Explorando a detecção de conflitos semânticos nas integrações de código em múltiplos métodos

José Antônio Alves Maciel[1]

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

jaam@cin.ufpe.br

Abstract. *During software development, integrating changes from different developers is crucial, yet it can cause the software to deviate from the intended behaviors by introducing semantic conflicts. Existing tools for detecting these conflicts are limited to simpler scenarios, focused on changes to a single method. This study broadens the approach by creating a sample with 613 synthetic merge scenarios to simulate more complex situations and evaluate the effectiveness of tools in detecting these conflicts. The identification of 230 semantic conflicts highlights the success of these adapted tools, contributing to the improvement of software development processes.*

Resumo. *Durante o desenvolvimento de software, a integração de mudanças por diferentes desenvolvedores é crucial, no entanto, pode levar o software a não preservar os comportamentos intencionados por eles ao introduzir conflitos semânticos. As ferramentas existentes para detectar esses conflitos se limitam a cenários mais simples, focados em alterações em um único método. Este estudo amplia a abordagem ao criar uma amostra com 613 cenários sintéticos de merge para simular situações mais complexas e avaliar a eficácia das ferramentas na detecção desses conflitos. A identificação de 230 conflitos semânticos destaca o sucesso dessas ferramentas adaptadas, contribuindo para o aprimoramento dos processos de desenvolvimento de software.*

1. Introdução

Dentro dos processos modernos de desenvolvimento de *software*, a colaboração e fluxos de trabalho paralelos são fundamentais, exigindo a frequente integração de mudanças no código feitas por diversas pessoas. Esse processo, embora essencial, carrega o risco de introduzir comportamentos inesperados — consequência da combinação de alterações que, individualmente, apresentam os resultados esperados, mas que, de maneira coletiva, geram um resultado imprevisto.

Tais comportamentos inesperados podem ser categorizados de várias maneiras, com base em suas características. Neste estudo, adotamos a terminologia utilizada por [Silva et al. 2020], usando conflitos de *merge* [Clementino et al. 2021] e conflitos textuais como sinônimos. Conflitos de *build* referem-se a conflitos semânticos sintáticos e estáticos, enquanto conflitos semânticos comportamentais estão relacionados a conflitos de teste e produção. Seguindo a abordagem de Silva et al. para brevidade, nos referirmos

a este último tipo simplesmente como conflitos semânticos, omitindo o termo ”comportamental”. Mais adiante, exploramos as distinções entre esses tipos de conflitos.

Enquanto ferramentas para identificar conflitos de *merge* [Clementino et al. 2021] e de *build* [Silva et al. 2020] decorrentes de integração de código são amplamente difundidas e eficazes [Cavalcanti et al. 2017], o desafio de detectar conflitos semânticos permanece considerável. Conflitos semânticos surgem não apenas de diferenças textuais diretas, mas das mudanças mais sutis no comportamento do código, implicando em uma maior dificuldade na detecção desses tipos de conflitos, particularmente quando essas alterações se distribuem por vários métodos ou componentes. Por exemplo, quando as modificações feitas por um desenvolvedor afetam o estado de elementos que são monitorados e empregados em partes do código alteradas por outro desenvolvedor. Este último pode ter baseado seu trabalho em uma determinada configuração do estado desses elementos, a qual deixa de ser válida após a integração das mudanças.

Para identificar conflitos semânticos, Silva et al. [Silva et al. 2020] introduziram a ferramenta SMAT, que analisa alterações comportamentais nas quatro distintas versões de código que contemplam um processo de *merge*. Tais alterações são identificadas pelo uso de ferramentas de geração automática de testes unitários e de heurísticas desenvolvidas pelos autores para comparar os resultados desses testes gerados. Embora tenham obtido resultados positivos, os experimentos conduzidos limitaram-se a amostras reduzidas formadas apenas por cenários de integração cujas modificações restringiram-se a um único método. Essa limitação sugere que os resultados podem não capturar integralmente a complexidade inerente aos processos de *merge* em contextos práticos.

Para preencher essa lacuna, criamos uma amostra abrangente contendo 613 cenários sintéticos de *merge*, consistindo em quádruplas de versões de um projeto - incluindo o *commit Merge*, os *commits Parents* e o *commit Base* - projetados para refletir situações reais onde integrações de código levam a conflitos semânticos. Esta criação de novos cenários visa oferecer um extenso campo de experimentação para identificar conflitos semânticos em uma variedade de cenários mais complexos, onde as integrações contemplam alterações em múltiplos métodos. Nesse contexto, conduzimos um estudo aplicando o SMAT em conjunto com ferramentas de geração de testes automatizados, como *EvoSuite* [Fraser and Arcuri 2014], *Randoop* [Pacheco et al. 2007], *Randoop Clean* [SILVA 2022] e *Modified EvoSuite* - uma versão customizada do *EvoSuite* por nós - para identificar conflitos nesses cenários avançados. A detecção de 230 conflitos destaca a eficácia do SMAT na identificação de conflitos semânticos em integrações complexas e enfatiza a importância de adotar múltiplas estratégias e ferramentas de teste automatizadas na análise de *software*.

A Seção 2 deste documento é dedicada a uma exploração do conceito de conflito semântico e ao papel do SMAT na identificação desses conflitos, conforme descrito por [Silva et al. 2020]. Prosseguindo para a Seção 3, detalhamos a metodologia empregada neste estudo, incluindo o desenvolvimento da nossa amostra de cenários sintéticos e a integração do *Modified EvoSuite* ao SMAT, uma ferramenta projetada para otimizar a geração de testes focando exclusivamente nos métodos alterados. Na Seção 4, avaliamos o desempenho de SMAT frente aos cenários de *merge* contidos em nossa base de dados e contrastamos os resultados alcançados neste trabalho com os encontrados em [SILVA 2022]. Na seção 5, fizemos uma análise dos trabalhos relacionados. Por fim, a

Seção 6 é dedicada à conclusão desse estudo, além de esboçar direções para pesquisas futuras.

2. Entendendo conflitos de integração

Conforme mencionado previamente, a prática de desenvolvimento de *software* tem se inclinado cada vez mais para abordagens colaborativas e trabalho em paralelo, visando potencializar a produtividade das equipes. Essa estratégia permite que os esforços individuais dos desenvolvedores sejam efetivamente combinados por meio de um processo de *merge* subsequente.

Dessa forma, adotando a terminologia utilizada por [Silva et al. 2020], podemos caracterizar um cenário de *merge* como a composição de quatro versões distintas do *software* em desenvolvimento, cada uma vinculada a um *commit* específico. O *commit Base* representa a versão compartilhada inicialmente pelos desenvolvedores. Seguem-se os *commits Parent*, denominados *Left* e *Right*, que refletem as modificações paralelas realizadas por diferentes desenvolvedores. Por último, o *commit Merge* simboliza a integração dessas alterações.

Embora a colaboração entre desenvolvedores possa aumentar significativamente a eficiência dos times, o processo de integrar os trabalhos realizados pode, por vezes, resultar em conflitos. Como mencionado anteriormente, esses conflitos se manifestam de diversas formas, incluindo os conflitos textuais, que ocorrem quando as modificações feitas por desenvolvedores incidem sobre a mesma seção do código, impossibilitando a integração. Cabe destacar as outras duas categorias de conflito citadas previamente: os conflitos de *build* e os conflitos semânticos. Os conflitos de *build* surgem de alterações em distintas partes do código que, ao serem combinadas, geram uma versão que não pode ser compilada. Já os conflitos semânticos, referem-se a divergências que não são aparentes durante a integração ou a compilação, mas que se revelam durante a execução do *software*, quando o comportamento observado diverge das intenções originais dos desenvolvedores.

2.1. Conflitos Semânticos

Para ilustrar a ocorrência de conflitos semânticos, considere o exemplo do *Quality Bank*, um banco de investimentos, que decide atualizar sua rotina de classificação de clientes para melhorar a indicação de produtos de investimento. No *commit Base* do *software*, demonstrado na Figura 1, as desenvolvedoras Lice e Rafa recebem tarefas distintas: Lice deve aprimorar o algoritmo de recomendação de produtos, incluindo a consideração do perfil do cliente, enquanto Rafa é encarregada de refinar a classificação de perfis dos clientes com base em parâmetros específicos. As versões do software de Lice (*Left*) e Rafa (*Right*) podem ser observadas na Figura 2.

```java
public class ClienteService {
    public void atualizaCliente(int clienteId) {
        Cliente cliente = repository.getCliente(clienteId);
        (...)
        cliente.setProdutosRecomendados(calculaProdutosCliente(cliente));
        processTransacoes(cliente);
        (...)
    }
    private List<Produtos> calculaProdutosCliente(Cliente cliente) {
        (...)
    }
}
```

Figura 1. Representação do código no *commit Base*

Lice

```java
public class ClienteService {
    public void atualizaCliente(int clienteId) {
        Cliente cliente = repository.getCliente(clienteId);
        (...)
        cliente.setProdutosRecomendados(calculaProdutosCliente(cliente));
        processTransacoes(cliente);
        (...)
    }
    private List<Produtos> calculaProdutosCliente(Cliente cliente) {
        (...)
        return filtraPorPerfil(produtos, cliente.getPerfil());
    }
}
```

Rafa

```java
public class ClienteService {
    public void atualizaCliente(int clienteId) {
        Cliente cliente = repository.getCliente(clienteId);
        (...)
        cliente.setProdutosRecomendados(calculaProdutosCliente(cliente));
        processTransacoes(cliente);
        cliente.setPerfil(calculaNovoPerfil(cliente));
        (...)
    }
    private List<Produtos> calculaProdutosCliente(Cliente cliente) {
        (...)
    }
}
```

Figura 2. Acima as modificações introduzidas por Lice. Em seguida, as alterações feitas por Rafa.

Após a integração das modificações (Figura 3) e o lançamento em produção, o banco começa a enfrentar reclamações de clientes que recebem recomendações de investimentos incompatíveis com seus perfis atuais. A análise do problema revela que as recomendações feitas por Lice baseiam-se em informações de perfil que são posteriormente atualizadas por Rafa, levando a sugestões desalinhadas com os perfis atualizados dos clientes.

```java
public class ClienteService {
    public void atualizaCliente(int clienteId) {
        Cliente cliente = repository.getCliente(clienteId);
        (...)
        cliente.setProdutosRecomendados(calculaProdutosCliente(cliente));
        processTransacoes(cliente);
        cliente.setPerfil(calculaNovoPerfil(cliente));
        (...)
    }
    private List<Produtos> calculaProdutosCliente(Cliente cliente) {
        (...)
        return filtraPorPerfil(produtos, cliente.getPerfil());
    }
}
```

Figura 3. Acima as modificações introduzidas por Lice. Em seguida, as alterações feitas por Rafa

Este cenário sublinha a natureza sutil dos conflitos semânticos, que podem não ser evidentes durante as fases de integração e compilação. Embora a adoção de boas práticas de desenvolvimento, como a clara definição de requisitos, revisão de código e testes robustos, possa mitigar esses problemas, [SILVA 2022] observa que, mesmo em projetos Java amplamente conhecidos que seguem essas práticas, conflitos semânticos ainda podem surgir, ressaltando a necessidade de ferramentas específicas para sua detecção.

2.2. SMAT

Na tentativa de resolver o desafio de identificar conflitos semânticos, Silva et al. [Silva et al. 2020] introduzem a ferramenta SMAT, que se vale de ferramentas de geração de testes automatizados para criar especificações parciais. Estas são posteriormente analisadas utilizando um conjunto de heurísticas específicas destinadas à detecção de tais conflitos.

Como mencionado anteriormente, SMAT emprega reconhecidas ferramentas de geração automática de testes, incluindo EvoSuite [Fraser and Arcuri 2014], Randoop [Pacheco et al. 2007] e Randoop Clean [SILVA 2022], com o objetivo de desenvolver suítes de testes unitários que reflitam as intenções dos commits *Left* e *Right*. Um exemplo de teste possível de ter sido gerado por SMAT, ilustrativo do cenário descrito anteriormente, é apresentado na Figura 4.

```java
@Test
public void test() {
    clienteService.atualizaCliente(42);
    Cliente cliente = getClient(42);
    List<Produtos> produtos = cliente.getProdutosRecomendados();
    assertTrue(produtos.allMatches(produto -> produto.getPerfil()
    == cliente.getPerfil()));
}
```

Figura 4. Exemplo de teste que visa refletir a intenção de Lice com sua mudança para adequação de perfil

Essa suíte de testes é executada nas quatro versões do software que definem um

cenário de merge. Os resultados são então avaliados segundo critérios heurísticos para detecção de conflito, que incluem:

- Testes que falham tanto no commit base quanto no merge, mas são aprovados em pelo menos um dos commits parents. O oposto também é considerado um indicativo de conflito.
- Testes que falham no commit base e nos commits parents, mas são aprovados no merge, e vice-versa.

Estes critérios servem para identificar a presença de comportamentos conflitantes decorrentes da integração de mudanças. Por exemplo, uma execução que resulta em falha tanto no *commit Base* quanto no *Merge*, mas é bem-sucedida no *Left*, indica que o *Left* introduziu um comportamento ausente no *Base* e que foi perdido no *Merge*, sugerindo uma possível interferência pelas modificações do *Right*. Por outro lado, uma execução que falha no *Base* e nos *commits Parents*, mas é bem-sucedida no *Merge*, revela que a integração das mudanças produziu um comportamento, que não foi observado nas versões anteriores.

Portanto, o teste ilustrado na Figura 4 revelaria um conflito semântico ao ser avaliado pelo SMAT. A razão é que a asserção não seria bem-sucedida tanto no *commit Base*, que carece do controle dos produtos sugeridos com base no perfil do cliente — funcionalidade introduzida pelas modificações de Lice —, quanto no *commit Merge*, possivelmente devido à desatualização do perfil utilizado para a elaboração da lista de produtos recomendados, e que posteriormente foi atualizado pelas mudanças implementadas por Rafa. Contudo, o teste seria aprovado no *commit Left*, refletindo a intenção de Lice com suas modificações. Este cenário enquadra-se no primeiro critério de detecção de conflitos do SMAT, evidenciando a utilidade da ferramenta na identificação de conflitos semânticos.

Diante disso, podemos resumir o SMAT como uma ferramenta projetada para gerar testes unitários para os *commits Parents*, utilizando um conjunto de ferramentas de geração automática de testes, a fim de capturar a intenção subjacente às modificações de cada desenvolvedor. Com os testes assim gerados, o SMAT procede à execução destes em todas as quatro versões que definem o cenário de *merge*, submetendo os resultados obtidos a uma análise com base em seus critérios. Quando ao menos um deles é satisfeito, o SMAT identifica a presença de um conflito.

Adicionalmente, propomos uma ampliação das capacidades do SMAT por meio da inclusão do *Modified EvoSuite*. Esta nova versão do *EvoSuite* é ajustada para focar exclusivamente em critérios relacionados aos métodos durante seus cálculos de algoritmos evolutivos. Isso difere da versão original do *EvoSuite*, que considera critérios aplicáveis a toda a classe. Essa modificação visa refinar a geração de testes do SMAT, ao introduzir uma ferramenta capaz de focar em métodos indicados, e dessa forma, buscando aumentar a capacidade de SMAT em detectar conflitos.

2.3. Modified EvoSuite

[Fraser and Arcuri 2014] destacam *EvoSuite* como uma ferramenta de geração automática de suítes de testes para código Java. Sua principal funcionalidade é desenvolver suítes de testes que otimizam a satisfação de diversos critérios de cobertura, como por exemplo a cobertura de linhas, *branches* e exceções, por meio do emprego de algoritmos evolutivos.

Esta abordagem permite ao *EvoSuite* selecionar as suítes de testes mais eficazes para alcançar os objetivos de cobertura definidos pelo usuário.

Um aspecto notável observado em nossas análises dos testes gerados pelo *EvoSuite* é sua tendência a incluir uma ampla gama de métodos da classe sob teste (CUT), muitos dos quais não passaram por modificações. Conforme ilustrado na Figura 5, o *EvoSuite*, na sua configuração padrão utilizada pelo SMAT, não se limita a analisar apenas as linhas de código dos métodos especificados, mas estende sua cobertura para todas as linhas dentro da CUT. Como pode ser observado pela iteração da linha 6, que passa por todos os métodos da classe, e posteriormente, adiciona cada uma das linhas desse método ao seu conjunto de objetivos, como é notado pela iteração da linha 10 e adição da linha 12. Muitas vezes resultando em testes que cubram extensivamente métodos fora do escopo de interesse, sejam priorizados em detrimento de testes que focam em áreas alvo mais restritas.

```
1   public List<LineCoverageTestFitness> getCoverageGoals() {
2     List<LineCoverageTestFitness> goals = new ArrayList<>();
3     for (String className : LinePool.getKnownClasses()) {
4       if (!isCUT(className))
5         continue;
6       for (String methodName : LinePool.getKnownMethodsFor(className)) {
7         if (isEnumDefaultConstructor(className, methodName))
8           continue;
9         Set<Integer> lines = LinePool.getLines(className, methodName);
10        for (Integer line : lines) {
11          (...)
12          goals.add(new LineCoverageTestFitness(className, methodName, line));
13        }
14      }
15    }
16    return goals;
17  }
```

Figura 5. Código de EvoSuite responsável por selecionar linhas alvo da ferramenta.

Considerando que o *EvoSuite* opera com base em um limite de tempo, gerando testes até que os objetivos de cobertura sejam atingidos ou o tempo disponível expire, identificamos uma oportunidade de aprimoramento. Modificamos a ferramenta para que concentrasse seu tempo apenas nas linhas de código dos métodos especificados pelo usuário, de forma que ela possa não perder tempo gerando testes que exercitem outros métodos da CUT.

Para tal, atualizamos a versão do *EvoSuite* e adicionamos uma forma de validar as assinaturas dos métodos, de modo que os únicos métodos que serão analisados são aqueles que indicamos na entrada usada por SMAT. Optamos por implementar essa validação através de expressões regulares, pois o *EvoSuite* utiliza as nomenclaturas dos métodos em sua assinatura ASM, isso é, como aquele método é identificado no *bytecode*, o que difere da assinatura do método no código fonte, que é a forma oferecida na entrada da ferramenta. Desse modo, optamos pelo uso de expressões regulares para simplificar a validação da correspondência de nomenclaturas.

Esta modificação visa direcionar mais precisamente o esforço de teste às áreas de interesse, potencializando a eficiência da geração de testes ao focar especificamente nas partes do código que são cruciais para a análise.

3. Metodologia

Embora os experimentos conduzidos com a ferramenta SMAT tenham apresentado resultados encorajadores [Silva et al. 2020] [SILVA 2022], eles revelam uma limitação significativa devido à análise de um número restrito de cenários. Além disso, as modificações nos *commits Parents* nos cenários avaliados limitam-se exclusivamente a alterações em um único método, de modo a não explorar sua eficiência diante de cenários mais complexos, por exemplo, aqueles em que múltiplos métodos realizam operações de leitura e escrita sobre um mesmo componente do código, como nosso exemplo citado acima no qual os métodos alterados por Lice e Rafa consultavam ou alteravam o estado do perfil do cliente.

Para abordar essa restrição e abranger cenários mais variados e complexos, este estudo introduz uma nova amostra contendo 613 cenários de sintéticos de *merge*. Esses cenários foram elaborados utilizando o *Defects4J* [Just et al. 2014] e pretendem simular situações reais de integração que envolvem conflitos semânticos, especialmente aqueles que englobam modificações em vários métodos que têm ação em um componente comum, proporcionando um espectro mais amplo de testes.

Com a ampliação da amostra, o SMAT foi ajustado para incorporar uma diversidade de ferramentas com objetivos distintos para a geração de testes automatizados. Esse arranjo visa explorar a habilidade dessas ferramentas em captar as nuances dessas integrações e, consequentemente, identificar os conflitos semânticos inerentes. Tal abordagem permite uma avaliação mais abrangente e detalhada das potenciais discrepâncias que emergem durante o processo de *merge*, destacando a capacidade do SMAT de se adaptar e responder às demandas de cenários de desenvolvimento de *software* cada vez mais intrincados.

3.1. Construção da amostra

Na elaboração da amostra, utilizamos o *Defects4J* para criar os *commits Base* e *Parents*, seguindo a metodologia proposta por [Ji et al. 2022]. Após a execução do *merge* entre os *parents*, coletamos informações cruciais do cenário, incluindo as classes e métodos modificados durante a integração. Esses dados foram então utilizados para compilar um arquivo JSON, que serve como insumo para o SMAT, facilitando a análise automatizada dos cenários de *merge*.

3.1.1. Defects4J

O *Defects4J* é um repositório que consiste em uma coleção de pares de versões de projetos Java autênticos. Cada par é composto por uma versão que contém um *bug* específico e outra versão que, derivada da primeira, inclui a correção para o *bug* em questão. Em ambas versões, existe pelo menos um teste que comprova essa correção, dessa forma, o teste falha na versão *bug* e passa na versão corrigida. A dimensão do repositório, incluindo a quantidade de versões com *bugs*, é detalhada na Tabela 1 apresentada a seguir.

Adicionalmente, o *Defects4J* disponibiliza uma funcionalidade que permite a geração de versões mutantes do código. Essa característica é particularmente valiosa para o desenvolvimento deste estudo, pois será empregada na construção de cenários de integração sofisticados.

Tabela 1. Quantidade de bugs ativos por projeto.

	Número de bugs ativos
Chart	26
Cli	39
Closure	174
Codec	18
Collections	4
Compress	47
Csv	16
Gson	18
JacksonCore	26
JacksonDatabind	112
JacksonXml	6
Jsoup	93
JxPath	22
Lang	64
Math	106
Mockito	38
Time	26

3.1.2. Geração de cenários

Para a elaboração dos nossos cenários de *merge*, adotamos uma abordagem que utiliza os pares de versões disponíveis no repositório do *Defects4J* e os testes do projeto. Nessa metodologia, a versão que contém um *bug* será designada como o *commit Base*, enquanto a versão corrigida desse *bug* representará o *commit Left*. A partir do *commit Base*, exploraremos a funcionalidade do *Defects4J* que permite a criação de múltiplas versões mutantes do código. Essas versões mutantes simulam diversas alterações que poderiam ser realizadas por desenvolvedores distintos, assumindo o papel do nosso *commit Right*.

Com esses elementos em mãos, procederemos à integração de cada versão mutante (*Right*) com a versão corrigida (*Left*), culminando na geração dos *commits Merge* (como pode ser observado na Figura 6). Essa estratégia simula um processo real de *merge* em que alterações paralelas são feitas a partir de um estado base do código e, em seguida, integradas para formar uma nova versão consolidada.

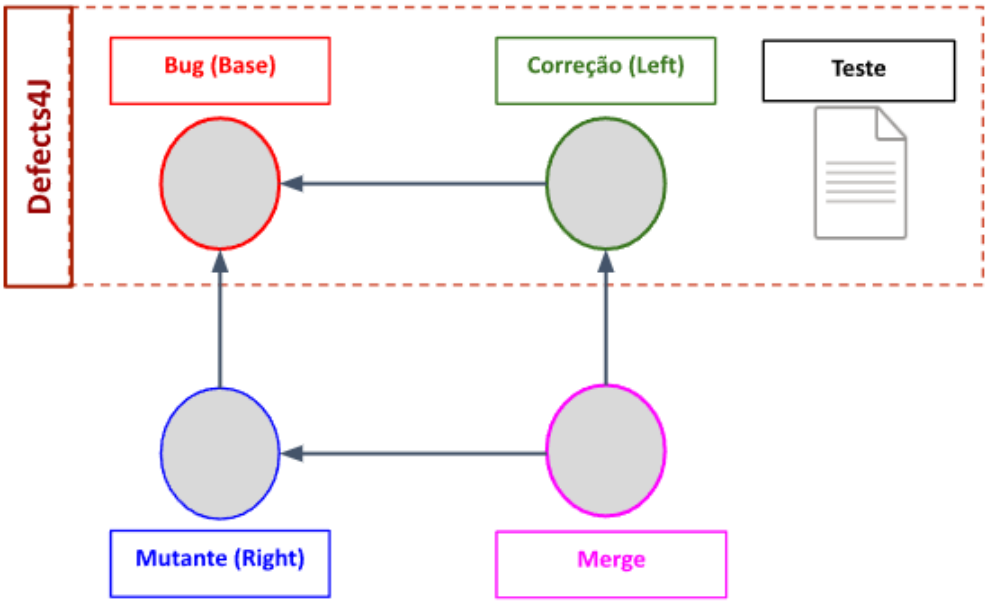

Figura 6. Processo de criação de cenário de *merge*

Para validar cada cenário de *merge* gerado, executamos a suíte de testes do *commit Left* nas quatro versões presentes no cenário (*Base*, *Left*, *Right*, e *Merge*). Utilizando as heurísticas previamente estabelecidas para a detecção de conflitos, filtramos os cenários em que foi possível identificar um conflito. Abaixo, na Tabela 2, podemos observar a quantidade de cenários gerados para cada projeto.

Tabela 2. Quantidade de cenários por projeto.

	Número de cenários criados
Chart	19
Cli	37
Closure	100
Codec	17
Collections	3
Compress	43
Csv	12
Gson	12
JacksonCore	8
JacksonDatabind	74
JacksonXml	2
Jsoup	83
JxPath	4
Lang	57
Math	97
Mockito	27
Time	18

Portanto, é importante enfatizar que todos os cenários de *merge* incluídos na amostra apresentam um conflito semântico identificado pela execução dos testes do próprio projeto. Dada essa característica, certas análises, como o cálculo da precisão (*precision*) do experimento, não são aplicáveis ou relevantes.

Após a identificação de um cenário de *merge* que envolve um conflito semântico, coletamos informações críticas para a operação da ferramenta SMAT, tais como os nomes das classes e dos métodos que sofreram modificações. Esses dados são essenciais para que o SMAT possa realizar uma análise precisa dos conflitos semânticos dentro dos cenários de *merge*.

Em seguida, procedemos com uma série de etapas em cada uma das versões dos cenários criados. Inicialmente, executamos uma rotina para alterar os modificadores de acesso das classes alvo da nossa análise, convertendo o acesso de campos, métodos ou construtores para público. Esse procedimento tem o objetivo de facilitar o acesso desses elementos pelas ferramentas de geração de testes, assegurando que possam interagir livremente com todas as partes da classe.

Em seguida, compilamos o projeto e reunimos os arquivos compilados, juntamente com suas dependências, gerando um arquivo JAR para cada versão. Esse pacote consolidado permite uma manipulação mais eficiente durante os testes e análises subsequentes.

O processo culmina com a geração de um arquivo JSON, que organiza e armazena todas as informações relevantes em uma estrutura padronizada. Nela, vemos que "targets" indica os alvos de geração de testes para SMAT, ao indicar o nome da classe e quais seus métodos alvo. É possível indicar mais de uma classe. Este arquivo assemelha-se ao modelo apresentado na Figura 7. Assim, cada cenário de *merge* é meticulosamente preparado para a análise, garantindo que as ferramentas de teste tenham acesso irrestrito aos componentes necessários para a avaliação eficaz dos conflitos semânticos identificados.

Neste estudo, utilizamos uma versão adaptada do *EvoSuite*, denominada *Modified EvoSuite*, em duas configurações distintas: uma, que chamaremos de *Modified EvoSuite Plus* quando for necessário distinguir entre as duas configurações da ferramenta, visa atender a todos os critérios de cobertura oferecidos pela ferramenta, incluindo linhas, *branches*, exceções, mutações, entre outros, para a geração da suíte de testes; e a outra foca exclusivamente nos critérios de linha e de *branches*, que correspondem aos condicionais dos métodos. O propósito dessa dualidade é explorar como a ferramenta pode eficazmente detectar conflitos semânticos ao se focar tanto em objetivos de cobertura abrangentes quanto em mais específicos e fundamentais.

```
{
    "projectName": "Mockito",
    "runAnalysis": true,
    "scenarioCommits": {
        "base": "76e0b5e14aa7c92f9bcce8b89041842b14f3fa11",
        "left": "00cd36406d48373c49aee099e7c40be1e85afcca",
        "right": "3540eec6ee9d0c96a9a96dc652f5287ee78d2584",
        "merge": "2fe0e69d566b39c43361f07172a87bad87eb6a4d"
    },
    "targets": {
        "org.mockito.internal.invocation.Invocation": [
            "expandVarArgs(boolean| Object)",
            "callRealMethod()",
            "getRawArguments()",
            "getArgumentsCount()",
            "getLocation()"
        ]
    },
    "scenarioJars": {
        "base": "/home/CIN/jaam/jars/Mockito/Mockito_36/base.jar",
        "left": "/home/CIN/jaam/jars/Mockito/Mockito_36/left.jar",
        "right": "/home/CIN/jaam/jars/Mockito/Mockito_36/right.jar",
        "merge": "/home/CIN/jaam/jars/Mockito/Mockito_36/merge.jar"
    },
    "jarType": "transformed"
}
```


Figura 7. Exemplo de entrada para SMAT

Por fim, ajustamos o SMAT para analisar os 613 cenários de *merge* criados, utilizando um total de cinco ferramentas: *EvoSuite*, *Randoop*, *Randoop Clean* e duas configurações do *Modified EvoSuite*. Cada uma dessas ferramentas recebeu 60 segundos para sua execução em cada um dos *parents* de cada cenário. O experimento, com essa configuração, foi executado em duas diferentes máquinas. Essa abordagem multifacetada permite uma análise comparativa detalhada, avaliando não apenas a eficácia de cada ferramenta individualmente, mas também o impacto das diferentes configurações de cobertura do *Modified EvoSuite* na detecção de conflitos em ambientes de *merge* complexos.

4. Resultados

Devido ao caráter não determinístico das ferramentas empregadas pelo SMAT, observou-se uma ligeira variação nos resultados obtidos pelas diferentes máquinas: uma identificou 202 conflitos, enquanto a outra registrou 200 conflitos. Para fins de análise neste estudo, optamos por considerar a união dos resultados das duas execuções.

Considerando as duas execuções do experimento, a ferramenta demonstrou capacidade para identificar 230 conflitos semânticos, isto é 37,52% dos conflitos da amostra. Como a amostra é formada apenas de casos positivos de conflito, isso corresponde ao recall, que é 0,3752. A análise dos dados, como apresentado na Tabela 3, revela que o *EvoSuite* se destacou na detecção de conflitos, identificando 185 ocorrências, sendo 76 delas exclusivas, ou seja, conflitos não detectados por nenhuma outra ferramenta. Em seguida, *Modified EvoSuite Plus* identificou 97 conflitos, com 10 deles sendo exclusivos. *Modified EvoSuite*, com foco apenas em *branches* e linhas, detectou 85 conflitos, com 5 exclusivos.

Randoop Clean conseguiu identificar 63 conflitos, apresentando 7 exclusividades e *Randoop* encontrou 55 conflitos, dos quais 5 foram únicos. Por fim, consideramos também destacar a união dos conflitos detectados pelas duas versões de *Modified EvoSuite*, capaz de identificar 109 conflitos, sendo 26 deles exclusivos; assim como a união de *Randoop* e *Randoop Clean* detectando 78 conflitos, com 16 destes sendo capturados somente por eles.

A Figura 8 exibe um diagrama de Venn, facilitando a compreensão do desempenho individual e comparativo das ferramentas na detecção de conflitos semânticos. Para tornar a análise mais acessível, agrupamos as detecções sob duas categorias simplificadas: ”Randoop”abrange os resultados obtidos tanto pelo *Randoop* quanto pelo *Randoop Clean*, e ”Modified EvoSuite”consolida os achados das duas configurações do *Modified EvoSuite* empregadas no experimento. Esse agrupamento permite uma visão clara das sobreposições e das contribuições únicas de cada ferramenta no contexto da identificação de conflitos.

Tabela 3. Conflitos detectados por cada ferramenta.

	Conflitos detectados	**Conflitos exclusivos**
Evosuite	185	76
Modified Evosuite (ME)	85	5
Modified Evosuite Plus (MEP)	97	10
Randoop (R)	55	5
Randoop Clean (RC)	63	7
ME $\cup$ MEP	109	26
R $\cup$ RC	78	16

Uma análise aprofundada dos resultados destaca o *EvoSuite* como a ferramenta mais eficiente na detecção de conflitos, com as detecções exclusivas por esta ferramenta representando 33,04% do total de conflitos identificados. Tal eficiência ilustra a vantagem de uma abordagem generalista que visa cobrir todas as linhas da classe, como mencionado anteriormente, revelando sua capacidade superior em criar testes que efetivamente exploram os conflitos. Uma possível explicação para este sucesso é que o exercício de vários métodos da classe leva à geração de um *pool* mais variado de objetos nos testes, destinados a satisfazer os critérios de cobertura de toda a classe. Essa diversidade de objetos pode ser crucial para explorar efetivamente partes dos métodos modificados, permitindo a descoberta de conflitos.

Figura 8. Diagrama de Venn dos conflitos detectados por cada ferramenta.

As configurações do *Modified EvoSuite*, embora tenham detectado um número menor de conflitos, contribuíram significativamente ao identificar 26 (11,3%) conflitos que não foram descobertos por outras ferramentas. Essa constatação enfatiza a importância de empregar ferramentas com abordagens distintas e integrar seus resultados para uma análise mais completa. A estratégia mais direcionada dessas configurações prova ser eficaz ao investigar minuciosamente os métodos modificados, desvendando conflitos que a abordagem mais genérica do *EvoSuite* poderia não perceber. Um exemplo disso ocorreu no projeto *JacksonDatabind*, especificamente na classe *TypeFactory*, que experimentou alterações nos métodos *constructType(Type, Class)* e *constructType(Type, JavaType)*. Enquanto as suítes de teste criadas pelo *EvoSuite* invocaram os métodos modificados 49 vezes, as geradas pelo *Modified EvoSuite* fizeram-no 126 vezes, examinando as mudanças de modo mais reforçado e identificando exclusivamente um conflito.

Da mesma forma, a análise das razões pelas quais as configurações do *Modified Evosuite* identificaram diferentes conjuntos de conflitos revela que uma quantidade reduzida de critérios de cobertura pode limitar a ferramenta a gerar uma menor quantidade de testes. Em alguns casos, isso pode até resultar no uso subótimo do tempo de execução disponível, uma vez que os objetivos de cobertura são rapidamente atingidos e assim a ferramenta encerra a geração. Por outro lado, uma configuração que abarca uma gama mais ampla de critérios tem o potencial de desenvolver a diversidade de testes necessária para detectar conflitos. Contudo, essa abrangência pode também diluir o foco da ferramenta, levando à geração de suítes de teste que, ao tentarem satisfazer uma ampla variedade de critérios, desviam-se do objetivo principal de descobrir conflitos. Portanto, uma abordagem que utiliza menos critérios, concentrando-se nos aspectos mais cruciais

das modificações — como linhas e *branches* — pode ser mais eficaz em revelar alguns conflitos ao focar na cobertura de elementos fundamentais do código.

Randoop Clean e *Randoop* registraram as taxas de sucesso mais baixas na detecção de conflitos. Uma análise da Tabela 4, que detalha o número de suítes de testes geradas por cada ferramenta, revela que ambas produziram a menor quantidade de suítes, o que naturalmente afeta diretamente suas capacidades de detecção. No entanto, é essencial destacar que, juntas, essas ferramentas identificaram 16 (6,95%) conflitos semânticos não detectados por outras ferramentas, sublinhando seu valor no ecossistema do SMAT. Esse resultado indica o potencial sub explorado de *Randoop Clean* e *Randoop* e sugere que a adoção de versões atualizadas ou ajustes nas configurações para aumentar a produção de suítes de testes poderia melhorar significativamente sua eficácia em pesquisas futuras.

Tabela 4. Suítes geradas por cada ferramenta.

	Suítes geradas
Evosuite	1135
Modified Evosuite	1210
Modified Evosuite Plus	1196
Randoop	906
Randoop Clean	686

Destarte, é instrutivo comparar os achados deste estudo com os de [SILVA 2022]. O experimento conduzido por Silva avaliou o SMAT configurado para utilizar o *EvoSuite*, *Randoop*, *Randoop Clean* e o *EvoSuite Diferencial* — este último não sendo objeto de análise no nosso estudo — em uma base de dados contendo 85 cenários de *merge* de projetos Java reais, com modificações limitadas a um único método. Destes, apenas 28 cenários apresentavam conflitos. A pesquisa de Silva resultou na identificação de 9 conflitos, correspondendo a 10,6% da amostra e um recall de 0,321. Comparativamente, portanto, os resultados do nosso estudo reforçam a robustez do SMAT e destacam a capacidade das ferramentas para detectar conflitos semânticos com diferentes particularidades, uma vez que mesmo diante de cenários de *merge* mais amplos e complexos, foi possível obter um recall ligeiramente maior. Isso mostra a importância de adotar múltiplas estratégias para uma detecção mais eficiente dessas discrepâncias em integrações mais desafiadoras.

5. Trabalhos relacionados

Nesta seção, exploramos trabalhos relevantes ao nosso estudo. Inicialmente, abordamos a pesquisa de [Cavalcanti et al. 2017], que realizou um estudo empírico sobre cenários de *merge*, comparando o desempenho de técnicas de resolução de *merge* variadas: não estruturada, semi-estruturada e estruturada. Contudo, a ferramenta proposta por eles não conseguiu identificar conflitos semânticos comportamentais, focando apenas em conflitos semânticos sintáticos e estáticos, ou conflitos de *build*.

[Castanho 2021] adota uma abordagem semelhante à nossa ao empregar ferramentas de geração automática de testes para desenvolver suítes de teste, usando os resultados para detectar conflitos semânticos na ferramenta UNSETTLE. Uma diferença marcante

entre nossos estudos é o critério para definir um conflito semântico. Castanho introduz o conceito de comportamento emergente, indicando conflito quando um comportamento não intencionado por nenhum dos desenvolvedores surge após a integração. Para isso, um teste deve falhar nos *commits Parents* e ser bem-sucedido no *commit Merge*, sem levar em consideração o *commit Base*, diferentemente da nossa abordagem. Além disso, enquanto buscamos criar testes como especificações parciais das intenções dos desenvolvedores, Castanho gera suítes de teste para o *commit Merge*. Ele conclui que as principais limitações das ferramentas de geração de testes são a criação de objetos relevantes para abordar problemas de integração e desenvolver asserções adequadas para examinar esses objetos em busca de conflitos, conclusões que ecoam nossas observações.

Para superar o desafio de gerar objetos complexos, [SILVA 2022] propõe uma técnica de serialização de objetos relevantes usando testes existentes no projeto. Quando um objeto passa pelo método-alvo durante a execução de um teste, ele é serializado e disponibilizado no arquivo jar final para uso na etapa de geração de testes. Optamos por não seguir essa abordagem no estudo, pois os testes do projeto são empregados no desenvolvimento da amostra sintética. Assim, utilizar esses testes, que já são conhecidos por detectar conflitos, para criar objetos em testes automatizados poderia introduzir um viés em nosso experimento.

6. Conclusão e trabalhos futuros

Os resultados obtidos neste estudo não apenas reforçam a capacidade do SMAT na identificação de conflitos semânticos em cenários de *merge* mais complexos que abarcam alterações em múltiplos métodos, mas também ampliam a compreensão sobre a importância de uma abordagem diversificada na utilização de ferramentas de teste automatizadas. Ao compararmos nossa análise com os trabalhos anteriores, como o realizado por [SILVA 2022], evidencia-se um avanço na capacidade de detectar uma gama mais ampla de conflitos semânticas, graças à avaliação de cenários sintéticos e à adaptação de ferramentas como o *Modified EvoSuite*. Este estudo contribui para o campo do desenvolvimento de software colaborativo, oferecendo insights valiosos para a melhoria contínua dos processos de integração de código e apontando caminhos para futuras pesquisas na detecção e gestão de conflitos semânticos, um desafio persistente na engenharia de software moderna.

Como destacado previamente, as ferramentas utilizadas pelo SMAT na geração de testes apresentam natureza não determinística, o que significa que seus resultados podem divergir devido a variáveis distintas durante a execução do experimento. Diante dessa variabilidade, reconhecemos a importância de uma análise mais detalhada das execuções experimentais, incorporando métricas adicionais relevantes, como média, mediana e desvio padrão, para fornecer uma compreensão mais profunda e abrangente dos resultados obtidos.

Para futuras investigações, sugerimos aprimorar as metodologias de ferramentas de geração de testes automatizados como um meio potencial de avançar na detecção de conflitos semânticos. Uma estratégia promissora envolve buscar um equilíbrio entre a abordagem generalista do *EvoSuite* e o foco específico do *Modified EvoSuite*. Isso incluiria, na geração de testes, não apenas os métodos diretamente modificados, mas também aqueles que, apesar de não sofrerem alterações no processo de *merge*, interagem com os

métodos-alvo. O objetivo dessa abordagem balanceada é otimizar a eficácia na detecção de conflitos, mesclando diversidade e precisão ao examinar áreas críticas do código.

Além disso, uma limitação observada nas ferramentas atuais é sua dificuldade em gerar testes que refletem as intenções dos desenvolvedores quando modificam certas partes do código. Esses testes muitas vezes atendem aos critérios de cobertura sem capturar variações significativas no comportamento do código entre diferentes versões do cenário, um aspecto crucial para a detecção de conflitos com base nos critérios heurísticos adotados para esse estudo. Ferramentas como o *EvoSuite Differencial*, utilizado por Silva, e o UNSETTLE [Castanho 2021] — detentor de uma variante do *EvoSuite* que prioriza a variação de estados entre versões —, mostram promessa em gerar testes mais alinhados com nossos critérios, potencialmente identificando um número maior de conflitos.

Finalmente, uma área ainda pouco explorada, mas com grande potencial, é a aplicação de Grandes Modelos de Linguagem na análise do contexto e das intenções dos desenvolvedores, traduzindo essa compreensão em testes complexos que possam efetivamente executar tais intenções. Testes com asserções detalhadas poderiam oferecer uma inspeção mais acurada se os comportamentos esperados são mantidos ou alterados nas diferentes versões testadas, abrindo novas avenidas para a detecção eficiente de conflitos semânticos em ambientes de desenvolvimento colaborativo e complexo.

Agradecimentos

Expresso minha profunda gratidão aos professores Paulo Borba e Leuson Da Silva pela sua dedicação a este trabalho, oferecendo revisões e insights preciosos que contribuíram significativamente para seu desenvolvimento. Meu agradecimento também se estende aos integrantes do Software Productivity Group, em especial a Thais Burity pelo excepcional esforço na criação das amostras sintéticas, essenciais para a realização deste estudo, e a João Pedro Duarte pelas valiosas melhorias implementadas nas ferramentas utilizadas na pesquisa. Além disso, agradeço ao INES (Instituto Nacional de Engenharia de Software) pelo apoio crucial, disponibilizando os servidores necessários para a execução dos experimentos.

Referências

Castanho, N. G. N. (2021). *Semantic Conflicts in Version Control Systems*. PhD thesis.

Cavalcanti, G., Borba, P., and Accioly, P. (2017). Should we replace our merge tools? In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 325–327.

Clementino, J., Borba, P., and Cavalcanti, G. (2021). Textual merge based on language-specific syntactic separators. In *35th Brazilian Symposium on Software Engineering (SBES 2021)*, pages 243–252.

Fraser, G. and Arcuri, A. (2014). A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2).

Ji, T., Chen, L., Mao, X., Yi, X., and Jiang, J. (2022). Automated regression unit test generation for program merges. *Science China Information Sciences*, 65(9):199103.

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA. Association for Computing Machinery.

Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. (2007). Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84.

Silva, L., Borba, P., Mahmood, W., Berger, T., and Moisakis, J. (2020). Detecting semantic conflicts via automated behavior change detection. In *36th IEEE International Conference on Software Maintenance and Evolution (ICSME 2020)*, pages 174–184.

SILVA, L. M. P. d. (2022). *Detecting, understanding, and resolving build and test conflicts*. PhD thesis, Recife, PE, Brazil.