Reestruturação e aperfeiçoamento de uma ferramenta para detecção de conflitos semânticos de código

João Pedro Henrique Santos Duarte[1]

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

jphsd@cin.ufpe.br

Abstract. *In software development, in order to maximize collaboration and parallelize development, it is common to use branches which are later integrated through merges. One of the main problems in this integration are merge conflicts, which negatively affect the project due to the cost involved in its resolution. Although there are tools capable of detecting textual conflicts, semantic conflicts - where changes in software behavior occur - still lack efficient detection tools. The present work proposes to enhance a semantic conflict detection tool based on automated tests, as well as restructure and improve overall code quality of it.*

Resumo. *No desenvolvimento de software, a fim de maximizar a colaboração e o desenvolvimento em paralelo, é comum a utilização de branches que são posteriormente integradas através de merges. Um dos principais problemas nesta integração são os conflitos de merge, que afetam o projeto negativamente pelo custo envolvido em sua resolução. Embora existam ferramentas capazes de detectar conflitos textuais, conflitos semânticos - onde ocorrem mudanças no comportamento do software - ainda carecem de ferramentas eficientes de detecção. O presente trabalho propõe o aperfeiçoamento de uma ferramenta de detecção de conflitos semânticos baseada em testes automatizados, como também uma reestruturação com o objetivo de incrementar a qualidade do código.*

1. Introdução

O desenvolvimento de *software* é um processo essencialmente colaborativo. Em diversos projetos, as tarefas são distribuídas entre os desenvolvedores, que trabalham utilizando o conceito de *branches*. *Branches* permitem que a implementação de diferentes funcionalidades do *software* sejam realizadas de maneira independente sendo posteriormente integradas através de processos de *merge*.

Embora esta alternativa facilite o gerenciamento do projeto, *branches* e *merges* trazem com si o surgimento de conflitos de *merge*, que ocorrem durante o processo de integração das modificações de duas *branches* distintas. Alguns destes conflitos já são detectados por ferramentas existentes atualmente, entretanto, especialmente na ocasião dos conflitos semânticos em tempo de execução - onde mudanças realizadas em diferentes partes do *software* acabam resultando em um comportamento diferente em tempo de execução - ainda carecem de ferramentas eficientes de detecção [Cavalcanti et al. 2017].

[Silva et al. 2020] propõe a ferramenta SMAT, que realiza a detecção de conflitos semânticos por meio da observação automática de mudanças de comportamento entre as diferentes versões do *software* presentes no processo de *merge*. Essa detecção é possível

graças a utilização de ferramentas que realizam a geração de testes unitários de forma automática, permitindo que especificações parciais de cada versão sejam construídas e posteriormente comparadas a partir de uma heurística proposta pelos autores.

Apesar de resultados positivos já terem sido observados em SMAT, a implementação da ferramenta enfrenta problemas como a baixa modularidade e dificuldade de extensibilidade e manutenção do código fonte, tal como a baixa configurabilidade, não permitindo customizar aspectos importantes da ferramenta, e a dificuldade de reprodução dos experimentos conduzidos pelo caráter não-determinístico presente na ferramenta. O presente trabalho consiste em uma reestruturação e um aperfeiçoamento desta ferramenta, a nível de arquitetura e implementação, visando solucionar as lacunas supracitadas.

Na seção 2, discutimos um pouco mais a respeito dos conceitos teóricos por trás dos conflitos de integração de código e soluções existentes na literatura para a sua detecção. Na seção 3, apresentamos a ferramenta formulada por [Silva et al. 2020] para detecção de conflitos semânticos, discutindo suas funcionalidades, implementação e apresentando alguns pontos de melhoria. Na seção 4, discutimos como os pontos de melhoria citados previamente foram implementados a fim de aperfeiçoar a ferramenta de [Silva et al. 2020]. Por fim, na seção 5, apresentamos e discutimos os resultados obtidos, bem como apresentamos possíveis trabalhos futuros.

2. Conflitos de Integração de Código

Conforme mencionado previamente, durante o processo de desenvolvimento de *software* é comum que desenvolvedores realizem suas tarefas em paralelo utilizando *branches* que são posteriormente integradas utilizando *merges*.

Um cenário de *merge* é geralmente composto por dois *commits parents* que serão integrados, usualmente denominados *Left* e *Right* e um *commit* resultado da integração entre *Left* e *Right*, denominado *Merge*. Além destes *commits*, adicionaremos também a informação do primeiro *commit* ancestral que é comum ao tronco de *Left* e *Right*, aqui denominado *Base* [SILVA 2022].

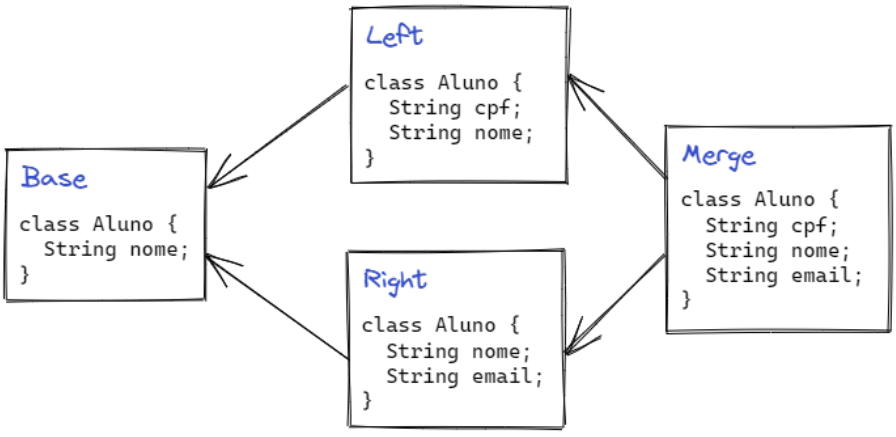

Figura 1. Visualização simplificada de um cenário de integração de diferentes versões de código. O *commit merge* é gerado pela integração das modificações introduzidas em *left* e *right*

Embora *branches* e *merges* permitam aumentar consideravelmente a produtividade de um time, o processo de integração pode resultar em conflitos, que acabam afetando a produtividade do time - haja vista a deficiência de ferramentas que resolvam estes conflitos automaticamente, como, em casos mais extremos, afetando a corretude do sistema - na ocasião em que os conflitos sequer são detectados [Cavalcanti et al. 2017].

2.1. Conflitos de merge

A fim de explorar esta primeira categoria de conflitos, suponhamos o seguinte cenário: dois desenvolvedores, João e Laura, trabalham na mesma equipe de desenvolvimento de *software* para a UFPE. Este projeto possui inicialmente a classe Aluno apresentada na Figura 2.

```
1 class Aluno {
2   private UUID codigo;
3   private String cpf;
4   private String email;
5 }
```

Figura 2. Classe de Aluno presente no projeto.

Durante uma de suas tarefas, João percebeu que é possível cadastrar um estudante com um email inválido. Em uma nova *branch* denominada *feat-validacao-email*, João decide adicionar um novo método para realizar a validação de um endereço de *email* fornecido chamado *validarEmail(email)*.

Simultaneamente, Laura percebe que um defeito pelo qual ela é responsável é fruto da ausência de validação do CPF. De forma similar ao raciocínio de João, em uma nova *branch fix-validacao-cpf*, ela decide também adicionar um método utilitário que realiza a validação do CPF chamado *validarCpf(cpf)*. A Figura 3 mostra as diferentes versões do código de João e Laura.

João (feat-validacao-email)

```
class Aluno {
  private UUID codigo;
  private String cpf;
  private String email;

  void validarEmail(String email) {
     (...)
  }
}
```

Laura (fix-validacao-cpf)

```
class Aluno {
  private UUID codigo;
  private String cpf;
  private String email;

  void validarCpf(String cpf) {
     (...)
  }
}
```

Figura 3. À esquerda, a modificação introduzida por João em sua *branch*. À direita, a modificação introduzida por Laura em sua *branch*

Ao realizar a integração das modificações de João e Laura, ocorre um conflito de *merge*: ambos os desenvolvedores introduziram modificações diferentes no mesmo trecho de um mesmo arquivo. Este tipo de conflito é detectado pela ferramenta de controle

de versão, que altera o arquivo da classe Aluno para reportar a ocorrência do conflito, utilizando uma notação especial, conforme mostra a Figura 4.

```
1  class Aluno {
2    private UUID codigo;
3    private String cpf;
4    private String email;
5
6  <<<<<<<<<<<<<<
7    void validarEmail(String email) {
8      (...)
9    }
10 ==============
11   void validarCpf(String cpf) {
12     (...)
13   }
14 >>>>>>>>>>>>>>
15 }
```

Figura 4. Classe Aluno após a integração das modificações realizadas por João e Laura. As linhas 6, 10 e 14 contém caracteres adicionados automaticamente pelo sistema de gerenciamento de versão para assinalar um conflito de *merge*

Ainda que este tipo de conflito seja de fácil detecção, a estratégia de *merge* padrão do sistema de controle de versão (merge não-estruturado) não consegue resolvê-lo automaticamente, exigindo uma intervenção manual do desenvolvedor para que o conflito seja solucionado. A literatura propõe a utilização de ferramentas que possuem conhecimento a respeito da linguagem de programação utilizada, para construir *merges* semi-estruturados e estruturados de forma que conflitos semelhantes ao apresentado no exemplo possam ser solucionados sem ser necessária a intervenção do desenvolvedor [Seibt et al. 2021].

2.2. Conflitos semânticos

Apesar dos conflitos de *merge* já terem considerável impacto no processo de desenvolvimento, exigindo que um desenvolvedor interrompa suas atividades para resolvê-los, conflitos de integração podem aparecer de diferentes formas, potencialmente causando ainda mais problemas do que os conflitos já discutidos.

Em algumas ocasiões, mesmo que conflitos de *merge* não sejam reportados, o resultado da integração ainda pode resultar em uma versão inválida do *software*, seja por um problema de compilação ou por ter a sua corretude afetada, como ilustrado mais a seguir. Neste trabalho, utilizaremos a nomenclatura adotada por [SILVA 2022] para ambos os cenários supracitados, os quais discutimos com mais profundidade nas próximas sessões.

2.2.1. Conflitos semânticos em tempo de compilação

A fim de discutir essa categoria de conflitos de integração, suponhamos o seguinte exemplo, ainda dentro do contexto de uma equipe de desenvolvimento de *software* acadêmico

na UFPE. Desta vez, João está trabalhando na refatoração de uma classe no projeto responsável por realizar chamadas a uma API externa, cuja implementação inicial é mostrada pela Figura 5.

```
class GovApiService {
   public Student getStudentFromGov(Cpf aCpf) {
      (...)
   }
}
```

Figura 5. Código Java da versão inicial da classe modificada por João

Durante a refatoração, João optou por alterar o nome do método *getStudentFromGov* para *fetchStudentFromGov*. Em paralelo e em outra *branch*, Laura está desenvolvendo uma nova funcionalidade que realizará uma chamada à API externa utilizando a classe modificada por João. Por uma falha de comunicação e planejamento, Laura ainda não possui a versão atualizada de João e, portanto, realiza a chamada utilizando o nome *getStudentDataFromGov*.

João

```
class GovApiService {
 public Student fetchStudentFromGov(Cpf aCpf) {
    (...)
 }
}
```

Laura

```
class AlunoController {
  public Student createStudent(Cpf aCpf) {
    Student aStudent = govService.getStudentFromGov(aCpf);
    (...)
  }
}
```

Figura 6. Na parte superior, a modificação introduzida por João em um arquivo (a renomeação de um método). Já na parte inferior, a modificação introduzida por Laura em outro arquivo (que envolve uma chamada ao método que foi renomeado por João).

Ao realizarem a integração deste código, a ferramenta de versionamento de código não reporta nenhum conflito de *merge*, de fato, as mudanças ocorreram em arquivos diferentes. Entretanto, a nova versão resultado da integração de ambas as versões não passa nas checagens de integração contínua do projeto, sendo reportado um erro de compilação do tipo símbolo não-existente, resultado da invocação do método *getStudentFromGov* por *right*, método este, que foi renomeado na classe *GovApiService* em *left*.

Mesmo que, à primeira vista, a situação demonstrada acima possa parecer ter ocorrência rara em ambientes reais de desenvolvimento de *software*, este tipos de conflitos ocorrem com considerável frequência. Além disso, diferentemente dos conflitos de

merge, que possuem ferramentas maduras para detecção e ferramentas promissoras para a sua solução, conflitos em tempo de compilação geralmente exigem que os próprios desenvolvedores realizem correções manualmente de modo que o *software* seja compilável novamente [Da Silva et al. 2022], custando tempo e qualidade para o projeto.

2.2.2. Conflitos semânticos em tempo de execução

Até aqui, discutimos categorias de conflitos que, apesar de afetarem a produtividade, ainda são detectados durante a etapa de desenvolvimento, inclusive já dispondo de ferramentas que conseguem detectá-los com certa facilidade. Nesta seção, discutiremos uma categoria de conflitos que surgem durante integrações que geram versões compiláveis do *software*, mas que em tempo de execução fornecem resultados diferentes dos esperados.

A fim de compreender como estes conflitos podem ocorrer na prática, tomemos como exemplo a seguinte situação: suponha que dois desenvolvedores estão trabalhando no método de validação de uma determinada classe de forma independente. O código resultado da integração de ambas as contribuições está representado na Figura 7.

```
1 class Aluno {
2    public static void validarAluno(String email) {
3        StringUtils.assertMinLength(email, 15);
4        (...)
5        StringUtils.assertMaxLength(email, 10);
6    }
7 }
8
```

Figura 7. Código Java resultado da integração entre as versões. A linha 3 é oriunda de uma contribuição de *left*, enquanto a linha 5 é oriunda de *right*.

Observemos a ausência de conflitos de *merge*, de fato, as modificações ocorrem em trechos distintos do código, e que a versão obtida após a integração é compilável, apontando a ausência de conflitos semânticos em tempo de compilação. Todavia, se executarmos o programa resultante, observamos um comportamento diferente do esperado.

Analisando as versões a serem integradas de forma isolada, podemos extrair especificações implícitas do comportamento esperado por cada desenvolvedor ao realizar as suas modificações. No exemplo da Figura 7, enquanto *left* especifica que o *email* deve ter no mínimo 15 caracteres, *right* especifica que o valor não pode ultrapassar 10 caracteres. Embora estas especificações estejam implementadas corretamente em ambas as versões, a coexistência de ambas é logicamente inconsistente resultando em um programa que lança exceções para entradas que seriam aceitas apenas por *left* ou *right*.

Esta situação aponta a presença de um conflito durante a integração que se manifesta apenas em tempo de execução. Como a detecção desta categoria de conflitos carece de ferramentas automáticas, as últimas barreiras para a detecção destes conflitos em uma equipe é a adoção de boas práticas como a utilização de suítes de testes automatizadas e políticas de revisões de código. Mesmo assim, inclusive projetos que possuem boa cobertura de testes e políticas estritas de revisão correm o risco de negligenciar tais conflitos,

permitindo que estes cheguem ao *software* em produção [SILVA 2022].

2.2.3. Detectando conflitos semânticos em tempo de execução

Detectar conflitos semânticos em tempo de execução possui elevada dificuldade. De fato, detectar tais conflitos envolveria construir uma ferramenta que fosse capaz de compreender o comportamento esperado do *software*, e o efetivamente implementado em cada um dos *parents* envolvidos na integração, podendo então avaliar a existência de interferências e potenciais conflitos entre cada uma das versões. Entretanto, neste contexto, esta avaliação é um problema que não é sequer computável [Cavalcanti et al. 2017].

Mesmo assim, soluções heurísticas são propostas na literatura, geralmente utilizando ferramentas de análise estática, a fim de detectar tais conflitos. Neste trabalho, focaremos na ferramenta SMAT, proposta por [Silva et al. 2020], que utiliza uma estratégia diferente das apresentadas até então.

3. Ferramenta SMAT

Conforme apresentado na seção anterior, detectar conflitos semânticos em tempo de execução é um processo difícil e ainda são poucas as propostas de ferramentas capazes de atacar esse problema de forma eficaz. Neste contexto, [Silva et al. 2020] propõe a ferramenta SMAT que utiliza uma estratégia diferente das utilizadas por outras ferramentas que possuem a mesma finalidade.

Para o método modificado onde se quer investigar a ocorrência de conflitos semânticos, aqui denominado **método alvo**, SMAT utiliza ferramentas que realizam a geração automática de suítes de testes já propostas na literatura, como *Evosuite* [Fraser 2018] e *Randoop* [Pacheco et al. 2007], para construir especificações parciais do método alvo em ambos os *parents*. Como exemplo, suponha o teste da Figura 8, que especifica parcialmente a modificação introduzida por *right* no exemplo da seção anterior.

```
1 @Test
2 public void test() {
3   Exception e = assertThrows(StringException.class, () -> {
4     String email = "12caracteres"; // string com tamanho 12
5       Aluno.validarAluno(email);
6   });
7   assertTrue(e.getMessage()
8     .equals("String não pode ter mais que 10 caracteres");
9 }
```

Figura 8. Caso de teste de exemplo para a modificação introduzida por *right* no exemplo da Figura 7

SMAT então realiza comparações destas especificações em cada versão do *software* através da execução dos casos de teste gerados em cada uma das versões contidas no cenário de *merge*. O resultado da execução dos casos de testes são avaliados com o objetivo de detectar conflitos que atendam aos seguintes critérios heurísticos:

- A execução do teste falha em pelo menos um dos *parents* e passa em *Base* e *Merge*. Analogamente, um conflito também é detectado se a execução do teste passa em pelo menos um dos *parents* mas falha em *Base* e *Merge*.
- A execução do teste falha em ambos os *parents* e em *Base*, mas passa em *Merge*. Analogamente, um conflito também é detectado se a execução do teste passa em ambos os *parents* e em *Base* mas falha em *Merge*.

Como o caso de teste da Figura 8 passa na versão do código de *right* e falha em *base*, afinal nenhuma validação a respeito do tamanho da *String* é realizada. Embora o teste também falhe em *merge*, a exceção levantada pela validação é pelo fato da *String* de teste possuir comprimento inferior ao permitido, comportamento introduzido por *left*. Este cenário de interferência satisfaz o primeiro critério para detecção de um conflito semântico, sendo então reportado por SMAT ao usuário.

Por outro lado, suponha o cenário da Figura 9 que é o código resultado da integração de duas *branches* diferentes, e as linhas ocultas não modificam o valor da variável *x*. A Figura 10 representa um teste gerado como uma especificação parcial para a modificação introduzida por *right*.

```
1  public static int some_random_method() {
2    int x = 0;
3    (...)
4    x += 1;
5    (...)
6    x += 1;
7    (...)
8    if (x >= 2) {
9      throw new RuntimeException();
10   }
11   return x;
12 }
```

Figura 9. Neste cenário, a linha 4 foi introduzida por *left* e a linha 6 por *right*.

```
1 @Test(expected = None.class)
2 public void test() {
3   int result = RandomClass.some_random_method();
4   assertEquals(10, result);
5 }
```

Figura 10. Caso de teste de exemplo para a Figura 9

É válido observar que, tanto em *base* quanto em ambos os *parents*, o teste da Figura 10 passa (afinal, a variável *x* nunca possui o valor 2). Entretanto, quando integradas, as modificações introduzidas por *left* e *right* acabam fazendo com que a variável *x* assuma o valor 2, lançando uma exceção em tempo de execução e quebrando o teste em *merge*, satisfazendo assim o segundo critério heurístico de SMAT.

A fim de evitar que testes que tenham comportamento *flaky*, isto é, que não apresentam resultados consistentes ao longo de várias execuções, possam interferir no re-

sultado fornecido pela ferramenta, SMAT realiza 3 execuções de cada caso de teste, descartando-o na análise na presença de qualquer disparidade entre os resultados.

Outra preocupação existente é com testes que venham a falhar por exercitarem elementos que não estão presentes em alguma versão do código que está sendo analisado - suponha, por exemplo, a introdução ou remoção de um método em algum dos *parents*. Neste contexto, a avaliação seria inconclusiva e, portanto, tais testes são descartados durante a análise.

3.1. Arquitetura e Implementação de SMAT

A implementação original de SMAT[1] é realizada em *Python* com aproximadamente 3800 linhas de código, e é fruto de um *fork* da ferramenta Nimrod, originalmente utilizada para experimentos com teste de mutação.

A ferramenta pode ser estruturada conceitualmente como sendo composta por 4 módulos que coincidem com as 4 etapas sequenciais da execução de SMAT: geração das suítes de testes *(Test Generation)*, execução destas suítes *(Test Execution)*, análise dos resultados *(Dynamic Analysis)* e geração de relatórios *(Output Generation)*, estruturados arquiteturalmente na aplicação como mostra a Figura 11.

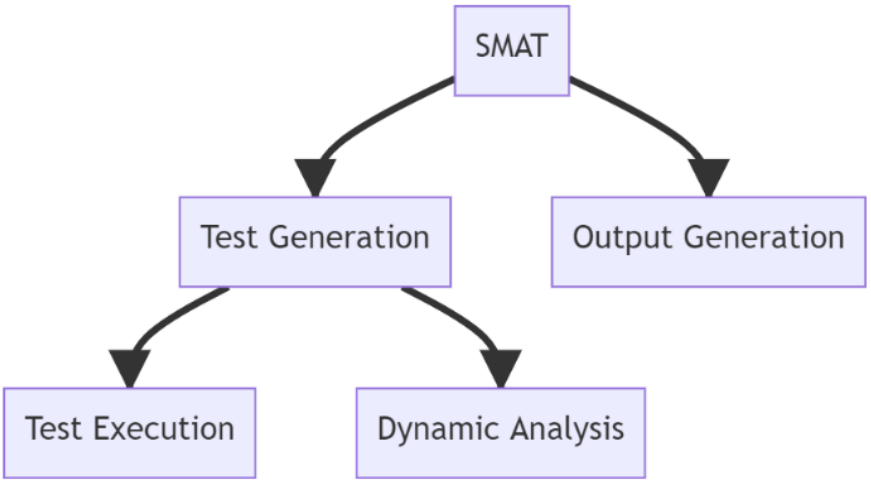

Figura 11. Diagrama da macro-arquitetura de SMAT.

SMAT recebe como entrada um ou mais cenários de merge dispostos em um arquivo CSV que obedece a seguinte estrutura:

- Nome do projeto a ser analisado;
- Um booleano indicando se o cenário deve ser analisado ou não;
- Seguem 4 colunas com as *hashes* dos *commits* que serão analisados (*base, left, right e merge*);
- *Fully-Qualified Class Name (FQCN)* da classe alvo. Mais de uma classe pode ser informada utilizado o caractere — como separador;
- Assinatura do método alvo. Somente um método pode ser fornecido. Os argumentos do método devem ser informados utilizando o caractere — como separador;

[1] https://github.com/spgroup/SMAT/tree/ae33ccdc49

- Seguem 4 colunas com o caminho para os JARs de cada SHA (*base, left, right e merge*). Estas versões são necessárias para que sejam informadas ao gerador de testes.

3.2. Limitações e pontos de melhoria de SMAT

Apesar de SMAT ter apresentado resultados positivos na detecção dos conflitos, tendo sido capaz de detectar corretamente mais de 30% dos casos presentes na amostra estudada em trabalho anterior [Silva et al. 2020], a ferramenta possui algumas limitações e pontos de melhoria relacionados à sua implementação e arquitetura.

Pela natureza exploratória das ferramentas de geração de testes utilizadas, é possível que o método alvo da análise não seja exercitado por nenhum caso de teste que atenda aos critérios heurísticos implementados. Assim, como SMAT não leva em consideração os métodos executados em um caso de teste durante a análise, é possível que um conflito seja reportado sem que o método alvo tenha sido executado, consistindo assim em um falso positivo.

Junto a isto, SMAT possui a limitação de conseguir analisar apenas um único alvo por execução. Por exemplo, se no cenário da seção anterior estivéssemos interessados em executar a análise para outro método, teríamos que realizar uma nova chamada a SMAT e, por consequência, realizar uma nova chamada as ferramentas geradoras de testes e novas chamadas a execução dos testes, o que aumentaria consideravelmente o tempo de execução da ferramenta.

Do ponto de vista arquitetural, embora as 4 etapas da execução possuam um isolamento conceitual, a atual hierarquia dos módulos, representada no diagrama da Figura 11, apresenta um acoplamento entre a geração, execução e posterior análise dos testes gerados. Desta forma, um desenvolvedor que esteja encarregado de adicionar um novo gerador de testes, ou mesmo corrigir um *bug* relacionado a um deles, deveria estar atento e conhecer detalhes relacionados a outros aspectos da aplicação, aumentando a complexidade envolvida neste processo representando uma área onde a modularidade do código pode ser melhorada [Ousterhout 2018].

Ainda no contexto arquitetural, a aplicação possui um modelo anêmico, geralmente representando conceitos do domínio da aplicação utilizando-se de estruturas de dados como listas. A ausência de um modelo estruturado e que explicite conceitos relevantes para o funcionamento da aplicação aumenta drasticamente a complexidade do *software*, fazendo com que, a longo prazo, sua compreensão, manutenção e extensão torne-se impraticável [Evans 2004].

Outro aspecto importante a ser discutido na implementação de SMAT é a dificuldade de customização do comportamento da ferramenta em tempo de execução. Suponhamos, como exemplo, que estamos interessados em alterar o tempo disponível para a geração de testes utilizando a ferramenta Evosuite. A Figura 12 mostra o método responsável pela chamada de Evosuite[2].

[2]https://github.com/spgroup/SMAT/blob/ae33ccdc49/nimrod/tools/evosuite.py#L16

```
class Evosuite(SuiteGenerator):
    (...)
    def _exec_tool(self):
        params = [
            '-jar', EVOSUITE,
            '-projectCP', self.classpath,
            '-class', self.sut_class,
            '-Dtimeout', '10000',
            '-Dassertion_strategy=all',
            '-Dp_reflection_on_private=0',
            '-Dreflection_start_percent=0',
            '-Dp_functional_mocking=0',
            '-Dfunctional_mocking_percent=0',
            '-Dminimize=false',
            #'-Dassertions=false',
            '-Dsearch_budget=300',
            '-Djunit_check=false',
            '-Dinline=false',
            '-DOUTPUT_DIR=' + self.suite_dir
        ]

        params += self.parameters

        return self._exec(*tuple(params))
    (...)
```

Figura 12. Código do método responsável pela chamada de Evosuite para a geração de suítes de testes para uma determinada classe.

Neste cenário, observemos que não é possível realizar essa alteração a) sem modificar diretamente o código fonte, b) sem conhecer os argumentos disponíveis para a execução de Evosuite, neste caso seria necessário modificar o atributo *-Dsearch-budget* [Fraser 2018]. Enquanto a introdução da configurabilidade em tempo de execução pode ser vista aqui como somente um simples aperfeiçoamento, esta modificação abre a possibilidade da introdução de uma interface comum e extensível para a geração de suíte de testes que permita que o usuário realize a customização de aspectos da execução de SMAT, como escolher quais geradores serão executados, como também customizar características dos próprios geradores, como o tempo de busca exploratória disponível e a utilização de geração determinística, sem que seja necessário conhecer os detalhes da implementação [Ousterhout 2018].

Outro ponto que é válido ressaltar é que, pelo caráter não-determinístico da implementação utilizada na geração de suíte de testes, execuções sucessivas de SMAT no mesmo cenário podem levar a resultados diferentes. Esta característica torna difícil a reprodução dos experimentos conduzidos, algo importante para a validação dos resultados apresentados por [Silva et al. 2020] e que poderia ser resolvida tornando a etapa de geração de testes determinística, mesmo que os algoritmos de geração tenham um determinado grau de aleatoriedade.

Por último, é importante observar que SMAT não possui anotações de tipos nos

elementos utilizados em seu código. Apesar de Python ser uma linguagem que consegue determinar o tipo de suas variáveis somente em tempo de execução, é possível utilizar ferramentas que introduzem anotação de tipos [Lehtosalo et al. 2021] que podem ser posteriormente analisadas de maneira estática, melhorando significativamente a produtividade [Hanenberg 2009] e a manutenção do *software* [Kleinschmager et al. 2012].

4. Atacando as lacunas de SMAT

Conforme discutido anteriormente, o objetivo deste trabalho é de propor aperfeiçoamentos e soluções para as limitações encontradas em SMAT. Nas seções seguintes, discutimos as estratégias utilizadas durante este processo.

4.1. Alterando a interface de entrada

A primeira proposta ofertada neste trabalho foi a de construir uma nova interface para o fornecimento da lista dos cenários a serem analisados por SMAT com o objetivo de facilitar a utilização da ferramenta pelos seus clientes. Para isto, decidimos alterar a formatação da entrada para utilizar arquivos JSONs, em substituição aos antigos CSVs, para permitir a representação de estruturas mais complexas e substituir notações exclusivas da ferramenta, *e.g.:* a separação de parâmetros dos métodos por caracteres — ao invés do habitual separador (,).

Outro objetivo a ser atingido com a modificação era o de permitir que a ferramenta pudesse detectar conflitos em uma quantidade arbitrária de alvos em uma mesma execução, ao invés do único alvo, que era a estratégia implementada até então. Realizar a análise com todos estes alvos em uma única execução simplifica o uso da ferramenta pelo cliente mas também diminui o tempo de execução da ferramenta em cenários realistas onde, geralmente, diversos alvos são modificados em um mesmo cenário de integração.

Para atender a este objetivo, a declaração dos alvos a serem avaliados foi reestruturada e é informada utilizando um dicionário, que tem como chaves os *FQCNs* das classes alvos e como valores uma lista com os métodos a serem observados naquela execução. A Figura 13 mostra um exemplo com a nova entrada de SMAT.

```
[
  {
    "projectName": "spring-boot",
    "runAnalysis": false,
    "scenarioCommits": {
      "base": "7578f2f824aac027529878810b76ee176b39e73a",
      "left": "0d00039ae7d01538de3f813b17d125dc5fdd5706",
      "right": "1c21f54bf91283d70e04c49ea09a4c05a885d7ac",
      "merge": "3eebbe1c8aed529e6903e78476492592ce0b0049"
    },
    "targets": {
      "br.ufpe.cin.entidades.Aluno": [
        "validarEmail(String anEmail)",
        "validarCpf(String aCpf)",
      ]
    },
    "scenarioJars": {
      "base": "caminho-do-jar-base.jar",
      "left": "caminho-do-jar-left.jar",
      "right": "caminho-do-jar-right.jar",
      "merge": "caminho-do-jar-merge.jar"
    }
  },
]
```

Figura 13. Representação de exemplo da nova interface de SMAT. Neste cenário, estamos avaliando a presença de conflitos semânticos nos métodos *validarEmail* e *validarCpf* da classe *br.ufpe.cin.entidades.Aluno*

4.2. Detectando os alvos envolvidos em um conflito e descartando falsos positivos

A introdução da possibilidade de utilização de diversos alvos durante a mesma execução apresentada na seção anterior traz consigo uma nova necessidade dentro da aplicação. Suponhamos que estamos interessados em avaliar a existência de conflitos semânticos nos métodos *x()* e *y()* de uma determinada classe A. Na ocasião da existência de um conflito, como identificar em qual método este conflito ocorreu?

De fato, é importante observar que, a princípio, o conflito pode ter ocorrido em *x()* ou *y()*, ou até mesmo em ambos os métodos, no caso de uma interação. Assim, é importante que a ferramenta indique ao desenvolvedor onde exatamente este conflito ocorreu e quais os alvos envolvidos em cada um destes conflitos. Para isto, a ferramenta precisa ser capaz de detectar os alvos que foram efetivamente executados nos testes onde os conflitos foram detectados. Esta detecção pode ser realizada utilizando duas estratégias diferentes:

1. Através de uma análise estática do código fonte do caso de teste gerado e do código fonte do projeto sob análise, a fim de verificar os métodos alvos que podem ser diretamente ou indiretamente invocados pelos casos de teste envolvidos.
2. Através de uma análise do relatório de cobertura do código fonte durante a execução do teste conflitante no commit de merge, a fim de verificar se ao menos uma linha do método alvo foi invocada durante a execução do teste.

A fim de aproveitar a infraestrutura de coleta e geração de reportes de cobertura de código já implementada em SMAT para outras finalidades, adotamos a estratégia de análise do relatório de cobertura citada anteriormente, mesmo que esta introduza um custo devido a instrumentação do código e da coleta de cobertura em si. Uma consequência interessante desse aperfeiçoamento é que a mesma estratégia pode ser trivialmente utilizada para avaliar a ocorrência de falsos positivos como descritos por [SILVA 2022]: a situação em que nem *x()* e nem *y()* são executados pelo teste. Desta forma, nossa ferramenta pode dar uma resposta mais significativa para os seus usuários.

4.3. Arquitetura

Conforme discutido previamente, o principal problema arquitetural presente em SMAT é o acoplamento existente entre os módulos de geração das suítes, execução dos testes e análise dinâmica. Desta forma, realizamos a reestruturação dos módulos conforme mostra a Figura 14, a fim de permitir a evolução independente destes contextos da aplicação.

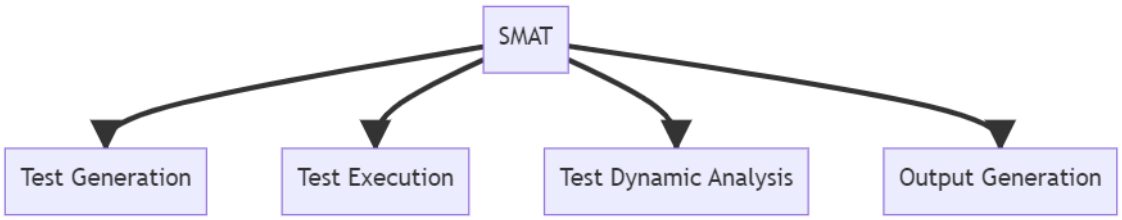

Figura 14. Diagrama da nova macro-arquitetura proposta para SMAT.

Cada módulo expõe uma interface pública que permite um ponto único de acesso para as funcionalidades disponíveis e que são então consumidas pela classe principal de controle do sistema responsável por realizar as chamadas na ordem de execução de SMAT. Aspectos utilitários da aplicação, como decodificação da entrada, logging, configuração e a instanciação de objetos foram abstraídos para um módulo separado.

Com o objetivo de aumentar a customização da ferramenta, é possível alterar alguns parâmetros referentes à execução a partir de um arquivo de configuração JSON, cuja estrutura está descrita no Apêndice A.

A fim de mitigar os problemas relacionados ao fraco modelo presente na aplicação, adotamos técnicas de modelagem utilizando o paradigma de orientação a objetos com a finalidade de melhor representar as interações entre as diferentes entidades e serviços envolvidos durante a execução da ferramenta. Nas seções a seguir, discutimos os detalhes e decisões envolvidas no aperfeiçoamento e implementação deste modelo em cada um dos módulos de execução da aplicação.

4.4. Geração das Suítes de Testes

A primeira etapa da execução de SMAT consiste da geração de suítes de testes para cada um dos *parents* (de um cenário de *merge*) utilizando ferramentas automáticas de geração de testes unitários como Evosuite e Randoop. Na implementação original de SMAT, um dos principais problemas era o acoplamento entre as etapas de geração, execução e análise dinâmica. Tal acoplamento atingia inclusive a classe de controle, de modo que esta tinha, obrigatoriamente, conhecimento de quais geradores de testes estavam sendo executados

conforme mostra o diagrama da Figura 15, sendo assim difícil adaptar SMAT para usar um conjunto diferente de ferramentas de geração.

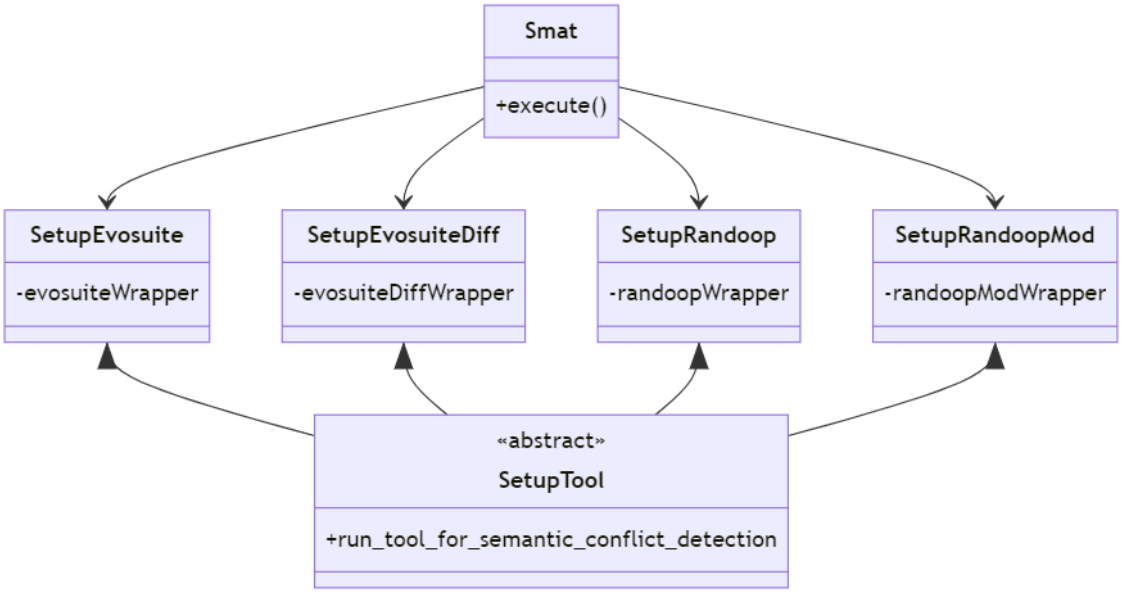

Figura 15. Diagrama de classes simplificado de SMAT. Na implementação original, a classe abstrata SetupTool mistura aspectos de geração, execução e análise dinâmica. Suas subclasses utilizam *wrappers* para realizar a chamada a cada gerador de testes.

Nossa primeira proposta foi a de agrupar todos os conceitos da geração de suíte de testes em um único módulo profundo, que expusesse uma interface simples mas que atendesse às funcionalidades e comportamentos esperados por seus clientes [Ousterhout 2018], chegando então a arquitetura apresentada na Figura 16

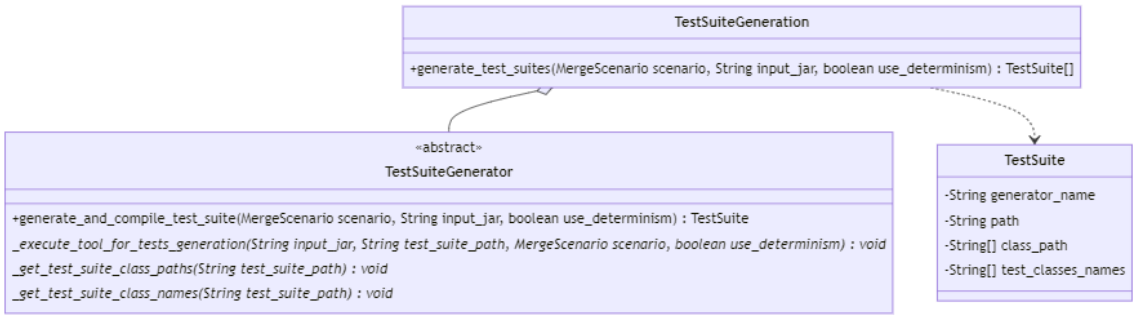

Figura 16. Diagrama de classes do módulo de Geração das Suítes de Testes.

Neste novo *design*, a classe *TestSuiteGeneration*, que publica a interface deste módulo, expõe um único método que encapsula as sucessivas chamadas a cada um dos

geradores de testes implementados. Isso isenta os clientes de conhecerem quais geradores estão sendo executados, simplificando o código da classe de controle e permitindo a possibilidade de customizar os geradores de testes utilizados em tempo de execução, comportamento este implementado a partir de um arquivo de configuração.

Nesta arquitetura, as subclasses de *TestSuiteGenerator* oferecem através de sua interface o método *generate_and_compile_test_suites*, responsável por realizar a chamada a ferramenta de geração de testes e a posterior compilação do código fonte gerado.

Como cada ferramenta de teste possui características específicas, tais como a invocação da ferramenta, a determinação de quais arquivos gerados devem ser compilados e onde estão localizados, bem como determinar o *classpath* necessário para a compilação da suíte gerada, o método *generate_and_compile_test_suites* foi implementado utilizando o padrão *template method* [Gamma et al. 1995], de modo que o comportamento específico de cada gerador fosse implementado pelas suas subclasses.

Outro ponto de melhoria importante levantado na seção anterior é adicionar a possibilidade de termos determinismo na geração dos testes. Pelo grau de aleatoriedade presente nas ferramentas utilizadas para a geração de testes, atualmente, execuções sucessivas de SMAT podem resultar em resultados diferentes pelas diferenças geradas em cada suíte.

Entretanto, as ferramentas utilizadas disponibilizam funcionalidades que permitem tornar a execução determinística, através do fornecimento de uma *seed* e da alteração em alguns parâmetros durante a invocação da ferramenta, a saber a remoção das restrições de tempo e determinação de novos critérios de parada.

Optamos por abstrair o conceito da geração determinística através de um parâmetro na interface do módulo de geração das suítes, que pode ser configurado pelo usuário final a partir do arquivo de configuração, permitindo dinamicamente habilitar o comportamento determinístico sem que seja necessário alterar o código fonte. Além disto, também é possível customizar a *seed* fornecida às ferramentas através do arquivo de configuração, sendo utilizado um valor padrão caso este não seja informado pelo usuário.

Por último, a introdução da classe *TestSuite* permitiu estruturar os dados das suítes geradas em um objeto, diferentemente da estruturação em n-tuplas utilizada previamente. Embora seja mais verboso, o objeto permite armazenar e acessar com facilidade informações da suíte que serão utilizados em etapas posteriores da execução, como a localização dos arquivos gerados, o gerador utilizado, bem como o nome das classes geradas e o *classpath* dos arquivos compilados.

4.5. Execução das Suítes de Testes

Uma vez que a geração das suítes é concluída, é necessário realizar a execução destas suítes em cada uma das versões do *software* presentes no cenário de integração. Para isto, SMAT utiliza-se de um executor de testes, no caso JUnit 4. De forma similar à seção anterior, encapsulamos o comportamento da execução da suíte em um módulo, arquitetado conforme mostra a Figura 17.

Neste design, além de realizar as chamadas às ferramentas que realizam a execução dos testes, a classe *TestSuiteExecutor* também é responsável por realizar a decodificação da saída de *JUnit*, convertendo-a para as representações internas da

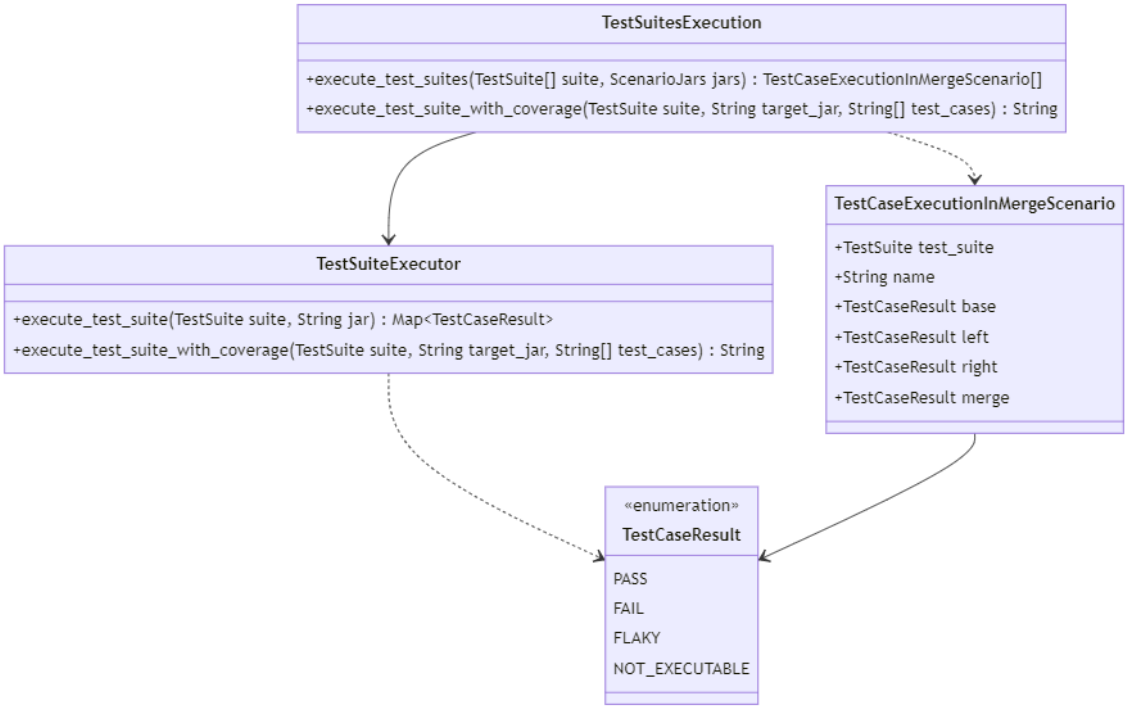

Figura 17. Diagrama de classes do módulo de Execução das Suítes de Testes.

aplicação, atuando como um Adapter [Gamma et al. 1995]. Este design permite facilmente substituir o executor de suítes de teste sem ter de se preocupar em eventuais danos a outros aspectos do modelo, contribuindo diretamente com a extensibilidade da ferramenta.

Além disso, é importante discutir os diferentes resultados que um caso de teste pode ter em uma determinada execução, para além dos casos triviais: falhar e passar. Como discutido na Seção 3, é possível que alguns dos testes gerados previamente possuam comportamento *flaky*, isto é, não possuem resultados consistentes em sucessivas execuções. A fim de tentar detectar este tipo de comportamento, SMAT realiza 3 execuções sucessivas de cada caso de teste marcando-o como *flaky* caso algum resultado seja diferente.

Outro ponto de atenção é na situação em que o caso de teste falha pela ausência de um determinado símbolo que foi introduzido ou removido por alguma modificação em algum dos *commits*. De fato, suponha o trecho de código apresentado na Figura 18 em *merge*, onde a linha 3 indica uma chamada a um método adicionado por *left*.

```
class Aluno {
    public void validarAluno(String email, String cpf) {
        this.validarEmail(email);
        this.validarCpf(cpf);
    }
}
```

Figura 18. Classe de exemplo. O método *validarEmail* foi adicionado por *left*

Ao executarmos um caso de teste que exercita o método *validarEmail* em *right* ou *base* seremos surpreendidos com um erro do tipo *NoSuchMethodError*, decorrido da ausência da implementação deste método na classe Aluno nestas versões. Desta forma, embora o teste falhe, esta ocorrência é resultado do fato de o teste sequer ser executável nestas versões do *software*. Ao detectar estes tipos de erros (métodos, atributos e classes faltantes), SMAT classifica tais testes como *NOT_EXECUTABLE*.

Outra mudança foi a introdução de um novo método que executa a coleta de cobertura de código enquanto realiza a execução dos testes. Na implementação original de SMAT, esta coleta ocorre em todas as chamadas ao módulo de execução, introduzindo uma penalidade considerável em performance, especialmente pela estratégia de executar os testes mais de uma vez. Esta decisão permite que o fluxo secundário de coleta de cobertura seja separado do fluxo primário da execução, resultando em um ganho de performance sem prejuízo semântico ao cliente da aplicação [Ousterhout 2018].

SMAT realiza a coleta de cobertura de código utilizando *Jacoco* [Hoffmann et al. 2009]. Para isto, é necessário realizar uma instrumentação da versão a ser analisada, para que seja possível detectar quais comandos do código foram executados. Na implementação original, o processo de instrumentação é realizado previamente pelo cliente da classe, que somente então invoca a execução dos testes fornecendo a versão instrumentada.

Esta decisão contribui para que detalhes da implementação da coleta de cobertura estejam expostos ao cliente (a necessidade de realizar um pré-processamento na versão a ser analisada), sinalizando uma quebra do encapsulamento do algoritmo implementado, aumentando a carga cognitiva e complicando a interface do módulo. Em casos extremos, por exemplo, o usuário desta API pode ser surpreendido pelo comportamento do módulo caso forneça uma versão não instrumentada do código, algo que acarretará com que o reporte não seja gerado corretamente.

[Ousterhout 2018] afirma que é preferível que um módulo tenha uma interface simples mesmo que isso aumente a complexidade de sua implementação. Baseado neste princípio, decidimos encapsular a lógica de instrumentação da versão a ser analisada no método *execute_test_suite_with_coverage*. Desta forma, um cliente precisa somente informar a versão na qual deseja realizar a coleta de cobertura, e a implementação será responsável por realizar a instrumentação — inclusive pulando esta etapa caso uma versão instrumentada já esteja presente — e realizar a posterior execução e tratamento dos testes.

4.6. Análise Dinâmica

Após a execução, é necessário que SMAT realize a análise das execuções dos testes, a fim de verificar se algum deles satisfaz algum dos critérios heurísticos de interferência apresentados na Seção 3. Vale ressaltar que, durante a fase de análise, os testes que possuem comportamento *flaky* ou que não são executáveis em alguma versão do *software* são descartados.

Além disso, é importante ressaltar que é possível que exista mudança de comportamento entre diferentes versões do *software* sem que necessariamente ocorra a presença de um conflito. Assim, para fins de estudo, SMAT também computa mudanças de comportamento entre as versões do *software* verificando pares de versões que possuem resultados diferentes para um mesmo caso de teste.

Na implementação original, tanto a verificação de conflitos quanto de mudanças de comportamento são realizadas por uma única classe[3]. Nesta versão, os casos de testes são separados em diferentes conjuntos: os que passaram, os que falharam e os que possuem comportamento *flaky* ou que não foram executados. As verificações são realizadas então utilizando operações entre estes diferentes conjuntos, o que torna difícil a compreensão de como cada critério é definido apenas pela leitura do código.

Além disso, a ausência do uso de objetos e pela estratégia de dividir os casos de testes em diferentes conjuntos polui a interface dos métodos, que acabam tendo elevado número de parâmetros resultando em um *code smell* [Fowler 2018]. A ausência de um modelo orientado a objetos também resulta em um uso abusivo de tipos primitivos, que acabam por não expressar de forma clara o comportamento e as relações entre os diferentes componentes do *software*.

Neste sentido, nosso primeiro passo é construir um modelo que consiga refletir a relação existente entre a execução de um caso de teste em um determinado cenário e se ele satisfaz ou não um determinado critério heurístico da ferramenta. Para modelar esta relação bem como os possíveis múltiplos critérios existentes para a detecção de conflitos, encapsulamos tal comportamento na interface *SemanticConflictCriteria*. Na situação de uma execução de um caso de teste satisfazer algum destes critérios, estes são encapsulados em uma instância da classe *SemanticConflict*. Essa visão parcial do modelo de análise dinâmica está representado no diagrama da Figura 19.

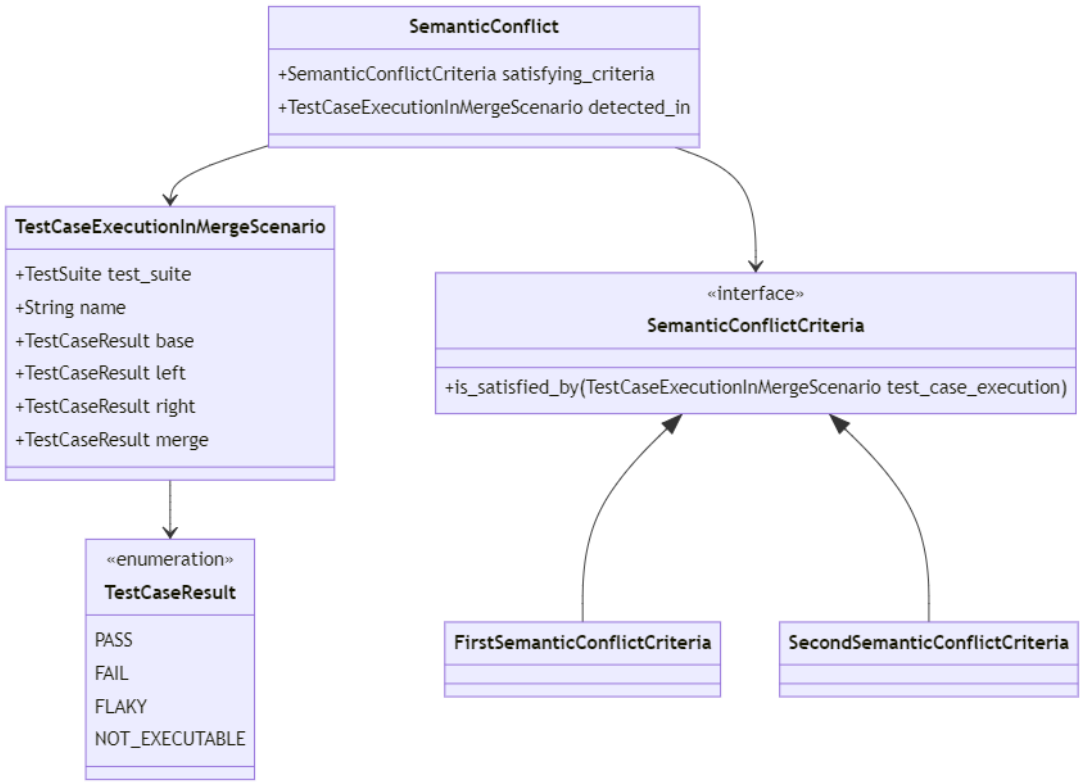

Figura 19. Diagrama de classes da estruturação da análise de conflitos semânticos.

[3]https://github.com/spgroup/SMAT/blob/ae33ccdc49/nimrod/setup_tools/behaviour_check.py

Com este modelo, a implementação dos critérios de conflitos tem sua implementação drasticamente simplificada, se assemelhando bastante a sua descrição em linguagem natural, conforme mostra a Figura 20 que ilustra a implementação para o segundo critério apresentado na seção 3

```
class SecondSemanticConflictCriteria(SemanticConflictCriteria):
  def is_satisfied_by(self, test_case_execution):
    fails_in_base_and_both_parents_but_passes_in_merge = \
        test_case_execution.base == TestCaseResult.FAIL \
        and test_case_execution.left == TestCaseResult.FAIL \
        and test_case_execution.right == TestCaseResult.FAIL \
        and test_case_execution.merge == TestCaseResult.PASS
    passes_in_base_and_both_parents_but_fails_in_merge = \
        test_case_execution.base == TestCaseResult.PASS \
        and test_case_execution.left == TestCaseResult.PASS \
        and test_case_execution.right == TestCaseResult.PASS \
        and test_case_execution.merge == TestCaseResult.FAIL
    return fails_in_base_and_both_parents_but_passes_in_merge or \
    passes_in_base_and_both_parents_but_fails_in_merge
```

Figura 20. Implementação da detecção do segundo critério de conflitos

Uma estratégia similar pode ser utilizada para identificar as mudanças de comportamento entre diferentes versões do *software*. De fato, o modelo resultante é semelhante, conforme mostra a Figura 21.

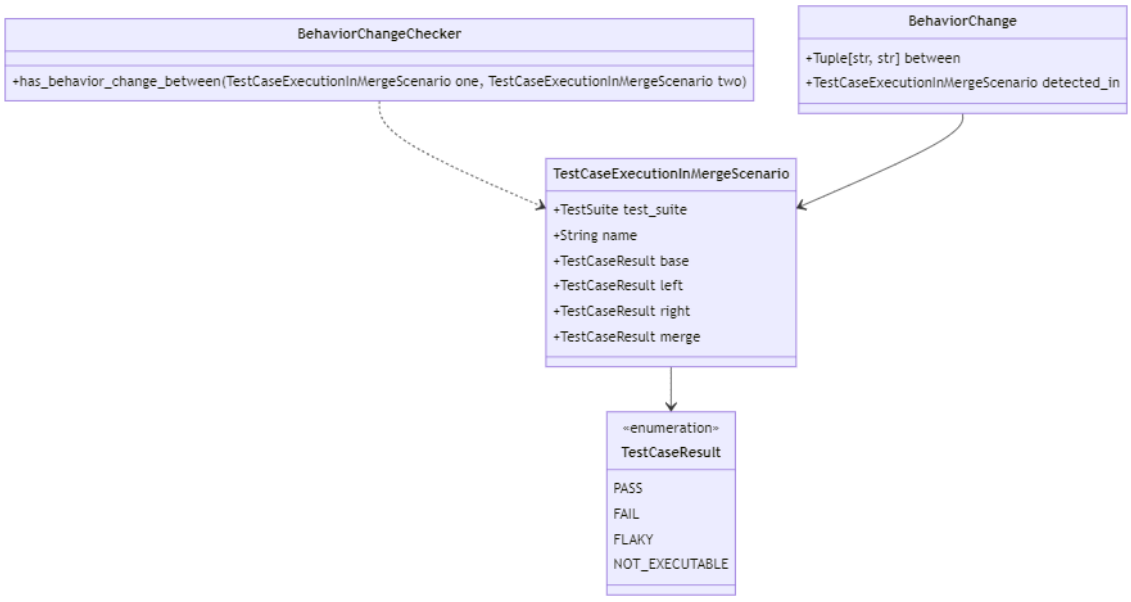

Figura 21. Diagrama de classes da estruturação da análise de mudanças de comportamento.

A fim de simplificar o consumo por parte de seus clientes, as implementações do módulo de Análise Dinâmica foram agrupadas em uma Fachada [Gamma et al. 1995], que agrupa as funcionalidades principais fornecidas pelo módulo. Desta forma, uma visão geral do módulo é apresentada na Figura 22.

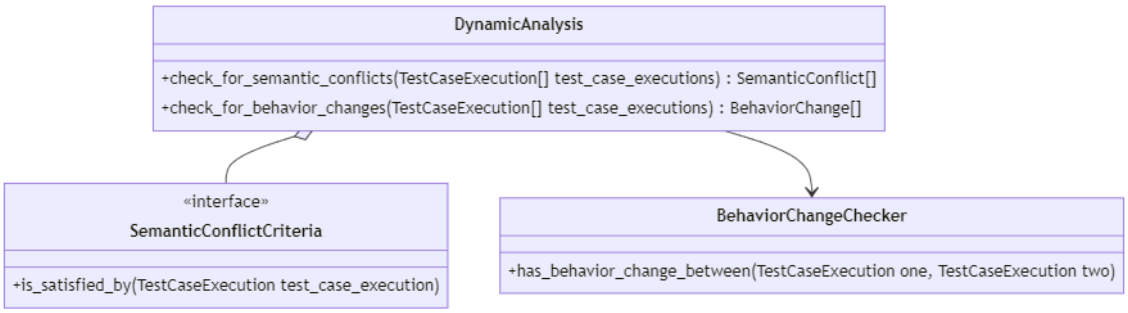

Figura 22. Diagrama de classes em alto nível do módulo de análise dinâmica.

4.7. Geração de relatórios

Com as análises finalizadas, a última etapa da execução de SMAT consiste em construir relatórios com os dados coletados para que estes possam ser consumidos e analisados pelo usuário ou por outras ferramentas.

A ferramenta permite a confecção de 3 tipos de relatórios: um com informações a respeito dos conflitos semânticos detectados, outro possuindo informações a respeito das mudanças de comportamento observadas entre as diferentes versões do cenário e um último reporte contendo informações a respeito das suítes de testes geradas. A Figura 23 mostra a estrutura de cada um dos relatórios gerados.

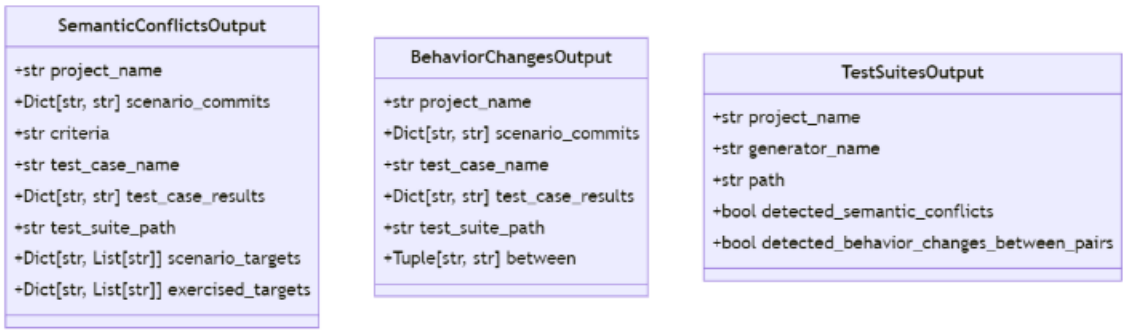

Figura 23. Estrutura de cada um dos relatórios gerados. Cada relatório é composto de uma lista de instâncias destas classes.

Seguindo a mesma estratégia utilizada nas outras seções, decidimos por isolar a geração de relatórios em um módulo na aplicação. Este módulo possui uma classe abstrata *OutputGenerator* responsável por realizar a geração de um relatório. Embora na versão original de SMAT os relatórios fossem gerados em formato CSV, optamos por utilizar JSON em nossa implementação, pela possibilidade de representar mais facilmente estruturas como dicionários e listas.

Como cada relatório possui uma estratégia diferente para sua geração, consultando dados diferentes e aplicando processamentos específicos, cada subclasse deve sobrescrever o método abstrato *_generate_report_data* introduzindo o comportamento necessário para a confecção do relatório, resultando assim em mais uma aplicação do padrão *template method* [Gamma et al. 1995]. A Figura 24 apresenta a arquitetura do módulo de geração de reportes.

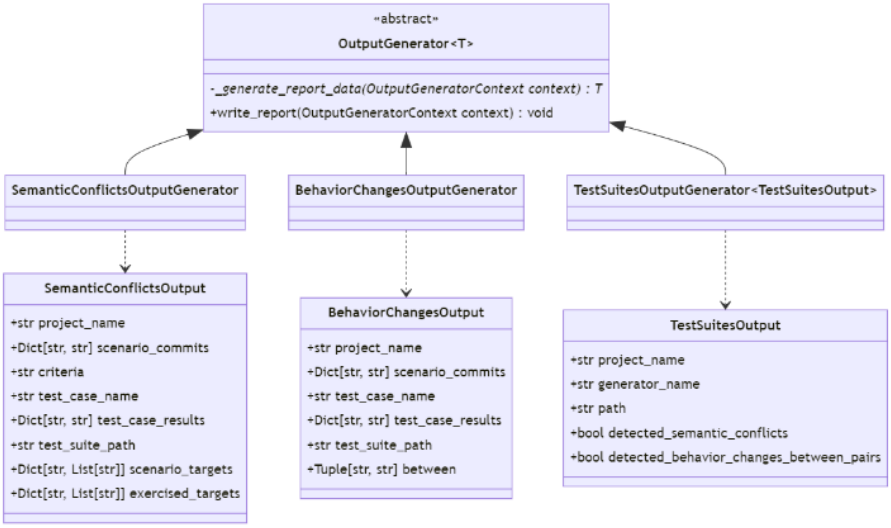

Figura 24. Diagrama de classes do módulo de geração de relatórios

Um aspecto importante a ser considerado neste módulo é a necessidade de consultar valores computados em diferentes etapas da execução. Por exemplo, os reportes de conflitos semânticos necessitam de informações obtidas durante a etapa de análise dinâmica, enquanto o reporte de suítes de teste consulta informações da geração de suítes e de análise dinâmica.

A fim de preservar uma interface comum para os diferentes geradores de relatório, o que simplifica a consumação da classe pelos seus clientes, decidimos por ter como parâmetro da geração de relatório as diferentes entidades das etapas anteriores. Entretanto, para evitar o *code smell Long Parameter List* [Fowler 2018], quando um método possui diversos parâmetros, decidimos por encapsular estes parâmetros em um único objeto, utilizando a técnica de *Introduce Parameter Object* [Fowler 2018].

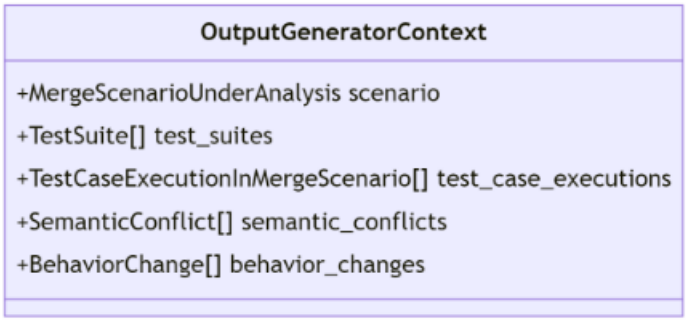

Figura 25. Diagrama do objeto *OutputGeneratorContext*, que guarda as diversas informações a serem consultadas durante a geração de um relatório.

4.8. Documentação e gerência do projeto

A fim de melhorar a manutenibilidade do projeto, iniciamos a produção de documentações relevantes. A fim de documentar as diferentes decisões arquiteturais tomadas ao longo do projeto, adicionamos um documento de arquitetura ao projeto, que possui diagramas que discutem em alto nível as decisões, implicações e *trade-offs* levados em consideração durante a fase de projeto.

Do ponto de vista de gerenciamento do projeto, a principal preocupação foi a de permitir o aumento da utilização de ferramentas de verificação estática do código a fim de encontrar possíveis defeitos ainda em tempo de compilação. Neste sentido, foram adicionadas anotações de tipos estáticos em todas as novas implementações, bem como adicionadas verificações destes tipos a esteira de integração contínua do projeto.

5. Discussão e Trabalhos Futuros

SMAT é uma ferramenta que tenta encontrar conflitos semânticos em um cenário de integração de código a partir da detecção automática de mudanças de comportamento utilizando geração de testes automáticos. Durante este trabalho, tivemos a oportunidade de revisitar, reestruturar e aperfeiçoar diversos aspectos desta ferramenta. Nesta seção, discutimos os resultados dessas modificações e como elas podem contribuir na evolução e manutenção do produto, bem como apresentamos possibilidades de trabalhos futuros.

Em primeiro lugar, é importante destacar que a nova implementação de SMAT permitiu que a ferramenta atingisse um maior nível de configurabilidade, de forma que diversos aspectos possam ser configurados diretamente em tempo de execução, como definir quais serão as ferramentas utilizadas durante a etapa de geração de testes, bem como customizar parâmetros relevantes para a execução como o tempo disponível para a busca e definir quais reportes serão elaborados na etapa de geração de saída.

Outro aperfeiçoamento introduzido na ferramenta foi a possibilidade de execuções determinísticas da ferramenta, permitindo que execuções sucessivas com a mesma entrada tenham sempre o mesmo resultado. Este aperfeiçoamento facilita a reprodutibilidade dos experimentos realizados com a ferramenta, um aspecto que foi observado como um dos principais pontos de melhoria da ferramenta.

Do ponto de vista arquitetural, a ferramenta possui agora um modelo que facilita o entendimento, evolução e manutenção do *software*. Através de um maior entendimento de aspectos importantes do domínio da ferramenta, foi possível introduzir abstrações que permitissem estruturar as diversas entidades da aplicação e como estas se relacionam durante a execução da ferramenta.

Além disso, a nova arquitetura dos módulos facilita a extensibilidade do *software* sem que seja necessário modificar componentes já existentes. Em todos os contextos da aplicação, adicionar novas ferramentas ou funcionalidades se limita a implementar uma única subclasse. É o caso na adição de novos geradores de testes, novos critérios heurísticos para detecção de conflitos e novos geradores de relatórios.

Vale ressaltar também que a remoção do acoplamento entre as etapas de geração, execução e análise dinâmica permite agora que um desenvolvedor responsável por uma tarefa que envolva apenas uma dessas etapas não tenha de se preocupar ou conhecer de-

talhes da implementação das etapas adjacentes, diminuindo a carga cognitiva exigida e reduzindo a probabilidade do surgimento de defeitos.

A introdução de novas interfaces de entrada e saída para os clientes da ferramenta utilizando JSON também contribuiu para facilitar o uso da ferramenta. Ao termos a possibilidade de representar estruturas complexas exigidas pela ferramenta de maneira trivial, sem ser necessário a substituição de caracteres ou utilização de separadores customizados, eliminamos um trabalho considerável do cliente em construir uma entrada que adote as notações utilizadas bem como seja capaz de converter a saída para sua representação interna.

A introdução de uma interface que permitisse que uma mesma execução de SMAT fosse capaz de buscar conflitos semânticos em mais de um alvo, trouxe um ganho de performance considerável, especialmente nos cenários onde vários alvos precisam ser investigados. Além disso, a estratégia introduzida para detectar quais destes alvos foram executados durante um conflito, permitiu também resolver uma das fontes de falsos positivos discutidas pelos autores em trabalhos anteriores.

Como trabalhos teóricos futuros, temos como principal horizonte a expansão de SMAT para que ela seja capaz de atender outros projetos para além da linguagem Java. Vale ressaltar aqui que tal expansão depende da existência de ferramentas que sejam capazes de realizar a geração de testes automáticos para diferentes linguagens, e que estas ferramentas ainda estão em desenvolvimento dentro da literatura.

Outro campo de exploração é verificar os ganhos introduzidos com as nossas contribuições como, por exemplo, analisando se houve melhora na performance da ferramenta, ou verificar como a nova estratégia de detecção de falsos positivos comporta-se na prática em um cenário real.

A nível de implementação, é importante observar que SMAT é apenas uma ferramenta do arcabouço produzido pelo trabalho de [SILVA 2022]. Desta forma, como a ferramenta é consumida por outros clientes, é importante que estes clientes sejam atualizados para utilizar as novas interfaces providas por SMAT. No momento, as interfaces antigas ainda são suportadas, mas foram depreciadas, podendo ser removidas em futuras versões da ferramenta.

Outro aspecto relevante para o projeto é o de aumentar a cobertura de testes automatizados da aplicação. É importante ressaltar que com o advento das execuções determinísticas implementadas neste trabalho, é possível construir valiosas suítes de testes de integração que podem, com baixo custo de desenvolvimento, serem capazes de encontrar com facilidade possíveis regressões em tarefas de desenvolvimento futuras.

Além disso, o isolamento da execução dos geradores de suítes de testes permite explorar possibilidades de execução da geração em paralelo. Embora uma análise menos criteriosa aponte para a possibilidade de um ganho de performance, é necessário considerar como este comportamento influenciaria a qualidade e o tempo disponível para a geração de cada uma das suítes, especialmente quando utilizando a busca exploratória limitada por tempo.

Por último, contribuições na documentação da ferramenta, na remoção de trechos de código não mais utilizados e na implementação de uma política de refatoração contínua

são também possíveis áreas de interesse para trabalhos futuros.

Referências

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

Da Silva, L., Borba, P., and Pires, A. (2022). Build conflicts in the wild. *Journal of Software: Evolution and Process*, 34(4):e2441.

Evans, E. J. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Fraser, G. (2018). A tutorial on using and extending the evosuite search-based test generator. In *International Symposium on Search Based Software Engineering*, pages 106–130. Springer.

Gamma, E., Helm, R., Johnson, R., Johnson, R. E., Vlissides, J., et al. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

Hanenberg, S. (2009). What is the impact of static type systems on programming time. In *PLATEAU Workshop at OOPSLA'09*.

Hoffmann, M., Janiczak, B., Mandrikov, E., and Friedenhagen, M. (2009). Jacoco code coverage tool.

Kleinschmager, S., Robbes, R., Stefik, A., Hanenberg, S., and Tanter, E. (2012). Do static type systems improve the maintainability of software systems? an empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 153–162. IEEE.

Lehtosalo, J., Rossum, G. v., Levkivskyi, I., Sullivan, M. J., Fisher, D., Price, G., Lee, M., Seyfer, N., Barton, R., Ilinskiy, S., et al. (2021). Mypy: Optional static typing for python. *URL: http://mypy-lang. org/[cited 2021-11-30]*.

Ousterhout, J. K. (2018). *A philosophy of software design*, volume 98. Yaknyam Press Palo Alto.

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

Seibt, G., Heck, F., Cavalcanti, G., Borba, P., and Apel, S. (2021). Leveraging structure in software merge: An empirical study. *IEEE Transactions on Software Engineering*, pages 1–1.

SILVA, L. D. (2022). *Detecting, Understanding, and Resolving Build and Test Conflicts*. PhD thesis, Universidade Federal de Pernambuco.

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

A. Estrutura do arquivo de configuração de SMAT

SMAT permite que o usuário customize alguns aspectos de sua execução. Estas modificações podem ser realizadas editando o arquivo *nimrod/tests/env-config.json*. Este documento discute as opções disponíveis para customização.

A.1. Geral

As propriedades a seguir se referem a aspectos gerais da execução de SMAT.

A.1.1. java_home

Por padrão, SMAT utilizará a variável de ambiente *JAVA_HOME* para popular o caminho de instalação do Java. Entretanto, é possível sobrescrever esse valor configurando a variável *java_home* no arquivo de configuração.

A.1.2. maven_home

Por padrão, SMAT utilizará a variável de ambiente *MAVEN_HOME* ou MVN_HOME para popular o caminho de instalação do Maven. Entretanto, é possível sobrescrever esse valor configurando a variável *maven_home* no arquivo de configuração.

A.1.3. logger_level

Esta propriedade permite alterar o nível mínimo de mensagens que serão exibidas pela CLI. O valor padrão é *INFO*, podendo ser alterada para: *CRITICAL*, *ERROR*, *WARNING* ou *DEBUG*.

A.1.4. input_path

Esta propriedade contém o caminho absoluto para o arquivo JSON que contém a descrição dos cenários a serem analisados pela ferramenta.

A.2. Geração de Suítes de Teste

As propriedades a seguir estão relacionadas com aspectos da etapa de Geração de Suítes de Teste.

A.2.1. test_suite_generators

Um *array* com o nome das ferramentas a serem utilizados durante a etapa de geração. Se não for informado, todos os geradores implementados serão utilizados. Valores válidos são: *randoop, randoop-modified, evosuite* e *evosuite-differential*.

A.2.2. test_suite_generation_search_budget

Permite customizar o tempo em segundos disponível para cada gerador durante a etapa de Geração de Suítes de Testes. O valor padrão é de 300 segundos. Observe que esta opção será ignorada se a geração de testes for determinística.

A.2.3. generate_deterministic_test_suites

Se configurada para *true*, SMAT utilizará versões determinísticas de seus gerados, i.e., as suítes geradas serão sempre as mesmas independente de quantas vezes a ferramenta seja executada.

A.3. Output Generation

As propriedades a seguir estão relacionadas com aspectos da etapa de Geração de Reportes.

A.3.1. output_generators

Um *array* contendo os relatórios que devem ser escritos durante a etapa de geração de relatórios. Se não for informada, todos os relatórios implementados serão gerados. Valores válidos são: *behavior_changes, semantic_conflicts, test_suites*.