semente escolhida na geração de testes, como ambas as geradoras de testes são baseadas em algoritmos genéticos, a escolha da semente impacta diretamente os resultados obtidos.

Na Tabela 2 podemos observar o comparativo entre o *EvoSuiteR* e o *Differential EvoSuite*. O *Diferential EvoSuite* é uma versão que traz como modificações a tentativa de gerar testes que revelam diferenças de comportamento entre duas versões diferentes do código (Da Silva et al., 2024), de forma semelhante ao *EvoSuiteR*, entretanto este último tenta encontrar diferenças entre três versões distintas do código, como descrito na Seção 4.

Tabela 2. Casos de conflitos semânticos detectados pelo *EvoSuiteR* e *Differential EvoSuite*

EvoSuiteR			*Differential EvoSuite*		
Projeto	Classe	JAR	Projeto	Classe	JAR
Antlr4	Python2Target	O & T	Antlr4	Python2Target	O & T
Antlr4	Python3Target	O & T	Antlr4	Python3Target	O & T
Fitnesse	SlimTableFactory	O & T	Fitnesse	SlimTableFactory	T
			Fitnesse	SlimTableFactoryTest	T
			Spring-Boot	AtomikosProperties	O & T
			Storm	KafkaSpoutConfig	T

Observando os resultados da Tabela 2, verificamos que no estudo realizado por Da Silva et al. (2024) o *DIfferential EvoSuite* foi capaz de identificar também o caso de conflito semântico para o projeto *Fitnesse* para a classe *SlimTableFactory*, de forma que, nos nossos estudos, não identificamos por parte do *EvoSuiteR* nenhum caso novo de indicação de cenário de conflito semântico. Quando comparamos os resultados obtidos pelas duas

ferramentas, além de considerar os pontos destacados na comparação entre o *EvoSuiteR* e o *EvoSuite*, recordamos que nesse caso, comparando o *EvoSuteR* e o *Differential EvoSuite*, ambas baseiam a geração dos prefixos dos testes em regressões realizadas entre as versões do código, isto é, ambas as ferramentas buscam gerar prefixos que explorem diferenças quando executadas nas diferentes versões, a partir desse fato esperaria-se observar um comportamento semelhante em ambas as ferramentas, mas não é isto que é observado. Essa fato pode ser explicado pela forma como o *EvoSuiteR* elabora seus prefixos, considerando três versões de código em vez de duas como o *Differential EvoSuite*. Ao levar em consideração uma maior quantidade de commits, o

Na Tabela 3 temos a comparação dos resultados obtidos pelo *EvoSuiteR* com os resultados obtidos pelo *Randoop* e *Randoop Clean*, com essas duas últimas ferramentas sendo capazes de identificar os mesmos cenários de conflitos semânticos no estudos de Da Silva et al. (2024). O *Randoop Clean* é uma versão modificada do *Randoop* com o intuito de trazer testes que tragam melhores resultados na identificação de conflitos semânticos.

Tabela 3. Casos de conflitos semânticos detectados pelo *EvoSuiteR* e pelo *Randoop / Randoop Modified*

EvoSuiteR			*Randoop / Randoop Modified*		
Projeto	Classe	JAR	Projeto	Classe	JAR
Antlr4	Python2Target	O & T	Spring-Boot	AtomikosProperties	O & T
Antlr4	Python3Target	O & T	Storm	KafkaSpoutConfig	T
Fitnesse	SlimTableFactory	O & T			

Comparando o *EvoSuiteR* com as versões do *Randoop* percebemos que os casos de conflitos detectados pelas ferramentas não possui interseção, isso pode ser entendido pela forma como as ferramentas possuem abordagens de geração de testes distintos, levam a diferentes casos de identificação de cenários de conflito semântico.

Finalmente, na Tabela 4 analisamos os casos observados de falsos positivos para cada geradora de testes pelo nosso trabalho por Da Silva et al. (2024), esses resultados foram separados por geradora para que pudéssemos traçar comparativos entre elas.

Tabela 4. Casos de falsos positivos observados no nosso estudo e no estudo de (Da Silva et al., 2024)

Nosso estudo			(Da Silva et al., 2024)		
Geradora	Projeto	Classe	Geradora	Projeto	Classe
EvoSuiteR	Netty	ChannelBuffer	*EvoSuite*	Spring-Boot	AtomikosProperties
			Randoop Clean[1]	Spring-Boot	AtomikosProperties

[1]O caso para a geradora *Randoop Clean* detectou o conflito para ambos os casos do arquivo compilado do tipo original quanto transformado, dessa forma, (Da SIlva et al., 2024) considera três casos de falso positivo detectado por seu estudo

Observando os resultados obtidos sobre casos de falso positivos, vemos que apesar de Da Silva et al., (2024) afirmar que que detectou três casos de conflito semântico distintos, vemos que eles estão relacionados ao mesmo cenários de merge, mudando somente a

ferramenta e o tipo de código compilado utilizado. Segundo Da Silva et al. (2024), os casos detectados de falso positivos em seu estudo estão relacionados com testes *flaky*, isto é, teste que possuíam diferentes comportamentos em diferentes execuções sobre o mesmo código. Para o nosso caso, o falso positivo emergente não possui como causa ser um teste *flaky*, mas sim, essa ocorrência está diretamente relacionada à forma como o *EvoSuiteR* elabora os serus *asserts* no corpo dos testes. Além disso, investigango essa observação com maior profundidade foi possível observar que a forma de gerar *asserts* sugerida por Castanho (2021) no *EvoSuiteR* pode induzir a muitos casos de acusações de falsos positivos quando utilizamos essa geradora de testes em cenários reais de *merge* textual, as motivações para isso serão elaboradas na sessão 7.

Finalmente, podemos resumir os resultados observados da seguinte forma, ainda que performando de maneira inferior à suas demais versões, o *EvoSuiteR* ainda assim foi capaz de ter melhor resultado quando comparado ao *Randoop* e ao *Randoop Clean*. Essa melhor desempenho está diretamente relacionada à forma como essas ferramentas utilizam seus algoritmos na geração de prefixos de testes, enquanto as variantes do *Randoop* possuem forma de busca um pouco mais aleatória, as versões do *EvoSuite* conseguem direcionar de maneira mais efetiva o formato ideal de prefixo de teste, criando de forma mais completa diversos tipos de objetos complexos, esses que carregam consigo os conflitos semânticos. Quando comparamos o *EvoSuiteR* como suas geradoras “irmãs”, vemos que esta sub performa em relação às duas, isso pode ser entendido principalmente pela forma como os prefixos dos testes são gerados que ocasiona, com certa frequência, a não execução desse teste em uma das versões do código, e que, por nossas heurísticas, não são levados em consideração na verificação da ocorrência de conflitos semânticos.

7. Investigação do caso falso positivo observado

Após a execução do SAM utilizando o *EvoSuiteR*, e a observação de um novo caso de falso positivo foi gerado, isto é, que não foi detectado por Da Silva et al. (2024) em seu estudo, iniciamos uma investigação das motivações para a ocorrência deste, e pudemos observar que este caso gerado está diretamente ligado à forma como o *assert* foi gerado, isto é, ao uso de um *hashcode* para compreender o objeto completo.

Apesar desta nova forma de gerar *asserts* ter sido implementada com o objetivo de aumentar a capacidade de identificação de conflitos semânticos de ferramentas como o SAM, para alguns casos de integração textual que são comuns de serem encontrados em projetos de desenvolvimento, essa modificação pode levar, na verdade, a um aumento na quantidade de falsos positivos.

Considerando o caso falso positivo levantado por nossa execução, que foi referente ao projeto Netty, após inspeção manual, constatamos que nesse cenário de integração textual houve a adição de um atributo à classe *LengthFieldBasedFrameDecoder* pelo desenvolvedor *Left* da integração, da mesma forma o desenvolvedor *Right* também adiciona um novo atributo à mesma classe e essa modificação também mantida na versão *Merge*. De tal forma que esta adição de atributo foi o suficiente para que o SAM, utilizando o teste gerado pelo *EvoSuiteR*, indicasse que nesse cenário de integração textual houvesse conflito semântico, onde na verdade não há.

Isso ocorre devido aos diferentes valores do *hashcode* observados para o objeto do tipo *LengthFieldBasedFrameDecoder* criado no corpo do teste para as versões *Base*, *Left*, *Right* e *Merge*. Explicando o motivo desses valores distintos, imaginemos a execução do método como descrito na seção 4. Ao executarmos o método sobre o objeto em *Base*, esta versão do código terá um quantitativo *x* de atributos para a classe referida, dessa forma apresentando um valor retornado pela função *allFieldsMethod*. Ao executarmos esse teste em *Right*, que apresenta um atributo a mais do que *Base*, o método retornará um valor

diferente do observado para *Base*, visto que a presença desse atributo influencia no valor retornado pelo método. De maneira análoga, em *Left*, que adiciona um atributo não existente em *Base*, e pelo fato deste ter tipo, nome ou valor distinto do atributo adicionado por *Right*, o método *allFieldsMethod* retornará também um valor distinto ao de *Base* e ao de *Right* ao executarmos esse teste na versão *Left*. Finalmente, em *Merge*, com a presença dos atributos dos dois *commits* pai de *Merge*, também teremos um valor diferente retornado pela função ao executar esse teste em *Merge.*

Notamos também que Castanho (2021) apresenta em seu estudo um teste gerado pelo *EvoSuiteR* para o projeto *Netty* muito semelhante ao que nos encontramos em nosso estudo, entretanto ele indica que o teste foi capaz de evidenciar o conflito semântico nesse cenário de *merge*. Apesar do projeto possuir um conflito semântico na sua integração textual, o teste apresentado por ele, similarmente ao encontrado por nós, não evidencia o conflito. Por uma coincidência ele acusa a existência do conflito devido às heurísticas utilizadas, entretanto, devido ao *assert* que falha não estar verificando diretamente o método conflituoso, o consideramos um caso de falso positivo.

Na Figura 8 é possível observar o atributo adicionado pelo desenvolvedor *Left* em sua versão do código, e na Figura 9 é possível ver o teste gerado pelo *EvoSuiteR* que, quando executado pelo SAM, o fez acusar a existência de um conflito semântico.

```
1 public class LengthFieldBasedFrameDecoder extends FrameDecoder{
2   private final int maxFrameLength;
3   private final int lengthFieldOffset;
4   private final int lengthFieldLength;
5   private final int lengthFieldEndOffset;
6   private final int lengthAdjustment;
7   private final int initialBytesToStrip;
8   private final boolean lengthFieldIncludedInFrameLength;
9   private boolean discardingTooLengFrame;
10  private long tooLongFrameLength;
11  private long bytesToDiscard;
12 }
```

Figura 8. Classe modificada no cenário de integração textual do projeto Netty presente no *mergedataset*. A adição do atributo pelo desenvolvedor está destacada em verde.

```
1 @Test(timeout = 4000)
2 public void test0() throws Throwable {
3   int into0 = 2055;
4   int int1 = 3;
5   int int2 = 2;
6   LengthFieldBasedFrameDecoder lengthFieldBasedFrameDecoder0 = new
  LengthFieldBasedFrameDecoder(int0, int1, int2);
7   ...
8     long long0 = AllFieldCalculator.allFieldsMethod(lengthFieldBasedFrameDecoder0);
9   assertEquals((-2375578926732272529L), long0);
10 }
```

Figura 9. Teste gerado pelo *EvoSuiteR* que quando executado em *Merge* passa, mas quando executado em *Base, Left e Right* falha, indicando um conflito segundo o segundo critério da nossa heurística

A verificação desse caso de falso positivo nos motivou a investigar com mais profundidade os casos nos quais o novo método de cálculo de *asserts* poderia gerar casos falsos positivos, visto que um cenário de adição de atributo a uma classe pode gerar a acusação de conflito e o inverso, a remoção de atributo, também pode ocasionar o mesmo comportamento observado. Com isso construímos alguns cenários hipotéticos para ilustrar as limitações apresentadas pela nova ferramenta, principalmente no que diz respeito à geração de *hashcodes* e a utilização destes em *asserts* de testes.

Supondo o seguinte cenário de integração textual: uma classe denominada Foo possui dois métodos implementados, *setMean* e setDistance, em *Base* o método *setMean* calcula uma média aritmética entre os dois atributos da classe, *x* e *y*, já o método *setDistance* calcula a distância euclidiana entre uma coordenada representada pelos dois atributos e a origem. Esses métodos não possuem chamada ao outro, e não existem outros métodos na classe que não seja o construtor, métodos *get* e *set*. Dois desenvolvedores, *Left* e *Right*, irão modificar esse métodos da versão *Base*, set*Distance* e set*Mean*, respectivamente. O desenvolvedor *Left* modifica o método set*Distance* para que ele passe a retornar a distância Manhattan entre um ponto representado pelos atributos da classe e a origem, e o desenvolvedor *Right* modifica o método *setMean*, onde a média calculada é, agora, uma média ponderada entre os dois atributos. Durante a etapa de integração textual não são detectados conflitos textuais entre as versões, visto que os desenvolvedores modificaram porções distintas do código, e a versão final compila normalmente. Além disso, não há conflito semântico nesse cenário de integração de código, dentro da classe o comportamento dos métodos são independentes. Na Figura 10 podemos ver os atributos e o construtor da classe Foo e na Figura 11 podemos ver o cenário de integração textual descrito.

```
1 public class Foo{
2   private double x;
3   private double y;
4   private double mean;
5   private double distance;
6   public Foo(double x, double y){
7     this.x = x;
8     this.y = y;
9     setMean();
10    setDistance();
11  }
12 }
```

Figura 10. Atributos e construtor da classe Foo

Base

```
1 public void setMean(){
2   this.mean = (this.x + this.y) / 2
3 }
4 public void setDistance(){
5   this.distance = Math.pow((Math.pow(this.x,2)
6                           + Math.pow(this.y,2)),0.5);
7 }
```

Left

```
1 public void setMean(){
2   this.mean = (this.x + this.y) / 2;
3 }
4 public void setDistance(){
5   this.distance = this.x + this.y;
6 }
```

Right

```
1 public void setMean(){
2   this.mean = (this.x * 0.3) + (this.y * 0.7);
3 }
4 public void setDistance(){
5   this.distance = Math.pow((Math.pow(this.x,2)
6                           + Math.pow(this.y,2)),0.5);
7 }
```

Merge

```
1 public void setMean(){
2   this.mean = (this.x * 0.3) + (this.y * 0.7);
3 }
4 public void setDistance(){
5   this.distance = this.x + this.y;
6 }
```

Figura 11. Cenário de integração textual onde não há conflitos semânticos provocados pelas mudanças dos desenvolvedores

Sob o cenário apresentado anteriormente, e considerando um prefixo de teste ilustrado na Figura 12, caso executamos esse prefixo nos diferentes *commits*, teríamos os seguintes valores para cada atributo da classe Foo, como pode ser visto na Figura 13.

```
1 @Test
2 public void test0() throws Throwable {
3   double double_0 = 3.0;
4   double douboe_1 = 5.0;
5   Foo foobar = new Foo(double_0, double_1);
6 }
```

Figura 12. Exemplo de prefixo de teste gerado para a classe Foo.

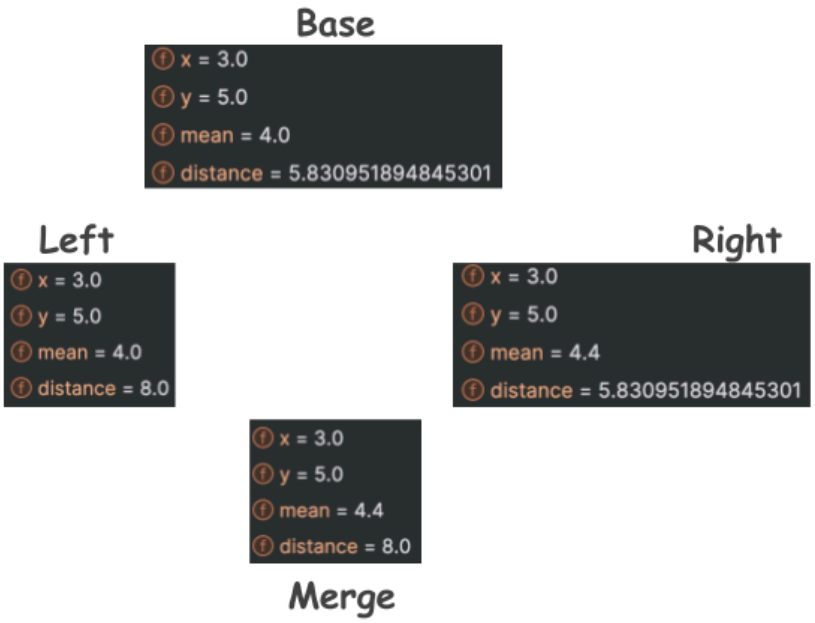

Figura 13. Valores observados para cada atributo do objeto foobar criado no prefixo do teste da Figura 12 após execução do prefixo em cada *commit*.

Supondo agora a criação de um *assert* para o teste, onde é calculado um hashcode para o objeto foobar, esse valor será utilizado como comparação dentro do assert. Para esse caso consideramos que o valor utilizado no assert foi aquele levando em consideração que o teste deve passar em *Merge*, da mesma forma como é observado no *EvoSuiteR*, onde este recebe as versões *Left, Right* e *Merge*, produzindo um teste que passa na última versão. Na Figura 14 podemos ver o teste agora com o cálculo do *hashcode* para o objeto e a presença do *assert* levando em consideração o valor de *Merge*. Na Figura 15 podemos ver os valores observados para o *hashcode* do objeto foobar nas demais versões de código, e o resultado do teste ao ser executado nas outras versões do código.

```
1 @Test
2 public void test0() throws Throwable {
3   double double_0 = 3.0;
4   double douboe_1 = 5.0;
5   Foo foobar = new Foo(double_0, double_1);
6   long long_0 = AllFieldsCalculator.allFieldsMethod(foobar);
7   assertEquals(4915846405332L, long_0);
8 }
```

Figura 14. Teste contendo *assert* que calcula o valor do *hashcode* para o objeto *foobar*.

Figura 15. Valores observados para o *hashcode* do objeto foobar ao final da execução do teste e os correspondentes resultados da execução. A partir das nossas heurísticas de conflitos temos um caso de comportamento emergente para o cenário

Outro cenários construído para entender com mais profundidade o problema da asserção de objetos completos buscou verificar se o mesmo comportamento é observado para ferramentas geradoras de testes unitários onde estas geram *asserts* tradicionais. Nesse caso, para compreender uma asserção de objeto completo, supomos o caso em que o teste gerado verifica todos os atributos da classe, com o intuito de verificar mudanças de estado no objeto. Para este caso, ainda consideramos a classe Foo utilizada no exemplo anterior, e o mesmo prefixo do teste. Na Figura 16 podemos verificar o novo teste gerado, utilizando o prefixo do exemplo anterior, mas agora com asserts tradicionais, um para cada atributo da classe. Esse novo teste foi executado sobre as quatro versões de código e os resultados das execuções, bem como o resultado da nossa heurística pode ser observado na Figura 17.

```
1 @Test
2 public void test0() throws Throwable {
3   double double_0 = 3.0;
4   double double_1 = 5.0;
5   Foo foobar = new Foo(double_0, double_1);
6   double double_2 = foobar.getX();
7   assertTrue(3.0 == double_2);
8   double double_3 = foobar.getY();
9   assertTrue(5.0 == double_3);
10  double double_4 = foobar.getMean();
11  assertTrue(4.4 == double_4);
12  double double_5 = foobar.getDistance();
13  assertTrue(8.0 == double_5);
14 }
```

Figura 16. Segundo exemplo de teste para investigação de indicação de casos falso positivos para asserts de objetos completos

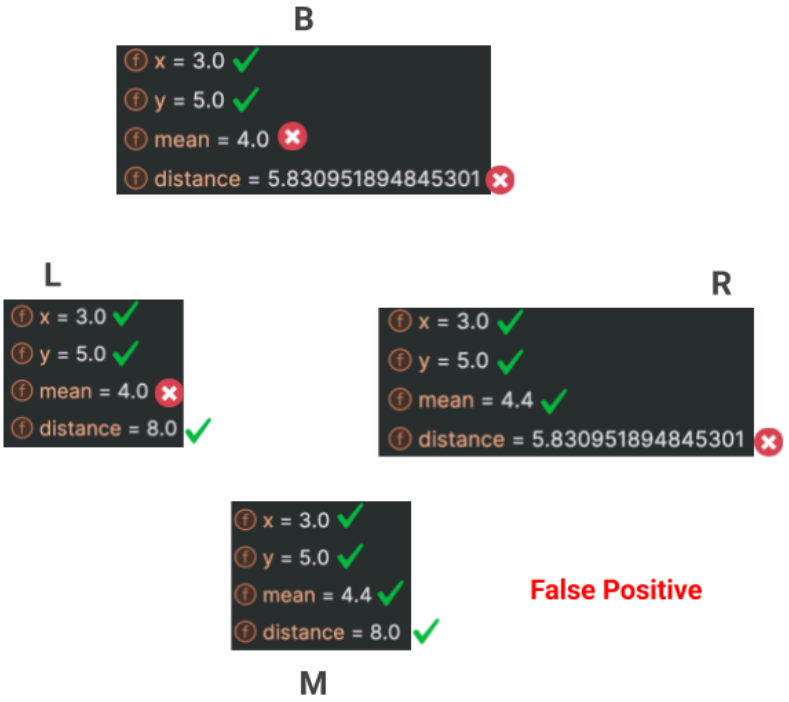

Figura 17. Resultado da execução e valores obtidos para cada atributo do objeto foobar, bem como o resultado da nossa heurística de conflitos semânticos

A partir das análises vemos que testes que buscam verificar por completo objetos complexos criados neles, seja através de métodos que calculam um *hashcode* associado ao estado desse objeto, seja através da geração de *asserts* para cada atributo do objeto, quando associados a heurísticas de conflitos semânticos em cenários de integração textual podem levar a casos falso positivos. E, segundo nossas análises, os dois casos para os quais esses testes podem gerar falsos positivos são cenários comuns de integração textual em projetos de desenvolvimento, o primeiro deles diz respeito à adição e remoção de atributos de uma classe, o segundo diz respeito a modificações paralelas em métodos onde esses métodos alteram atributos distintos da classe.

8. Trabalhos relacionados

Quando observamos a investigação de conflitos de *merge* em cenários de integração textual, usualmente focamos a abordagem na investigação de casos de conflito existentes em uma mesma classe de um projeto, o que pode simplificar o cenário das análises. Numa abordagem para ampliar a complexidade dos cenários de integração textual Maciel et al. (2024) realizou um estudo sobre 613 cenários de *merge* explorando cenários de conflito semânticos mais complexos, utilizando SAM acoplado a novas ferramentas geradoras de testes unitários, onde pode obter com sucesso uma boa taxa de acurácia para a ferramenta utilizada sobre o dataset produzido, evidenciando o bom desempenho e capacidade dessas ferramentas em detectar conflitos inclusive em cenários complexos. Em seu estudo, Maciel et al. (2024) elaborou uma base de dados sintética para a validação de suas observações, para nosso estudo preferimos nos ater a cenários de projetos reais de merge, entendendo que esses trazem consigo ainda uma maior complexidade às geradoras de testes no momento da geração dos prefixos, dessa forma, aumentando o rigor para a validação do *EvoSuiteR* em nossos experimentos.

Da Silva et al. (2020) propõe além de modificações nas ferramentas geradoras de testes unitários, modificações diretas no código fonte para melhorar a cobertura dessas ferramentas. Segundo ele, são dois os pontos principais de dificuldade enfrentados pelas

geradoras de testes: criação de objetos significativos no prefixo do teste e geração de *asserts* significativos capazes de captar as mudanças. Seguindo suas indicações de trabalhos futuros, analisamos o *EvoSuiteR*, uma ferramenta que tem como objetivo melhorar a elaboração de asserts trazendo análises sobre objetos criados no prefixo do teste por completo, observando se esta nova ferramenta, que não foi utilizada por Da Silva et al. (2024), sobre sua eficácia na detecção de de conflitos semânticos.

Com o intuito de melhorar a taxa de acurácia dessas ferramentas, Castanho (2021) propõe uma nova ferramenta para detecção de conflitos semânticos, UNSETTLE, consistindo de uma DSL para detecção de cenários candidatos de conflito semântico seguido da geração de testes através de sua implementação do *EvoSuite*, o *EvoSuiteR*, seu estudo teve a intenção de ser capaz de gerar testes com *asserts* que explorasse a complexidade dos objetos criados no teste. Entretanto, como evidenciado por nossos resultados obtidos e explicados na seção 7, esse modelo de asserts tem o viés de gerar casos falsos positivos em cenários de *merge* de projetos reais, o que pode ir na contramão dos objetivos do uso de ferramentas como o SAM na detecção de de conflitos semânticos.

Moraes et al. (2024) propôs um método de detecção de conflitos semânticos em tempo de execução em cenários de *merge* textual para projetos JavaScript. Essa abordagem traz consigo a vantagem de se detectar conflitos em projetos de linguagens não compiladas e fracamente tipadas. Entretanto seu estudo se limita à verificação de conflitos semânticos de *overriding assignments*, isto é, contribuições dos diferentes desenvolvedores modificam em tempo de execução a mesma variável. Entretanto, a capacidade de detecção de conflito por parte de sua ferramenta depende da capacidade do *script* em executar as porções do código que possuem as infecções. No nosso caso, por nosso estudo ser baseado em cenários de merge de projetos que utilizam linguagem compilada e fortemente tipada, dependemos da capacidade de das geradoras de testes criarem objetos relevantes no corpo do teste e criarem *asserts* relevantes que sejam capazes de evidenciar esses conflitos.

Outra abordagem para a detecção de conflitos semânticos em cenários de *merge* reside no uso de ferramentas de análise estática, onde algoritmos são executados sobre versões compiladas do código em análise. De Jesus et al. (2024) propõem uma abordagem eficiente para detecção de conflitos semânticos utilizando análises estáticas, visto que trabalhos anteriores relacionados possuem alto custo computacional. Apesar de trazer melhorias em tempo de execução para essas ferramentas De Jesus et al. (2024) demonstra que essa abordagem ainda levanta um alto número de casos falsos positivos, diferentemente de propostas como o SAM que focamos em nosso estudo, onde essa ferramenta apresenta um baixo quantitativo de casos de falsos positivos, mas ainda não apresenta a mesma taxa de *Recall* das ferramentas baseadas em análises estatísticas.

9. Conclusão

Os resultados obtidos neste estudo mostram que *asserts* complexos, sejam eles utilizando *hashcodes*, sejam eles tradicionais, mas que compreendam todos os atributos de um objeto, podem ser imprecisos quando utilizados em ferramentas detectoras de conflitos semânticos como o SAM. Isso reside no fato de que os testes que buscam analisar a completude do objeto serão sensíveis a todas as mudanças realizadas pelos desenvolvedores, inclusive aquelas que não geram conflitos entre versões.

A ferramenta proposta por Castanho (2021) foi capaz de detectar 3 dos 28 casos de conflitos semânticos em nossa base de dados, apresentando menos casos de falsos positivos e menor taxa de *recall* do que aquela observada por Da Silva et al. (2024). Isso pode ser explicado por algumas razões. A primeira delas pode estar relacionada a um

“azar” na escolha da semente para a geração dos testes utilizando o *EvoSuiteR*, como esta geradora de testes se baseia na execução de um algoritmo heurístico para a formulação de prefixos de teste, devemos levar em consideração que essa escolha de semente não favoreceu a criação de objetos que evidenciam conflitos. Outro ponto que pode ser levado em consideração é que diferentemente da heurística adotada por Castanho (2021) não consideramos testes que falham nas execuções para verificar a existência de de conflitos semânticos, pois compartilhamos da ideia de Da Silva et al. (2024) de que a consideração desses casos pode contribuir para a acusação de casos falso positivos.

Para futuras investigações, é sugerido que o método de geração de asserts seja capaz de verificar as diferentes versões do código e incorporar nele essas mudanças, de forma que a geração de um *assert* observando uma única versão do código pode trazer falsas conclusões sobre cenários de integração sobre nossas heurísticas de deteção de conflitos semânticos. Por exemplo, imaginemos uma forma de gerar *asserts* que leve em consideração uma formulação lógica entre os valores observados em *Base* e *Merge* e que seja executado em *Left* e *Right* para a verificação de conflitos. A utilização dos valores observados nos dois primeiros *commits* evitaria a acusação de casos falsos positivos, pois permitiria a verificação das duas heurísticas utilizadas nesse estudo em um único *assert*, a execução em *Left* e *Right* poderia, então, confirmar os casos de conflito semântico.

Outro ponto para futuras investigações é o estudo de variações das condições iniciais das ferramentas geradoras de testes. Essa variação deve ser feita a partir dos valores de semente adotados para a execução das geradoras de testes, de modo que sejam gerados testes mais diversos, e esses sejam capazes de detectar mais casos de conflitos semânticos. Como sabemos, ferramentas geradoras de testes unitários são baseadas, geralmente, em algoritmos heurísticos, e estes dependem fundamentalmente das condições iniciais de sua população gerada para exploração do espaço de soluções. Entretanto, variar tais condições implicam num aumento proporcional do tempo de busca dessas ferramentas, o que em muitos casos de integração de código, pode ser um impedimento na adoção dessas ferramentas.

Agradecimentos

Gostaria de agradecer às colaborações dadas por Léuson da SIlva e Toni Maciel, que durante nossos encontros semanais sempre contribuíram com ideias e observações precisas e fundamentais para o desenvolvimento deste estudo, bem como à orientação do professor Paulo Borba, que esteve sempre apoiando o desenvolvimento e guiando os melhores caminhos para a realização do trabalho. Agradeço também ao INES (Instituto Nacional de Engenharia de Software) pela disponibilização da infraestrutura de laboratórios e servidores, estes últimos fundamentais para a execução dos experimentos.

Referências

CASTANHO, Nuno Guilherme Nunes. Semantic conflicts in version control systems. 2021. Dissertação de Mestrado. Universidade de Lisboa (Portugal).

DA SILVA, Léuson et al. Detecting semantic conflicts with unit tests. Journal of Systems and Software, v. 214, p. 112070, 2024.

DA SILVA, Leuson et al. Detecting semantic conflicts via automated behavior change detection. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2020. p. 174-184.

DE JESUS, Galileu Santos et al. Lightweight Semantic Conflict Detection with Static

Analysis. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings. 2024. p. 343-345.

FRASER, Gordon; ARCURI, Andrea. Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011. p. 416-419.

MACIEL, Toni et al. Explorando a detecção de conflitos semânticos nas integrações de código em múltiplos métodos. In: Simpósio Brasileiro de Engenharia de Software (SBES). SBC, 2024. p. 181-191.

MORAES, Amanda; BORBA, Paulo; DA SILVA, Léuson. Semantic conflict detection via dynamic analysis. In: Simpósio Brasileiro de Linguagens de Programação (SBLP). SBC, 2024. p. 53-61.

PACHECO, Carlos et al. Feedback-directed random test generation. In: 29th International Conference on Software Engineering (ICSE'07). IEEE, 2007. p. 75-84.

SHAMSHIRI, Sina. Automated Unit Testing of Evolving Software. 2016. Tese de Doutorado. University of Sheffield.