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

Toni Maciel
Centro de Informática
Universidade Federal de Pernambuco
Brasil
jaam@cin.ufpe.br

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

Léuson Da Silva
Polytechnique Montreal
Canadá
leuson-mario-pedro.da-silva@polymtl.ca

Thaís Burity
Universidade Federal do Agreste de Pernambuco
Brasil
thais.burity@ufape.edu.br

RESUMO

Durante o desenvolvimento de software, integrar mudanças dos diferentes desenvolvedores é crucial. No entanto, essa ação pode resultar em uma versão do sistema que não preserva os comportamentos individuais pretendidos por eles, causando o que chamamos de conflitos de *merge* semânticos. As ferramentas atuais para detectar esses conflitos são limitadas a cenários mais simples, onde contribuições conflitantes ocorrem dentro do mesmo método. Para superar essa limitação, este artigo adapta e avalia uma ferramenta que detecta conflitos, considerando a interferência causada por mudanças feitas em diferentes métodos e classes. Para alcançar isso, a ferramenta explora a criação de testes utilizando ferramentas de geração de testes. Para avaliar a eficácia da ferramenta proposta, foi realizado um estudo empírico com uma amostra de 613 cenários sintéticos de *merge* criados com conflitos, representando uma amostra seis vezes maior em comparação com estudos anteriores. Como resultado, foi possível observar a detecção de 230 conflitos pela ferramenta, demonstrando seu potencial para detectar conflitos ao explorar múltiplas mudanças e apoiar ferramentas existentes. Além disso, os resultados reforçam para a importância em explorar diferentes ferramentas de geração de testes em conjunto.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*; **Software reliability**; **Software testing and debugging**.

KEYWORDS

Detecção de conflitos semânticos, Integração de múltiplos métodos, Geração de testes, Mudança de comportamento, Teste diferencial

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 de forma independente por diversas pessoas. Esse processo, embora essencial, carrega o risco de afetar negativamente a qualidade do *software* sendo construído ou a produtividade da equipe que o desenvolve [5, 23]. Isso é uma consequência natural da combinação de alterações que, individualmente, são adequadas e estão corretas, mas que, quando integradas, geram um resultado imprevisto ou indesejado.

Tais resultados indesejados podem ser categorizados com base em sua natureza linguística, ou de acordo com a fase do processo de desenvolvimento em que os mesmos acontecem. Contemplando as duas categorizações, neste artigo adotamos a terminologia utilizada por Da Silva et al. [13], que usa os termos conflitos de *merge* [9–12, 22, 28] e conflitos textuais como sinônimos. De forma similar, conflitos de *build* [6, 27, 29] referem-se a conflitos sintáticos e de semântica estática. Por fim, conflitos de semântica dinâmica, ou conflitos semânticos comportamentais, estão relacionados a conflitos de teste e produção [6]. Seguindo a abordagem de Da Silva et al. [13], por brevidade nos referirmos a este último tipo simplesmente como conflitos semânticos, omitindo o termo "comportamental". Mais adiante, exploramos mais profundamente as distinções entre esses tipos de conflitos.

Enquanto ferramentas para identificar conflitos de *merge* [9–12, 22, 28] e de *build* [14] decorrentes de integração de código são bem estabelecidas e sempre eficazes [9–11], o desafio de detectar conflitos semânticos permanece considerável. Conflitos semânticos surgem não apenas de diferenças textuais diretas, mas de mudanças potencialmente sutis no comportamento do código. Por exemplo, esse tipo de conflito ocorre 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. Por isso, a detecção desse tipo de conflito normalmente envolve maior dificuldade, particularmente quando essas alterações se distribuem por vários métodos ou classes do sistema.

Para detectar conflitos semânticos, Da Silva et al. [13] introduziram a ferramenta SMAT, que analisa alterações comportamentais em *cenários de merge*, isto é, nas quatro distintas versões de código que contemplam um processo de integração: o *commit* de *merge* em si; os dois pais imediatos desse *commit*, chamados aqui de *esquerda* e *direita*; e o mais recente ancestral comum a esses dois pais, chamado aqui de *base*. As alterações nessas quatro versões são identificadas por SMAT através de dois componentes principais: uso de ferramentas de geração automática de testes unitários que são executados nas quatro versões do código; e heurísticas para comparar os resultados desses testes gerados. Embora tenham obtido resultados positivos com SMAT, os experimentos conduzidos limitaram-se a

amostras de tamanho reduzido, e formadas apenas por modificações de *esquerda* e *direita* em um mesmo método. Dadas essas limitações, os resultados podem não capturar integralmente a complexidade inerente aos processos de *merge* em contextos práticos.

Para resolver essas limitações, criamos uma amostra pelo menos seis vezes maior que a de trabalhos anteriores na área, contendo 613 cenários sintéticos de *merge* que refletem situações realistas de integrações de código que levam a conflitos semânticos. Essa criação de novos cenários visa oferecer um extenso campo de experimentação para identificar conflitos semânticos em uma variedade de cenários mais complexos que a de trabalhos anteriores, com integrações que contemplam alterações de *esquerda* e *direita* em múltiplos métodos.

Utilizando essa amostra, conduzimos um estudo empírico para avaliar o potencial de uma versão adaptada de SMAT para detectar conflitos semânticos em cenários de merge mais complexos. Avaliamos também o potencial de geração de testes de cada uma das ferramentas de geração de testes que são utilizadas por essa versão adaptada de SMAT: *EvoSuite* [16], *Randoop* [26], *Randoop Clean* [29], e *Focused EvoSuite*— uma versão customizada de *EvoSuite* que propomos aqui para identificar conflitos em cenários de merge com mudanças integradas em diferentes métodos. Como resultado do estudo, 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 para detectar esse tipo de conflito.

Assim, como contribuições deste trabalho podemos destacar:

- Avaliação do uso de ferramentas de geração de testes para detecção de conflitos semânticos baseado em mudanças em múltiplos métodos;
- Proposta de *Focused EvoSuite*, uma extensão da ferramenta oficial;
- Disponibilidade do dataset de conflitos semânticos sintéticos e dos *scripts* usados, apoiando a replicação bem como a execução de novos estudos.

O resto deste artigo está organizado da seguinte forma. A Seção 2 ilustra o conceito de conflito semântico e explica o papel do SMAT na identificação desse tipo de conflito. Prosseguindo para a Seção 3, detalhamos a metodologia empregada neste estudo, incluindo a criação da nossa amostra de cenários sintéticos e a integração do *Focused 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 Da Silva et al. [13]. Na Seção 6, analisamos trabalhos relacionados. Por fim, a Seção 7 é dedicada à conclusão desse estudo, além de esboçar direções para pesquisas futuras.

2 ENTENDENDO CONFLITOS DE INTEGRAÇÃO

Ao longo do tempo, a prática de desenvolvimento de *software* tem adotado 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. Adotando a terminologia utilizada por Da Silva et al. [13], podemos caracterizar um cenário de *merge* como a composição de quatro versões distintas do *software* em desenvolvimento, onde cada versão está vinculada a um *commit* específico. O *commit base* representa a versão compartilhada inicialmente pelos desenvolvedores. Posteriormente, observa-se os *commits pais*, denominados *esquerda* e *direita*, que refletem as modificações individuais e paralelas realizadas por diferentes desenvolvedores. Por último, o *commit merge* simboliza a integração das alterações prévias (*esquerda* e *direita*).

Embora a colaboração entre desenvolvedores possa aumentar significativamente a eficiência dos times, o processo de integração das diferentes contribuições realizadas pode 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. É importante destacar as outras duas categorias de conflito citadas previamente: os conflitos de *build* e os conflitos semânticos comportamentais. 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 comportamentais referem-se a divergências que não são aparentes durante a integração ou a compilação, mas que são observados 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. Atualmente, o algoritmo responsável pelas indicações dos produtos financeiros baseia-se unicamente no saldo atual do cliente. No entanto, essa abordagem pode não capturar completamente as preferências e necessidades individuais dos clientes.

Diante disso, é requisitado às desenvolvedoras Lice e Rafa que aprimorem essa rotina com o intuito de trazer uma lista mais acurada de produtos, bem como refinar também outras partes da rotina, como o enquadramento de perfil do cliente. Enquanto Lice deve aprimorar o algoritmo de recomendação de produtos, incluindo uma associação entre os produtos recomendáveis para cada classificação de perfil de cliente, bem como outras adições; Rafa é responsável por refinar essa classificação de perfis com base em parâmetros específicos, como por exemplo histórico de transações, tipos de investimentos anteriores, idade, metas financeiras declaradas, e até mesmo análise de comportamento do cliente em relação às movimentações financeiras. Na Figura 1 podemos observar um possível *merge* das alterações de Lice (em vermelho) e Rafa (em verde).

Após a integração das modificações e o eventual lançamento das novas mudanças em produção, o banco começa a enfrentar reclamações de alguns clientes, que agora estão recebendo recomendações de investimentos desalinhadas com seus perfis e preferências financeiras. Esta falta de precisão nas recomendações tem gerado desconforto e insatisfação entre os clientes, que esperam receber orientações que estejam mais alinhadas com seus objetivos, hábitos e tolerância ao risco.

```
1  class ClienteService {
2   void atualizaCliente(int id) {
3    Cliente cliente = repository.getCliente(id);
4    (...)
5    cliente.setProdutos(calculaProdutos(cliente));
6    processTransacoes(cliente);
7  + cliente.setPerfil(calculaPerfil(cliente));
8    (...)
9   }
10
11  List<Produtos> calculaProdutos(Cliente cliente) {
12   List<Produtos> produtos = repository.getProdutos();
13   (...)
14 - return filtraProdutosPorSaldo(cliente.getSaldo());
15 +  produtos = filtraProdutosPorSaldo(cliente.getSaldo());
16 + (...)
17 + return filtraProdutosPorPerfil(produtos, cliente.getPerfil());
18  }
19
20  Perfil calculaPerfil(Cliente cliente) {
21  +  List<Transacoes> transacoes = cliente.getTransacoes();
22  +  int idade = cliente.getIdade();
23  +  (...)
24  +  return algoritmoPerfil(transacoes, idade, ...);
25  +  cliente.setPerfil(calculaPerfil(cliente));
26  }
27 }
```

Figura 1: Representação do *merge* das alterações de Lice e Rafa.

Uma análise mais minuciosa do problema revela que as novas recomendações, atualizadas após as mudanças de Lice, são baseadas em informações dos perfis dos usuários que são posteriormente atualizadas pelas mudanças de Rafa. Como consequência, observa-se sugestões desalinhadas com os perfis atualizados dos clientes.

Este cenário evidencia a natureza sutil dos conflitos semânticos, que pode não ser perceptível durante as fases iniciais do processo de integração de código e compilação. Eles podem ser tanto difíceis de detectar quanto de resolver. Embora a adoção de boas práticas de desenvolvimento possam evitar conflitos, como a definição de requisitos claros e atualizados, a adoção de revisão rigorosa de código e o projeto e execução de testes robustos, elas podem não ser suficientes para detectar uma parte significativa dos conflitos. Em particular, para detectar esse tipo de conflito, a revisão de código precisaria envolver um profundo entendimento semântico das modificações e suas interações, o que pode consumir bem mais tempo do que o planejado para revisão de código em alguns contextos. Por outro lado, fortes suítes de teste de integração podem detectar conflitos semânticos. No entanto, a análise prévia de projetos evidencia que os testes tal como projetados normalmente não são suficientes para detectar esse tipo de conflito [13]. Assim, espera-se que alguns conflitos não sejam capturados por essas práticas, sendo necessário ferramentas específicas para a detecção de conflitos semânticos.

```
@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 2: Exemplo de teste que visa refletir a intenção de Lice com sua mudança para adequação de perfil

2.2 SMAT

Na tentativa de apoiar a detecção de conflitos semânticos, Da Silva et al. [13] apresentam a ferramenta SMAT, que se fundamenta no uso de ferramentas de geração automática de testes unitários, como EvoSuite [16], Randoop [26] e Randoop Clean [29]. Esses testes são usados como especificações parciais para as novas contribuições realizadas pelos desenvolvedores, sendo posteriormente analisados com base em um conjunto de heurísticas específicas destinadas a orientar a detecção de conflitos. As suítes de testes unitários geradas refletem as intenções dos *commits esquerda* e *direita*, uma vez que são elaboradas com base nesses *commits*. Dessa forma, ao considerar novamente o exemplo de conflito semântico discutido anteriormente, é possível visualizar um caso de teste potencialmente gerado pelo SMAT, conforme apresentado na Figura 2. O propósito deste teste é evidenciar a intenção de Lice ao aprimorar o cálculo dos produtos recomendados, garantindo que estes estejam em conformidade com o perfil de cada cliente. Essa garantia é realizada por meio de uma asserção.

As suítes de testes geradas por SMAT são executadas nas quatro versões do software que definem um cenário de *merge*. Os resultados são então avaliados segundo heurísticas para a detecção de conflito, que incluem:

- Testes falham nos *commits base* e *merge*, mas passam em pelo menos um dos *commits pais*. A situação oposta também é considerada um indicativo de conflito.
- Testes que falham nos *commits base* e *pais*, mas são bem-sucedidos no *commit merge*, e vice-versa.

As heurísticas propostas podem ser aplicadas a qualquer teste do cenário em análise, seja um teste já existente no projeto ou gerado por ferramentas como as utilizadas pelo SMAT. Em nosso experimento, utilizamos apenas os testes das suítes geradas. Esses 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 *esquerda*, indica que o *esquerda* introduziu um comportamento ausente no *base* e que foi perdido no *merge*, sugerindo uma possível interferência pelas modificações do *direita*. Por outro lado, uma execução que falha no *base* e nos pais, mas é bem-sucedida no *merge*, revela que a integração das mudanças produziu um comportamento que não era observado nas versões anteriores.

Portanto, o teste ilustrado na Figura 2 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 esquerda*, 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 pais*, 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 *Focused EvoSuite*. Esta nova versão do *EvoSuite* é ajustada para projetar testes exclusivamente focados em critérios relacionados aos métodos que podem conter conflitos, durante seus cálculos de algoritmos evolutivos. Isso difere da versão do *EvoSuite* usada em SMAT, que considera critérios aplicáveis à toda a classe. Essa modificação visa refinar a geração de testes do SMAT, buscando aumentar a capacidade desta ferramenta em detectar conflitos.

2.3 Focused EvoSuite

Fraser e Arcuri [16] 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 3, 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 isso resulta em testes que cobrem extensivamente métodos fora do escopo de interesse, que podem ser priorizados em detrimento de testes que focam em áreas alvo mais restritas e de maior interesse para a detecção de conflitos semânticos.

Considerando que o *EvoSuite* opera com base em um limite de tempo, gerando testes até que os objetivos determinados sejam atingidos ou o tempo disponível expire, identificamos uma oportunidade de aprimoramento. Modificamos a ferramenta para que concentre seu tempo apenas nas linhas de código dos métodos passados como entrada pelo usuário, de forma que ela possa focar todo

```
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 3: Código de EvoSuite responsável por selecionar linhas alvo da ferramenta.

o seu tempo em gerar testes que exercitem os métodos de interesse do usuário. No nosso caso, o papel de usuário é exercido pelo SMAT, que passa para o *Focused EvoSuite* a lista dos métodos modificados por pelo menos um dos desenvolvedores cujas contribuições são integradas pelo cenário de merge sob análise do SMAT.

Para tal, atualizamos a versão do *EvoSuite* e adicionamos uma forma de validar as assinaturas dos métodos alvo para geração de testes, de modo que os únicos métodos que são 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 anteriormente conduzidos com a ferramenta SMAT tenham apresentado resultados encorajadores [13, 29], eles revelam uma limitação significativa devido à análise de um número restrito de cenários. Além disso, as modificações nos *commits pais* nos cenários avaliados limitam-se exclusivamente a alterações em um único método, assim não avaliando a capacidade da ferramenta diante de cenários mais complexos, como aqueles em que múltiplos métodos são atualizados e afetam um mesmo elemento de estado (variável, atributo, etc.). Este é o caso do exemplo apresentado anteriormente, 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 sintéticos de *merge*. Esses cenários foram elaborados utilizando o *Defects4J* [20], e pretendem simular situações reais de integração que envolvem conflitos semânticos, especialmente aqueles que englobam cenários nos quais as modificações de *esquerda* e *direita* contemplam diversos métodos, proporcionando uma amostra mais ampla para experimentação.

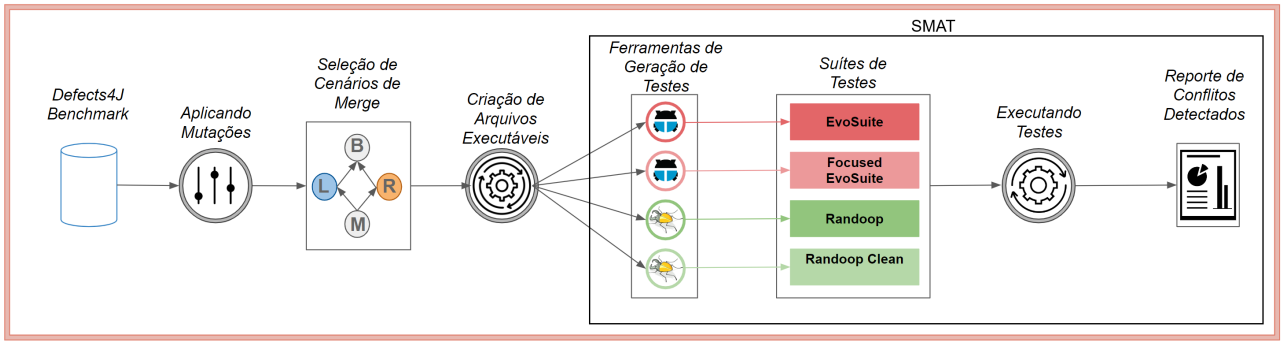

Figura 4: Metodologia adotada para o estudo

Essa amostra ampliada é então utilizada para avaliar o potencial de detecção de conflitos de uma versão estendida do SMAT que propomos aqui. Essa versão incorpora 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 conflitos semânticos. 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. A Figura 4 resume a metodologia adotada neste trabalho, detalhada nas seções a seguir.

3.1 Construção da amostra

Para a construção da nossa amostra, utilizamos o *Defects4J* [20] para construir cenários de merge que contêm conflitos semânticos. Cada unidade da amostra do *Defects4J* consiste de um *commit bug*, um *fix*, e pelo menos um teste que revela o problema, isto é, falha no primeiro e passa no segundo *commit*, respectivamente. Assim, usamos cada par de commit *bug* e *fix* do *Defects4J* como sendo os *commits base* e *esquerda* de um cenário de merge da nossa amostra, respectivamente, seguindo a metodologia proposta por Ji et al. [17]. Em seguida, aplicamos mutações ao *commit base*, gerando um novo *commit direita*, que quando integrado com o *commit esquerda* cria o commit *merge*, criando um potencial cenário para a nossa amostra. Executamos então testes do *Defects4J* nos *commits base, esquerda e merge* do cenário, e incluímos o cenário na amostra se o teste que falha em *base* e passa em *esquerda* falha também em *merge*, o que caracteriza a existência de pelo menos um conflito. Por fim, coletamos informações cruciais do cenário, incluindo as classes e métodos modificados durante a integração. Esses dados são então utilizados para compilar um arquivo JSON, que serve como insumo para o SMAT, facilitando a análise automatizada dos cenários de *merge*. A seguir, apresentamos em detalhes todos estes passos.

3.1.1 Defects4J. O *Defects4J* é um *benchmark* que consiste em uma coleção de pares de *commits* de projetos Java autênticos. Cada par é composto por um *commit* que contém um *bug* específico e outra que, derivado do primeiro, inclui a correção para o *bug* em questão. A fim de comparar a mudança de comportamento entre as diferentes versões, existe pelo menos um teste que comprova o comportamento. Para tanto, um dado teste em questão falha no *commit bug*, enquanto é bem-sucedido na versão corrigida. Dada a diversidade e quantidade de *bugs* reportados, *Defects4J* é amplamente utilizado em estudos prévios, evidenciando sua relevância [18, 32]. Sobre o status atual do *benchmark*, a Tabela 1 apresenta detalhes sobre os repositórios e número de *bugs* associados.

Tabela 1: Quantidade de bugs ativos em cada projeto e quantidade de cenários gerados por projeto.

	Qtd. de bugs ativos	Qtd. de cenários criados
Chart	26	19
Cli	39	37
Closure	174	100
Codec	18	17
Collections	4	3
Compress	47	43
Csv	16	12
Gson	18	12
JacksonCore	26	8
JacksonDatabind	112	74
JacksonXml	6	2
Jsoup	93	83
JxPath	22	4
Lang	64	57
Math	106	97
Mockito	38	27
Time	26	18
TOTAL	835	613

Dentre as diferentes funcionalidades providas por *Defects4J*, há a possibilidade de aplicar mutações no código de cada uma das versões dos *bugs*. Essa característica é particularmente valiosa para o desenvolvimento deste estudo, pois é empregada na construção de cenários de integração sofisticados.

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 associados a cada projeto. Dessa forma, a versão que contém um *bug* será designada como o *commit base*, enquanto a versão corrigida desse *bug* representará o *commit esquerda* (*fix*). A partir do *commit base*, exploramos a funcionalidade do *Defects4J* que permite a criação

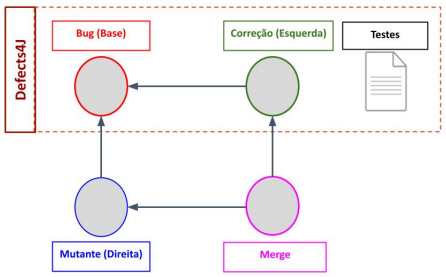

Figura 5: Processo de criação de cenário de *merge*. Os círculos indicam os *commits*, enquanto as setas representam o relacionamento de precedência entre eles.

Listing 1: Commit esquerda (fix)

```
1       String consumeToEnd() {
2   -       String data = input.substring(pos, input.length() - 1);
3   +       String data = input.substring(pos, input.length());
4           pos = input.length();
5           return data;
6       }
```

Listing 2: Commit direita (mutante)

```
1           private final String input;
2           private final int length;
3   -       private int pos = 0;
4   +       private int pos = 1;
```

Figura 6: Representação do *merge* das alterações do fix e mutante.

de múltiplas versões mutantes do código por meio do *framework* Major [19] com os seus 8 operadores padrão. Essas versões mutantes simulam diversas alterações que poderiam ser realizadas por desenvolvedores distintos, assumindo o papel do *commit direita* de um possível cenário de merge.

Para determinar quais métodos devem ser alvo da geração de mutantes, analisamos os métodos cobertos durante a execução dos testes que falham no *commit bug* e passam na versão corrigida. O objetivo é simular um processo de *merge*, sem conflitos textuais e de *build*, onde um desenvolvedor está focado na correção de um *bug* específico, enquanto outro realiza modificações por razões distintas, sem conhecimento prévio do *bug* existente. Essa abordagem captura a complexidade do desenvolvimento colaborativo de software, onde as correções de *bugs* podem ser mescladas com modificações posteriores que, embora não relacionadas ao *bug* original, têm o potencial de afetar o código de maneiras imprevisíveis.

Com esses elementos em mãos, procederemos à integração de cada *commit* mutante (*direita*) com a versão corrigida (*esquerda*), culminando na geração dos *commits merge* (como pode ser observado na Figura 5). Essa estratégia simula um processo realista 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.

Um exemplo de cenário criado foi derivado de *Jsoup*,[1] como mostrado nas alterações presentes na Figura 6. Nela observamos que o *commit esquerda (fix)* corrigiu um comportamento de *consumeToEnd* para incluir também o último carácter do *input*. O nosso *commit direita (mutante)*, por outro lado, altera o valor inicial de `pos` de 0 para 1. Essa mudança simula um desenvolvedor que não compreendeu bem onde o atributo `pos` deveria estar: se na posição a ser lida ou na imediatamente após, uma situação plausível considerando que o contexto das alterações do *commit esquerda* está relacionado com os limites de intervalo para consumir a *string*. A partir desse exemplo, podemos compreender o valor do *dataset* sintético como um bom simulador de situações reais, expandindo assim nossa capacidade de experimentação.

[1]Esse caso se refere ao *commit* de *merge* 729efc do projeto Jsoup.

Para validar cada cenário de *merge* gerado, executamos os testes que evidenciam mudança de comportamento, aqueles que falham em *bug* e passam em *esquerda*, na versão resultante da integração (*merge*). Utilizando as heurísticas previamente estabelecidas para a detecção de conflitos (ver Seção 2), filtramos os cenários em que foi possível identificar algum conflito. Na Tabela 1, podemos observar a quantidade de cenários gerados para cada projeto. Cenários de merge onde (i) conflitos textuais aconteceram durante a integração, ou (ii) o resultado da integração (*commit merge*) não era compilável foram descartados. Isto porque nossa abordagem necessita executar os testes do sistema, e assim verificar o status do teste previamente falho no *commit base*.

Portanto, é importante enfatizar que todos os cenários de *merge* incluídos na amostra apresentam pelo menos 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. Ao final deste processo, 613 cenários de *merge* com conflitos foram criados compondo a amostra deste estudo.

3.2 Geração de Arquivos Executáveis

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 um dos *commits* 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 do código da classe. Para cada um dos quatro *commits* do cenário de *merge*, compilamos o programa e reunimos os arquivos compilados, juntamente com suas dependências, gerando um arquivo JAR.

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.

3.3 Ferramentas de Geração de Testes

Neste estudo, consideramos cinco ferramentas de geração de testes integradas ao SMAT. Inicialmente, consideramos *Randoop* [26] e sua versão estendida *Randoop Clean* [29]. Em relação ao Evosuite, nós adotamos a versão oficial [16] bem como *Focused EvoSuite*, nossa versão customizada (ver Seção 2.3). É importante ressaltar que *Focused EvoSuite* provê duas configurações distintas, comumente avaliadas neste estudo:

- *plus*, que 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;
- *branches*, que foca exclusivamente nos critérios de cobertura de linha e *branches*, que correspondem aos condicionais dos métodos.

O propósito dessa dualidade é explorar como a ferramenta pode de maneira eficaz detectar conflitos semânticos ao focar tanto em objetivos de cobertura abrangentes quanto em outros 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 as cinco ferramentas. Cada uma dessas ferramentas recebeu 60 segundos para sua execução em cada um dos *commits pais* de cada cenário de *merge*. O experimento, com essa configuração, foi executado em duas máquinas por duas razões principais. Primeiro, para reduzir o tempo necessário para obter os resultados, dado que a execução completa do experimento demandaria aproximadamente quatro dias. Em uma máquina, o experimento foi realizado com a lista completa de cenários na ordem padrão, enquanto na outra máquina foi executado com a lista em ordem reversa. Dessa forma, após dois dias, em vez de quatro, obtivemos o resultado completo cobrindo todos os cenários da nossa amostra. Em segundo lugar, a execução duplicada do experimento em dois servidores distintos permitiu avaliar a natureza não-determinística das ferramentas utilizadas e como isso pode impactar os resultados. Essa abordagem revelou como os resultados podem variar, ainda que de forma sutil, mesmo em máquinas com configurações idênticas.

4 RESULTADOS

Nesta seção, apresentamos e discutimos os resultados do estudo realizado. Inicialmente, discutimos a detecção de conflitos de maneira geral, e depois, comparamos os resultados observados para cada ferramenta.

4.1 Detecção Geral de Conflitos

Devido ao caráter não-determinístico das ferramentas invocadas pelo SMAT, observou-se uma ligeira variação nos resultados obtidos pelas diferentes máquinas em que executamos o experimento. Enquanto a primeira execução reportou 202 conflitos, a segunda registrou 200 conflitos, com uma grande interseção de conflitos detectados nos dois conjuntos, mas com diferenças também. Para fins de análise neste estudo, optamos por considerar a união dos resultados das duas execuções, já que aproximaria o uso da ferramenta em um contexto prático. Considerando as duas execuções do experimento, a ferramenta identificou 230 conflitos semânticos *distintos*, representando 37,5% dos conflitos de toda a amostra.

A análise dos dados, como apresentado na Tabela 2, revela que o *EvoSuite* contribuiu mais na detecção de conflitos por parte do SMAT, identificando 185 ocorrências, sendo 76 delas exclusivas, ou seja, conflitos não detectados por nenhuma outra ferramenta. Em seguida, *Focused EvoSuite Plus* (F.E.P) ajudou SMAT a identificar 97 conflitos, com 10 deles sendo exclusivos. *Focused EvoSuite Branches* (F.E.B), com foco apenas em *branches* e linhas, contribuiu na detecção de 85 conflitos, com 5 exclusivos. *Randoop Clean* auxiliou na identificação de 63 conflitos, apresentando 7 exclusividades, e *Randoop* reportou 55 conflitos, dos quais 5 foram únicos. Por fim, consideramos também destacar a união dos conflitos detectados pelas duas versões de *Focused EvoSuite*, totalizando 109 conflitos, sendo 26 deles exclusivos; assim como a união de *Randoop* e *Randoop Clean* que contribuiram com 78 conflitos, com 16 destes sendo capturados somente por eles. Nossos resultados reforçam estudos anteriores sobre a viabilidade de agrupar diferentes ferramentas a fim de alcançar melhores resultados de deteção de conflitos [13, 29].

4.2 Comparação entre Ferramentas

A Figura 8 apresenta um diagrama de Venn para facilitar a compreensão do desempenho individual e comparativo das ferramentas

na detecção de conflitos semânticos com o SMAT. Para tornar a análise mais simples, agrupamos as detecções sob três categorias simplificadas:

- **EvoSuite**, em azul, representa as contribuições individuais de *EvoSuite*;
- **Randoop**, em roxo, abrange os resultados encontrados tanto pelo *Randoop* quanto pelo *Randoop Clean*;
- **Focused EvoSuite**, em laranja, consolida os achados das duas configurações do *Focused EvoSuite* avaliadas no experimento (*plus* e *branches*).

Esse agrupamento permite uma visão clara das sobreposições e das contribuições únicas de cada ferramenta no contexto da identificação de conflitos.

Tabela 2: Conflitos detectados por cada ferramenta.

	Conflitos detectados	Conflitos exclusivos
EvoSuite	185	76
F.E.P	97	10
F.E.B	85	5
Randoop Clean (R.C)	63	7
Randoop (R)	55	5
F.E.P ∪ F.E.B	109	26
R.C ∪ R	78	16

Uma análise aprofundada dos resultados destaca o *EvoSuite* como a ferramenta mais eficiente na geração de testes para detecção de conflitos, com uma taxa de 33,04% de detecções exclusivas do total de conflitos identificados. Esse resultado reforça estudos anteriores, que também apresentam *EvoSuite* figurando no topo das ferramentas com maior poder de gerar testes que detectam conflitos [13, 29]. 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 alto índice de detecção se dá pelo fato da ferramenta exercitar vários métodos de uma dada classe, levando à geração de um conjunto 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 nuances no código, e consequentemente, a detecção de conflitos.

Em relação às configurações do *Focused EvoSuite*, embora seus testes tenham detectado um número menor de conflitos, suas detecções contribuíram de maneira significativa reportando 26 conflitos exclusivos (11,3%). Essa constatação enfatiza a importância de empregar ferramentas com abordagens distintas e integrar seus resultados para uma análise mais completa. O EvoSuite gera suítes de teste que atendem a uma série de critérios para toda a classe. Em contraste, o Focused EvoSuite concentra seus esforços apenas nos métodos modificados. Assim, o Focused EvoSuite utiliza todo o seu tempo alocado para satisfazer os critérios de teste dos métodos modificados, onde os conflitos são mais prováveis de ocorrer. Portanto, podemos dizer que sua abrangência é menor, mas ele explora mais sistematicamente o que abrange. Um exemplo disso ocorreu no

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

projeto *JacksonDatabind*,[2] 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, *Focused EvoSuite* realizou 126 chamadas, examinando as mudanças de modo mais reforçado e identificando exclusivamente um conflito.

Da mesma forma, a análise das razões pelas quais as configurações do *Focused 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 sub-ótimo 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 3, que detalha o número de suítes de testes geradas sem erros de compilação por cada ferramenta, revela que ambas produziram a menor quantidade de suítes compiláveis, o que naturalmente afeta diretamente suas capacidades de detecção. No entanto, é essencial destacar que, juntas, essas ferramentas identificaram 16 conflitos semânticos exclusivos (6,95%), reforçando sua importância no ecossistema do SMAT.

[2]Esse caso se refere ao commit merge fffaf14 do projeto JacksonDatabind

Tabela 3: Suítes compiláveis geradas por cada ferramenta.

	Suítes compiláveis geradas
Evosuite	1135
Focused EvoSuite	1210
Focused EvoSuite Plus	1196
Randoop	906
Randoop Clean	686

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 poderiam melhorar significativamente sua eficácia em pesquisas futuras.

Destarte, é instrutivo comparar os achados deste estudo com os do experimento conduzido por Silva et al. [29], que 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 de dois desenvolvedores em um mesmo método. De toda a amostra, apenas 28 cenários apresentavam conflitos, enquanto 9 destes conflitos foram identificados, correspondendo a 10,6% da amostra e um recall de 0,32. 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 superior (0,37).

De uma forma geral, os resultados mostram o potencial do SMAT para detectar conflitos semânticos, mesmo em cenários complexos de integração de código, indo além do que foi observado em estudos anteriores. É interessante observar que, com ferramentas mais direcionadas para geração de testes— como a *Focused Evosuite* que propomos aqui— conseguimos obter na versão adaptada do SMAT taxa similar de detecção de conflitos à observada anteriormente para cenários mais simples. Os resultados também enfatizam a importância de adotar múltiplas estratégias e ferramentas de teste automatizadas para ajudar na detecção de conflitos semânticos.

5 AMEAÇAS À VALIDADE

Nosso estudo apresenta algumas ameaças que discutimos a seguir. Inicialmente, os cenários de *merge* e conflitos semânticos investigados neste estudo representam uma amostra sintética. Apesar de representarem situações realistas de integração de código, não são integrações reais, limitando o poder de generalização dos nossos resultados. Contudo, apesar dos conflitos avaliados não serem oriundos de cenários de *merge* reais, os conflitos detectados neste estudo ainda são válidos, considerando que mudanças de comportamento entre os diferentes *commits* dos cenários de *merge* podem ser observadas, por meio de um caso de teste.

Devido às restrições adotadas para a criação da nossa amostra, não é possível avaliar apropriadamente a precisão das ferramentas para a detecção de conflitos neste trabalho. Uma vez que nossa amostra é formada apenas por cenários de *merge* com conflitos semânticos, não há casos que seriam classificados automaticamente como falsos positivos (cenários sem conflitos e reportados como tais). Em relação aos casos da nossa amostra, a detecção de conflitos ocorre a partir da verificação do resultado das suítes de testes e dos critérios adotados. Assim, mesmo que os testes exercitassem código não alterado pelos mutantes, eles ainda seriam considerados como conflitos, considerando que diferentes comportamentos foram observados nos diferentes *commits* do cenário de *merge*.

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.

Com o intuito de aumentar a testabilidade do código dos cenários de *merge* em análise, transformações de testabilidade foram aplicadas alterando os modificadores de acesso de atributos, métodos e construtores para públicos. Embora estas transformações quebrem o encapsulamento do código, elas não representam uma mudança semântica. Com isso, caso um conflito pudesse ser observado diretamente por meio da chamada de um método privado, as ferramentas poderiam enfrentar desafios para indiretamente alcançarem estes métodos sem as transformações.

Nossos resultados se limitam a projetos Java e open-source. Por fim, a aplicação da abordagem deste estudo para outras linguagens de programação requer ferramentas de geração de testes apropriadas bem como ferramentas de mutação de código associadas.

6 TRABALHOS RELACIONADOS

Nesta seção, discutimos alguns trabalhos relacionados ao problema abordado em nosso estudo. Inicialmente, Cavalcanti et al. [9–11] realizam um estudo empírico sobre a ocorrência de conflitos de *merge*, comparando o desempenho de diferentes técnicas de resolução de conflitos. (*merge* não-estruturado, semi-estruturado e estruturado). Contudo, as diferentes ferramentas exploradas não podem ser usadas para detectar conflitos semânticos comportamentais, focando apenas em conflitos semânticos sintáticos e estáticos, ou conflitos de *build*. Vale ressaltar que essas ferramentas se complementam, cada uma contribuindo com sua própria abordagem para uma detecção abrangente de conflitos. Integrar essas ferramentas e utilizá-las em conjunto pode representar um caminho promissor para uma gestão mais eficaz de conflitos durante o desenvolvimento de *software*.

Brun et al. [7] e Kasi e Sarma [21] investigam a detecção de conflitos de testes. Para tanto, os autores usam as suítes de testes originais dos projetos analisados, executando-as localmente no *commit* de *merge* do cenário em análise. Uma vez que as suítes podem ser ineficientes ou incompletas, conflitos podem não ser detectados. Assim como em trabalhos anteriores, neste trabalho, exploramos o potencial de gerar novos testes e usá-los para a detecção de conflitos.

Castanho [8] adota uma abordagem semelhante ao nosso estudo, explorando ferramentas de geração automática de testes para produzir suítes de teste. Posteriormente, estas suítes são usadas para detectar conflitos semânticos por meio da ferramenta UNSETTLE, abordagem previamente explorada por Da Silva et al. [13]. Uma diferença entre nossos estudos é o critério adotado para definir um conflito semântico. Castanho [8] 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 pais* e ser bem-sucedido no *commit merge*; entretanto, eles não consideram o resultado do teste no *commit base*, como consideramos neste estudo. Além disso, enquanto buscamos criar testes como especificações parciais das intenções dos desenvolvedores, Castanho [8] gera suítes de teste para o *commit merge*. Por fim, o autor conclui que as principais limitações das ferramentas de geração de testes diz respeito à criação de objetos relevantes para abordar problemas de integração e desenvolver asserções adequadas para examinar esses objetos em busca de conflitos, conclusões que corroboram com nossas observações.

Da Silva et al. [29] também investigam a detecção de conflitos usando ferramentas de geração de testes. Buscando utilizar objetos complexos, os autores adotam a serialização de objetos relevantes como entrada para as ferramentas de geração de testes [15]. Estes objetos serializados seriam identificados por meio da execução das suítes de testes originais dos projetos avaliados. Desta forma, quando um objeto passa pelo método-alvo durante a execução de um teste, ele é serializado e disponibilizado no arquivo jar final, para então ser usado na etapa de geração de testes. Neste trabalho, nós optamos por não seguir essa abordagem, pois os testes originais do projeto já são usados para a criação da amostra sintética. Assim, utilizar esses testes novamente, que já são conhecidos por detectar as mudanças de comportamento entre as versões com e sem *bug* do *benchmark* para criar objetos em testes automatizados poderia introduzir um viés em nosso experimento.

Nguyen et al. [25] apresentam Semex, uma ferramenta que detecta quais combinações de mudanças integradas durante um cenário de *merge* causam um conflito baseado em sua técnica previamente proposta, chamada execução com reconhecimento de variabilidade [24]. Semex separa as mudanças feitas por cada *commit pai* no cenário de *merge* e codifica-as em volta de condicionais (declarações `if`), integrando todas as mudanças em um único programa. Em seguida, Semex executa os testes do projeto neste programa único, quando disponíveis, explorando todas as diferentes combinações de mudanças codificadas (declarações `if`). Entretanto, reportar conflitos baseado apenas no resultado dos testes executados apenas na versão integrada do sistema pode introduzir falsos positivos. Por exemplo, se o teste que falha no *commit merge* também falha em um dos *commits pai*, a falha no *commit merge* pode ser resultante de comportamento falho herdado do *commit pai*. Neste trabalho, nós adotamos uma abordagem mais conservadora ao reportar conflitos que exploram o resultado dos testes nos diferentes *commits* que compõem um cenário de *merge*.

Wuensche et al. [31] também investigam a ocorrência de conflitos, mas adotando uma abordagem baseada em análise estática. Baseado nas mudanças durante um cenário de *merge*, os autores propõem (re)construir um *call graph* e assim, realizar a detecção de conflitos, a partir de potenciais dependências entre os fragmentos de código do cenário de *merge*. Como resultado, 22 de 1489 cenários de *merge* foram reportados com potenciais conflitos pela ferramenta proposta. Para validar os potenciais conflitos, os autores realizaram uma busca por reportes de *bug* datados após cada cenário de *merge* com conflito; entretanto, os autores reportam que nenhum *bug* foi identificado.

Sousa et al. [30] seguem na mesma direção, propondo *SafeMerge*, uma ferramenta que explora a ausência de conflitos em um cenário de *merge* por meio de verificação composicional. Para tanto, os autores realizam um estudo empírico analisando 52 cenários de *merge*, onde 75% destes cenários estavam livres de conflitos, com uma taxa de 15% de falsos positivos. Entretanto, ao analisar alguns cenários reportados com conflitos, nós observamos que eles não representavam conflitos baseados nos critérios adotados aqui. Nestes casos, as mudanças não eram conflitantes entre si ou eram apenas *refactorings*, que não levariam a mudanças de comportamento, e consequentemente, não constituindo uma interferência.

7 CONCLUSÃO

Neste trabalho investigamos o potencial de SMAT para detectar conflitos semânticos resultado de alterações envolvendo múltiplos métodos. Os resultados obtidos reforçam a capacidade de SMAT na identificação de conflitos semânticos em cenários de *merge* complexos, 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 [13, 29], evidencia-se um avanço na capacidade de detectar um maior número de conflitos semânticos, devido à avaliação de cenários sintéticos e à adaptação de ferramentas como o *Focused 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, como a utilização de um conjunto de ferramentas que explorem a geração de testes a partir de diversas abordagens, assim como contribuímos com a elaboração de uma amostra sintética cerca de seis vezes maior que a utilizada em estudos anteriores e com integrações mais complexas, como alterações em múltiplos métodos, bem como apontamos caminhos para futuras pesquisas na detecção e gestão de conflitos semânticos, um desafio persistente na engenharia de software moderna.

Para futuras investigações, sugerimos buscar um equilíbrio entre a abordagem generalista do *EvoSuite* e o foco do *Focused EvoSuite*. Isso incluiria, na geração de testes, não apenas os métodos diretamente modificados, mas também aqueles que 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.

DISPONIBILIDADE DE ARTEFATOS

Para apoiar a replicação e a execução de novos estudos, disponibilizamos em nosso apêndice online: (i) *dataset* com 613 cenários de merge com conflitos semânticos [1] , (ii) código-fonte de *Focused EvoSuite*[2] e SMAT[3], (iii) *scripts* para a geração de cenários de merge com conflitos semânticos sintéticos[4].

AGRADECIMENTOS

Agradecemos aos integrantes do Software Productivity Group, em especial a João Pedro Duarte pelas valiosas melhorias implementadas nas ferramentas utilizadas na pesquisa. Agradecemos também ao INES (Instituto Nacional de Engenharia de Software) pelo apoio crucial, disponibilizando os servidores necessários para a execução dos experimentos, e ao CNPq (projeto 309235/2021-9), FACEPE (projetos IBPG-0546-1.03/15 e APQ/0388-1.03/14), e CAPES.

REFERÊNCIAS

[1] Apêndice Online. 2024. Disponível em: https://github.com/ToniMaciel/Syntetic-Mergesdataset.

[2] Apêndice Online. 2024. Disponível em: https://github.com/ToniMaciel/evosuite/tree/focused-evosuite.

[3] Apêndice Online. 2024. Disponível em: https://github.com/spgroup/SMAT.

[4] Apêndice Online. 2024. Disponível em: https://github.com/thaisabr/SyntheticMergeSample.

[5] Christian Bird and Thomas Zimmermann. 2012. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[6] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Crystal: precise and unobtrusive conflict warnings. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 444–447. https://doi.org/10.1145/2025113.2025187

[7] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 168–178.

[8] Nuno Guilherme Nunes Castanho. 2021. *Semantic Conflicts in Version Control Systems*. Ph. D. Dissertation.

[9] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'17)*. 59:1–59:27.

[10] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Should We Replace Our Merge Tools?. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 325–327. https://doi.org/10.1109/ICSE-C.2017.103

[11] Guilherme Cavalcanti, Paulo Borba, Georg Seibt, and Sven Apel. 2019. The Impact of Structure on Software Merging: Semistructured versus Structured Merge. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. 1002–1013.

[12] Jônatas Clementino, Paulo Borba, and Guilherme Cavalcanti. 2021. Textual merge based on language-specific syntactic separators. In *35th Brazilian Symposium on Software Engineering (SBES 2021)*. 243–252.

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

[14] Léuson Da Silva, Paulo Borba, and Arthur Pires. 2022. Build conflicts in the wild. *Journal of Software: Evolution and Process* 34, 4 (2022), e2441.

[15] Sebastian Elbaum, Hui Nee Chin, Matthew B Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 253–264.

[16] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 8 (dec 2014), 42 pages. https://doi.org/10.1145/2685612

[17] Tao Ji, Liqian Chen, Xiaoguang Mao, Xin Yi, and Jiahong Jiang. 2022. Automated regression unit test generation for program merges. *Science China Information Sciences* 65, 9 (2022), 199103.

[18] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal testing: understanding defects' root causes. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 87–99.

[19] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 international symposium on software testing and analysis*. 433–436.

[20] René Just, Darioush Jalali, and Michael D. Ernst. 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* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[21] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive conflict minimization through optimized task scheduling. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 732–741.

[22] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. 2023. A Formal Investigation of Diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science: 27th International Conference, New Delhi, India, December 12-14, 2007. Proceedings* (New Delhi, India). Springer-Verlag, Berlin, Heidelberg, 485–496. https://doi.org/10.1007/978-3-540-77050-3_40

[23] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. 2017. Software practitioner perspectives on merge conflicts and resolutions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 467–478.

[24] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. 2014. Exploring variability-aware execution for testing plugin-based web applications. In *International Conference on Software Engineering*. IEEE.

[25] Hung Viet Nguyen, My Huu Nguyen, Son Cuu Dang, Christian Kästner, and Tien N Nguyen. 2015. Detecting semantic merge conflicts with variability-aware execution. In *Symposium on the Foundations of Software Engineering*. ACM.

[26] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*. 75–84. https://doi.org/10.1109/ICSE.2007.37

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

[28] Georg Seibt, Florian Heck, Guilherme Cavalcanti, Paulo Borba, and Sven Apel. 2021. Leveraging Structure in Software Merge: An Empirical Study. *IEEE Transactions on Software Engineering* (2021), 1–1.

[29] Léuson Silva, Paulo Borba, Toni Maciel, Wardah Mahmood, Thorsten Berger, João Moisakis, Aldiberg Gomes, and Vinícius Leite. 2024. Detecting semantic conflicts with unit tests. *Journal of Systems and Software* (2024), ?–?

[30] Marcelo Sousa, Isil Dillig, and Shuvendu K Lahiri. 2018. Verified three-way program merge. *ACM Transactions on Programming Languages and Systems* 2, OOPSLA (2018), 1–29.

[31] Thorsten Wuensche, Artur Andrzejak, and Sascha Schwedes. 2020. Detecting Higher-Order Merge Conflicts in Large Software Projects. In *International Conference on Software Testing, Validation and Verification*. IEEE.

[32] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.