Usando Transformações de Código para Melhorar Detecção de Conflitos de Teste através de Ferramentas de Geração de Testes de Unidade

João Pedro Moisakis
Orientador: Paulo Borba
Coorientador: Léuson da Silva

Centro de Informática (CIn), Universidade Federal de Pernambuco

I. Introdução

Durante o desenvolvimento colaborativo de software, diferentes desenvolvedores podem realizar mudanças a um mesmo conjunto de arquivos dependentes entre si. Neste sentido, sistemas de controle de versão são amplamente utilizados com o objetivo de aumentar a produtividade do time, considerando que contribuições individuais de desenvolvedores podem ser realizadas em paralelo. Por exemplo, a criação de branches de desenvolvimento para uma dada contribuição individual, e posteriormente, sua integração com outras contribuições, processo que é conhecido como *cenário de merge*.

Apesar dos benefícios alcançados pelo uso de ferramentas de controle de versão, durante a tentativa de integrar contribuições de diferentes desenvolvedores, conflitos podem ocorrer impactando negativamente a produtividade do time e a qualidade do software desenvolvido [1], [2]. Estes conflitos podem se manifestar em diferentes estágios durante um cenário de *merge*. Conflitos de merge ocorrem já durante a tentativa falha de integração das contribuições, enquanto que conflitos de *build* e teste ocorrem durante a tentativa falha de gerar um arquivo executável do programa, que inclui o processo de compilação e a execução dos testes durante o processo de *build*. Ferramentas de *merge* convencionais apresentam limitações no reporte de conflitos de *merge* com o alto número de falsos positivos. Mesmo usando ferramentas mais inteligentes, nem todos os conflitos podem ser detectados e tratados. Assim, a resolução destes conflitos é realizada via intervenção do integrador, que ainda assim pode resolver estes conflitos e ocasionar outros problemas posteriores.

Conflitos de teste são causados por mudanças de comportamento não esperadas durante a integração, quando a *build* e o *merge* não apresentaram nenhuma falha, mas o resultado do *merge* apresenta um comportamento inesperado do sistema. Estes conflitos representam um desafio no que diz respeito à sua detecção e resolução, por seu impacto ser a nível semântico, torna-se necessário por parte de desenvolvedores um conhecimento mais profundo sobre o comportamento do sistema, estando além da capacidade das ferramentas tradicionais de *merge*. Como resultado, os impactos causados na qualidade do software podem aumentar, uma vez que estes conflitos podem ser identificados muito tardiamente no ciclo de vida do projeto. Caso sejam encontrados em um ambiente de produção, por exemplo, podem impactar diretamente o usuário final. Assim, atividades de resolução de conflitos de teste representam custos e tempo adicionais de desenvolvedores durante o processo de desenvolvimento.

Conflitos de teste poderiam ser detectados durante a fase execução dos testes durante o processo de *build*; por exemplo, ao comparar o resultado dos testes na *base*, *merge* e *commits parents*. Porém, esta abordagem nem sempre pode representar uma opção viável devido à fragilidade e baixa cobertura das suíte de testes de um projeto. Para solucionar essa limitação, ferramentas como Evosuite [3],[4] e Randoop [5] poderiam ser usadas para a criação de suítes de testes unitários para uma dada classe, e estes testes poderiam ser usados na detecção de conflitos de teste. Da Silva et al. [6] propõem a ferramenta SMAT, que para um cenário de *merge*, utiliza essas ferramentas para gerar testes unitários, executá-los e posteriormente, detectar mudanças que causam conflitos de teste. Entretanto, essas ferramentas apresentam limitações, considerando que nem sempre os testes unitários gerados podem diretamente exercitar os métodos/atributos de uma classe envolvidos nas mudanças do cenário de *merge*. Através do uso de transformações de testabilidade, podemos tornar elementos de uma classe diretamente referenciáveis pelos testes gerados sem alterar a semântica do código, permitindo a exploração das mudanças realizadas em cenários de *merge*.

Este trabalho busca criar e realizar essas transformações em cenários de *merge* com o objetivo de avaliar o código transformado através de um estudo empírico, comparando a testabilidade do código antes e depois das transformações de testabilidade utilizando a ferramenta SMAT citada anteriormente.

II. Exemplo de Motivação

Para ilustrar a dificuldade na identificação de um conflito de teste, considere como exemplo o cenário de **merge** real do projeto Fitnesse[1], representado na Fig. 1. No exemplo, a

[1] https://github.com/unclebob/fitnesse/commit/4d9ba9d

```
1  public class SlimTableFactory {
2    private static final Logger LOG = Logger.getLogger(
         SlimTableFactory.class.getName());
3    private final Map<String, Class<? extends SlimTable
         >> tableTypes;
4    private final Map<String, String> tableTypeArrays;
5    public  SlimTableFactory() {
6      tableTypes = new HashMap<String, Class<? extends
           SlimTable>>(16);
7      tableTypeArrays = new HashMap<String, String>();
8      addTableType("dt:", DecisionTable.class);
9      addTableType("decision:", DecisionTable.class);
10     addTableType("ddt:",
11         DynamicDecisionTable.class);
12     addTableType("dynamic decision:",
13         DynamicDecisionTable.class);
14     addTableType("ordered␣query:", OrderedQueryTable.
           class);
15     addTableType("subset␣query:", SubsetQueryTable.
           class);
16     addTableType("query:", QueryTable.class);
17     addTableType("table:", TableTable.class);
18     addTableType("script", ScriptTable.class);
19     addTableType("script:", ScriptTable.class);
20     addTableType("scenario", ScenarioTable.class);
21     addTableType("import", ImportTable.class);
22     addTableType("library", LibraryTable.class);
23   }
24   ...
```

Fig. 1. A integração de duas mudanças (cada parent adicionou uma das linhas destacadas) que são semanticamente conflitantes.

```
1  public void test0()  throws Throwable  {
2      SlimTableFactory slimTableFactory0 = new
           SlimTableFactory();
3      Class<LibraryTable> class0 = LibraryTable.class;
4      try {
5           slimTableFactory0.addTableType("ddt:",
                class0);
6           fail("Expecting␣exception:␣
                IllegalStateException");
7
8      } catch(IllegalStateException e) {
9          10verifyException("fitnesse.testsystems.slim.
               tables.SlimTableFactory", e);
10         assertTrue(e.getMessage().equals("A␣table␣
               type␣named␣'ddt:'␣already␣exists"));
11     }
12 }
```

Fig. 2. Caso de teste gerado pelo EvoSuite Diferencial no código não transformado da Fig. 1

```
1  public void test0()  throws Throwable  {
2      SlimTableFactory slimTableFactory0 = new
           SlimTableFactory();
3      Map<String, Class<? extends SlimTable>> map0 =
           slimTableFactory0.gettableTypes();
4      assertEquals(12, map0.size());
5  }
```

Fig. 3. Caso de teste gerado pelo Evosuite Diferencial no código transformado da Fig. 1

classe `SlimTableFactory` é o resultado de um *merge* que integra as mudanças dos commits *left* e *right*. As linhas 10 a 13 foram adicionadas pelo *commit left*, enquanto a linha 19 foi adicionada pelo *commit right*. As outras linhas são originárias do commit ancestral comum mais recente de *left e right*. Como as chamadas ao método `addTableType()` das linhas 14 a 18 separaram as mudanças a serem integradas, não existe conflito de merge sintático.

Suponha que a intenção do desenvolvedor L que estava trabalhando em *left*, foi de adicionar uma nova opção de tipo de `SlimTable` (`DynamicDecisionTable`) que poderia ser criada pela *factory*. Já o desenvolvedor R que estava trabalhando em *right*, pensou em uma nova forma de viabilizar a criação de uma `ScriptTable`. Entretanto, R não estava ciente das modificações que L fez, e a partir do momento que o código foi integrado, houve uma interferência local não prevista no comportamento na classe, consequentemente, um conflito semântico.

Como ferramentas de testes atuais não são capazes de detectar conflitos semânticos, esses geralmente se tornam custosos para detectar e resolver. A menos que o projeto tenha suites de testes robustas e um processo de revisão de código bem estabelecido, a maioria dos conflitos semânticos pode passar despercebido, e impactar os usuários finais. No exemplo apresentado, seria necessário uma análise aprofundada da intenção dos desenvolvedores e de como essas mudanças interfeririam uma na outra para que a interferência pudesse ser detectada.

A fim de reduzir o custo relacionado à detecção e resolução de conflitos semânticos, propõe-se o uso de ferramentas de geração de teste para ajudar na identificação de interferência. A ideia central seria utilizar essas ferramentas para gerar testes e identificar mudanças de comportamento. Entretanto, devido a limitações das ferramentas de geração de teste, nem sempre os elementos que estão envolvidos no conflito são explicitamente exercitados pelos testes gerados. A fim de aumentar a testabilidade da classe em análise e viabilizar o uso de ferramentas de geração de testes na detecção de conflitos semânticos, propõe-se o uso de transformações de código, que serão detalhadas posteriormente.

Para ilustrar o impacto das transformações na detecção de conflitos semânticos, utilizaremos a classe representada na Fig. 1, e exemplos reais retirados das suites de testes gerada para versão original e transformada da classe em left, Fig. 2 e Fig. 3 respectivamente.

Na Fig 2 está representado um caso de teste gerado pelo Evosuite Diferencial utilizando como entrada a classe *left* sem transformações. Podemos verificar que o teste espera que aconteça uma exceção ao tentar adicionar um `TableType` que já existe, nesse caso foi utilizado como argumento o recém adicionado em *left* `DynamicDecisionTable`. Esse teste passa quando executado em *left*, dado que a exceção acontecerá como esperado. O mesmo teste quebra quando executado em *base* como esperado, já que a `DynamicDecisionTable` identificada pela string `"ddt"` não está presente. Dessa forma é possível afirmar que o teste revela uma mudança de comportamento de *base* para *left*. Quando o teste é executado em *merge*, o teste passa, visto que a exceção seria lançada. Dessa forma, não é possível afirmar que durante o *merge* as mudanças de *right* interferem no *commit left*, uma vez que o teste não detectou uma mudança de comportamento de *left* para *merge*. Se pudéssemos encontrar testes que falham em *base*, passem em *left* e falhem em *merge*, poderíamos afirmar que as mudanças de *right* interferem em *left*. Isso é essencialmente o critério aplicado para detecção de interferência no restante deste estudo.

A Fig. 3 contém um caso de teste gerado a partir de uma versão transformada da classe *SlimTableFactory* representada na Fig. 1. Durante a aplicação das transformações de testabilidade,

```
1  public class IntegratedConfiguration(){
2
3      private final int configurationSize = 0;
4      private Configuration configuration;
5
6      public IntegratedConfiguration(Configuration
           configuration){
7          this.configurationSize = 10;
8          this.configuration = configuration;
9
10     }
11
12     boolean applyIntegratedConfiguration() {...}
13
14     private class Inner_Demo {
15         {...}
16     }
17
18 {...}
19 }
```

Fig. 4. Classe exemplo IntegratedConfiguration não transformada.

getters e *setters* são adicionados para os atributos da classe, viabilizando o acesso direto das ferramentas de geração de aos atributos. Como podemos observar na Fig. 3, linha 3, a adição do getter para o atributo `tableTypes`, permitiu que o EvoSuite Diferencial acessasse o atributo, criando um teste que o exercite. Nesse caso, ao ser executado em *left* o teste passa, visto que a assertion verifica o tamanho de `tableTypes`, e que para *left* é 12. Falha na *base*, uma vez que o tamanho do *map* na *base* é 10. E falha em *merge*, devido ao commit *right* ter inserido um novo objeto a`tableTypes`, tornando seu tamanho 13. A partir disso, podemos observar que o critério de interferência foi alcançado, e por conseguinte, o conflito semântico foi detectado graças às transformações de testabilidade.

III. Aumentando a Testabilidade do Código aplicando Transformações de Testabilidade

Transformações de testabilidade são alterações feitas no código-fonte de uma classe Java de modo a aumentar sua testabilidade, durante o processo de geração de suítes de testes por ferramentas de geração de testes. Estas transformações foram motivadas a partir do uso de ferramentas de geração de testes em um subconjunto dos cenários analisados nesse trabalho e projetos toy. Nesses experimentos, observamos que elementos modificados por dois desenvolvedores e que, possivelmente, causavam a ocorrência de conflitos de teste, não eram explicitamente exercitados pelos testes gerados. Este comportamento acontecia devido os elementos modificados terem modificadores não públicos, e, portanto, eles não poderiam ser diretamente referenciados pelos casos de testes. Assim, para aumentar a testabilidade da classe que detém os elementos modificados por dois desenvolvedores, será proposta a aplicação de transformações de testabilidade. Com isso, esperamos que as ferramentas de geração de teste de unidade, exercitem mais os os elementos modificados, aumentando a detecção de conflitos de teste.

Inicialmente, as transformações foram implementadas em *Python* utilizando bibliotecas de expressão regular para encontrar os elementos a serem transformados. Entretanto, conforme a demanda do estudo, o escopo das transformações

```
1  public class IntegratedConfiguration(){
2
3  public final int configurationSize = 0;
4  public Configuration configuration;
5
6      public IntegratedConfiguration(Configuration
           configuration){
7          this.configurationSize = 10;
8          this.configuration = configuration;
9
10     }
11
12     public IntegratedConfiguration(){};
13
14     public boolean applyIntegratedConfiguration() {...}
15
16     public class ConfigurationBuilder {
17         {...}
18     }
19
20     public int getConfigurationSize(){...}
21
22     public Configuration getConfiguration(){...}
23
24     public void setConfiguration(){...}
25
26    {...}
27 }
```

Fig. 5. Classe exemplo IntegratedConfiguration após a aplicação das transformações de testabilidade.

aumentou, e se tornou muito custoso criar expressões regulares robustas suficientes que atendessem as necessidades sem causar efeitos colaterais. Em vista disso, as transformações foram refeitas em Java utilizando o parser Eclipse JDT, para facilitar a manutenção e implementação. As transformações de testabilidade estão disponíveis em formato *jar*, sendo possível aplicá-las, a princípio, para qualquer classe Java desejada.

A seguir, estas transformações propostas são apresentadas, bem como suas motivações e como foram implementadas. É possível ver o impacto das transformações ao comparar a classe `IntegratedConfiguration` destacada nas Fig. 4 e Fig 5.

A. Transformação de modificadores de acesso de atributos e métodos

Modificadores de acesso de atributo e métodos são modificados para *public*. Esta transformação é necessária para permitir o acesso direto das ferramentas de geração de testes aos métodos e atributos da classe em análise. Pode ser observada nas linhas 3, 4 e 14 da Fig. 5.

B. Transformação de modificadores de acesso de Classes

Modificadores de acesso de classes são alterados para *public*. Esta é uma transformação necessária para atender uma limitação da ferramenta Randoop, que durante a execução dos experimentos foi possível notar que os testes não são criados no pacote da classe sendo exercitada, consequentemente, a ferramenta não consegue acessar classes do projeto que possuem modificadores de acesso não públicos. Pode ser observada na linha 16 da Fig. 5.

C. Adição de getters e setters

A adição de *getters* e *setters* permite que as ferramentas diretamente acessem ou modifiquem atributos de uma classe

em análise. Mesmo com os atributos públicos, as ferramentas apresentam dificuldades em referenciar estes atributos, assim, busca-se facilitar essas referências para que as elas passem a utilizar os *getters* e *setters* para acessar o atributo.

Getters são adicionados para todos os atributos, enquanto setters somente para atributos não *final*, uma vez que, em java atributos *final* não podem ter seu valor modificado após serem inicializados. Pode ser observada nas linhas 20, 22 e 24 da Fig. 5.

D. Adição de construtor vazio

Em alguns cenários, não é necessário criar um objeto atribuindo valores a todos os atributos, uma vez que, estes objetos nem sempre são essenciais para fazer a chamada ao método modificado, e posteriormente, detectar o conflito. Dessa forma, quando não existe um construtor vazio, sua adição permite criar um objeto simples, sem atributos inicializados, facilitando o uso das ferramentas geração de testes, pois nem sempre são capazes de criar objetos complexos, com dependências internas e externas.

Construtores vazios são adicionados somente para aquelas classes em que não tem atributos *final*, uma vez que em Java atributos *final* precisam ser inicializados nos construtores. Pode ser observada na linha 12 da Fig. 5.

IV. Metodologia

Para avaliar a testabilidade do código após aplicar as transformações de testabilidade apresentadas no capítulo anterior, um estudo empírico foi realizado, onde analisamos 50 cenários de merge de projetos Java do GitHub. Para cada um destes cenários, foram criadas arquivos *jars* para cada um de seus *commits* (*base, left, right, merge*). Inicialmente, foram gerados *jars* para os *commits* originais, e em seguida, *jars* para os *commits* com as transformações de testabilidade. Ao final, para cada cenário de *merge*, 8 arquivos *jars* foram criados. Em seguida, utilizamos os *jars* gerados no passo anterior como entrada para o SMAT, que utiliza as ferramentas de geração de teste de unidade para criar e executar teste nestes arquivos *jars*. Em seguida, SMAT verifica se o critério de interferência foi atendido, e posteriormente, o reporte de conflitos de teste. Para tanto, nosso trabalho consiste em 5 passos, representados na Fig. 6. A seguir, nós explicamos cada um destes passos.

A. Amostra

Inicialmente, selecionamos um conjunto de 50 cenários de *merge* de projetos Java hospedados no GitHub utilizados em estudos anteriores [6][7]. Estes cenários apresentam modificações em um mesmo elemento por diferentes desenvolvedores durante um cenário de *merge*. Projetos Java foram escolhidos devido a sua utilização em trabalhos relacionados, a ferramentas de geração de teste de unidade produzirem testes prioritariamente para Java e unificar a implementação das transformações em um único parser, nesse caso, o JDT.

B. Geração de Jars

Para cada um desses cenários de *merge*, *builds* com dependências foram geradas manualmente para oito versões do projeto: *merge, left, right, base* e 4 outras *builds* para suas respectivas versões transformadas. Entretanto, durante o processo de build, foram observados alguns problemas, posteriormente corrigidos. Entre eles, estão:

1) Dependências não disponíveis: Em alguns projetos, os repositórios de dependências não foram capazes de encontrar as versões das dependências declaradas nos *commits* do cenário de *merge*, causando a falha na *build*. Portanto, para que fosse possível utilizar os cenários no estudo, as dependências foram atualizadas para versões mais recentes disponíveis nos repositórios.

2) Falhas ocasionadas pelas transformações: Durante o processo de geração dos *jars* transformados para o cenário de *merge*, foi possível observar que em determinados projetos, o *jar* das transformações causou erros de sintaxe. Esse problema foi ocasionado devido à falta de robustez das transformações, consequentemente acarretando na falha da *build*. Os problemas reportados estão relacionados a alterações no código que violavam a sintaxe de herança. Isso ocorre devido a uma limitação das transformações, que, na versão atual, é capaz de alterar apenas uma classe por vez. Estes erros aconteceram quando a classe a ser transformada está inserida em uma hierarquia de herança; logo, a adição de um construtor vazio em uma subclasse só é válida se há um construtor vazio na super classe. Nestes casos, foi necessária a remoção manual do recém adicionado construtor vazio.

No mesmo sentido, quando a classe transformada está inserida numa hierarquia de herança, a mudança dos modificadores de acesso de *protected* para *public* só é válida se suas subclasses também forem transformadas, visto que os métodos das subclasses que sobrescrevem os métodos da superclasse também devem ser públicos. Nesse caso, para solucionar o problema, foi necessário alterar o modificador de acesso desses métodos para *public*.

Por fim, quando o projeto adota ferramentas de análise estática, algumas checagens podem falhar durante o processo de *build* após a aplicação das transformações. Por exemplo, ferramentas de checagem de estilo de código, como o *Checkstyle*, que checa se o código fonte está de acordo com as regras de codificação previamente estabelecidas. Deste modo, o descumprimento destas orientações resulta na falha do processo de *build*. Para este estudo, o descumprimento destas orientações de estilo de código após a aplicação das transformações não é relevante, visto que estas orientações não tem efeito na testabilidade do código. Diante disso, as declarações de uso destas ferramentas foram manualmente removidas dos arquivos POM, e posteriormente, nenhuma checagem foi realizada durante o processo de *build*.

C. Geração e Execução de Testes

Em seguida, para cada conjunto de *jars* resultantes do passo anterior foram agrupados por cenário de *merge* e usados como entrada para a ferramenta SMAT. Essa ferramenta utiliza as ferramentas de geração de teste de unidade para gerar testes

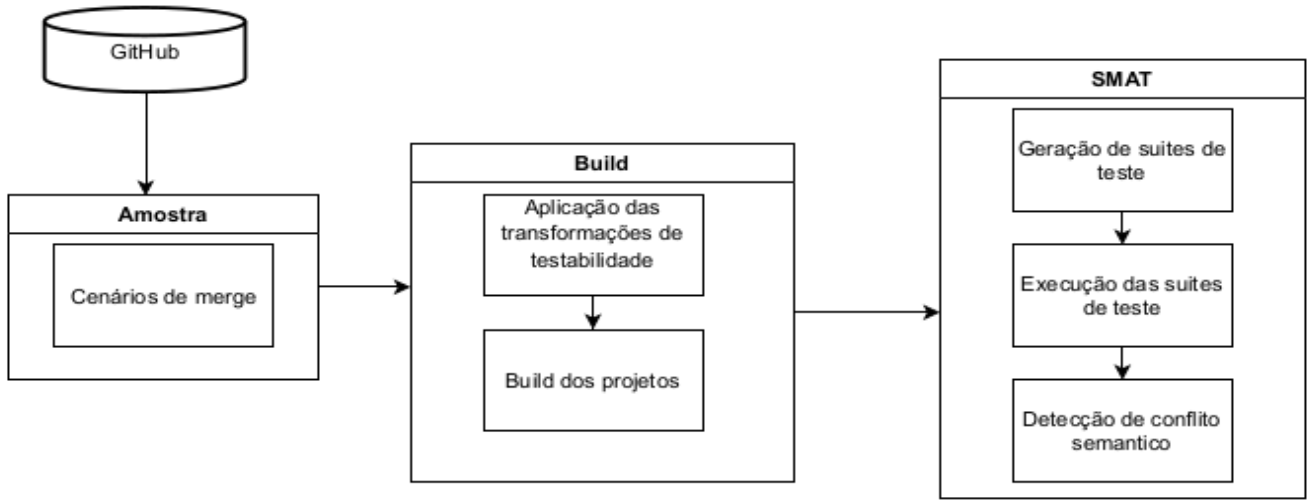

Fig. 6. Descreve as etapas da metodologia utilizada neste estudo.

baseados nos arquivos *jars* de um cenário de *merge*. Esses testes são executados nas diferentes versões do cenário de acordo com um critério de interferência, e a partir de seus resultados podemos identificar conflitos de teste. Inicialmente, SMAT aplica Evosuite, Evosuite Diferencial e Randoop para os arquivos *jars* dos commits pais de um cenário de *merge* (*left* e *right*, e suas respectivas versões transformadas). Em seguida, as suítes de testes geradas são executadas nos *commits base, merge* e *commit* pai usado para a criação das suítes.

Esse processo é repetido três vezes, obtendo 12 suítes de teste para cada ferramenta, portanto 36 suítes para cada cenário de *merge*. Para cada suíte gerada, os casos de testes são executados três vezes para cada uma das versões, *base, parent* e *merge*, resultando em 9 execuções, ou seja, para cada cenário de *merge* com 36 suites, são executados nove vezes, no total, 324 execuções.

D. Detecção de Conflitos

Essas execuções são agrupadas em conjuntos de 81 execuções, e associados com as versões em que foram executados (*base, left, right e merge)*. Para cada conjunto, SMAT registra os testes que obtiveram o mesmo resultado nos *commits base* e *merge*, mas um resultado diferente no commit pai (*left* ou *right*). Estes registros são utilizados para verificar se o critério de interferência foi alcançado, e posteriormente reportar o conflito de teste. Para um conjunto de execuções, devem existir testes que passem no *commit parent*, mas falhem na *base e merge*. O mesmo se aplica para os casos em que os testes falham no *parent*, mas passam nos commits *base e merge.*

E. Ground Truth

No estudo de Da Silva et al. [6] foi feita uma análise manual dos cenários tentando estabelecer um ground truth de interferência, de modo a avaliar a corretude da detecção de conflitos de teste. Para isso, foram coletadas informações sobre falsos positivos(casos em que não há interferência, mas o critério de interferência é observado) e falsos negativos (casos em que há interferência, mas o critério não é observado) com o objetivo de entender o potencial das ferramentas de geração de teste e do critério de interferência.

Para os cenários de *merge* em que foi identificado interferência, tentaram criar um teste que poderia revelar o problema, a fim de reduzir erros humanos e possíveis equívocos. Similarmente, nos casos em que não foram encontrados indícios de interferência, o mesmo foi feito, entretanto nem sempre foi possível criar tal caso de teste. Por exemplo, quando uma das modificações foi uma refatoração estrutural que não afeta o comportamento do código integrado.

Com os resultados reportados pelo SMAT, cada cenário foi analisado manualmente. Para os cenários de *merge* reportados com conflito, suas suites de teste foram analisadas para ter certeza que os testes recém criados exploram a interferência identificada durante a análise manual. Essa análise é essencial para que se possa identificar possíveis falsos positivos introduzidos pelas transformações de testabilidade, já que é possível que algumas delas tenham alterado a semântica do código. Para isso, as asserções que falharam nos casos de teste foram verificadas, a fim de observar se exploram os efeitos colaterais ocasionados pelo conflito.

Durante a análise, dois falsos positivos foram identificados, um caso antes das transformações e um após. Foi possivel observar que apesar do critério de interferência ter sido seguido, suas asserções para *base* e *merge*, falham em diferentes pontos do código, que não foram explorados pelas mudanças dos desenvolvedores. Isso pode ter ocorrido devido a falhas e limitações das ferramentas de geração de teste, visto que não puderam diminuir o escopo dos testes.

Com objetivo de entender as limitações das ferramentas de geração de teste de unidade Da Silva et al. [6] analisou as suítes de teste que tiveram como resultados falso negativos. Baseado

nos testes criados durante a fase de análise do ground truth, tentaram modificar os testes gerados de modo a identificar a interferência. Com isso, melhorias puderam ser identificadas, além de auxiliar o entendimento do quão perto as ferramentas estariam de criar um teste que revelaria o conflito.

V. Resultados

Agora serão apresentados os resultados da execução do estudo nas 50 integrações de código, presentes nos 48 cenários de *merge* de projetos Java retirados do Github. Em seguida, será apresentado o processo de detecção de conflitos de teste utilizando ferramentas de geração de teste e o critério de interferência. Por fim, será avaliado o potencial das transformações em aumentar a testabilidade do código.

Recapitulando, com o objetivo de verificar o potencial de testes de regressão para detectar conflitos de teste, nós selecionamos 50 integrações de código de 48 cenários de *merge* de projetos hospedados no GitHub. Em seguida, geramos arquivos *jars* para cada um dos *commits* dos cenários de *merge*, bem como aplicamos transformações de testabilidade. Por fim, utilizamos SMAT para criar e executar suítes de teste de forma automatizada. Inicialmente, serão discutidos os casos com conflitos de teste, e em seguida, os casos sem conflitos. Por fim, discutimos o impacto das transformações na melhora da testabilidade.

A Fig. 7 ilustra os resultados da detecção de conflitos em nossa amostra; o ramo esquerdo agrupa as integrações que apresentam conflitos de teste, enquanto o ramo da direita representa as integrações sem conflitos. No total, pudemos detectar automaticamente três conflitos de teste (em dois cenários de *merge*) de 50 integrações (6%). Pode-se observar que no ramo esquerdo (Fig. 7), com base na análise manual realizada previamente nas integrações, que dos 15 conflitos de teste, 3 foram detectados, enquanto os 12 restantes foram falsos negativos.

Voltando ao exemplo da Fig. 1, que mostra uma integração retirada do cenário de *merge* do projeto Fitnesse, aplicado ao código transformado. Nesse cenário de merge, o *commit left* adicionou duas novas chamadas ao metodo `addTableType`(Fig. 1 , linhas 10 a 13) passando como parâmetro `DynamicTable`, enquanto o *commit right* adicionou uma nova chamada ao metodo `addTabletype` passando como parâmetro uma `ScriptTable` (Fig. 1 linha 19).

Left e right modificam o comportamento do programa individualmente (adição de diferentes objetos ao `map` de `TableTypes`) de acordo com suas necessidades no momento do commit. Essas mudanças puderam ser integradas sem a ocorrência de conflitos de merge (Fig. 1 linhas 14 a 18 separam as mudanças) e *build* (programa resultante compilável). A menos que exista uma suíte de teste no projeto que explore o comportamento desta classe, em particular, seu atributo `tableTypes`, este conflito não seria detectado.

Na Fig. 3, pode-se observar o teste gerado pelo Evosuite Diferencial, usando o *commit left*. Como destacado no caso de teste (Linha 3 Fig. 3), a asserção espera que o tamanho do objeto `tableTypes` seja 12. O caso de teste passa no *commit left* como esperado. Para o *commit base*, o teste falha, uma vez que neste commit, o tamanho do `map` é 10. Isso é esperado acontecer, considerando que o *commit left* deseja alterar o comportamento do programa em relação a base. Analogamente, no *commit* de *merge*, o caso de teste também falha, mas devido a integração com as mudanças feitas no *commit right*, pois o tamanho do map seria 13 com a modificação feita no *commit right* (Fig. 1 linha 19) .

Os outros dois conflitos detectados tem alguns aspectos comuns ao exemplo discutido. Primeiro, o conflito ocorre porque os *commits* pais modificam o mesmo objeto. Segundo, as ferramentas tinham acesso direto ao objeto envolvido no conflito. Consequentemente, o caso de teste deve ter ao menos uma asserção explorando o conteúdo desse objeto para que o conflito seja detectado.

Para esses conflitos, aplicar transformações de testabilidade mostrou-se promissor para a detecção do conflito de teste, visto que as transformações aumentam a testabilidade do código. Após a aplicação das transformações de testabilidade, as ferramentas puderam ter acesso direto aos objetos, e utilizá-los em suas asserções. Isso levou a detecção de um conflito adicional comparado a execução das mesmas ferramentas na mesma amostra, mas sem as transformações aplicadas.

Em Da Silva et al. [6] foi feita uma análise sobre os falsos negativos, buscando o porquê de não terem sido detectados. Em seguida, foram aplicadas algumas mudanças nos casos de teste visando viabilizar o acesso aos elementos envolvidos no conflito, dessa forma, os conflitos puderam ser detectados em sete das 12 integrações com falsos negativos. Durante essa análise foi possível observar que mesmo para falsos negativos, as suítes puderam alcançar o objeto que revelaria o conflito, mas falharam em criar asserções relevantes que explorassem o conteúdo do objeto. Para esses casos, as ferramentas chegaram perto de detectar o conflito. Em outros casos, não obtiveram o mesmo sucesso. Os métodos que detinham o conflito foram chamados, mas os argumentos passados impediram que os testes explorassem o local da interferência, por exemplo: O método foi chamado com argumentos `null`, que ocasionava exceções antes de alcançar o conflito. O mesmo pôde ser observado para os 2 casos de falsos positivos, nos quais os testes executados no *commit base e merge*, falharam em asserções que não eram relevantes para a identificação do conflito, devido a falta de robustez das ferramentas de geração de teste.

Focando no ramo direito da Fig. 7, temos as 35 mudanças sem conflitos, considerados os verdadeiros negativos. Os cenários de *merge* contidos neste conjunto, podem ser classificados em grupos:

A. Falta de suporte para geração e execução de testes

Alguns cenários foram considerados não suportados pelas ferramentas de geração de teste, por as mudanças envolvidas no merge estão localizadas em classes de teste. Os testes não puderam ser gerados em função de que o *framework* de teste associado e o ambiente de execução não foram providos. Entretanto, foi possível observar através da analise manual e critério de interferência, que esses casos não tem conflito de teste.

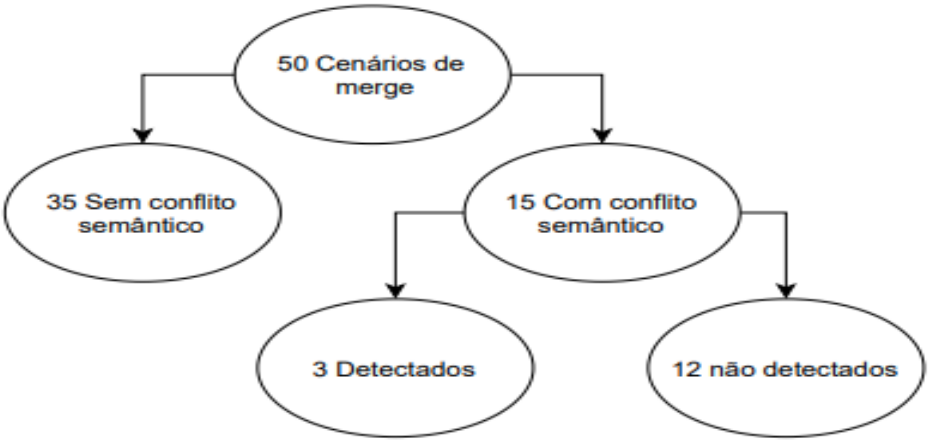

Fig. 7. Resultados da detecção de conflitos semânticos

.

B. Alteração individual da semântica

Casos em que os parents (*left e right)* tiveram alterações semânticas individualmente, mas quando integrados, não tiveram interferências uns nos outros localmente. Entretanto, é possível que essas modificações tenham interferências globais. Nesses casos, não é esperado que seja reportado conflito, já que o estudo foca em interferências locais .

C. Refatorações estruturais

Como mencionado anteriormente no passo de estabelecimento do ground truth, mesmo que um dos commits parents alterem a semântica do código, não e esperando um conflito de teste.

Foi feita uma análise quantitativa dos resultados, visando entender o quanto as transformações impactam na qualidade das suites de teste geradas pelas ferramentas. Os resultados foram observados e avaliados por duas métricas: número de casos de teste gerados e número de chamadas ao método/atributo com potencial conflito.

Na execução das 50 integrações, em 9 casos não foi possível gerar testes, mesmo com a aplicação das transformações de testabilidade. Isso pode ter se dado devido a uma sequência de fatores como, limitações de recursos da máquina em que os testes foram gerados, e, como mencionado anteriormente, limitações das ferramentas de geração de teste.

As 41 integrações restantes aplicamos a análise quantitativa por meio das métricas informadas anteriormente. Em seguida, comparamos os resultados destas métricas entre as suítes geradas para os *jars* com e sem transformações. Avaliando a testabilidade das transformações do ponto de vista de número de casos de teste, observa-se que em 32 casos houve um aumento no número de casos de teste, em 8 casos houve uma diminuição, e em 1 caso o número de testes foi inalterado.

A piora no número de casos de teste gerados, nesses 8 casos, pode ser atribuída à natureza não determinística das ferramentas de geração de teste. Apesar de o estudo ter sido executado sequencialmente na mesma máquina para versões originais e transformadas, é possível que por uma falta de recursos no momento da execução, as ferramentas tenham performado ligeiramente pior em certos momentos, causando a diminuição de testes gerados. Entretanto, em 3 dos 8 casos em que houve diminuição da quantidade de casos de teste foi possível observar um aumento no número de chamadas ao método envolvido na integração, o que de certa forma contribui para a validação dessa hipótese.

Em relação ao número de chamadas ao método ou atributo com potencial conflito de teste, das 41 integrações, houve aumento em 28 casos, em 7 o número de chamadas foi inalterado, e em 6 observou-se uma redução. Apesar da diminuição no número de chamadas para 6 casos, observou-se que para esses casos, o número de chamadas estava diretamente atrelado a quantidade de testes gerados; em somente 1 houve uma diminuição expressiva.

VI. Ameaças à Validade

Este trabalho tem como foco principal a detecção de conflitos de teste através de modificações localmente observáveis. Assim, acreditamos que seria custoso e complexo analisar todo o projeto buscando por eventuais interferências durante um cenário de *merge*. Contudo, é possível que as modificações envolvidas no processo de integração tenham interferências globais não-previstas, e, consequentemente, apresentar uma diferença no número de falsos positivos e falsos negativos comparado ao que foi reportado nos resultados. Entretanto, interferências globais podem ser detectadas se gerarmos testes para outras classes além das envolvidas no processo de integração.

Com o objetivo de aumentar a testabilidade do código sendo analisado pelas ferramentas de geração de teste de unidade, foram aplicadas transformações de testabilidade. Por exemplo, ao transformar os modificadores de acesso de uma classe para *public* não altera o programa semanticamente, mas permite que os elementos da classe possam ser alcançados pelas ferramentas de geração de teste. Em uma situação em que o conflito semântico pode ser observado em um campo privado de uma classe,

as ferramentas poderiam enfrentar problemas ao tentar acessar esse atributo indiretamente sem a aplicação das transformações.

Quando estabelecendo o ground truth da interferência, saber os resultados das ferramentas de geração de teste antes da análise manual pode ter influenciado o veredito. Por exemplo, ao saber que para alguns cenários as ferramentas foram mal sucedidas, há o risco de impossibilitar uma análise mais elaborada. Para reduzir essa ameaça, em Da Silva et al. [6] para cada cenário foi elaborada uma explicação do porquê não haver interferência. Neste sentido, o risco é significativamente reduzido para os casos em que as ferramentas foram bem sucedidas, já que foi feito uma análise de interferência, execução dos testes, e verificações manuais se as asserções no caso de teste exercitam os elementos modificados.

Os Resultados apresentados são específicos para projetos Java *open-source* hospedados no GitHub. As transformações de código, como discutimos, impactam nossos resultados positivamente e contribuem para aumentar a testabilidade do código, e em alguns casos detectar o conflito. Entretanto, observou-se a ocorrência de falsos positivos, que mesmo seguindo o critério de interferência não apresentam conflitos de teste reais. Para mitigar esse risco, os dados coletados por estudos anteriores podem ser utilizados para melhorias nas ferramentas, aumentando a chance de criar asserções relevantes para o elemento modificado. Para aplicar esse estudo para outras linguagens de programação, seria necessário ferramentas de geração de teste para a linguagem desejada e transformações de testabilidade que se apliquem aos contextos dessas ferramentas.

VII. Trabalhos Relacionados

Sarma et al.[1] propõe a ferramenta Palantir, que notifica os desenvolvedores caso haja uma mudança paralela ao mesmo artefato, visando minimizar a ocorrência de conflitos. Brun et al. [8] propõe incorporar análise especulativa para detectar e prevenir conflitos. Com essa finalidade, eles analisaram projetos Java, e utilizaram suites de testes dos projetos, que nem sempre se mostram robustos o suficiente, como mencionamos anteriormente. Se a *build* dos *commits* falha devido a uma falha nos testes, consideram o cenário como conflito semântico. Os testes não são executados nos *commits base* e *parent* o que resultaria em um número maior de falsos positivos , visto que as falhas nos testes podem ter sido ocasionadas por mudanças de um só *commit parent*.

Da Silva et al.[6] propõem um estudo similar a este, utilizando ferramentas de geração de teste de unidade e transformações de código de modo a melhorar a detecção de conflitos semânticos. Os autores investigaram a detecção de conflitos de teste utilizando ferramentas de geração de teste automatizado. Para tanto, eles aplicaram um critério de interferência sobre uma amostra de cenários de *merge* de projetos Java hospedados no GitHub. Combinando ferramentas de geração de teste com transformações de testabilidade no código a ser analisado, os autores mostraram a viabilidade de uma ferramenta de merge semântico baseado na geração de testes de regressão. Por fim, os autores discutem melhorias que poderiam ser aplicadas às ferramentas de geração de testes. No entanto, os autores não exploram as métricas gerais das suites de teste geradas, como a análise quantitativa de testabilidade do código transformado que foi exposta neste estudo. Comparativamente, os estudos compartilham parte do *dataset*, o processo de execução das ferramentas e critérios de interferência, entretanto houveram diferenças nos resultados. O subconjunto do *dataset* referente aos cenários que apresentam conflitos semânticos foi comum para ambos os projetos, contudo, a execução do experimento em Da Silva et al.[6] foi capaz de detectar um conflito a mais no projeto *storm*. Essa diferença pode ter ocorrido a natureza não determinística das ferramentas de geração de teste, além de falhas que podem ter ocorrido durante a execução. Apesar disso, foi possível chegar a uma mesma conclusão quanto à melhora da testabilidade do código avaliado através do uso de transformações de testabilidade.

VIII. Trabalhos Futuros

Como mencionado anteriormente, as ferramentas de geração de teste de unidade apresentam diversas limitações, visando solucionar esses tipos de situações foram implementadas transformações de testabilidade. Entretanto, devido à falta de robustez das transformações, problemas relacionados à violação da sintaxe de herança foram reportados, necessitando intervenção manual. Em função da indisponibilidade de tempo para a conclusão da dissertação, recomenda-se para trabalhos futuros melhorias no código das transformações, além de um estudo mais aprofundado das limitações das ferramentas de geração de teste, e implementação de melhorias para as mesmas.

IX. Conclusão

Em um contexto colaborativo de software, a utilização de branches de desenvolvimento e integração de código são práticas comuns no dia a dia de desenvolvedores. Apesar de aumentar a produtividade dos desenvolvedores, o surgimento de conflitos à medida que as modificações são integradas tendem a ser custosos. Ferramentas de merge convencionais tendem a prevenir diversos conflitos, entretanto são limitadas a conflitos sintáticos, enquanto conflitos semânticos são mais difíceis de serem detectados e corrigidos. Este estudo abordou a detecção automatizada de conflitos de teste utilizando um critério de interferência estabelecido por estudos anteriores. Ferramentas de geração de teste foram combinadas a transformações de testabilidade a fim de melhorar a testabilidade da classe sendo analisada. Foi feita uma análise quantitativa das suites de teste geradas, para avaliar o aumento da testabilidade após a aplicação das transformações. Como resultado do experimento foi possível detectar três conflitos entre 50 integrações, isso demonstra que utilizar ferramentas de geração de teste associados ao critério de interferência poderia ajudar os desenvolvedores a detectar conflitos de teste mais cedo. A aplicação das transformações viabilizaram a detecção de conflito em um dos 3 casos apontados com interferência, apesar de percentualmente bom, o ganho foi de somente um caso. Avaliando a testabilidade das transformações do ponto de vista de número de casos de teste e número de chamadas ao método ou atributo com potencial conflito de teste pode-se observar

uma melhora na maioria das integrações, para esse fim, as transformações se mostraram bem promissoras.

REFERENCES

[1] A. Sarma, D. F. Redmiles, and A. Van Der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 889–908, 2012.

[2] S. McKee, N. Nelson, A. Sarma, and D. Dig, "Software practitioner perspectives on merge conflicts and resolutions," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 467–478.

[3] M. M. Almasi, H. Hemmati, G. Fraser, and A. Arcuri, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *39th International Conference on Software Engineering, ICSE 2017, Software Engineering in Practice Track*. IEEE, 2017, pp. 263–272.

[4] G. Fraser, "A tutorial on using and extending the evosuite search-based test generator," in *Search-Based Software Engineering*. Springer, 2018, pp. 106–130.

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

[6] L. Da Silva, P. Borba, T. Berger, and J. Moisakis, "Detecting semantic conflicts via automated behavior change detection," *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020.

[7] R. S. Barros Filho, "Using information flow to estimate interference between developers same-method contributions," 2017. [Online]. Available: https://twiki.cin.ufpe.br/twiki/pub/SPG/GenteAreaThesis/diss_-_rsmbf_-_final.pdf

[8] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, 2013.