UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

IARA RAMOS BRITO

# Uma Metodologia para a Geração de Testes Unitários Baseada em Extração de Modelos

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Lucio Mauro Duarte
Co-orientadora: Profa. Dra. Érika Fernandes Cota

Porto Alegre
2021

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Brito, Iara Ramos
Uma Metodologia para a Geração de Testes Unitários Baseada em Extração de Modelos / Iara Ramos Brito. -- 2021.
72 f.
Orientador: Lucio Mauro Duarte;

Coorientadora: Érika Fernandes Cota.

Dissertação (Mestrado) -- Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2021.
1. Verificação de Modelos. 2. Testes Unitários. 3. Extração de Modelos. I. Duarte, Lucio Mauro, orient. II. Cota, Érika Fernandes, coorient. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões
Vice-Reitor: Profª. Patricia Pranke
Pró-Reitor de Pós-Graduação: Profª. Cíntia Inês Boll
Diretor do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenadora do PPGC: Profª. Luciana Salete Buriol
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

## AGRADECIMENTOS

Á minha mãe, que nunca mediu esforços para investir em minha educação, que sempre me incentivou a estudar e sempre acreditou em mim.

Ao meu companheiro Braulio Nogueira de Oliveira pelo apoio fundamental dado na construção deste trabalho, pelo incentivo e compreensão.

Ao meu orientador professor Lucio Mauro Duarte e minha co-orientadora professora Érika Fernandes Cota, pelas orientações, repasses de conhecimentos e sugestões valiosas dadas ao longo da construção deste trabalho.

Á Universidade Federal do Rio Grande do Sul, pelo apoio concedido.

Á Deus, por tantas coisas boas que me concedeu.

## RESUMO

A geração de testes em sistemas já existentes, sem documentação atualizada ou sem testes legados, usualmente não é uma tarefa fácil, pois exige do desenvolvedor o trabalho de descobrir ou supor as regras que os testes devem executar para testar o sistema corretamente. Neste trabalho, propõe-se uma metodologia com o objetivo de possibilitar a criação semiautomática de testes unitários nesse cenário. A metodologia propõe a integração das técnicas de testes e verificação de modelos, utilizando a extração de modelos. A extração de modelos é usada para possibilitar a obtenção de um modelo de comportamento a partir de rastros de execução de alguns casos de teste executados sobre o código de interesse. Este modelo é analisado em uma ferramenta de verificação de modelos, gerando contraexemplos, os quais descrevem caminhos a partir do estado inicial que são diferentes daqueles constantes no modelo atual. Estes contraexemplos gerados são analisados a fim de verificar se permitem a criação de algum teste válido. A criação de um novo teste implica a geração de um novo rastro e, por consequência, a construção de um novo modelo contendo este novo comportamento do sistema, o que torna a abordagem iterativa. O processo de extração de modelos utilizado neste trabalho usa as ferramentas Labelled Transition Systems Extractor (LTSE) para a geração de um modelo no formato de Labelled Transition Systems (LTS) a partir de rastros deixados na execução da classe instrumentada. Nos experimentos realizados, foi analisada a eficácia da metodologia para a criação de testes unitários. Para isso, foram analisadas quatro classes de quatro sistemas de acesso público. A metodologia foi aplicada nessas quatro classes e foram verificadas três questões principais em relação a metodologia: se a metodologia permite a geração de testes, se os testes gerados permitiam o aumento de cobertura de branch ou linha e se a metodologia permite descobrir erros. Os resultados obtidos no experimento mostraram que, em geral, a metodologia permite a geração de testes e que a geração desses testes pode implicar o aumento de cobertura de branch ou linha. Foram realizadas análises dos resultados dos experimentos avaliando as possíveis relações entre algumas métricas de software de cada classe e os seus diferentes resultados obtidos na aplicação da metodologia. Além disso, foi mostrado no trabalho como a metodologia pode contribuir para a descoberta de erros na aplicação.



**Palavras-chave**: Testes unitários. Extração de Modelos. Verificação de Modelos.

**A Methodology for the Generation of Unit Tests Based on Model Extraction**

## ABSTRACT

Generating tests for systems without documentation, documentation out-of-date or without legacy tests is not easy task because the developer has to discover the rules that tests must follow to correctly test the system. In this work, a methodology is presented to allow the semi-automatic creation of unit tests in this scenario. The methodology proposes the integration of Software Testing and Model Checking through Model Extraction. Model Extraction allows the generation of a behavior model from recorded execution traces, which can be obtained during the execution of some test performed on the code of interest. The generated model can then be analyzed in a model checking tool to generate counterexamples, which describe paths that are different from those currently in the model. The generated counterexamples are analyzed in the methodology in order to verify whether they allow the creation of any valid new test. The approach of the methodology is iterative because the generation of a new test implies the generation of a new trace and, consequently, the generation of a new model that includes the new observed behavior. The model extraction process applied in this work uses the Labelled Transition Systems Extractor (LTSE) tool to generate a model as a Labelled Transition System (LTS) through traces recorded during the execution of an instrumented component. In the experiments, the effectiveness of the methodology for creating unit tests was analyzed. For this analysis, four open source systems were used. The methodology was applied to components of these systems and three research questions were verified in relation to the methodology: whether the methodology allowed the generation of new tests, whether the generated tests allowed an increase in branch and/or line coverage, and whether the methodology could be used to uncover errors. The results of the experiments showed that, in general, the methodology allows the generation of new tests and that these new tests can lead to an increase in branch and line coverage. The possible relations between some software metrics for each component and their different results obtained in the application of the methodology were evaluated. In addition, it was shown how the methodology can contribute to the discovery of errors in the application.

**Keywords**: Unit Testing. Model Checking. Model Extraction.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| LTSE | Labelled Transition Systems Extractor |
| LTSA | Labelled Transition Systems Analyser |
| LTS | Labelled Transition Systems |
| FSP | Finite State Processes |
| LP | Lista de Possíveis Casos de Testes |
| LT | Lista de Casos de Testes Criados |
| LN | Lista de Contraexemplos Não Executáveis |

**SUMÁRIO**

# 1 INTRODUÇÃO

Sistemas de software estão cada vez mais presentes nas mais diversas áreas da sociedade, e tais sistemas estão gradativamente mais complexos e mais presentes no dia a dia (BAIER; KATOEN, 2008). Considerando-se software utilizado em aplicações que controlam operações complexas, como na área financeira e em controles aviônicos, é natural que se espere que estes nunca falhem (AMMANN; OFFUTT, 2008), pois falhas nestes sistemas podem causar danos irreparáveis. Assim, cada vez mais tem-se buscado aumentar a correção de sistemas, principalmente, os mais complexos, que exigem maior esforço de análise. Além da questão da crescente necessidade de correção de sistemas, muitas vezes é necessário gerar testes em sistemas já existentes e que não possuem documentações ou mesmo documentações atualizadas. Nesse sentido, a geração de teste em sistemas nesse contexto descrito não é uma tarefa fácil. O presente trabalho explora e propõe uma solução para a geração de testes nesse contexto descrito.

A técnica de teste de software tem sido utilizada como padrão para identificar erros em sistemas por muito tempo (BEYER; LEMBERGER, 2017). Teste de software (AMMANN; OFFUTT, 2008) pode ser aplicada em todos os níveis de desenvolvimento e em diferentes tipos de software. Apesar disto, um dos principais problemas desta técnica é permitir apenas a identificação da presença do erro, não a sua ausência. O teste de software permite aumentar a confiabilidade do programa (MYERS; BADGETT; SANDLER, 2012). Nesse sentido, aumentar a confiabilidade do programa significa encontrar e remover erros. Na criação de testes realizada manualmente existe a possibilidade de influência por parte do programador, tais como: experiência do testador e propósito de geração do teste.

Outras técnicas de verificação de software são as baseadas em métodos formais, em especial as baseadas em modelos abstratos. Estas técnicas são mais eficazes na verificação de sistemas complexos, oferecendo uma análise do sistema com rigor matemático em um determinado nível de abstração. Em projetos de sistemas de software complexos, foi constatado que pode haver um maior gasto de tempo e maior esforço na fase de verificação do que na própria construção da aplicação (BAIER; KATOEN, 2008), por isso técnicas de verificação que facilitem e reduzam o esforço na verificação são essenciais. Um dos principais motivos para isto envolve o esforço necessário e conhecimento técnico requerido para se obter um modelo do sistema, necessário para a análise. A verificação de modelos (CLARKE, 1999) é uma técnica formal usada para verificação de sistemas a partir de modelos. Apesar de permitir

análises que seriam impossíveis no sistema real, construir um modelo nem sempre é uma tarefa trivial.

Um modelo de comportamento é obtido através do processo de extração de modelos. A extração de modelos (HOLZMANN; SMITH, 1999) é o processo pelo qual é possível gerar automaticamente um modelo a partir de um sistema existente. Dessa forma, é possível obter-se um modelo para habilitar o uso das técnicas baseadas na análise de modelos.

Na entrega de projetos de desenvolvimento de sistemas, cada vez mais há uma constante necessidade de rapidez para cumprir prazos e muitos sistemas, quando entregues, não apresentam documentações ou mesmo têm a sua documentação atualizada quando há alterações no sistema. Além disto, é comum que ocorram atualizações e alterações ao longo do tempo em sistemas devido aos mais diversos fatores. No que se refere à documentação destes sistemas, quando existentes, muitas vezes é possível que não sejam atualizadas no momento das alterações devido ao custo e esforço necessário. Em relação aos testes legados, quando são utilizados para verificar os sistemas, quando existentes, muitas vezes permitem uma análise apenas superficial e podem acabar não sendo atualizados quando ocorrem alterações nos sistemas, especialmente devido ao custo dos testes para o projeto. Diante destes problemas, propomos uma metodologia que possibilita integrar a técnica de testes de softwares, e de verificação de modelos, utilizando como base a extração de modelos. A metodologia tem como objetivo principal permitir a geração de testes não aleatórios, em particular, em sistemas que não possuem documentação ou têm documentação desatualizada, ou ainda, em sistemas sem testes legados. Os objetivos específicos são: extrair um modelo do sistema já existente e assim obter pelo menos um tipo de documento para o sistema; gerar testes a partir de contraexemplos que podem ser gerados na análise do modelo extraído do sistema.

A integração de verificação de modelos e teste de software utilizando a extração de modelos, na qual essa metodologia é baseada, foi proposta em Duarte (2011). Os trabalhos de Beyer et al. (2004), Seijas e Thompson (2018), Walkinshaw, Derrick e Guo (2009), Majerkowski (2012), também serviram como base para a construção da metodologia, principalmente o trabalho de Majerkowski (2012), que emprega a integração de verificação de modelos e teste de software para melhoria de detecção de erros em sistemas.

A principal contribuição da metodologia é possibilitar a geração de testes mesmo quando não há documentações atualizadas dos requisitos do sistema, combinando teste de software, verificação de modelos e extração de modelos. Apesar da indisponibilidade da documentação não ser o cenário ideal em sistemas de software, é comumente visto em muitas empresas. Além disso, como a metodologia se baseia no uso de modelos e ferramentas de

análise, mesmos testadores inexperientes podem tirar proveito de seus resultados. Nesse sentido, utilizamos ferramentas de geração de testes automática para criar um conjunto de testes necessário para iniciar a aplicação da abordagem de extração de modelos utilizada na metodologia. Isto visa suprir a inexperiência de testadores, sendo que os passos seguintes são dirigidos pela análise do modelo gerado. Assim, a metodologia tem como foco, principalmente, auxiliar testadores.

Na construção da metodologia e ao longo dos experimentos realizados foi percebido que é necessário que o usuário aplicador da metodologia tenha conhecimento básico de teste de software, tais como: propósito do teste e estrutura básica de testes. Além disso, tenha conhecimentos básicos de verificação de modelos e saiba utilizar a ferramenta de verificação de modelos utilizada no experimento. Outro requisito necessário é que o usuário tenha conhecimento básico sobre a abordagem de extração de modelos utilizada no experimento, tais como: processo para a geração de modelos, conhecimento do conceito de rastros e de propriedades e contraexemplos.

O trabalho está organizado em capítulos conforme a descrição que segue: o Capítulo 2 introduz conceitos teóricos fundamentais para o entendimento do trabalho; o Capítulo 3 apresenta todos os trabalhos que influenciaram a construção da metodologia; o Capítulo 4 descreve a metodologia e como foi criada; o Capítulo 5 apresenta os experimentos realizados com a aplicação da metodologia, procedimento, discussões e resultados; e o Capítulo 6 apresenta todas as conclusões e discussões sobre trabalhos futuros.

# 2 CONCEITOS TEÓRICOS

Neste capítulo serão apresentados todos os conceitos e definições utilizados na composição da metodologia envolvendo teste de software, verificação de modelos e extração de modelos. Os conceitos de testes de softwares foram utilizados na fase inicial da metodologia e são de fundamental importância para a execução do sistema e geração dos rastros necessários para a geração do modelo de comportamento do sistema, conforme abordagem de extração de modelos (DUARTE, 2007). Na metodologia, essa abordagem de extração de modelos foi utilizada para permitir a obtenção do modelo de comportamento do sistema. Na metodologia, verificação de modelos foi utilizada na análise do modelo extraído da execução do sistema, o que pode possibilitar a geração de contraexemplos, utilizados como base na construção dos testes resultantes da aplicação da metodologia.

## 2.1 Teste de Software

Teste de software (AMMANN; OFFUTT, 2008) pode ser definido como a avaliação do sistema baseada na observação de sua execução. Além disto, é a técnica de verificação mais utilizada comumente no processo de desenvolvimento de sistemas. A técnica de teste de software pode ser aplicada em todos os níveis do desenvolvimento. O teste de unidade pode ser considerado como um método sem parâmetros que executa uma sequência de chamadas de métodos de forma a executar o código sob teste (TILLMANN; DE HALLEUX; XIE, 2010). Este tipo de teste possibilita a análise no nível mais elementar do sistema, evitando, possivelmente, que erros gerem falhas no sistema (AMMANN; OFFUTT, 2008). O teste de unidade pode ser considerado como um importante meio de melhorar a confiabilidade do software, por permitir a identificação de erros no início do desenvolvimento de software (TILLMANN; DE HALLEUX; XIE, 2010). Ele permite verificar unidades do código fonte, em que uma unidade pode ser considerada uma parte do código que exibe um comportamento útil no sistema (LANGR; HUNT; THOMAS, 2015).

Como é comumente impossível testar-se um sistema por completo, são utilizados critérios de seleção de testes. Entre tais critérios, existem os critérios baseados na cobertura do conjunto de testes. Um critério de cobertura pode ser definido como uma ou mais regras que impõem requisitos de teste em um conjunto de testes. Nesse contexto, um requisito de teste é um elemento específico de um artefato de software que um caso de teste deve satisfazer (AMMANN; OFFUTT, 2008). Dentre os vários tipos de cobertura, os mais básicos são as

coberturas de linha e de *branch* por permitirem medir a cobertura em nível de código fonte. A cobertura de linha, ou cobertura de código, requer que cada linha do código seja executada ao menos uma vez (BURR; YOUNG, 1998). Já a cobertura de *branch* requer cobrir todos os pontos de decisão do programa. Assim, na medição da cobertura, existem avaliações a dois requisitos de teste, um para a decisão de avaliar como falso e um para a decisão de avaliar como verdadeiro (AMMANN; OFFUTT, 2008).

Em geral, os seres humanos tendem a cumprir instintivamente os objetivos os quais se propõem. Nesse sentido, se, por exemplo, o objetivo do testador é demonstrar que o sistema não tem erros, inconscientemente se tende a criar testes que tenham grande probabilidade de não causar falha no sistema e vice-versa (MYERS; BADGETT; SANDLER, 2012). Assim, neste trabalho, entendemos que os novos testes propostos gerados por meio da metodologia são *testes não aleatórios*, pois são baseados em sugestões geradas pela ferramenta de verificação e, portanto, não sofrem possíveis influências do objetivo do testador. Uma das principais limitações da técnica de teste de software é que ela permite apenas a identificação da presença do erro e não a sua ausência (AMMANN; OFFUTT, 2008). Apesar do surgimento crescente de ferramentas cada vez mais poderosas que auxiliam nos testes ou mesmo geram testes automaticamente, em geral não é possível realizar-se uma análise completa do software.

## 2.2 Verificação de modelos

Verificação de modelos (CLARKE et. al. 2018) é uma técnica de verificação de software baseada em métodos formais em que, dados um modelo de estados finito do sistema e uma especificação, é realizada uma verificação exaustiva do espaço de estados a fim de identificar violações da especificação (BAIER; KATOEN, 2008). Técnicas baseadas em métodos formais, tais como verificação de modelos, oferecem a possibilidade de melhorar a correção de sistemas, principalmente em sistemas mais complexos (BAIER; KATOEN, 2008). Através do uso de técnicas baseadas em métodos formais é possível descobrir possíveis ambiguidades, incompletudes e inconsistências nos sistemas. A aplicação de tais técnicas também é facilitada por ferramentas usadas para automatizar grande parte do processo de verificação.

A verificação de modelos é constituída por etapas de modelagem, especificação e verificação (CLARKE et. al. 2018). A modelagem consiste na etapa em que um modelo de comportamento do sistema é criado, sendo importante destacar que o modelo precisa corresponder ao formato de modelos aceito pela ferramenta de verificação de modelos utilizada.

A especificação consiste na etapa em que as propriedades que o modelo deve atender (especificação) são definidas. A etapa de verificação é a verificação realizada, idealmente, por uma ferramenta em que é analisado se o modelo atende às propriedades descritas.

Existem várias ferramentas disponíveis gratuitamente que possibilitam o uso da técnica de verificação de modelos em diferentes tipos de modelos do sistema, tais como LTSA[1] e Spin[2] . Ferramentas como estas permitem a checagem de todos os possíveis estados do modelo do sistema em relação às propriedades que deve atender. Quando o modelo não atende a alguma das propriedades definidas, a ferramenta encontra a violação e gera um contraexemplo (BAIER; KATOEN, 2008). Contraexemplos descrevem um caminho a partir do estado inicial que viola a propriedade por permitir um comportamento não desejado.

### 2.1.1 Modelos de comportamento

Como já mencionado, para aplicar a técnica de verificação de modelos é necessário que haja um modelo do sistema. A criação de um modelo de comportamento não é uma tarefa trivial, exigindo muitas vezes conhecimentos técnicos especializados. É possível somente a análise de modelos de estados finitos, o que permite que a verificação possa ser realizada de forma automatizada (CLARKE et. al. 2018).

Modelos de comportamento são representações abstratas do comportamento de um sistema baseado em fundamentos matemáticos (UCHITEL; KRAMER; MAGEE, 2003). Modelos podem ser descritos por meio de máquinas de estados (MAGEE; KRAMER, 2006). São usualmente representados utilizando o conceito da computação chamado sistemas de transição (BAIER; KATOEN, 2008), que consiste basicamente em um grafo direcionado em que os nodos representam estados do sistema e os arcos representam transições realizadas na mudança de um estado para outro.

Um modelo de comportamento pode ser de dois tipos básicos (DUARTE, 2007): abordagem baseada em ação e abordagem baseada em estado. A abordagem baseada em estado abstrai as ações e leva em consideração rótulos na sequência de estados (BAIER; KATOEN, 2008). Já a abordagem baseada em ações leva em consideração apenas os rótulos nas transições, abstraindo os estados no modelo. Neste contexto, ação pode ser uma referência a chamada de método ou qualquer outro evento significativo (MAGEE; KRAMER, 2006).

---

1 Disponível em: https://www.doc.ic.ac.uk/ltsa/. Acesso em: 30 ago. 2020.
2 Disponível em: http://spinroot.com/spin/whatispin.html. Acesso em:30 ago. 2020.

O modelo *Labelled Transition Systems* (LTS) (KELLER, 1976) é um modelo baseado em ações que tem propriedades matemáticas bem definidas, podendo ser utilizado na modelagem de sistemas sequenciais, concorrentes ou distribuídos (MAGEE; KRAMER, 2006). Um LTS pode ser definido como um grafo direcionado com um rótulo em cada arco, em que o rótulo define a ação associada ao arco. O modelo LTS é baseado em máquinas de estados e é geralmente usado para descrever modelos de comportamento de acordo com um determinado nível de abstração (UCHITEL; KRAMER; MAGEE, 2003).

Um LTS $M = (S, si, \Sigma, T)$ é um modelo em que:

- $S$ é um conjunto finito de estados;
- $si \in S$ representa um estado inicial;
- $\Sigma$ é um alfabeto (conjunto de nomes das ações);
- $T \subseteq S \times \Sigma \times S$ é a relação de transição entre estados (MAGEE; KRAMER, 2006).

Transições são rotuladas com nomes de ações que causam a progressão do sistema do estado corrente para o novo estado. Assim, sendo dados dois estados $s0, s1 \in S$ e a ação $a \in \Sigma$, a transição $(s0, a) = s1$ significa que é possível ir do estado $s0$ para o estado $s1$ através da execução de uma ação de nome $a$ (MAGEE; KRAMER, 2006).

## 2.3 Extração de modelos

Para que qualquer análise sobre um modelo possa ser confiável, é essencial que ele seja preciso e represente todos os requisitos mais relevantes do sistema (WALKINSHAW; DERRICK; GUO, 2009). Por isto, a criação manual de modelos pode custar caro, ser propensa ao erro, além de exigir conhecimentos técnicos específicos. Alguns trabalhos (DUARTE, 2007), (WALKINSHAW; DERRICK; GUO, 2009) propõem abordagens de geração de modelo a partir do sistema real, chamada extração de modelo. Extração de modelo (HOLZMANN; SMITH, 1999) é o processo em que é possível gerar automaticamente um modelo a partir de um sistema existente. Devido ao processo ser automático, diminui riscos de erros no modelo em relação ao processo realizado manualmente.

A abordagem de extração de modelos descrita em (DUARTE, 2007), que é realizada usando o conceito de contextos, foi a escolhida para ser utilizada na metodologia proposta neste trabalho. A escolha desta abordagem se deve ao fato de poder ser aplicada em sistemas desenvolvidos na linguagem Java, já que Java é uma das linguagens mais utilizadas no mundo, e ter sido usada com sucesso para geração de modelos para diversas análises (DUARTE, KRAMER; UCHITEL, 2017). Além disso, também foram consideradas a presença de algumas

características de grande relevância no processo de extração de modelos, tais como: simplicidade, baixo tempo de construção, fidelidade e geração do modelo em linguagem suportada por ferramenta de verificação de modelos. Em relação à característica que se refere à simplicidade consiste no quão simples é construir o modelo do sistema através do processo de extração como um todo. Já a característica referente ao baixo tempo de construção no tempo gasto no processo como um todo deve ser atrativo em relação a outras propostas (DUARTE, 2007).

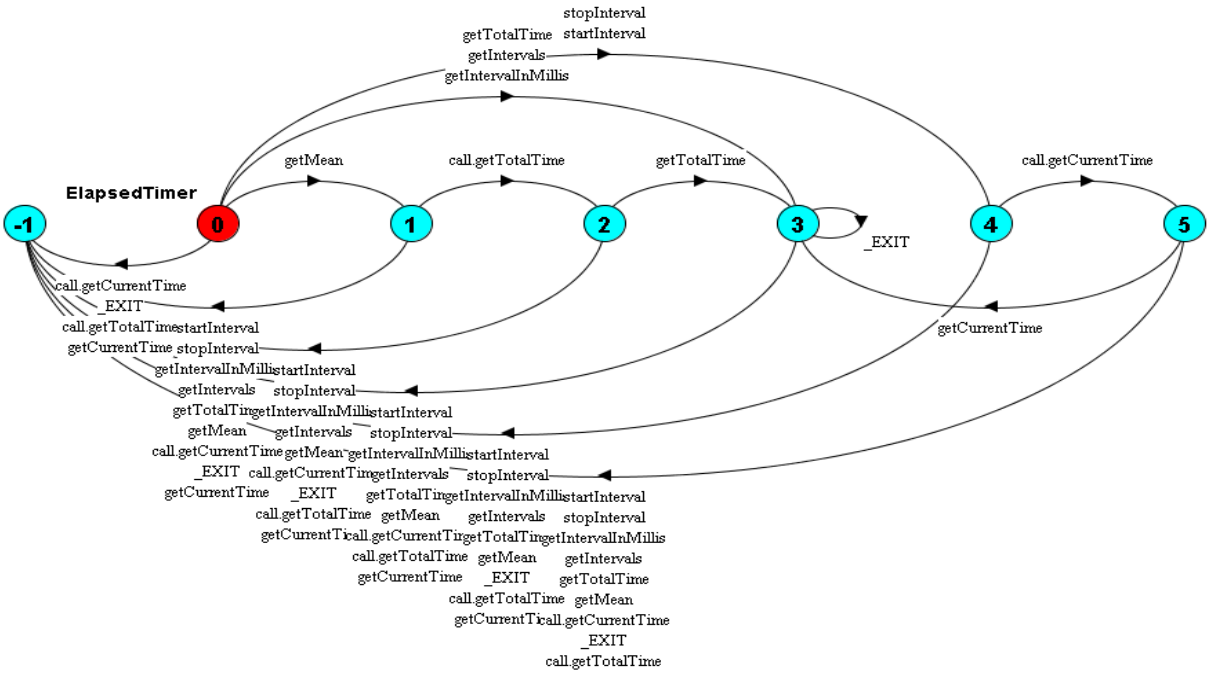
Inicialmente, a extração de modelos envolve a instrumentação do código do sistema de forma que pontos estratégicos recebam anotações através da ferramenta TXL (CORDY, 2002). As anotações permitem que informações estáticas e dinâmicas necessárias para formação do modelo de comportamento do sistema possam ser obtidas através de rastros deixados na execução do sistema. Os rastros obtidos na execução do sistema contêm informações do fluxo de controle e de valores de atributos do sistema (informação de contexto). A partir dos rastros gerados na execução do sistema instrumentado, o modelo de comportamento do sistema pode ser gerado através da ferramenta *Labelled Transition Systems Extractor* (LTSE). Resumidamente, a ferramenta LTSE combina as informações dos rastros, com base nas informações de contexto, de forma a gerar um modelo no formato LTS que serve de entrada para a ferramenta de verificação LTSA. Contexto se refere a combinações de informações estáticas e dinâmicas registradas nas execuções de ações no sistema (DUARTE, 2007).

Na abordagem da extração de modelos utilizada neste trabalho (DUARTE, 2007) é usada a ferramenta LTSA na fase de verificação de modelos. Na análise do modelo realizada na LTSA as propriedades devem ser verdadeiras para qualquer possível execução do sistema (MAGEE; KRAMER, 2006). Estas propriedades são divididas em categorias dentre as quais estão as propriedades de *Safety*, que serão utilizadas neste trabalho. Uma propriedade define um conjunto de comportamentos considerados corretos, sendo que comportamentos fora desse conjunto são definidos como incorretos. Uma propriedade de *Safety* define um processo determinístico no qual qualquer rastro ou ação inserido no alfabeto do modelo é aceito como verdadeiro. As ações que não fazem parte do comportamento descrito no modelo como verdadeiro são representadas no modelo, após a análise do modelo, como transições para o estado de erro (-1).

O modelo LTS pode ser descrito em forma de código textual da álgebra de processos *Finite State Processes* (FSP), usado como entrada para a LTSA. Assim, quando aplicado na ferramenta permite a análise e visualização gráfica do modelo LTS (MAGEE; KRAMER, 2006).

No código descrito em FSP e analisado por meio da ferramenta LTSA, um processo descrevendo uma propriedade é identificado por meio da palavra de controle *property* antecedendo a descrição do processo. Para que um processo seja definido como uma propriedade é necessário que ele seja determinístico, o que visa a facilitar a sua verificação. O LTSA permite a determinização de um modelo por meio da execução do acréscimo da palavra de comando *deterministic* antecedendo a descrição do processo que cria o modelo.

Figura 2.1 – Modelo no formato LTS com representação dos contraexemplos.




Fonte: Autor

Na Figura 2.1 está representado um modelo da classe ElapsedTimer[3], disponível no GtiHub. O modelo LTS foi obtido através da ferramenta LTSA, tendo sido extraído da classe com o uso da abordagem de extração de modelos. Os estados do modelo representados são: 0, 1, 2, 3, 4, 5 e o estado de erro -1. O alfabeto é composto pelas ações: *startInterval*, *stopInterval*, *getIntervalInMillis*, *getIntervals*, *getTotalTime*, *getMean*, *call.getCurrentTime*, *call.getTotalTime*, *getCurrentTime*, *getTotalTime*. Como o estado -1 representa o estado de erro, qualquer sequência de ações que leve a este estado é considera inválida, definindo um

[3] Disponível em: https://github.com/nasa/mct/blob/master/util/src/main/java/gov/nasa/arc/mct/util/internal/ElapsedTimer.java. Acesso em: 21 dez. 2020.

contraexemplo. No caso do modelo da Figura 2.1, por exemplo, a sequência *getMean* → *call.getTotalTime* → *stopInterval* é um contraexemplo, já que leva do estado inicial (0) até o estado de erro. Isto indica que tal sequência é considerada incorreta em relação à propriedade descrita.

# 3 TRABALHOS RELACIONADOS

A metodologia proposta neste trabalho foi baseada em trabalhos publicados que envolvem a integração das técnicas teste de software e verificação de modelos com o objetivo de aumentar a correção dos sistemas. Chen, Zhou e Bruda (2008) realizaram análises e descrição das características das técnicas de testes de software e verificação de modelos separadamente na aplicação e as características e benefícios da aplicação da integração de tais técnicas. O objetivo do trabalho era demonstrar que a combinação das técnicas de teste de software e verificação de modelos poderia ser benéfica e resolver problemas que podem aparecer na aplicação de somente uma das técnicas. Uma das conclusões descritas no trabalho foi de que as técnicas de teste de software e verificação de modelos são duas formas complementares de aumentar a confiabilidade da aplicação. Na proposta descrita no trabalho, a técnica de teste foi usada para analisar os resultados da verificação de modelos e para identificar diferenças entre o modelo abstrato e o código da aplicação.

O trabalho de Walkinshaw, Derrick e Guo (2009) destaca as dificuldades encontradas na engenharia reversa de se obter uma quantidade de amostras de rastros suficientes para gerar um modelo mais preciso. Nesse sentido, o trabalho propõe uma técnica iterativa de engenharia reversa para se obter um modelo de comportamento do sistema, a partir do código da aplicação real em um sistema desenvolvido na linguagem de programação Erlang. O trabalho propõe também que o modelo de comportamento do sistema, na forma de um *Partial LTS*, seja obtido a partir do sistema real, utilizando para isto teste baseado em modelo e verificação de modelos. Além disso, apresenta como novidade a geração de testes baseados em modelo como forma de melhorar o modelo de comportamento do sistema o quanto for possível. O trabalho concluiu que os modelos extraídos dos sistemas mostraram ser bem precisos tanto em termos de estrutura de grafos quanto na linguagem representada pelo modelo. Assim como no trabalho de Walkinshaw, Derrick e Guo (2009), a metodologia aqui descrita também é baseada na ideia de geração do modelo de comportamento a partir do sistema real e utiliza abordagem semelhante de geração de testes baseados no modelo de forma a aumentar a correção do sistema. No entanto, a geração do conjunto inicial de testes é realizada com uma abordagem diferente no sentido de que é proposta a criação de testes com a utilização de ferramentas de geração automática de testes.

O trabalho descrito em Beyer et al. (2004) propõe a extensão da ferramenta de verificação de modelos BLAST para gerar testes automaticamente para programas desenvolvidos na linguagem C com cobertura total em relação a um determinado predicado, sendo que um predicado é uma expressão com valor booleano. Segundo os autores, informações obtidas em uma ferramenta tradicional de verificação de modelos são limitadas. Nesse sentido, foi feita a extensão da ferramenta BLAST de tal forma que fosse possível obter-se o conjunto de todos os locais do programa *Q* tal que o predicado *p* seja verdadeiro. Segundo Beyer et al. (2004), o uso da ferramenta BLAST permite a geração de suítes de testes mais informativas do que qualquer outra ferramenta baseada somente em cobertura. Isto ocorre devido ao fato de a ferramenta permitir a identificação do conjunto de todos os locais de onde o programa apresentou a propriedade violada e não apenas um traço do programa que viola uma determinada propriedade temporal como os verificadores de modelos tradicionais permitem A metodologia proposta neste trabalho se baseou na ideia do artigo de Beyer et al. (2004) de gerar testes a partir de uma ferramenta de verificação de modelos. No entanto, na metodologia proposta, os testes gerados não são obtidos de forma automática, mas de forma manual, em que cada contraexemplo gerado pela ferramenta de verificação de modelos é analisado para verificar se pode gerar um caso de teste.

Em Duarte (2011) a utilização da extração de modelos foi proposta como forma de integrar as técnicas de teste de software e verificação de modelos. Nesse trabalho, o modelo do sistema era extraído a partir da execução de casos de testes do sistema real e no modelo gerado era aplicada a técnica de verificação de modelos. Baseado em Duarte (2011), Majerkowski (2012) aplicou a abordagem proposta, utilizando um conjunto inicial de testes criado a partir do conhecimento prévio dos requisitos do sistema, com o objetivo de aumentar a cobertura dos testes e a completude do modelo do sistema. Assim, para a aplicação do algoritmo proposto em Majerkowski (2012) era necessário que o documento de especificação do sistema fosse fornecido previamente. A metodologia aqui proposta baseia-se nessa abordagem de integração. No entanto, diferente do que é utilizado no trabalho de Majerkowski (2012), que tem como requisito primordial o conhecimento de testes para a criação do conjunto inicial de testes para a executar o sistema, a metodologia proposta não exige o conhecimento técnico de testes e nem o conhecimento da documentação de especificação do sistema. Nesse sentido, o trabalho aqui descrito contribui para a geração de testes também por desenvolvedores sem conhecimentos em técnicas de testes de softwares.

Em Seijas e Thompson (2018) foi descrita uma abordagem de criação de novos testes a partir de testes unitários legados e do modelo de máquinas de estados do sistema inferido destes testes legados. Seijas e Thompson (2018) utilizaram, na criação dos novos testes, a técnica de testes *Property-based testing (PBT)*. Esta é uma técnica que usa entradas de testes aleatórias de forma controlada para verificar se o sistema que está sob teste está em conformidade com um conjunto de propriedades esperadas (ARTS *et al.*, 2006). A ideia descrita pelos autores de gerar novos testes a partir de modelos de máquinas de estados do sistema inferidos de testes unitários legados também é utilizada nesta metodologia. No entanto, em vez da utilização de testes legados são utilizados testes gerados de forma automática por ferramentas.

Villani et. al. (2019) analisa o uso das técnicas de verificação e validação no desenvolvimento de softwares na área industrial: verificação de modelos e teste baseado em modelo. No trabalho, a escolha dessas duas técnicas tem como base que as duas técnicas possuem características complementares. Além disso, o trabalho aborda a integração de tais técnicas de forma automática para tornar tal integração mais eficiente, utilizando a ferramenta de verificação de modelos UPPAL e a ferramenta de teste baseado em modelo ConData. O trabalho explora dois processos de desenvolvimento: um processo busca identificar se teste baseado em modelo pode contribuir para identificar lacunas na especificação e suposições implícitas feitas por engenheiros ao aplicar a verificação de modelo; o outro processo foca em identificar como a verificação de modelo pode melhorar o desenvolvimento e verificação dos modelos que são usados para testes baseados em modelos. Nos resultados apresentados no trabalho são analisadas as vantagens e limitações dos processos de desenvolvimento apresentados, utilizando ferramentas de verificação de modelos e testes baseado em modelo e, além disso, compara os resultados obtidos com estudos realizados em trabalhos anteriores que utilizaram uma ou ambas as técnicas. A metodologia proposta neste trabalho também explora a integração de duas técnicas de verificação e validação considerando as características complementares de tais técnicas: testes de software e verificação de modelos. No entanto, na metodologia usamos a técnica de testes de forma diferente, pois exploramos a problemática de geração de testes em sistemas sem documentação e/ou sem testes. Assim, propomos a utilização da testes de software para a obter um conjunto de testes inicial utilizado na abordagem de extração de modelos e na geração manual de testes a partir do modelo extraído do sistema, pois a utilização dessa abordagem na metodologia possibilita

a geração de uma documentação através do modelo extraído do sistema e ainda permite a geração de testes não aleatórios.

O trabalho de Beyer e Lemberger (2017) faz um comparativo da evolução de teste de software e verificação de modelos, através da comparação de seis ferramentas de geração automática de casos de testes e quatro ferramentas de verificação de modelos. Para realizar a comparação, os autores criaram um framework de *test-based falsification (TBF)* que permite a comparação de diferentes casos de testes e com verificação de modelos para encontrar erros em programas. O trabalho de Beyer e Lemberger (2017) concluiu que ferramentas de verificação de modelos avaliadas são competitivas em relação a ferramentas de geração de testes automáticas avaliadas no trabalho. Nesse sentido, o nosso trabalho utiliza a integração de verificação de modelos e testes de software como forma de melhorar a correção do sistema e assim obter um melhor resultado do que a utilização isolada de uma das técnicas. Conforme já explicitado, para a integração das técnicas citadas utilizamos a extração de modelos para permitir a geração de testes unitários a partir do código fonte.

# 4 METODOLOGIA

A metodologia proposta neste trabalho foi criada, principalmente, para facilitar a geração de testes unitários em um contexto de sistemas reais. Consideram-se, em especial, sistemas que não possuem especificações documentadas, com ou sem testes legados, ou que possuam documentação desatualizada. Em teoria, esse contexto não é o ideal no desenvolvimento e manutenção de sistemas, mas representa, muitas vezes, o cenário comumente encontrado em empresas, muito por conta das limitações de prazo e recursos. Além disso, a metodologia foi pensada de forma que pudesse ser aplicada mesmo por desenvolvedores com pouca experiência na criação de testes de software e no uso das técnicas envolvidas.

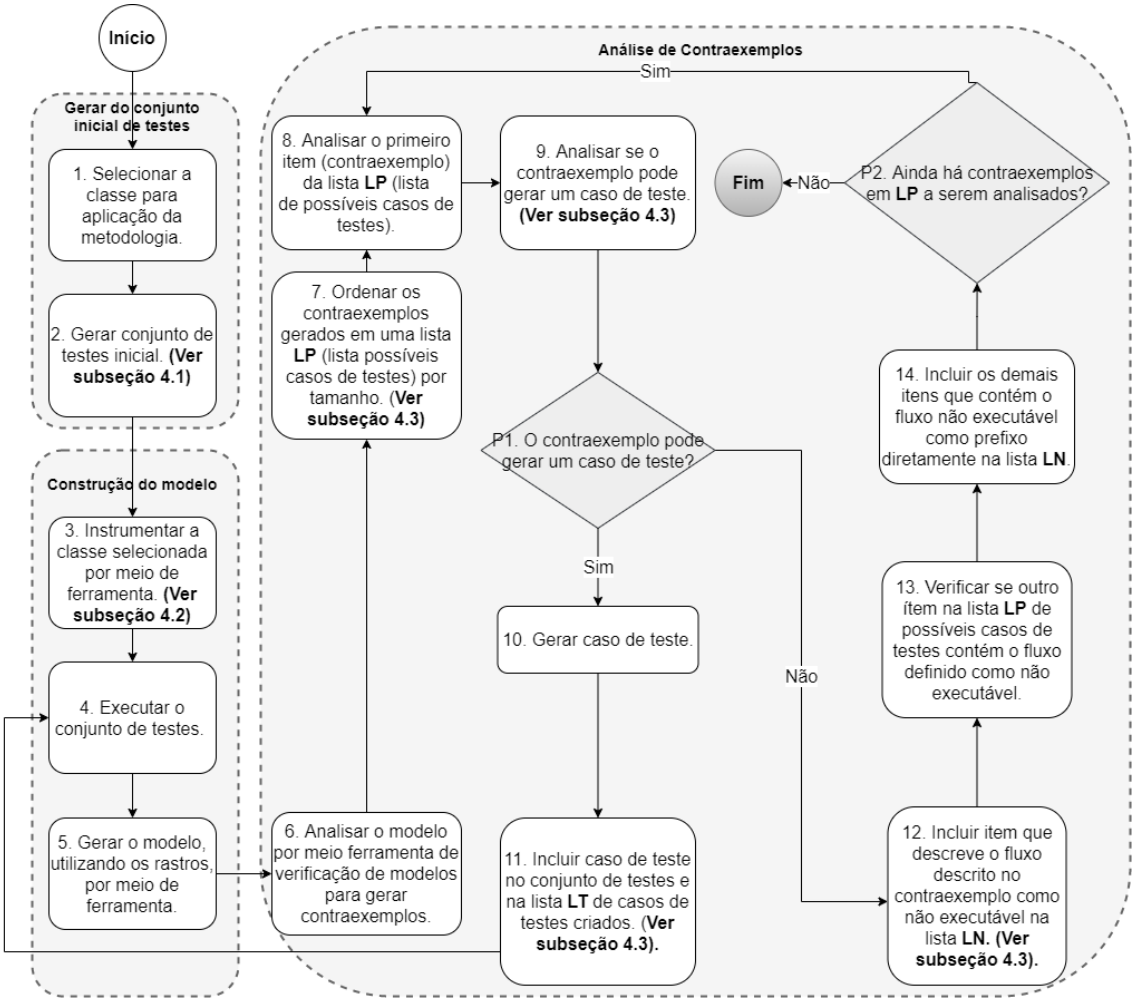
Na metodologia aqui descrita tem-se a integração das técnicas de teste de software com verificação de modelos, com o objetivo de possibilitar a geração de um conjunto de testes o mais completo possível. Um conjunto completo de testes executa todos os possíveis caminhos executados no sistema. Neste trabalho, para um sistema qualquer, o conjunto de testes mais completo possível faz referência ao conjunto de testes possíveis de serem identificados com a aplicação da metodologia.

Em síntese, a metodologia foi baseada nos trabalhos de Duarte (2011) e Majerkowski (2012) e na ideia da utilização de um modelo do comportamento do sistema para a criação de testes unitários. O resultado esperado com a aplicação da metodologia é a geração de testes concretos (atualmente, compatíveis com o padrão JUnit) a partir de um modelo obtido com a aplicação de um processo de extração de modelos, em que o modelo guia o desenvolvedor na identificação de novos potenciais casos de teste. Em tese, a metodologia pode ser aplicada em sistemas desenvolvidos em qualquer linguagem, mas, na forma como foi criada, é analisada sob o ponto de vista de sistemas desenvolvidos na linguagem orientada a objetos Java.

Na Figura 4.1 é apresentado o fluxo que o desenvolvedor deverá seguir para a aplicação da metodologia. É possível visualizar-se os passos para a geração do modelo que servirá de base para a obtenção dos testes. O modelo abstrato é uma representação da realidade do sistema (MAGEE; KRAMER, 2006), sendo que aqui ele representa as possíveis sequências de chamadas de métodos de uma classe Java. Para a geração do modelo, o desenvolvedor deverá seguir os passos 1 a 5, os quais se referem respectivamente a: seleção do componente (classe Java) ao qual a metodologia será aplicada (passo 1); geração do conjunto inicial de testes com uma ferramenta de geração

automática (passo 2); instrumentação do componente para inclusão de anotações necessárias para o processo de extração de modelo por meio uma ferramenta (passo 3); execução do conjunto de testes inicial sobre o código instrumentado, gerando rastros de execução (passo 4); e geração do modelo de comportamento com base nos rastros por meio da ferramenta LTSE (passo 5). O modelo gerado é, como dito anteriormente, uma representação abstrata do que foi executado no sistema, descrevendo, neste caso, as sequências de chamadas de métodos observadas nos testes realizados.

Figura 4.1 – Fluxo executado da metodologia




Fonte: Autor

No passo 6, tem-se a análise do modelo gerado a partir dos rastros na ferramenta LTSA com o objetivo de se obter os contraexemplos que podem derivar novos casos de testes na metodologia. Esta ferramenta permite identificar, dado um modelo, qual é o conjunto de comportamentos considerados “inválidos”. Como o modelo é construído sobre os comportamentos observados, tudo que não esteja representado no modelo é

considerado como comportamento inválido. Se considerarmos o modelo como uma especificação (parcial) do comportamento do código, então cada comportamento inválido seria um contraexemplo. Os contraexemplos identificados servem de base para a geração de novos casos de teste através da reprodução das sequências de chamadas de métodos representadas. Em alguns casos, a análise do modelo pode gerar uma grande quantidade de contraexemplos, o que pode dificultar a análise individual. Desta forma, o passo 7 se refere à organização destes contraexemplos em uma lista chamada lista de possíveis casos de testes em que cada item da lista se refere a um contraexemplo que pode virar um caso de teste. Ao longo da aplicação da metodologia criam-se 3 listas distintas: uma que armazena todos os contraexemplos que ainda não foram analisados (LP - Lista de Possíveis Casos de Testes), uma para os casos de testes gerados (LT - Lista de Casos de Testes Criados) e uma para contraexemplos que não geraram casos de testes (LN - Lista de Contraexemplos Não Executáveis).

Na fase de criação da metodologia, foi percebido que a análise dos possíveis casos de testes a partir dos contraexemplos mais curtos (i.e., com menos transições no modelo) contribui para a maior celeridade na aplicação da metodologia. Por isso, a lista LP é organizada por ordem de tamanho (comprimento do contraexemplo) e o passo 8 refere-se à seleção do primeiro item desta lista (ou seja, um dos contraexemplos com o menor número de transições no modelo). Também foi implementada a ideia de que, quando fosse identificado que um possível caso de teste era não executável, ele serviria de critério de exclusão de outros possíveis casos de testes ainda não analisados, que contivessem a sequência correspondente como prefixo. Isto eliminava muitos casos de teste mais longos que não gerariam testes executáveis, reduzindo o número total de casos a serem analisados. No contexto deste trabalho, casos de testes não executáveis referem-se a contraexemplos que não podem ser reproduzidos como uma execução do sistema por representarem sequências de chamadas de métodos não permitidas pela implementação.

A geração do caso de teste realizada manualmente é representada pelo passo 10 da Figura 4.1. Neste sentido, este passo se refere à ação do usuário aplicador da metodologia de: transformar o fluxo descrito no contraexemplo em caso de teste (chamadas de métodos da sequência do contraexemplo) e atribuir valores para o teste criado. A geração do caso de teste implica a análise do aplicador da metodologia que precisa identificar se o contraexemplo pode ou não gerar um caso de teste (passo 9). Entre os passos 9 e 10 existe a estrutura de decisão P1 que faz referência à decisão por parte do usuário aplicador da metodologia se o contraexemplo poderá gerar um caso de teste ou é

um contraexemplo não executável. Esta decisão é baseada na análise realizada no passo 9. Caso seja identificado que o contraexemplo pode gerar um caso de teste e feito o passo 10, o aplicador da metodologia deverá seguir para a execução do passo 11, que se refere à inclusão deste caso de teste criado no conjunto de testes existente e na lista LT.

A inclusão de um caso de teste no conjunto de testes existente possibilita a geração de um modelo extraído do sistema diferente do modelo atual, o que implicaria a necessidade de uma nova extração de modelo do sistema e posterior análise desse modelo para a geração de uma nova lista LP (contraexemplos). O ato de extrair um novo modelo a partir da inclusão de um novo caso de teste no conjunto de testes existente é considerada uma nova iteração da metodologia e implica a repetição dos passos descritos no fluxo da metodologia a partir do passo 4 até que haja a inclusão de outro novo caso de teste. Assim, a cada novo teste criado tem-se uma nova iteração da metodologia, já que a inclusão de um novo caso de teste no conjunto de testes, que gera um novo conjunto de testes, implicaria na geração de um novo modelo.

Caso o usuário identifique que o contraexemplo não pode representar um caso de teste, deverá haver a classificação deste contraexemplo como não executável para evitar possíveis retrabalhos na aplicação da metodologia. O passo 12 representa a inclusão do item da lista LT na lista LN. A LN armazena os contraexemplos avaliados como não executáveis, em que não executáveis significa que o fluxo de transições descrito no contraexemplo não pode representar um caso de testes válido

Já o passo 13 da Figura 4.1 representa a verificação manual do aplicador da metodologia na lista LP para verificar se outros itens desta lista apresentam o mesmo prefixo definido como não executável, podendo ser definidos como não executáveis. Assim, o passo 14 representa a inclusão de outros itens da lista LP que contenham o fluxo definido como não executável na lista de LN.

Todos os passos da metodologia são detalhados a seguir. Em 4.1, é descrito o processo pelo qual se obtém o conjunto inicial de testes utilizado na geração do modelo inicial. Além disto, foram listadas as considerações e princípios adotados na geração do conjunto inicial de testes. Já em 4.2, tem-se o processo da geração do modelo e as correlações com a metodologia, permitindo o entendimento do uso da extração de modelos na metodologia. Em 4.3, tem-se a descrição de como é realizada a geração de casos de teste, o que constitui o resultado que se pretende alcançar com a aplicação da metodologia.

### 4.1 Geração do conjunto de testes inicial

Na abordagem do processo de extração de modelos (DUARTE, 2007) um conjunto de testes é executado no componente instrumentado para a geração de rastros que são utilizados para a geração do modelo. Assim, para a geração de rastros é necessário que haja um conjunto de testes inicial para executar o sistema e gerar os rastros. Em síntese, os rastros gerados são aplicados na ferramenta LTSE que permite gerar o modelo de comportamento do sistema no formato de um modelo LTS.

Em Majerkowski (2012) e Seijas e Thompson (2018), que assim como esta metodologia utilizam um conjunto de testes para inferir modelos do sistema, o conjunto de testes foi construído com base em conhecimentos da aplicação e/ou técnicas de testes. Em Majerkowski (2012), o conjunto inicial utilizado no processo de extração de modelos foi construído manualmente a partir dos requisitos do sistema e utilizando conhecimento técnico de testes. Em Seijas e Thompson (2018) o modelo de comportamento do sistema foi gerado a partir de testes legados. A construção do conjunto de testes inicial descrito nesta metodologia difere dos trabalhos analisados por propor a geração do conjunto inicial de testes de forma automática, utilizando para isso ferramentas disponíveis gratuitamente e que precisam apenas da classe Java correspondente para gerar testes.

Existem diferentes ferramentas de geração automática de testes a partir de uma classe Java disponíveis na Internet, tais como: JUnit-Tools[4], EVOSUITE[5] e Randoop[6]. Cada ferramenta de geração de testes pode gerar diferentes testes no que se refere a características como estrutura do teste, cobertura do teste e valores utilizados nos testes gerados. Nesse sentido, uma determinada ferramenta pode gerar uma execução para cada método público da classe analisada em cada teste gerado e outra ferramenta pode gerar várias execuções para vários métodos públicos e não públicos em um mesmo teste.

Para facilitar o entendimento do fluxo da metodologia, foi utilizado o exemplo *ElapsedTimer.java*, retirado do sistema Open MCT[7] disponível no repositório GitHub, utilizando a ferramenta JUnit-Tools para a geração do conjunto de testes inicial. Na Figura 4.2 é possível visualizar o código fonte da classe *ElapsedTimer.java*, a qual contém métodos públicos que fazem chamadas a outros métodos. Conforme descrição

4 Disponível em: http://junit-tools.org/. Acesso: em 02 abr. 2020
5 Disponível em: http://www.evosuite.org/. Acesso em: 02 abr. 2020.
6 Disponível em: https://randoop.github.io/randoop/. Acesso em: 02 abr. 2020.
7 Disponível em: https://github.com/nasa/mct/. Acesso em: 21 dez. 2020

contida na própria classe, o propósito da classe é fornecer um cronômetro que pode ser usado para monitoramento de desempenho.

Figura 4.2 – Código fonte da classe ElapsedTimer.java.

```
public class ElapsedTimer { ...
   public ElapsedTimer() {
      overallTime = 0;
      intervals = 0; }
   public void startInterval() {
      elapsedTimeStart = getCurrentTime();
      elapsedTimeStop = 0;
   }
   public void stopInterval() {
      elapsedTimeStop = getCurrentTime();
      intervals++;
      overallTime += (elapsedTimeStop-elapsedTimeStart);
   }
   public long getIntervals() {
      return intervals;
   }
   public long getIntervalInMillis() {
      return elapsedTimeStop - elapsedTimeStart;
   }
   public long getTotalTime() {
      return overallTime;
   }
   public double getMean() {
      return getTotalTime()/((double)intervals);
   }
   long getCurrentTime() {
      return TimeUnit.NANOSECONDS.toMillis(System.nanoTime()); }}
```

Fonte: Parte do código ElapsedTimer[8] disponível no GitHub.

[8] Disponível em: https://github.com/nasa/mct/blob/master/util/src/main/java/gov/nasa/arc/mct/util/internal/ElapsedTimer.java. Acesso em: 21 dez. 2020.

Figura 4.3 – Parte do teste gerado pela ferramenta JUnit-Tools a partir da classe *ElapsedTimer.java*.

```
@Generated(value = "org.junit-tools-1.1.0")
public class ElapsedTimerTest {
	private ElapsedTimer createTestSubject() {
		return new ElapsedTimer();
	}
	@MethodRef(name = "startInterval", signature = "()V")
	@Test
	public void testStartInterval() throws Exception {
		ElapsedTimer testSubject;
		// default test
		testSubject = createTestSubject();
		testSubject.startInterval();
	}
	@MethodRef(name = "stopInterval", signature = "()V")
	@Test
	public void testStopInterval() throws Exception {
		ElapsedTimer testSubject;
		// default test
		testSubject = createTestSubject();
		testSubject.stopInterval();
	}
	@MethodRef(name = "getTotalTime", signature = "()J")
	@Test
	public void testGetTotalTime() throws Exception {
		ElapsedTimer testSubject;
		long result;
		// default test
		testSubject = createTestSubject();
		result = testSubject.getTotalTime();
	}
	@MethodRef(name = "getMean", signature = "()D")
	@Test
	public void testGetMean() throws Exception {
		ElapsedTimer testSubject;
		double result;
		// default test
		testSubject = createTestSubject();
		result = testSubject.getMean();
	}
```

Fonte: Autor.

O código do conjunto de testes gerado pela ferramenta JUnit-Tools a partir da classe *ElapsedTimer.java* é apresentado na Figura 4.3. Os testes gerados pela ferramenta JUnit-Tools são gerados no formato ".java" e são compatíveis com JUNIT. Para a geração

destes testes por meio da ferramenta é necessário instalar o plugin do JUnit-Tools na IDE Eclipse e executar a geração automática pelo comando específico através das opções de menu. Conforme representado na Figura 4.1, isso corresponde ao passo 2 do fluxo descrito da metodologia. Pode-se observar que os valores utilizados nos testes gerados pela ferramenta JUnit-Tools são pré-definidos e variam de acordo com o tipo de retorno do método e o tipo dos valores de parâmetros de entrada do método. Por exemplo: para os métodos com tipo de retorno *void* e parâmetro de entrada com uma variável do tipo *String*, a ferramenta gera o valor padrão (””) como parâmetro de entrada na execução dos métodos.

## 4.2 Geração do modelo

Conforme a abordagem de extração de modelos proposta em Duarte (2007), para a produção dos rastros utilizados na geração do modelo é necessário que o componente a ser analisado esteja instrumentado. Além disso, deve existir um conjunto de testes inicial que possibilite a execução do sistema, conforme descrito no capítulo 4.1.

A instrumentação (passo 3) é obtida por meio da ferramenta TXL[9] que permite inserir anotações de forma automática no código fonte, permitindo a captura das informações necessárias para se construir o modelo de comportamento quando o código instrumentado é executado, gerando rastros. As regras de anotação estão descritas na linguagem de entrada TXL, definindo a substituição de partes do código original por código com anotações. A transformação se baseia na gramática de Java, também descrita na linguagem da TXL. As regras de anotação usadas aqui foram definidas em Duarte (2007). O resultado esperado da instrumentação é a inclusão de anotações na classe de forma a permitir a geração dos rastros ao executar testes. A Figura 4.4 demonstra um trecho do código fonte do componente *ElapsedTimer.java* instrumentado. Resumidamente, a Figura 4.4 demonstra anotações no código fonte da classe que permitem a geração de rastros. Assim, tem-se a anotação que representa a entrada no método *startInterval* descrito pelo rótulo “*MET_ENTER*”, a representação de uma chamada dentro deste método a outro método *getCurrentTime* da classe representado pelo rótulo “*INT_CALL_ENTER”* e suas respectivas saídas representadas pelos rótulos *“INT_CALL_END”* e *“MET_END”*.

9 Disponível em: https://www.txl.ca/. Acesso em: 02 abr. 2020

Figura 4.4 – Parte do código fonte da classe *ElapsedTimer* instrumentada.

```
public class ElapsedTimer {
...
  /**
     * Default constructor to initialize the overall time and intervals.
     */
  public ElapsedTimer () {
    overallTime = 0;
    intervals = 0;
  }
  /**
     * The start time interval.
     */
  public void startInterval () {
    System.err.println ("MET_ENTER" + ":" + "startInterval" + "#" +
_thisClassName + "=" + this._thisInstanceID + "#" + "{" + "" + "overallTime" + "=" +
this.overallTime + "^" + "elapsedTimeStart" + "=" + this.elapsedTimeStart + "^" +
"elapsedTimeStop" + "=" + this.elapsedTimeStop + "^" + "intervals" + "=" +
this.intervals + "^" + "methodgetTotalTime" + "=" + this.methodgetTotalTime + "^" +
"}" + "#" + "8" + ";");
      {
       {
        System.err.println ("INT_CALL_ENTER" + ":" + "getCurrentTime" + "#"
+ _thisClassName + "=" + this._thisInstanceID + "#" + "{" + "" + "overallTime" + "="
+ this.overallTime + "^" + "elapsedTimeStart" + "=" + this.elapsedTimeStart + "^" +
"elapsedTimeStop" + "=" + this.elapsedTimeStop + "^" + "intervals" + "=" +
this.intervals + "^" + "methodgetTotalTime" + "=" + this.methodgetTotalTime + "^" +
"}" + "#" + "0" + ";");
        elapsedTimeStart = getCurrentTime ();
        System.err.println ("INT_CALL_END" + ":" + "getCurrentTime" + "#" +
_thisClassName + "=" + this._thisInstanceID + "#" + "0" + ";");
       } elapsedTimeStop = 0;
      } System.err.println ("MET_END" + ":" + "startInterval" + "#" +
_thisClassName + "=" + this._thisInstanceID + "#" + "8" + ";");
    }
```

Fonte: Autor.

Os rastros fornecidos à ferramenta LTSE permitem a geração de um modelo LTS (passo 4). A ferramenta LTSE permite a geração do código textual da álgebra de processos *Finite State Processes* (FSP) que usado na ferramenta LTSA permite a análise e visualização gráfica do modelo LTS.A geração do modelo através dos rastros por meio da ferramenta LTSE corresponde ao passo 5 da metodologia.

### 4.3 Geração dos casos de testes

A geração de casos de testes consiste, basicamente, no processo de criação de testes concretos JUnit a partir dos contraexemplos gerados na análise do modelo extraído do sistema real. Nesse caso, o usuário analisa os contraexemplos gerados na análise do modelo (passo 6) para gerar casos de testes manualmente. Os casos de testes são criados a partir da análise do fluxo de transições representadas nos contraexemplos gerados na análise do modelo. Assim, o aplicador da metodologia interpreta se os fluxos descritos nos contraexemplos podem descrever execuções de métodos da classe em casos de teste. O processo de geração de casos de testes descrito na metodologia foi influenciado pelo trabalho de Majerkowski (2012), que por sua vez se baseou nos trabalhos de Garganti (2007) e Groz et al., (2008).

No código do modelo extraído do sistema, descrito em FSP e analisado por meio da ferramenta LTSA, é possível definir um processo como uma propriedade por meio da palavra de controle *property* antecedendo a descrição do processo. Uma propriedade define um conjunto de comportamentos considerados corretos, sendo que comportamentos fora desse conjunto são definidos como incorretos. Para que um processo seja definido como uma propriedade é necessário que ele seja determinístico. O LTSA permite a determinização de um modelo por meio da execução do acréscimo da palavra de comando *deterministic* antecedendo a descrição do processo que cria o modelo. Ao se gerar um modelo com base em um processo definido como propriedade, o LTSA cria um estado de erro, representado por pelo identificador -1. Todos os comportamentos não presentes no processo descrito (i.e., não usados na criação do modelo), são direcionados para este estado de erro, indicando que eles são considerados violações da propriedade descrita (contraexemplos).

No cenário explorado neste trabalho, em que não se tem as especificações do sistema, o comportamento descrito no modelo extraído é considerado como o comportamento verdadeiro do sistema analisado. Desta forma, trata-se o modelo extraído como a “especificação” do sistema. Isto significa que comportamentos não presentes no modelo são, até prova em contrário, considerados incorretos. Assim, todos os comportamentos que a ferramenta LTSA identifica como incorretos (ou seja, contraexemplos) no modelo são considerados como possíveis casos de testes na metodologia. A análise do modelo e geração dos contraexemplos é representado na Figura 4.1 pelo passo 6. Resumidamente, os passos 7, 8 e 9 referem-se à organização dos

contraexemplos em uma lista para melhor análise, definição da ordem de análise dos contraexemplos e a análise de cada contraexemplo propriamente dita. Tais passos são explicados mais à frente, pois é necessário um entendimento do uso das listas usadas na aplicação da metodologia.

A geração do teste concreto (passo 10) é realizada basicamente através da reprodução da sequência de transições presentes nos contraexemplos em testes concretos JUnit. Assim, se o contraexemplo gerado representa a sequência de transições "*getMean->startInterval"* e após a análise do contraexemplo o aplicador da metodologia identificar que o contraexemplo pode gerar um teste concreto, o caso de testes criado (passo 10) deverá executar a sequência dos métodos representados no contraexemplo. Este exemplo é apresentado na Figura 4.5. Antes da geração do teste concreto propriamente dito, o usuário aplicador da metodologia deve identificar por meio de sua percepção e análise do código fonte da aplicação se o contraexemplo pode gerar um caso de testes, conforme descrito na Figura 4.1 pelo passo 9. O passo 9 na Figura 4.1 representa a análise manual do contraexemplo, em que o aplicador da metodologia avalia se o contraexemplo pode gerar um caso de teste. Nesta análise, é identificado se, por exemplo, o contraexemplo representa uma chamada a um método privado, o que indica um caso de teste não executável, ou mesmo chamadas a métodos públicos, que indica teste de caso viável. Além disto, no passo 9 também se tem a verificação do código fonte e conjunto de testes, em que o aplicador da metodologia verifica a viabilidade do caso de teste e valores de testes. A estrutura de decisão P1 da Figura 4.1 tem como objetivo representar o ponto em que se define se o contraexemplo pode ou não gerar um caso de testes. Esta estrutura está representada na Figura 4.1 para demonstrar que o usuário tem um ponto de decisão, após a análise do contraexemplo, em que ele deverá decidir se o contraexemplo pode ou não permitir a geração de um caso de testes.

Ao criar um novo caso de teste, o usuário aplicador da metodologia deverá incluí-lo no conjunto inicial de testes, pois a cada nova inclusão de um caso de teste no conjunto de testes que é executado, gera-se um novo modelo e, assim, potencialmente novos contraexemplos. Na Figura 4.1, a inclusão do novo teste no conjunto inicial de testes está representada no passo 11 e a nova iteração da metodologia é iniciada a partir do passo 4.

Influenciado pelo trabalho de Majerkowski (2012), que a partir de uma lista de contraexemplos realizou a análise exaustiva e individual dos contraexemplos gerados, este trabalho também propõe a utilização do uso de listas para organizar a análise de possíveis casos de testes: LP (Lista de Possíveis Casos de Testes), LT (Lista de Casos de

Testes Criados) e LN (Lista de Contraexemplos Não Executáveis). No entanto, a análise da lista de contraexemplos neste trabalho é realizada de forma mais otimizada, utilizando para isso uma abordagem em que um possível caso de teste identificado como não executável pode auxiliar na identificação de outros casos de testes não executáveis.

Figura 4.5 – Exemplo do teste concreto JUnit criado na aplicação da metodologia.

```
@Test
        public void novoTeste() throws Exception {
        ...
                instanciaClasse.getMean();
                instanciaClasse.startInterval();
        ...
        }
```

Fonte: Autor.

Na metodologia, para melhor entendimento, os contraexemplos gerados pela ferramenta de verificação de modelos foram organizados em uma lista chamada de LP que armazena possíveis casos de testes e que armazena todos os possíveis casos de teste (contraexemplos) gerados pela ferramenta de verificação de modelos. Ao se gerar um novo caso de teste, a partir de um contraexemplo, na aplicação da metodologia, guarda-se o caso de teste em uma outra lista chamada LT, a qual é usada na metodologia para armazenar os casos de testes criados em todas as iterações, podendo ser acessada em qualquer iteração para consulta de quais casos de testes que já foram criados e evitar repetição. Assim como os passos iniciais da metodologia, em que um conjunto de testes é utilizado para a geração de um modelo, a inclusão de um novo caso de teste no conjunto de testes existente gera um novo rastro, o que pode significar um novo comportamento no modelo. Na metodologia, chamamos de iteração a nova geração do modelo a partir do conjunto de testes acrescido do novo caso de testes. Ao final da aplicação da metodologia e todas as possíveis iterações necessárias, todos os casos de testes contidos na lista LT deverão estar contidos no conjunto de testes. Ao identificar que um dos itens da lista LP permitiu a geração de um caso de teste, é necessário atualizar a lista LP, esvaziando a lista LP e atualizando com os novos valores gerados na nova iteração.

No processo de geração manual dos testes, cada possível caso de teste é analisado para se identificar se pode gerar um caso de teste. Nessa análise, alguns contraexemplos podem ser identificados como não executáveis, ou seja, não representam uma sequência que poderia gerar um caso de teste. Nesse contexto, possíveis casos de testes não

executáveis são sequências de transições que não são permitidas pela implementação, tais como chamadas diretas a métodos privados. Estes contraexemplos não executáveis são separados em uma lista chamada LN. Esta lista LN pode também ser acessada em qualquer iteração para consulta de quais sequências já foram definidas como não executáveis e evitar esforço desnecessário de análise.

Visando facilitar e melhorar a análise dos possíveis casos de testes, foi considerada a organização da lista LP em uma lista ordenada por quantidade de transições contidas no contraexemplo (passo 7). Após a organização da lista LP tem-se a análise dos contraexemplos iniciando a partir do primeiro item (passo 8). Desta forma, um caso de teste identificado como não executável que contém cinco transições auxilia na identificação de outro caso de teste não executável de tamanho igual ou maior, através da identificação de outro possível caso de teste que tem como prefixo a sequência de transições já determinada como não executável. Neste caso, todos os possíveis casos de testes que se enquadram neste quesito são classificados como não executáveis (passos 13 e 14). Ao identificar-se que o fluxo de transições representado no contraexemplo pode ser transformado em um caso de teste (passo 9), a transformação em teste concreto é realizada manualmente (passo 10) e o caso de teste adicionado à suíte de testes (passo 11).

A inclusão de casos de testes na suíte de testes utilizada na geração do modelo do sistema implica a necessidade de repetição do processo de extração de modelos, análise do modelo e análise de possíveis novos casos de testes (contraexemplos), ou seja, uma nova iteração do fluxo descrito na Figura 4.1. Isto se faz necessário, pois a cada novo caso de teste criado e inclusão do mesmo no conjunto de testes inicial tem-se a possibilidade da geração de um novo modelo o que permite a uma nova lista de possíveis casos de testes. Neste sentido, adotou-se a estratégia de realizar uma nova iteração da metodologia a cada caso de teste criado, objetivando obter um melhor aproveitamento da metodologia. Dessa forma, a cada nova inclusão de um caso de teste, tem-se uma nova lista de contraexemplos e alguns contraexemplos da lista anterior podem acabar sendo eliminados pela inclusão de um novo caso de teste.

Um possível caso de teste pode ser definido como não executável de acordo com vários critérios, dependendo de cada sequência de transições, tais como: referência direta a um método privado no início de uma sequência de transição, sequência de métodos impossível de ser executada, entre outros. Nos contraexemplos gerados, a referência direta a um método privado pode se dar por meio da seguinte indicação no início do fluxo representado: “call.<Nome do Método Privado>”. Por exemplo, se em um contraexemplo

for identificado um fluxo de transições m1 –> m2, o que corresponde a executar uma chamada ao método público *m1()* e dentro deste método executar uma chamada ao método privado *m2()*, isto pode ser considerado um caso de teste possível, pois a chamada ao método privado não está sendo realizada diretamente pelo caso de teste. Já se, por exemplo, o contraexemplo descrever o mesmo fluxo de transições anterior, mas com a chamada ao método público *m1()* seguido de uma chamada ao método privado *m2()*, isso poderia ser considerado um caso de teste não executável, porque requer uma chamada direta a um método privado.

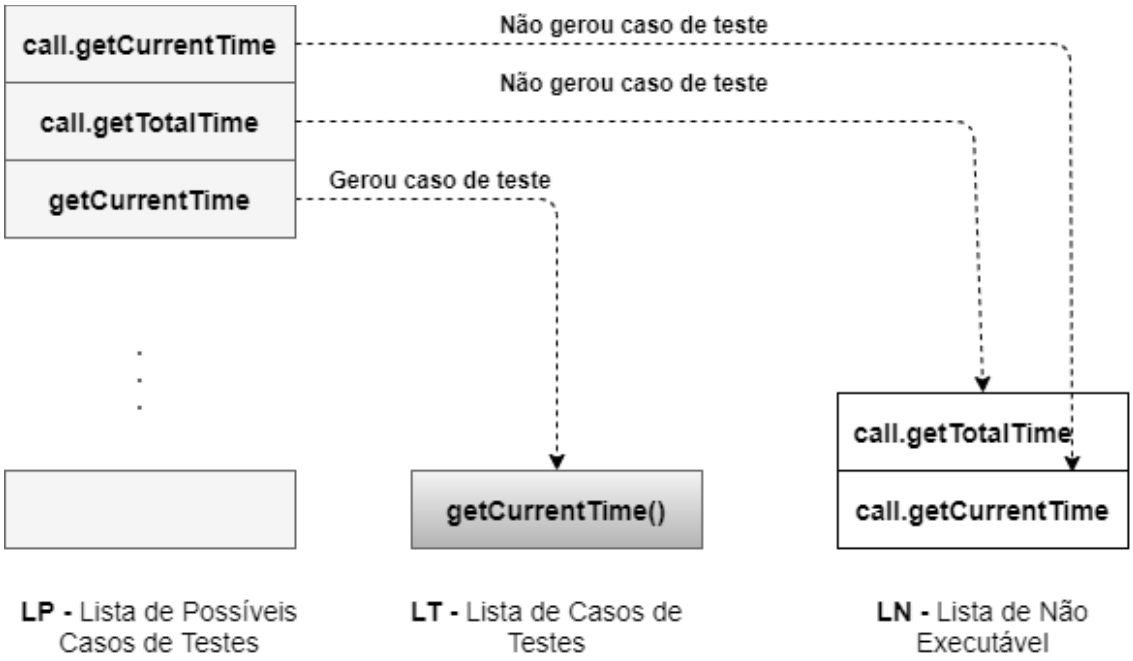
Quando um contraexemplo é identificado como não executável, ele é incluído na lista de não executáveis LN. A utilização desta lista tem como propósito evitar possíveis esforços desnecessários na identificação de outros casos de testes não executáveis, através da identificação em outros possíveis casos de testes do prefixo de sequência de transições já definida como não executável. Na Figura 4.1, o passo 12 descreve a ação de inclusão do item (contraexemplo) na lista LN. O passo 13 descrito na Figura 4.1 representa a busca na lista LP por outros contraexemplos que contenham a sequência definida como não executável como prefixo. Caso haja mais contraexemplos nesta situação, os mesmos são também incluídos na lista LN. O passo 14 descrito na Figura 4.1 representa a inclusão dos demais contraexemplos que contém a sequência descrita como não executáveis na lista LN.

Na Figura 4.6 é possível visualizar uma representação da interação entre as listas LP, LT e LN no decorrer da aplicação da metodologia. Nesta figura, se tem a representação de três contraexemplos da lista LP realizada na primeira iteração da aplicação da metodologia na classe *ElapsedTimer*, em que o aplicador da metodologia identificou que os contraexemplos *call.getCurrentTime* e *call.getTotalTime* não poderiam gerar casos de testes, já que o prefixo "*call*" representa uma chamada de um método dentro do corpo de outro método. Nesse sentido, tais contraexemplos são considerados não executáveis, pois não há como executar tal chamada de método sem estar-se executando o corpo de um método que contenha a chamada. Após a identificação do contraexemplo como não executável (passo 9), tem-se a estrutura de decisão P1, a qual representa a decisão de classificar ou não o contraexemplo como não executável. A inclusão do contraexemplo na lista LN como não executável está representado no passo 12 da Figura 4.1. Na fase de criação da metodologia e experimentos, foi identificado que poderia se classificar outros contraexemplos compostos do prefixo identificado como não executável e contidos na lista LP como não executável, o que otimizaria a aplicação da

metodologia e facilitaria a análise dos contraexemplos. Nesse sentido, os passos (13 e 14) representados na Figura 4.1 foram descritos no fluxo da aplicação para representar a identificação de outros contraexemplos não executáveis contidos na lista LP a partir da identificação do prefixo (contraexemplo) já identificado como não executável e a inclusão destes contraexemplos também na lista LN.

Na Figura 4.6 também é possível verificar que o contraexemplo *getCurrentTime* foi identificado como um caso de teste viável. Assim, houve a criação do caso de teste (passo 10) e inclusão do caso de teste na lista LT (passo 11). A estrutura de decisão P1 é respondida com base na análise realizada no passo 9. Conforme explicado anteriormente, a inclusão de um novo caso de teste implica uma nova iteração da metodologia.

Figura 4.6 – Listas LP, LT e LN.




Fonte: Autor.

No fluxo da metodologia descrita na Figura 4.1, é possível visualizar que ela é aplicada em iterações. Pode se observar na Figura 4.1 também que a condição de parada da aplicação da metodologia é a lista LP vazia, o que é representado na figura pela pergunta P2. Neste ponto o desenvolvedor que está aplicando a metodologia avalia se deve finalizar a aplicação da metodologia.

Após a análise de cada um dos itens da lista LP, em que se define se o possível caso de teste pode gerar um caso de teste ou se é não executável, se tem o ponto de parada da metodologia representado pela estrutura de decisão P2.

No ponto de estrutura de decisão P2 é verificado se o critério de parada definido foi atingido. Como uma nova iteração só é necessário se houver itens a analisar (i.e., contraexemplos na lista LP), o critério de parada adotado foi o de identificar que LP está vazia. Assim, nessa situação, todos os contraexemplos identificados na análise do modelo foram tratados, sendo que a lista LT contém todos aqueles que viraram casos de teste e LN, todos os contraexemplos que não puderam virar casos de teste. O fato de que, a cada iteração, eliminam-se múltiplos contraexemplos de LP, nos garante que tal critério será atingido em um tempo finito. A eliminação de um contraexemplo pode ocorrer de maneira direta ou indireta. Na direta, ele é excluído porque ou não é executável (vai para LN) ou porque virou caso de teste (vai para LT). Na indireta, ele é excluído porque tem mesmo prefixo que algum outro contraexemplo já em LN, sendo que ele também vai para LN, ou acaba sendo incluído no modelo pela adição de um novo rastro, o qual foi gerado por um contraexemplo que virou caso de teste. Neste último caso, o contraexemplo não é incluído nem em LN e nem em LT, já que o modelo o incorpora diretamente. Isto ocorra devido ao processo de extração baseado em contextos, que permite que a inclusão de um novo rastro leve à identificação de outros caminhos alternativos no modelo, podendo, assim, determinar que algo que seria um contraexemplo passa a ser uma execução válida. Todas estas situações fazem com que, a cada iteração, o número de contraexemplos a serem analisados seja substancialmente reduzido, inclusive com a eliminação de conjuntos infinitos, como no caso de eliminação por prefixo comum e mesmo pela inclusão indireta de comportamentos no modelo.

### 4.3.1 Aplicação da metodologia - Geração de casos de testes

A geração do teste a partir da metodologia conforme descrito em 4.3, é aplicada aqui utilizando o mesmo exemplo *ElapsedTimer.java*. O código FSP do modelo determinizado e com a palavra *property*, o que permite a identificação dos contraexemplos, pode ser visualizado na Figura 4.7 e o modelo com a representação dos contraexemplos gerado por meio da ferramenta LTSA pode ser visualizada na Figura 4.8. A determinização do modelo pode ser realizada por meio da ferramenta LTSA

O código FSP representado na Figura 4.7 descreve o modelo denominado *ElapsedTimer*. Os estados do modelo são representados por: Q0, Q1, Q2, Q3, Q4 e Q5. As transições realizadas do modelo são representados por: *startInterval*, *stopInterval*, *getIntervalInMillis*, *getIntervals*, *getTotalTime*, *getMean*, *call.getCurrentTime*,

*call.getTotalTime, getCurrentTime, getTotalTime*. Em resumo, o modelo tem o estado inicial Q0 de onde as transições *startInterval* e *stopInterval* partem para o estado Q1, as transições *getIntervalInMillis*, *getIntervals* e *getTotalTime*) partem para o estado Q2 e a transição *getMean* parte para o estado Q3, e assim sucessivamente. O estado final do modelo é o Q5, em que a transição *getTotalTime* parte para o estado Q2.

Figura 4.7 – Código FSP do modelo determinizado com a aplicação da palavra property.

```
property ElapsedTimer = Q0,
Q0      = ({startInterval, stopInterval} -> Q1
          |{getIntervalInMillis, getIntervals, getTotalTime} -> Q2
          |getMean -> Q3),
Q1      = (call.getCurrentTime -> Q4),
Q2      = (_EXIT -> Q2),
Q3      = (call.getTotalTime -> Q5),
Q4      = (getCurrentTime -> Q2),
Q5      = (getTotalTime -> Q2).
```

Fonte: Autor.

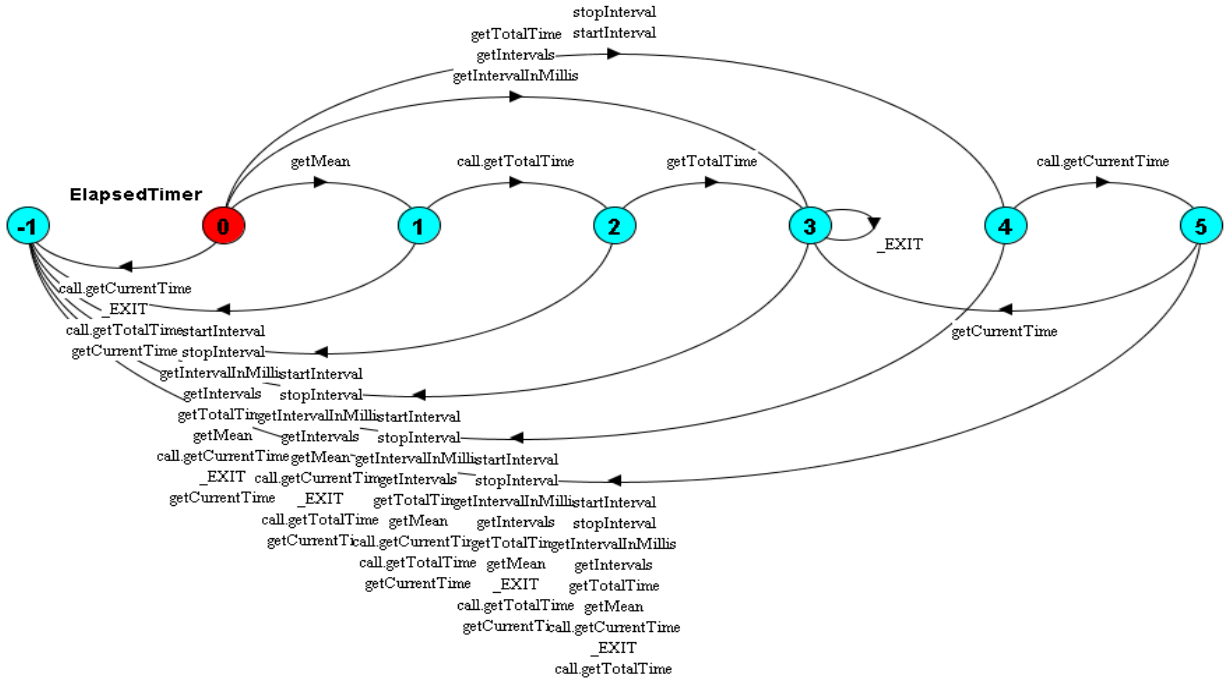
A Figura 4.8 representa o modelo gerado a partir da execução do código representado na Figura 4.7 na ferramenta LTSA. Em resumo, o modelo representa os contraexemplos partindo de algum dos estados válidos (0,1,2,3,4 e 5) e finalizando no estado de erro (-1). As sequências que começam no estado inicial 0 e terminam em um estado válido são execuções válidas do sistema, enquanto aquelas que iniciam em 0 e terminam em -1 são ditas inválidas, sendo por isso chamadas de contraexemplos.

A análise do modelo inicial na ferramenta LTSA resultou na criação de uma lista com 154 itens (contraexemplos). Logo, a lista LP continha estes 154 possíveis contraexemplos. Na primeira iteração da metodologia foi possível identificar que o contraexemplo *"getCurrentTime"* poderia gerar um novo caso de teste. Assim, o item passa a constar na lista LT e é incluído no conjunto de testes. A criação de um novo caso de teste implica a alteração da suíte de testes com a inclusão no caso de teste criado e por consequência uma nova iteração da metodologia e a repetição dos passos a partir do passo 4 da Figura 4.1, de forma que uma nova lista de contraexemplos é gerada, substituindo a lista anterior, e assim sucessivamente, até que não haja mais nenhum contraexemplo a ser analisado.

A Figura 4.9 representa parte dos testes gerados por meio da aplicação da metodologia. Neste exemplo, em relação aos valores de testes, foram usados nos casos de

testes os valores padrões gerados nos testes gerados de forma automática pela ferramenta JUnit-Tools. Na Figura 4.9 é possível visualizar exemplos dos casos de testes gerados a partir da aplicação da metodologia chamados *test1()*, t*est2()* e *test3()*. O caso de teste *test1()* executa chamada ao método *getCurrentTime*, o *test2()* executa chamadas aos métodos *getMean()* e *startInterval()* da classe ElapsedTimer. Já o caso de teste *test3()* executa chamadas aos métodos *getMean()* e *stopInterval*().

Figura 4.8 – Modelo com contraexemplos gerados através da aplicação da metodologia.




Fonte: Autor.

O código FSP referente ao modelo gerado ao final da aplicação da metodologia pode ser visualizado na Figura 4.10. Os estados do modelo são representados por: Q0, Q1, Q2, Q3, Q4, Q5 e o novo estado Q6. As ações do modelo são: *startInterval, stopInterval, getCurrentTime, getIntervalInMillis, getIntervals, getTotalTime, call.getCurrentTime, call.getTotalTime, getMean.* Em resumo, o modelo final teve um estado incluído (Q6) e mais transições, como as transições *startInterval, stopInterval* que saem do novo estado Q6. O modelo LTS gerado a partir da aplicação do código FSP do modelo final na ferramenta LTSA pode ser visualizado na Figura 4.11. Nesta figura que descreve o modelo final é possível verificar o novo estado incluído Q6 e as novas transições geradas a partir da inclusão dos novos testes criados com a aplicação da metodologia.

Figura 4.9 – Casos de testes gerados através da aplicação da metodologia.

```
import java.io.File;
import java.io.PrintStream;
import javax.annotation.Generated;
import org.junit.Test;
import org.junit.tools.configuration.base.MethodRef;

@Generated(value = "org.junit-tools-1.1.0")
public class ElapsedTimerTest {

        private ElapsedTimer createTestSubject() {
                return new ElapsedTimer();
        }

        // Testes criados
         @Test
         public void test1() throws Exception {
             ElapsedTimer testSubject;
             testSubject = createTestSubject();
             testSubject.getCurrentTime();
        }
         @Test(timeout = 4000)
          public void test2()  throws Throwable  {
             ElapsedTimer elapsedTimer0 = new ElapsedTimer();
             double double0 = elapsedTimer0.getMean();
             elapsedTimer0.startInterval();
          }
          @Test(timeout = 4000)
          public void test3()  throws Throwable  {
             ElapsedTimer elapsedTimer0 = new ElapsedTimer();
             double double0 = elapsedTimer0.getMean();
             elapsedTimer0.stopInterval();
          }
}
```

Fonte: Autor.

A cada análise de um possível caso de teste se indica que o desenvolvedor realize as seguintes reflexões:

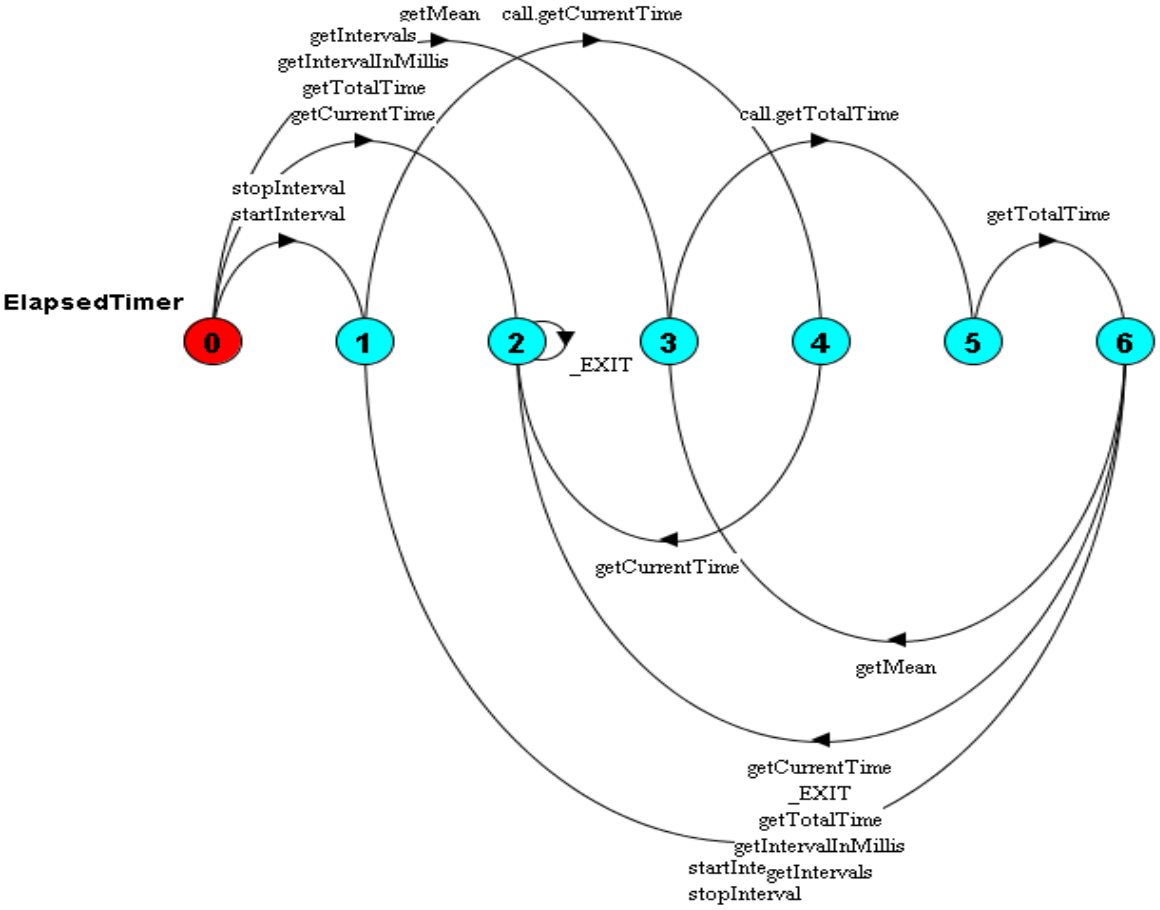
- O caso de teste representa uma sequência de execuções possíveis e que contribui de forma significativa para o propósito dos testes que se quer criar?
- Os valores dos casos de testes são adequados para o propósito dos casos de testes que se precisa gerar através da aplicação da metodologia?

Figura 4.10 – Código FSP do modelo final gerado pela ferramenta LTSE a partir dos rastros.

```
ElapsedTimer = Q0,
Q0    = ({startInterval, stopInterval} -> Q1
        |{getCurrentTime, getIntervalInMillis, getIntervals, getTotalTime} -> Q2
        |getMean -> Q3),
Q1    = (call.getCurrentTime -> Q4),
Q2    = (_EXIT -> Q2),
Q3    = (call.getTotalTime -> Q5),
Q4    = (getCurrentTime -> Q2),
Q5    = (getTotalTime -> Q6),
Q6    = ({startInterval, stopInterval} -> Q1
       |{_EXIT, getCurrentTime, getIntervalInMillis, getIntervals, getTotalTime} ->
Q2
       |getMean -> Q3).
```

Fonte: Autor.

Figura 4.11 – Visualização gráfica do modelo final gerado na aplicação da metodologia na classe *ElapsedTimer* gerado na ferramenta LTSA.




Fonte: Autor.

# 5 EXPERIMENTOS

Neste capítulo são apresentados resultados dos experimentos realizados aplicando a metodologia em alguns sistemas selecionados. Na descrição do fluxo da Figura 4.1, foram destacadas as ações realizadas em cada etapa do fluxo da aplicação da metodologia. Neste capítulo, cada ação da metodologia será detalhada indicando as ferramentas utilizadas. Nesse sentido, o desenvolvedor pode compreender o motivo de escolha de cada uma das ferramentas e decidir, caso seja necessário, por outras ferramentas que atendam aos requisitos da metodologia de acordo com o seu propósito de aplicação.

## 5.1 Questões de Pesquisa

Neste trabalho, buscou-se responder às seguintes três questões de pesquisa, como forma de avaliação da metodologia proposta e dos resultados de seu uso nos experimentos a serem descritos:

**1. A aplicação da metodologia permite a geração de novos testes?**

O objetivo desta questão é analisar se a metodologia contribui para a geração de testes diferentes dos testes do conjunto inicial, os quais serão denominados neste trabalho de *novos testes*. A análise desta questão se dará através dos resultados obtidos na aplicação da metodologia, dos quais serão contabilizadas as quantidades de novos testes criados.

**2. Caso sejam criados novos testes, estes podem contribuir para o aumento da cobertura de testes?**

O objetivo desta questão é analisar se há aumento de cobertura de linha ou *branch* quando se tem a inclusão de novos testes, gerados a partir da aplicação da metodologia na suíte de testes da classe. No trabalho são utilizadas as classes e métodos nas aplicações da metodologia. Desta forma, devido à granularidade dos objetos utilizados nas aplicações da metodologia foram utilizadas as coberturas de *branch* e linha para avaliar a metodologia por estas serem as coberturas mais indicadas para avaliar os objetos utilizados nos experimentos. Na análise desta questão, serão comparados os resultados

obtidos da cobertura de linha e *branch* antes (com base no conjunto de testes inicial) e após a aplicação da metodologia (conjunto de testes inicial mais os novos testes).

**3. A metodologia contribui para a descoberta de erros na aplicação?**

O objetivo desta questão é analisar se a aplicação da metodologia facilita a descoberta de erros na aplicação. Segundo Ammann e Offutt (2008), um erro pode ser definido como um estado interno incorreto que pode ou não resultar em uma falha no sistema. Na análise desta questão, são verificadas as possíveis contribuições da metodologia para a descoberta de erros na aplicação. Neste trabalho, a avaliação dos possíveis erros encontrados na aplicação da metodologia foi realizada através do uso da ferramenta de testes de mutação PIT. Resumidamente, a ferramenta possibilita avaliar a qualidade do teste a partir da porcentagem de mutações eliminadas, em que mutações são falhas propagadas no sistema. Mutações eliminadas indicam que o teste falhou.

## 5.2 Ferramentas Utilizadas

Todas as ferramentas utilizadas nas aplicações da metodologia descritas no capítulo da metodologia podem ser substituídas por outras ferramentas que possuem características semelhantes. Assim, na metodologia o importante é que seja utilizada uma ferramenta que melhor se adeque à necessidade do desenvolvedor, sendo importante que a ferramenta tenha o mesmo propósito necessário disposto na metodologia. Nesta seção serão descritos os critérios de escolha das ferramentas utilizadas nas aplicações da metodologia descritas neste trabalho.

Foram selecionados sistemas para a aplicação da metodologia descritas desenvolvidos na IDE Eclipse, em que os critérios de escolha para essa IDE foram: ampla utilização por desenvolvedores, por representar estabilidade, aceitabilidade no mercado, bem como pela multiplicidade de exemplos possíveis. O *framework* de testes utilizado foi o JUnit[10] , que se justifica por ser uma ferramenta de código aberto para a criação de testes unitários e compatível com a ferramenta IDE e linguagem JAVA utilizada nos sistemas utilizados no experimento.

10 Disponível em: https://junit.org/. Acesso em: 02 abr. 2020.

Para a geração do conjunto inicial de testes foram selecionadas as ferramentas JUnit-Tools e EVOSUITE por permitirem a geração automática de testes para sistemas desenvolvidos na linguagem JAVA e terem compatibilidade com o *framework* JUnit. Essas ferramentas de geração automática de testes selecionadas também apresentam diferentes aspectos, tais como: quantidade de testes gerados, critério de cobertura de testes e diferentes valores de testes. Além disto, outro fator determinante para a escolha dessas ferramentas foi a diferença dos testes gerados por essas ferramentas. Assim, a utilização de duas diferentes ferramentas permitiu avaliar o resultado da aplicação da metodologia utilizando diferentes conjuntos de testes inicial. Nos parágrafos a seguir foram explicadas as principais diferenças entre os testes gerados pelas ferramentas JUnit-Tools e EVOSUITE.

A ferramenta JUnit-Tools foi selecionada para ser utilizada nas aplicações da metodologia por gerar testes em formato simples e padronizados que, em geral, executam uma chamada a cada método público. Os testes gerados pela ferramenta utilizam valores restritos e padrões, tais como: 0, 1, -1 e *null*. O algoritmo utilizado pela ferramenta na geração de testes não utiliza critérios de testes específicos. Dessa forma, a suíte de teste gerada por essa ferramenta é a mais simples utilizada no experimento.

Já a ferramenta EVOSUITE utiliza o conceito de algoritmo genético na geração automática de suítes de testes JUnit, em que objetivam maximizar a cobertura do código. A suíte de testes gerada pela ferramenta EVOSUITE consiste em uma quantidade variável de casos de testes que executam sequências de instruções Java, por exemplo, chamadas a métodos (FRASER, 2018). A ferramenta faz ainda uso de uma combinação de diferentes critérios de cobertura na geração dos testes, o que permite a geração de melhores suítes de testes e, por utilizar algoritmo randomizado, gera diferentes resultados a cada geração de uma nova suíte de testes (CAMPOS; PANICHELLA; FRASER, 2019).

Na etapa de instrumentação dos componentes da metodologia, foi utilizada a ferramenta e a linguagem TXL. A ferramenta TXL permite instrumentação utilizando anotações que habilitam a construção do modelo LTS. Já na etapa de geração dos modelos, a ferramenta LTSE[11] foi utilizada para a geração dos modelos LTS. A seleção de tais ferramentas foi realizada devido a utilização destas na abordagem de extração de modelos de Duarte (2007) utilizada neste trabalho. Na análise dos modelos gerados, a ferramenta LTSA foi utilizada por ser a ferramenta utilizada na maioria dos trabalhos

11 Disponível em: https://github.com/lucioduarte/LTSE . Acesso em: 02 abr. 2020.

analisados que influenciaram a metodologia exposta. Além disso, permite a visualização do modelo do sistema no formato LTS, o que facilita a análise de fluxos executados no sistema, além de permitir a geração de uma lista de possíveis casos de testes a serem analisados. Das ferramentas de cobertura de testes analisadas, a EclEmma[12] foi utilizada para medir as coberturas de testes antes e após a aplicação da metodologia. A ferramenta EclEmma foi selecionada pela sua compatibilidade com a Eclipse IDE e por permitir a identificação da cobertura de testes visualmente no código através do destaque do código fonte.

A fim de comparar as classes selecionadas para o experimento e os diferentes resultados obtidos na aplicação da metodologia, utilizou-se a ferramenta Understand[13] para se obter métricas de software para cada classe. Dentre as métricas que a ferramenta permite medir, neste trabalho destacamos as métricas de software por permitirem a análise de diferentes características das classes analisadas: quantidade de linhas do código fonte da classe, média de complexidade ciclomática, quantidade de métodos da classe e o nível máximo de estruturas encadeadas (*if, else, while* etc). A Complexidade Ciclomática é uma métrica de software que permite avaliar o quanto o programa é complicado. Nesse sentido, essa medida permite medir a quantidade de caminhos independentes que podem ser executados no sistema, tendo sido desenvolvido por Thomas McCabe[14]. Na documentação da ferramenta Understand, tem-se a definição de *Average Cyclomatic Complexity* (AvgCyclomatic) como a média da complexidade ciclomática, *Public Methods* (CountDeclMethodPublic) como a quantidade de métodos públicos, *Nesting* (MaxNesting) como o nível máximo de estruturas encadeadas, como *if*, *else* e *while*, na função e *Source Lines of Code* (CountLineCode) como a quantidade de linhas do código.

Estas métricas de software permitem a análise dos resultados obtidos na metodologia, realizando uma comparação dos resultados obtidos na aplicação da metodologia e as métricas de software selecionadas. Assim, é possível, por exemplo, analisar, empiricamente, se a metodologia permitiu a criação de uma quantidade maior de testes na aplicação da metodologia realizada na classe que apresentou a maior média de complexidade ciclomática.

12 Disponível em: https://www.eclemma.org/. Acesso em: 02 abr. 2020.
13 Disponível em: https://scitools.com. Acesso em: 18 out. 2020.
14 Disponível em: https://scitools.com/support/mccabe-cyclomatic-complexity/. Acesso em: 18 out. 2020.

Na análise dos resultados da aplicação da metodologia foi utilizada também a ferramenta de teste de mutação PIT[15]. Em síntese, a ferramenta PIT gera mutações (falhas) no código e a suíte de testes é executada posteriormente. Se o teste falhar, a mutação é morta (*killed*); se o teste passar, então a mutação vive (*lived*). O propósito da utilização desta ferramenta é permitir melhor fundamentação na resposta da terceira questão de pesquisa sobre descoberta de erros na aplicação com o uso da metodologia. A ferramenta possibilita a avaliação da qualidade dos testes a partir da percentagem de mutações mortas.

### 5.3 Classes utilizadas para aplicação da metodologia

Conforme já dito, a metodologia é aplicada em uma determinada classe e para a utilização nos experimentos, inicialmente, foram selecionados sistemas baseados em alguns critérios. Tais sistemas precisavam possuir código fonte com acesso público, pois seriam utilizados nos experimentos, e publicados em repositórios de acesso público, como GitHub, ainda que não excluindo outros. Além disso, nos sistemas escolhidos, procurou-se selecionar sistemas que tratam de assuntos heterogêneos e de domínios variados, evitando assim um único domínio e algum viés derivado.

Seguindo os critérios citados, os sistemas selecionados foram *commons-lang*, *netflix-commons*, *commons-jxpath* e o *benchmark ristretto*. O sistema *commons-lang* se refere a um pacote de classes de utilitários Java para as classes que estão na hierarquia do *java.lang*. O *netflix-commons* é uma biblioteca disponível para dar suporte ao projeto do Netflix OSS. O *commons-jxpath* é um interpretador simples de uma linguagem de expressão chamada XPath. O *ristretto* faz parte de uma aplicação de gerenciamento de e-mail.

A seleção das classes nos sistemas selecionados considerou como critério de escolha a existência de métodos públicos, preferencialmente que contenham chamadas a outros métodos da própria classe ou de outras. O critério de seleção de classes que possuem métodos públicos baseia-se em boas práticas que orientam que testes que interagem diretamente com métodos não públicos violam noções de ocultações de informações (LANGR; HUNT; THOMAS, 2015). Outro critério adotado para a seleção

15 Site da ferramenta PIT. Disponível em: https://pitest.org/ Acesso em: 02 abr. 2020.

de classes para a aplicação da metodologia é que as classes contenham métodos com chamadas a outros métodos, o que permitiria a geração de sequências de chamadas.

### 5.4 Métricas de avaliação da eficácia da aplicação da metodologia

Aqui são descritas as métricas utilizadas para avaliar a metodologia. Tais métricas, foram: 1) a quantidade de novos testes criados por meio da aplicação da metodologia, que se refere a todos os testes gerados diferentes do conjunto inicial obtido das ferramentas de geração automática; 2) a cobertura dos testes, que envolveu análise dos possíveis aumentos de cobertura nos componentes e seus respectivos conjuntos de testes antes e após a aplicação da metodologia. 3) erros descobertos através da aplicação da metodologia, o que envolveu a aplicação da ferramenta de testes de mutações.

Neste trabalho, das métricas de software existentes utilizamos a média da complexidade ciclomática da classe (*AvgCyclomatic*), a quantidade de métodos públicos (*CountDeclMethodPublic*), quantidade de linhas de código fonte (*CountLineCode*) e o nível máximo de estruturas encadeadas *(MaxNesting).* A medida de quantidade de métodos (*CountDeclMethodPublic*) considera até mesmo os construtores das classes. No entanto, nos experimentos não foram considerados testes para construtores. Tais escolhas foram feitas por permitirem diferentes análises em relação às classes que compõem os experimentos e os resultados obtidos com a aplicação da metodologia. Além disso, considerou-se a escolha de duas medidas que analisam de forma diferente a complexidade de cada classe (*MaxNesting* e *AvgCyclomatic*) e duas medidas que poderiam dar uma noção de tamanho das classes envolvidas no experimento em relação à quantidade de linhas e quantidade de métodos (*CountDeclMethodPublic* e *CountLineCode*).

Conforme exposto na Tabela 5.1, nos experimentos foram considerados classes com diferentes médias de complexidade ciclomática, o que contribui para avaliar a metodologia em diferentes complexidades ciclomáticas. Na Tabela 5.2 estão listados os nomes e respectivos endereços dos sistemas utilizados nos experimentos. Os pacotes dos sistemas que contém as classes selecionadas para o experimento contém respectivamente: 37 classes (BitField/commons-lang); 10 classes (DataBuffer/netflix-commons); 9 classes (InfoSetUtil/commons-jxpath); 5 classes (SMTPProtocol/risttreto).

Tabela 5.1 – Tabela com os componentes e as métricas de software utilizadas no experimento: *AvgCyclomatic*, *CountDeclMethodPublic e CountLineCode*.

| Componente/ Sistema | Média da Complexidade Ciclomática *(AvgCyclomatic)* | Quantidade de métodos públicos *(CountDeclMethodPublic)* | Nível máximo de estruturas encadeados *(MaxNesting)* | Quantidade de linhas de código fonte *(CountLineCode)* |
|---|---|---|---|---|
| BitField/ commons-lang | 1,22 | 18 | 0 | 138 |
| DataBuffer/ netflix-commons | 1,73 | 10 | 3 | 186 |
| InfoSetUtil/ commons-jxpath | 9,25 | 4 | 2 | 370 |
| SMTPProtocol /risttreto | 3,41 | 22 | 4 | 1438 |

Fonte: Autor

Tabela 5.2 – Tabela com os componentes em que a metodologia foi aplicada, seus respectivos sistemas e o endereço para acessso a eles.

| Componente/Sistema | Endereço do Sistema |
|---|---|
| BitField/commons-lang | https://github.com/apache/commons-lang/ |
| DataBuffer/netflix-commons | https://github.com/Netflix/netflix-commons |
| InfoSetUtil/commons-jxpath | https://github.com/apache/commons-jxpath/ |
| SMTPProtocol/risttreto | https://sourceforge.net/projects/columba/files/Ristretto |

Fonte: Autor

### 5.4 Resultados e análise das questões de pesquisa

Considerando as questões de pesquisa propostas em 5.1, buscou-se respondê-las com base nos resultados obtidos da aplicação da metodologia nos sistemas selecionados. Para isso, utilizaram-se as ferramentas de geração automática de testes JUnit-Tools e EVOSUITE para a obtenção do conjunto inicial de testes.

A intenção do uso de uma ferramenta de geração automática de testes como o JUnit-Tools, que gera conjunto de testes simples, é simular a situação em que um desenvolvedor pretende aplicar a metodologia e possui um conjunto de testes o mais

simples possível, potencialmente por conta de sua pouca experiência ou pouco conhecimento sobre geração de testes, que efetua pelo menos uma chamada a cada método público da classe testada. Já a utilização de uma ferramenta de geração de testes como a EVOSUITE permite a análise do comportamento da aplicação da metodologia utilizando um conjunto de testes mais elaborado, considerando que esta ferramenta gera testes que pretendem atingir 100% de cobertura de *branch*.

A seguir, cada uma das questões de pesquisa propostas em 5.1 será analisada com base nos resultados obtidos nos experimentos realizados.

**1. A aplicação da metodologia permite a geração de novos testes?**

Para responder a esta questão foram consideradas a quantidade de testes gerados de forma automática através das ferramenta JUnit-Tools e EVOSUITE e a quantidade de novos testes gerados pela aplicação da metodologia. Além disso, foram realizadas análises dos resultados obtidos após a aplicação da metodologia em relação às métricas de software medidas para cada classe. O objetivo da análise desta relação foi verificar os possíveis diferentes resultados da metodologia obtidos em classes com diferentes medidas de software. Dessa forma, podemos avaliar melhor se em uma classe com uma complexidade maior foi possível gerar mais casos de testes do que em uma classe com uma complexidade menor.

Inicialmente, foram avaliados os resultados obtidos na aplicação da metodologia utilizando a ferramenta JUnit-Tools. Estes resultados são apresentados na Tabela 5.3. Nesta tabela, a primeira coluna indica o componente analisado, a segunda coluna descreve a quantidade de testes gerados automaticamente pela ferramenta JUnit-Tools, a terceira coluna descreve a quantidade de novos testes gerados ao final da aplicação da metodologia e a quarta coluna apresenta a porcentagem de novos testes gerados. Conforme definido no capítulo da metodologia, a cada novo teste gerado é realizada uma nova iteração da metodologia. Nesse sentido podemos considerar que para cada classe a quantidade de iterações da aplicação da metodologia será a quantidade de novos testes gerados.

Em relação às quantidades de novos testes gerados e às quantidades de testes gerados pela ferramenta, pode-se verificar que as quantidades de testes automáticos, em geral, são maiores do que as quantidades de novos testes. Isto pode ser explicado pelo fato de que a ferramenta JUnit-Tools, em geral, gera um teste para cada método público

da classe a ser testada. Assim, em classes com grande quantidade de métodos públicos, como *BitField*, em geral, tem-se uma quantidade de testes no conjunto inicial maior do que a quantidade de novos testes gerados por meio da aplicação da metodologia.

Tabela 5.3 – Tabela com a descrição das quantidades de testes gerados, utilizando o conjunto de testes inicial gerado pela ferramenta JUnit-Tools

| Componente | Quantidade de testes JUnit-Tools | Quantidade de novos testes | Porcentagem de novos testes gerados |
|---|---|---|---|
| BitField | 17 | 9 | 52,94% |
| DataBuffer | 10 | 4 | 40,00% |
| InfoSetUtil | 4 | 5 | 125,00% |
| SMTPProtocol | 20 | 0 | 0% |

Fonte: Autor

Conforme apresentado na Tabela 5.3, a aplicação da metodologia possibilitou a geração de novos testes em três das quatro classes analisadas. Através dos resultados apresentados, pode-se perceber que na maioria das aplicações da metodologia houve a criação de novos testes, o que indica que a metodologia permite a geração de testes, conforme o objetivo proposto da aplicação da metodologia.

Na aplicação da metodologia na classe *SMTPProtocol*, utilizando a ferramenta JUnit-Tools não foi possível gerar nenhum novo teste. Nesta classe foi observado, através do código fonte do componente e o modelo gerado a partir do conjunto inicial, que a regra de negócio contida no primeiro método executado neste conjunto de testes precisava de um valor de teste específico para ser executado. Como a ferramenta de geração de testes JUnit-Tools não conseguiu prever o valor de teste necessário para a execução deste método, foi necessário atribuir um valor específico, referente ao valor da porta utilizado em um dos métodos, para a conclusão da aplicação da metodologia, já que este método deveria ser o primeiro a ser executado antes de qualquer outro método do componente. Tal fato pode ser atribuído à simplicidade dos testes que, mesmo recebendo valores de testes compatíveis, não permitiram a exploração dos demais contextos executáveis. Além disto, através da análise do teste legado existente desta classe, foi possível observar que os métodos da classe precisavam serem executados em determinada ordem e com valores bem específicos, caso contrário os testes não poderiam serem executados.

Em relação à quantidade de novos testes gerados por meio da aplicação da metodologia e a medida da média de complexidade ciclomática de cada classe, por meio

dos resultados se observou, pela Tabela 5.1, que a metodologia permitiu a criação de novos testes tanto na classe *BitField*, que apresenta a menor média de complexidade ciclomática (1,22), quanto na classe que apresenta a maior média de complexidade ciclomática, a *InfoSetUtil* (9,25). No entanto, pode-se observar que na classe que apresentou a maior complexidade ciclomática, a metodologia possibilitou a criação de uma quantidade maior de novos testes do que a quantidade de testes geradas de forma automática. A coluna de porcentagem de novos testes gerados da Tabela 5.3 confirma tal fato ao indicar que o maior aumento de porcentagem (125 %) foi observado na classe *InfoSetUtil*. Ao analisar as classes *InfoSetUtil* e *BitField* se observou que na classe com maior complexidade ciclomática, ou seja, que possui mais caminhos a serem executados, a metodologia pode contribuir com a geração de uma maior quantidade de novos testes, porque possui mais possibilidades de caminhos diferentes a serem executados.

Em relação à medida de encadeamento das estruturas (*MaxNesting*) das classes e quantidade de novos testes gerados por meio da aplicação da metodologia, houve a geração de novos testes nas classes *BitField, DataBuffer* e *InfoSetUtil* que apresentaram, respectivamente, medidas de 0, 2 e 3, conforme visualizado na Tabela 5.1. Através dos resultados apresentados na Tabela 5.3 e as medidas de encadeamento das estruturas, pode-se mostrar que a metodologia permite a criação de novos testes tanto em classes que apresentam baixo nível de encadeamento de estruturas quanto em classes com um nível mais alto.

Em relação às medidas de software referentes a linhas dos códigos fontes das classes e as quantidades de novos testes obtidos através da aplicação da metodologia, não foi possível identificar nenhum resultado significativo em que fosse possível identificar que uma classe com mais quantidade de linhas apresente, com a aplicação da metodologia, uma maior quantidade de novos testes em relação a uma classe com menor quantidade de linhas de códigos. Por exemplo, foi observado que a classe *BitField* apresentou a menor quantidade de linhas de código e a maior quantidade de novos testes. Já na classe com a maior quantidade de linhas do código fonte, *SMTPProtocol*, não foi possível gerar novos casos de testes.

Para demonstrar os resultados obtidos pela aplicação da metodologia, com uso de um conjunto inicial mais elaborado do que o conjunto de testes gerado pela ferramenta JUnit-Tools, foi utilizada a ferramenta EVOSUITE. Na Tabela 5.4 é possível visualizar os resultados obtidos nas aplicações da metodologia utilizando essa ferramenta. A primeira coluna da Tabela 5.4 se refere ao nome dos componentes analisados no

experimento, a segunda coluna se refere à quantidade de testes criados automaticamente pela ferramenta EVOSUITE, a terceira coluna se refere à quantidade de novos testes gerados e a quarta coluna apresenta a porcentagem de novos testes gerados

Tabela 5.4 – Tabela com a descrição das quantidades de testes gerados automaticamente pela ferramenta EVOSUITE e novos testes criados após a aplicação da metodologia.

| Componente | Quantidade de testes gerados pela EVOSUITE | Quantidade de novos testes | Porcentagem de novos testes gerados |
|---|---|---|---|
| BitField | 52 | 9 | 17,30% |
| DataBuffer | 22 | 7 | 31,81% |
| InfoSetUtil | 26 | 4 | 15,38% |
| SMTPProtocol | 29 | 0 | 0% |

Fonte: Autor

Foi observado que a quantidade de testes criados pela ferramenta EVOSUITE foi bem maior do que a quantidade de novos testes gerados. Tal feito pode ser atribuído à completude dos testes criados pela ferramenta EVOSUITE, que utiliza um algoritmo mais robusto e gera testes baseados em diferentes critérios. Assim, através dos resultados apresentados na Tabela 5.4 foi possível observar que a metodologia permite a geração de novos testes mesmo com um conjunto de testes inicial mais completo do que um conjunto que executa apenas uma chamada a cada método público.

Em relação as porcentagens de novos testes gerados apresentados nas Tabelas 5.3 e 5.4, foi possível observar que as maiores porcentagens foram identificadas nas aplicações da metodologia utilizando a ferramenta JUnit-Tools. Isto pode ser atribuído às quantidades de testes gerados pelas ferramentas. Assim, as porcentagens de novos testes foram maiores quando se utilizou a ferramenta JUnit-Tools, devido ao fato dessa ferramenta ter gerado uma menor quantidade de testes para o conjunto inicial.

Em relação à classe *SMTPProtocol*, assim como no caso da aplicação da metodologia utilizando o JUnit-Tools, não foi possível gerar novos testes a partir da aplicação da metodologia, devido à necessidade de especificidade dos valores que deveriam ser utilizados nos testes para que fosse possível a execução do sistema. Além disto, analisando o código fonte foi possível identificar que os testes da maioria dos métodos desta classe não poderiam ser realizados de forma a testar uma chamada ao método em cada caso de teste da suíte. Assim, era necessário que cada caso de teste

executasse uma sequência de chamada de métodos específicos e com valores específicos para que fosse possível executar o teste unitário da classe. Nesse sentido, mesmo utilizando uma ferramenta de geração de testes mais completa não foi possível gerar um conjunto de testes inicial executável e assim executar a aplicação da metodologia. Essa conclusão foi possível, pois essa classe possuía um conjunto de testes legado, em que foi possível ver os valores de testes usados e comparar com os valores de testes gerados pelas ferramentas.

Em relação a um mesmo componente, comparando-se os resultados obtidos da aplicação da metodologia utilizando as ferramentas JUnit-Tools e EVOSUITE, foi possível observar que não houve uma grande diferença entre as quantidades de novos testes gerados a partir das duas ferramentas. Nesse sentido, notadamente, a diferença encontrada nas aplicações da metodologia utilizando as duas ferramentas foi no aumento da cobertura de testes. Isto pode ser atribuído aos diferentes testes gerados pelas ferramentas. Assim, como a ferramenta EVOSUITE gerou testes mais completos e com a maior cobertura possível utilizando o algoritmo da ferramenta, não foi possível observar um aumento significativo de cobertura de *branch* ou *linha* com a inclusão de novos testes. O tempo gasto na criação de um teste manual depende de vários fatores, tais como: experiência do desenvolvedor, conhecimento da aplicação, legibilidade do código, dentre outros. Nesse sentido, esse é um dado subjetivo. Comumente, na criação do teste manual, o desenvolvedor precisa de conhecimento das técnicas necessárias de testes e das especificações dos sistemas relativos ao componente que se pretende testar. Conforme já foi explicado, um dos objetivos da metodologia é permitir a criação de testes em sistemas sem documentação e/ou testes legados e o custo da aplicação da metodologia foi medido no intuito de se avaliar o custo-benefício da metodologia. O tempo gasto na geração do conjunto de testes inicial feito pelas ferramentas JUnit-Tools e EVOSUITE, realizado de forma automática, foi considerado igual para todas as aplicações da metodologia.

Apesar de a metodologia possuir uma etapa automática, a de geração de testes iniciais, pode-se observar que a aplicação da metodologia demanda um tempo gasto na geração dos novos casos de testes, que é realizada de forma manual. A etapa manual de geração desses novos testes inclui a análise do código, análise do conjunto de testes executado, viabilidade dos possíveis novos testes e análise do modelo gerado. Apesar do gasto de tempo demandado pela metodologia, é importante destacar que os novos testes têm o benefício de serem não aleatórios e baseados em análise de modelo. Isto permite a criação de testes mais fundamentados do que testes criados manualmente. Nesse contexto,

mais fundamentados significam testes não aleatórios e gerados a partir da integração da técnica de testes e verificação de modelos. Além disto, a geração de testes a partir da aplicação da metodologia permite, além da própria geração do caso de teste, a construção automática do modelo do sistema, o que produz uma forma de documentação do sistema e possibilita outros tipos de análises.

Em relação ao tempo gasto nas aplicações da metodologia, foi possível observar empiricamente que a aplicação da metodologia implica gasto de tempo variável. Foi percebido que o custo de tempo da aplicação é influenciado pela composição do conjunto inicial de testes e seus valores. Empiricamente, identificou-se que, nos casos em que o conjunto de testes inicial utilizado já explora consideravelmente os possíveis contextos de execução do sistema, a metodologia pode não contribuir para a geração de novos testes. Por outro lado, permite verificar a completude do conjunto de testes inicial, já que permite a geração e verificação do modelo. Além disso, a utilização na metodologia da técnica de testes, extração de modelos e análise do modelo pode auxiliar na identificação de possíveis fluxos incorretos, exceções não tratadas, trechos de códigos depreciados ou não executáveis, o que contribui para a melhoria da qualidade do código por possibilitar a identificação de erros.

Os resultados obtidos nos experimentos mostraram que, no geral, a metodologia permitiu a geração de novos testes nas classes analisadas. Além da quantidade de testes, a aplicação da metodologia permitiu a geração de testes não aleatórios, sem a obrigatoriedade do conhecimento prévio das especificações do sistema. Identificou-se ainda que a quantidade de novos testes pode ser influenciada por alguns fatores, tais como: chamadas de outros métodos dentro de um método, valores utilizados nos testes, tipos de retornos dos métodos testados. Isto ocorre por tais fatores poderem ser desencadeadores de novos caminhos a serem descobertos e assim gerar novos testes.

**2. Caso sejam criados novos testes, estes testes contribuem para o aumento da cobertura de testes?**

Esta pergunta busca analisar o possível aumento na cobertura de testes resultantes quando há a inclusão de novos testes. Para isso, foram medidas as coberturas de testes em relação a cada componente antes e após a aplicação da metodologia. As coberturas de linha e de *branch* foram selecionadas para avaliar a metodologia por se entender que são potencialmente eficazes nessa função, dado que na aplicação da metodologia são

utilizadas classes compostas do código fonte da aplicação e essas coberturas serem as mais indicadas para análise de cobertura de código fonte

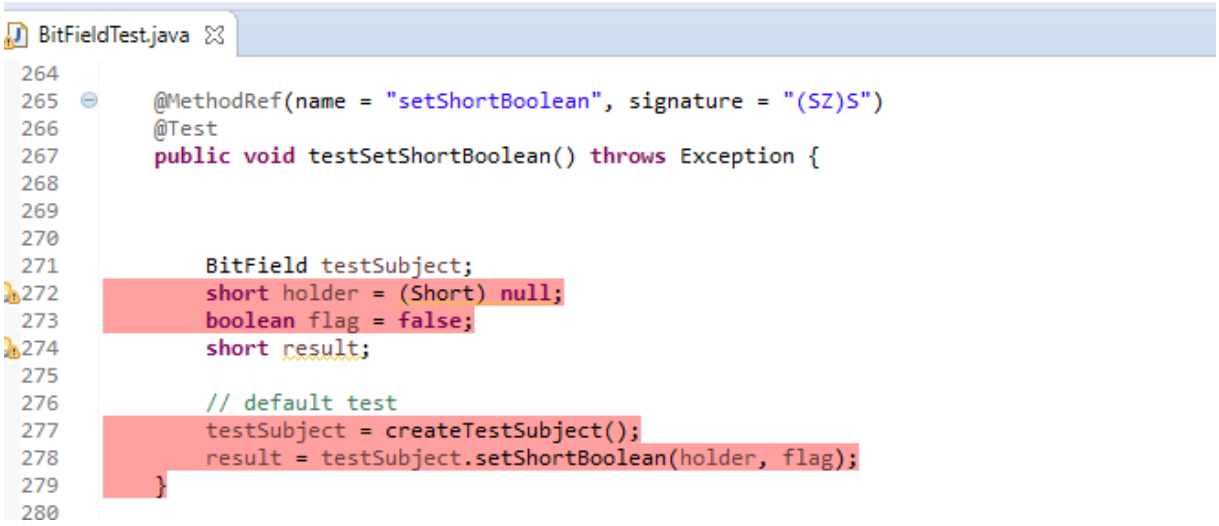
Nas aplicações da metodologia realizadas nos experimentos utilizando a ferramenta JUnit-Tools, foi possível constatar aumento de cobertura de linhas somente na classe *InfoSetUtil*. Neste caso, essa classe possui métodos com parâmetro de entrada *Object* que poderiam assumir os tipos *String, Number* ou *Boolean*, de acordo com os valores atribuídos nos testes, e que, assim, poderiam permitir execuções de diferentes caminhos independentes. No teste gerado pela ferramenta JUnit-Tools para esta classe, o teste assumiu somente um dos tipos possíveis. Assim, nos novos testes criados foram realizadas execuções aos métodos de forma que os outros tipos possíveis foram executados, o que resultou no aumento de cobertura de linha, já que foi possível executar outras linhas do código da classe. Em relação à classe *InfoSetUtil*, os resultados encontrados mostraram que houve um aumento da cobertura de linha deste componente de 40.8% para 49.3%. Nesta classe, pode-se destacar que, apesar de haver poucos métodos públicos (quatro no total), a média da complexidade ciclomática de 9,25 foi a mais alta dos componentes utilizados no experimento. Isto indica que essa classe possui maior quantidade de caminhos independentes que podem ser executados nos testes. Na aplicação da metodologia utilizando a ferramenta EVOSUITE, considerando o mesmo componente *InfoSetUtil*, não foi possível observar aumento da cobertura de linhas. Aliás, nenhuma aplicação da metodologia utilizando a ferramenta EVOSUITE apresentou aumento de cobertura de linhas, pois a ferramenta gerou testes com o máximo de percentual de coberturas de linhas possível.

Na análise das coberturas de linhas do experimento, observou-se que alguns testes gerados pela ferramenta JUnit-Tools não foram executados devido ao valor de teste gerado pela ferramenta JUnit-Tools, conforme a Tabela 5.5. Nesse sentido, se o valor de teste fosse alterado, o teste poderia ser executado. No entanto, como o definido anteriormente, para a normatização dos experimentos e facilitação das análises dos resultados obtidos neste experimento, os valores de testes gerados pelas ferramentas de geração de testes foram mantidos.

Na Figura 5.1, é possível visualizar um exemplo desse caso relatado em que o teste do método *setShortBoolean* gerado pela ferramenta JUnit-Tools não pode ser executado, já que o valor *(Short) null* foi atribuído para o parâmetro de entrada *holder* no teste. Na fase de análise do conjunto inicial de testes da metodologia foi identificado que, neste caso, o teste poderia ser executado se o valor da variável *holder* recebesse outros

valores tais como 0 ou 1. Neste contexto, a alteração do valor de teste permitiria a execução do método e resultaria no aumento da cobertura de linha e *branch*. Assim, em uma aplicação da metodologia em outro contexto não padronizado, o desenvolvedor poderia atribuir os valores que melhor conviesse para o propósito pretendido da aplicação da metodologia, podendo obter um melhor resultado.

Figura 5.1 – Figura que mostra o *testSetShortBoolean* gerado pela ferramenta JUnit-Tools e não executável devido ao valor de teste gerado pela ferramenta e atribuído à variável *holder*.




Fonte: Autor

Tabela 5.5 – Tabela com a descrição das coberturas de *branch* antes e após a aplicação da metodologia, utilizando o conjunto de testes inicial gerado pela ferramenta JUnit-Tools.

| Componente | Cobertura de *branch* do conjunto inicial de testes gerado pela JUnit-Tools | Cobertura de branch do componente após a inclusão de novos testes | Quantidade de novos testes |
|---|---|---|---|
| BitField | 41.7% | 58,3% | 9 |
| DataBuffer | 12,5% | 12,5% | 5 |
| InfoSetUtil | 31.9% | 50% | 4 |
| SMTPProtocol | 5.1% | 5.1% | 0 |

Fonte: Autor

Na Tabela 5.5 é possível verificar que nenhuma medição de cobertura apresentou um valor igual a 100%, provavelmente devido à simplicidade do conjunto de testes inicial gerado pela ferramenta JUnit-Tools. Evidentemente, os componentes com cobertura de testes menores do que 100% apresentam a possibilidade de aumento da cobertura com a inclusão de novos testes nos conjuntos de testes iniciais.

Ainda segundo os resultados apresentados na Tabela 5.5, identificou-se que as aplicações da metodologia nas classes *BitField* e *InfoSetUtil* apresentaram aumento de cobertura de *branch*. A classe *BitField*, que possui a menor média de complexidade ciclomática (1,22), apresentou um aumento de cobertura de *branch* de 41,7% para 58,3%. Já a classe *InfoSetUtil*, que apresentou a maior média de complexidade ciclomática, apresentou o maior aumento de cobertura de *branch*, de 31,9% para 50%. Isto pode ser atribuído ao fato da classe *InfoSetUtil* conter mais caminhos independentes a serem percorridos.

Nos resultados obtidos na aplicação da metodologia da classe *DataBuffer*, utilizando a ferramenta JUnit-Tools, foi observado que, apesar de ter havido a geração de novos testes e o conjunto de testes inicial ter apresentado um valor baixo de cobertura de *branch*, 12,5%, não foi possível verificar um aumento de cobertura na aplicação da metodologia. Após análise manual dessa classe *DataBuffer*, foi observado que se fossem utilizados determinados valores de testes nos novos testes, poderia haver aumento de cobertura de *branch*. No entanto, como já definido, neste experimento, os valores de testes gerados pelas ferramentas foram mantidos nos testes para padronizar e facilitar a análise dos resultados das aplicações da metodologia.

Tabela 5.6 – Tabela com a descrição das coberturas de *branch* antes e após a aplicação da metodologia, utilizando o conjunto de testes inicial gerado pela ferramenta EVOSUITE.

| Componente | Cobertura de *branch* do conjunto inicial de testes gerado pela EVOSUITE | Cobertura de *branch* do componente após a inclusão de novos testes | Quantidade de novos testes |
|---|---|---|---|
| BitField | 100% | 100% | 9 |
| DataBuffer | 75% | 75% | 7 |
| InfoSetUtil | 73,6% | 73,6% | 4 |
| SMTPProtocol | 6.1% | 6.1% | 0 |

Fonte: Autor

Conforme já esperado, o resultado apresentado na Tabela 5.6 mostrou que os conjuntos iniciais de testes gerados pela ferramenta EVOSUITE apresentaram uma maior porcentagem de cobertura de *branch* do que os testes gerados pela ferramenta Junit-Tools. Isto se deve às características dos testes gerados pela ferramenta EVOSUITE, que buscam gerar testes com alto percentual de cobertura. Nas aplicações da metodologia foi observado que os novos testes incluídos no conjunto inicial de testes nas aplicações utilizando a ferramenta JUnit-Tools resultaram em um aumento de cobertura de *branch*

maior do que as aplicações realizadas com a utilização da ferramenta EVOSUITE, porque os testes gerados pela ferramenta EVOSUITE já eram gerados com uma taxa de cobertura de *branch* alta devido ao algoritmo utilizado pela ferramenta desenvolvido para gerar testes com altas taxas de cobertura de *branch*. Na classe *BitField*, que apresentou a cobertura de *branch* de 100%, logicamente, mesmo com a inclusão de novos testes, não seria possível se obter um aumento de cobertura.

As aplicações da metodologia nas classes *DataBuffer* e *InfoSetUtil*, utilizando a ferramenta EVOSUITE não apresentaram nenhum aumento na cobertura de *branch*, o que também se deve às características dos testes gerados pela ferramenta EVOSUITE. Assim como na aplicação da metodologia, utilizando a ferramenta JUnit-Tools, pelos motivos já descritos, a aplicação da metodologia na classe *SMTPProtocol*, não apresentou a geração de nenhum novo teste.

Teoricamente, a metodologia pode possibilitar um aumento da cobertura de testes, já que permite a inclusão de novos testes. Através dos experimentos foi observado que em classes que contêm mais métodos que executam chamadas a outros métodos da mesma classe ou outra classe, a metodologia pode possibilitar um maior aumento da cobertura de *branch*, porque a metodologia permite a descoberta de novos caminhos que podem ser executados no sistema não identificados no conjunto de testes inicial.

É importante destacar que os novos testes são gerados a partir da junção das técnicas de análise do modelo e testes, portanto não são aleatórios. Os resultados sugerem ainda que a aplicação da metodologia em uma classe mais complexa poderia apresentar um aumento de cobertura e possibilitaria a execução de diferentes contextos. Além disso, indica também que a inclusão de novos testes, não implica, necessariamente, um aumento de cobertura.

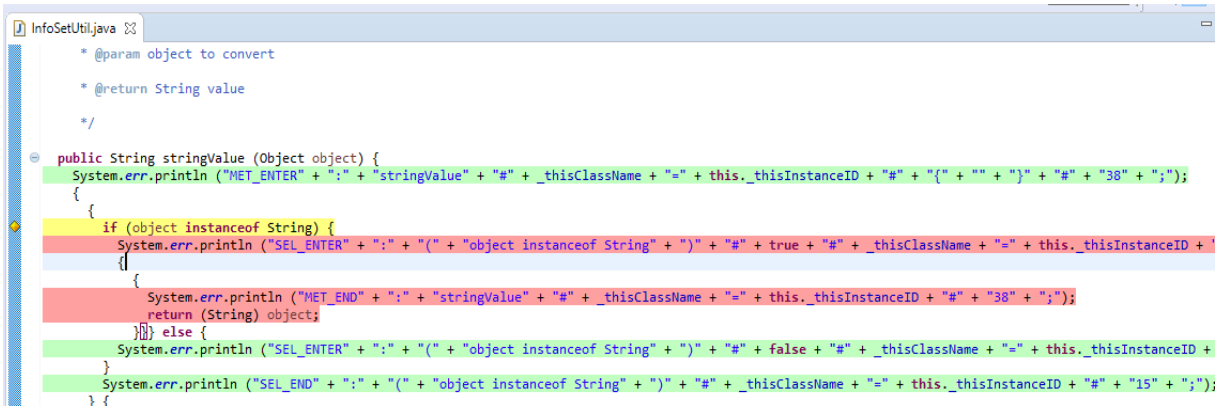
Considerando-se a abordagem de extração de modelos baseada em contextos usada na aplicação da metodologia, podemos observar nos resultados do experimento obtidos que o modelo extraído do sistema pode não sugerir algumas sequências de chamadas de métodos como contraexemplos. Isto porque a construção do modelo pode unificar rastros individuais pela combinação de contextos iguais, o que impediria a geração de contraexemplo para esta combinação. Por outro lado, esta característica ajuda a limitar o número de contraexemplos a serem analisados, colaborando para que se convirja para o critério de parada definido na metodologia.

**3. A metodologia contribui para a descoberta de erros na aplicação?**

Esta pergunta busca analisar principalmente a contribuição da metodologia na descoberta de erros na aplicação. Para isso, a discussão acerca dos erros na aplicação envolve diferentes elementos, tais como: testes gerados, cobertura dos testes e modelos gerados.

No exemplo da Figura 5.2, é possível visualizar um trecho do código fonte instrumentado do componente *InfoSetUtil*. A utilização do JUnit nos testes e da Ecclema permite a identificação através das cores dos códigos executados ou não. No trecho, as partes em verde representam os códigos executados pelo teste gerado automaticamente pela ferramenta JUnit-Tools. Já as partes sinalizadas pela cor vermelha representam as partes que não foram alcançadas por meio da execução do teste. Ciente dos valores de teste gerados pela ferramenta JUnit-Tools foi possível verificar que o trecho em vermelho não foi executado devido ao valor de teste utilizado. Nesse sentido, a utilização da ferramenta JUnit-Tools nos experimentos foi importante na avaliação da metodologia no que se refere a testes com valores limitados.

Figura 5.2 – Figura que mostra parte do código fonte instrumentado do componente InfoSetUtil com a sinalização da cobertura alcançada pelo teste executado.




Fonte: autor.

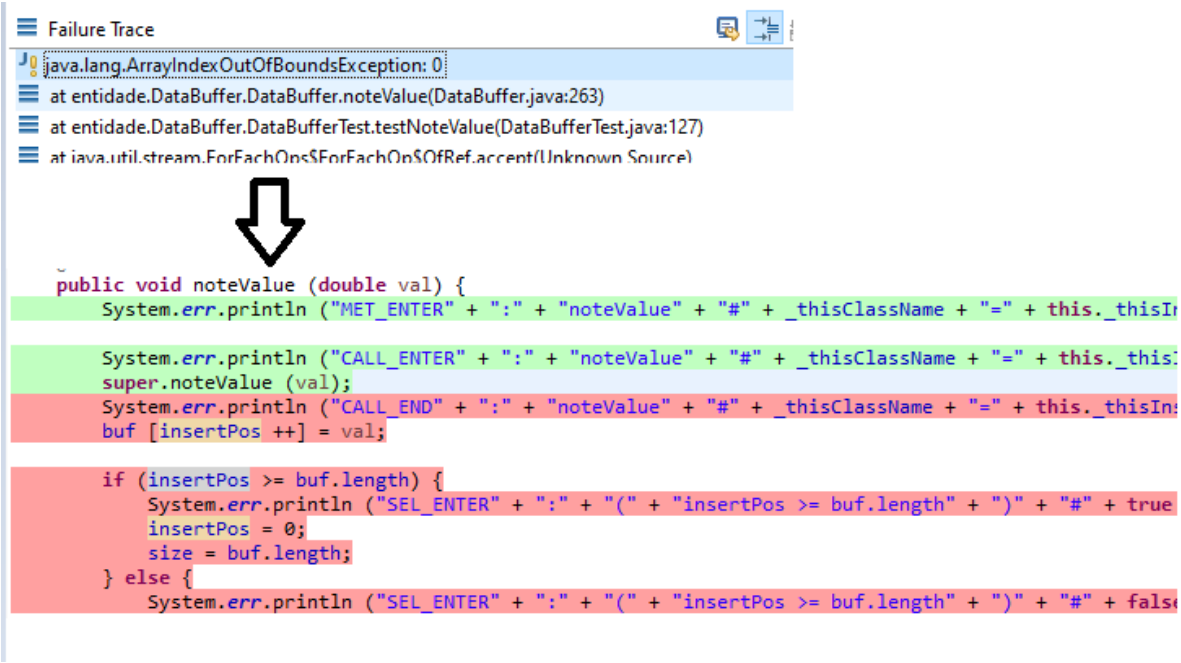
Neste exemplo, o valor de testes atribuído ao parâmetro *object* do método *stringValue* não permitiu a execução da estrutura *"if (object instanceof String)*" mostrada na Figura 5.2. Desta forma, o código dentro dessa estrutura não foi executado e a saída do método aconteceu sem que o restante do método pudesse ter sido percorrido. Desse modo, assim como o trecho exposto, a análise dos trechos não executados por testes ou

mesmo dos trechos executados pode resultar na identificação de códigos depreciados ou possíveis erros na aplicação. Assim, como a metodologia foi estruturada para permitir a geração de testes em sistemas sem a documentação, as análises a partir destes testes gerados, como as mostradas, podem auxiliar na identificação de erros.

Na Figura 5.3 é possível visualizar a falha apresentada por um teste gerado pela ferramenta JUnit-Tools e executado no JUnit, em que a falha foi percebida após a execução. Esta falha ocorreu devido ao valor de testes usado ser *null*.

Neste exemplo, houve a necessidade de se verificar o código fonte do componente analisado e/ou os testes para se verificar como a falha apresentada no teste poderia ser corrigida ou mesmo qual a ação necessária neste caso. Conforme definido anteriormente, os valores de testes gerados pela ferramenta de geração de testes foram mantidos na aplicação da metodologia. Nesse sentido, assim como nesse exemplo, as análises das falhas geradas nos testes poderiam facilitar a identificação de erros, principalmente, em um cenário de sistemas sem documentação como os cenários em que a metodologia busca auxiliar.

Figura 5.3 – Figura que mostra o método da classe DataBuffer e a falha identificada na execução do respectivo teste gerado pela ferramenta JUnit-Tools.




Fonte: Autor

Através dos experimentos também foi possível identificar que os modelos gerados nas aplicações da metodologia podem permitir a análise das interações dos métodos executados nos testes, inclusive as interações dos métodos públicos, privados e

protegidos. Isto é possível através das transições do modelo que representam estes métodos. Os modelos também podem servir como documentação.

A ferramenta PIT foi utilizada para verificar os possíveis erros identificados nos testes gerados. Mutações mortas são indicativos de falhas nos testes criados, em que mutações são alterações inseridas no código fonte para avaliar se os testes conseguem identificar as mutações inseridas no código. Quando os testes conseguem identificar as mutações, é dito que a mutação foi morta. Ainda conforme o site da ferramenta PIT, a porcentagem de mutações mortas pode ser utilizada como referência de qualidade dos testes. Em relação ao conjunto de testes gerado na aplicação da metodologia utilizando a ferramenta JUnit-Tools, foi possível notar que a classe *DataBuffer* apresentou 4 mutações mortas (*killed*), a classe *InfoSetUtil* 19 mutações mortas e a classe *BitField* apresentou 0. Conforme explicado nas seções em que foram analisados os resultados das aplicações da metodologia referentes às questões de pesquisa 1 e 2, a aplicação da metodologia na classe *SMTPProtocol* não foi possível devido a restrições específicas da classe. Assim, a classe *SMTPProtocol* não pode ser analisada pela ferramenta PIT. Nesse sentido, a classe com maior média de complexidade ciclomática apresentou a maior quantidade de mutações mortas. Em relação ao conjunto de testes gerado na aplicação da metodologia utilizando a ferramenta EVOSUITE, a ferramenta PIT Mutation não conseguiu gerar tais estatísticas, possivelmente por incompatibilidade dos testes gerados pela EVOSUITE com a ferramenta PIT.

## 5.6 Limitações e desafios da metodologia

Nesta seção são descritas as limitações e desafios identificados no decorrer das aplicações da metodologia. Nas limitações identificadas nos experimentos, são descritas algumas das soluções encontradas e aplicadas durante os experimentos.

Na fase de geração dos testes do conjunto inicial, um dos principais desafios do desenvolvedor na aplicação da metodologia é gerar um conjunto de testes inicial que possibilite a geração de um modelo que, ao final da aplicação da metodologia, contenha todas as possíveis transições do sistema e permita a geração do conjunto de testes final o mais completo possível. Desta forma, se o conjunto de testes inicial utilizado na aplicação da metodologia não executa chamadas a determinado método, este método provavelmente não aparecerá no modelo extraído do sistema e consequentemente nos novos testes gerados na aplicação da metodologia.

Na fase de instrumentação do código fonte da classe, o componente selecionado para o estudo precisa ser instrumentado por meio da ferramenta TXL, o que pode ser considerada uma limitação para aplicação da metodologia, já que, em alguns trechos do código dos componentes candidatos à aplicação da metodologia, não foi possível instrumentar o código, por restrições da gramática utilizada na instrumentação do código. Além disso, na instrumentação, em algumas classes, foi observado também que seriam necessárias algumas adaptações nas classes nas quais a metodologia foi aplicada, devido a restrições na instrumentação. Nas aplicações da metodologia foram necessárias realizar adaptações como essas em classes que continham situações de restrições da instrumentação da classe, tais como: chamadas de métodos no *return* do método, chamadas de métodos em estruturas como *for* e *if*.

Nas suítes de testes geradas automaticamente pela ferramenta EVOSUITE foi necessário realizar alterações devido a restrições da própria ferramenta EVOSUITE que, automaticamente, realiza a instrumentação dos testes gerados e ativa um "Java Agent", que intercepta qualquer carregamento de classe adicionando a sua própria instrumentação. Isso causa incompatibilidade da ferramenta EVOSUITE com as ferramentas que permitem a cobertura de testes, tais como: EclEmma, Cobertura, Clover e JMockit. Devido a esta restrição da ferramenta, foi necessário alterar o valor na propriedade *separateClassLoader* da suíte de testes de *true* para *false* para permitir a medição da cobertura pela ferramenta EclEmma. Isto permite a desativação da configuração do "Java Agent", conforme descrito na documentação da ferramenta. As possíveis alterações necessárias nos testes gerados pelas ferramentas de geração de testes automáticas não consistem em uma limitação ou desafio da metodologia em si, mas uma limitação da ferramenta utilizada nas aplicações da metodologia nos experimentos.

A criação de novos testes de forma automática é um desafio da metodologia a ser explorado em trabalhos futuros, pois a geração automática dos novos testes implica a automatização de todas as percepções que o usuário tem que avaliar na identificação se o contraexemplo pode gerar um caso de teste, tais como: valores de testes utilizados no caso de teste e análise da viabilidade de transformação do fluxo descrito no contraexemplo em caso de teste. Na metodologia descrita a análise e criação dos novos testes que é realizada de forma manual pelo usuário aplicado da metodologia, o que torna o processo de criação de novos testes passível de erro humano.

A seleção dos valores de testes utilizados nos novos testes é uma limitação da metodologia, já que o desenvolvedor deverá definir, de acordo com suas necessidades, o

método pelo qual definirá os valores de testes utilizados nos novos testes. O desenvolvedor pode utilizar diversos métodos para a escolha dos valores de testes utilizados nos casos de testes. Neste trabalho, para padronizar e melhor analisar as aplicações da metodologia, optou-se por utilizar ferramentas de geração de testes automática que geram suíte de testes com valores de testes.

Outra limitação da metodologia foi permitir a geração de testes em classes que contém métodos que precisam serem executados em uma ordenação específica e com determinados valores bem específicos. Nestes casos, foi necessário realizar nas aplicações uma melhor análise da classe e suas regras implementadas para tentar gerar testes válidos.

## 5.7 Discussões

Com base nos resultados apresentados nas aplicações da metodologia descritas nos experimentos, podemos dizer que a aplicação da metodologia mostrou resultados positivos na criação de testes em sistemas para os quais não se tem documentação e/ou testes legados, correspondendo ao principal objetivo ao qual a metodologia busca auxiliar, já que foi possível gerar testes na maioria das classes nas quais a metodologia foi aplicada

A metodologia propõe a geração de um conjunto inicial de testes sem especificar o meio pelo qual os testes são gerados, podendo ser manual, por meio de ferramentas, entre outros. Nas aplicações da metodologia descritas neste trabalho foram utilizadas ferramentas de geração automática de testes para a geração do conjunto inicial. O objetivo da utilização dessas ferramentas foi gerar testes para diferentes classes utilizando um mesmo padrão da ferramenta selecionada. Já a utilização de duas ferramentas que geram diferentes tipos de testes nos experimentos, possibilitaram a análise dos resultados da aplicação da metodologia em uma mesma classe utilizando diferentes conjunto de testes iniciais.

Em relação aos trabalhos analisados que influenciaram a criação desta metodologia, a metodologia apresentada mostra uma abordagem diferente no que se refere a possibilitar a criação de testes mesmo em sistemas que não tenham documentação e/ou testes legados. Conforme já dito, muitas vezes é um desafio para o desenvolvedor criar testes nesse cenário.

Considerando a quantidade de sistemas utilizados neste trabalho, a análise do conjunto de testes final obtido com a aplicação da metodologia pode produzir resultados

mais significativos com uma grande massa de dados. Percebeu-se que existem vários fatores que podem influenciar o conjunto de testes finais, tais como valores de testes e tipo de retorno do método. Diferentes valores de testes podem influenciar diretamente nos contextos executados do sistema, pois podem permitir a execução de novos caminhos.

Nas aplicações da metodologia em classes que não possuem testes legados e/ou documentação e é necessário a definição de valores de testes muito específicos para execução de testes unitários, a aplicação da metodologia utilizando o conjunto de testes inicial gerado por ferramentas de testes automática não foi proveitosa. Em aplicações da metodologia utilizando testes legados ou testes criados por desenvolvedores com conhecimento no código da aplicação a metodologia pode ser mais eficaz nesses casos.

# 6 CONCLUSÃO

O trabalho descrito nesta dissertação propõe uma metodologia que permite a geração de testes unitários em sistemas sem documentação e/ou testes legados, através da integração da técnica de testes e verificação de modelos, utilizando a extração de modelos. A metodologia contribui para a geração de um conjunto de testes mais completo na maioria das classes analisadas, tendo como base o modelo de comportamento extraído da execução do sistema, utilizando a técnica de teste de software e o processo de extração de modelos. Empiricamente, foi observado através dos resultados obtidos nos experimentos que a metodologia, em geral, possibilitou a geração de testes não aleatórios na maioria das aplicações, utilizando diferentes ferramentas de geração automática de testes para a geração do conjunto inicial de testes.

Em relação aos outros trabalhos analisados, as aplicações da metodologia realizadas neste trabalho se propõem à construção, de forma automática, do conjunto inicial de testes usado no processo de extração de modelo para simular a execução do sistema e assim extrair o modelo. Apesar disso, de forma geral, a metodologia pode ser aplicada com qualquer conjunto inicial de testes. A construção automática do conjunto inicial de testes foi definida na tentativa de reproduzir o caso do testador que não tem o total conhecimento das especificações do sistema.

Foi observado que a metodologia contribui para a análise e verificação da completude do conjunto de testes inicial. Assim, mesmo quando o conjunto inicial de testes é gerado por meio de uma ferramenta de geração automática de testes melhor ou por desenvolvedor que tenha conhecimento de técnicas de testes e conhecimento das regras do sistema, a metodologia pode contribuir para verificar a completude do conjunto inicial de testes.

A metodologia também contribui para a identificação de casos de teste não executáveis a partir de outros casos de testes já definidos como não executáveis. Esta identificação permite a melhoria da análise manual em relação a análise exaustiva como o trabalho proposto em Majerkowski (2012).

Nas aplicações da metodologia realizadas neste trabalho, utilizando a ferramenta JUnit-Tools e EVOSUITE para a geração do conjunto inicial de testes, foi observado que os resultados mais significativos da metodologia foram obtidos nas aplicações utilizando a ferramenta JUnit-Tools, em termos de aumento de cobertura e quantidade de novos testes gerados. Isto pode ser atribuído a simplicidade dos testes gerados por essa

ferramenta, que permitiram uma maior possibilidade de descobrir novos caminhos executados nos novos testes gerados.

Com base nos resultados obtidos nos experimentos foi possível observar também que, além da criação dos testes, os testes criados a partir da aplicação da metodologia também podem contribuir para o aumento de cobertura de linha e *branch* dos testes das classes.

A geração de novos testes na aplicação da metodologia realizada de forma automatizada é um tema a ser explorado em um trabalho futuro. Além disto, em trabalhos futuros seria possível explorar a realização de análise de forma mais eficiente dos contraexemplos que baseiam a geração dos testes, diminuindo o gasto de tempo da aplicação da metodologia.

Trabalhos futuros poderiam analisar a construção de conjunto de testes inicial mais eficaz, de forma a permitir a identificação dos valores de teste necessários para a aplicação e ordenação de chamadas de métodos necessários para a execução de testes. Também deveria ser feita a aplicação da metodologia em sistemas considerados complexos e analisar a aplicação da metodologia realizando experimento focado no usuário, de forma a avaliar as dificuldades e facilidades encontradas pelo usuário aplicador da metodologia.

As principais contribuições deste trabalho foram:

1. Geração de um conjunto de testes mais completo, tendo como base o modelo de comportamento extraído da execução do sistema, utilizando a técnica de teste de software e o processo de extração de modelos.
2. Identificação de casos de testes não executáveis a partir de outros casos de testes já definidos como não executáveis. Esta identificação permite a melhoria da análise manual em relação a análise exaustiva, como o trabalho proposto em Majerkowski (2012) .
3. Quando o conjunto inicial de testes é gerado por meio de uma ferramenta de geração automática de testes melhor ou por desenvolvedor que tenha conhecimento de técnicas de testes e conhecimento das regras do sistema a metodologia pode contribuir para verificar o quão completo é o conjunto de testes inicial.
4. Permitir a geração de testes unitários não aleatórios por desenvolvedores que não tem conhecimento profundo de técnicas de software e verificação de modelos. Esse é um dos principais objetivos da metodologia proposta e

inclusive foram utilizadas ferramentas de geração automática de testes na geração do conjunto inicial de testes para tornar a aplicação da metodologia possível mesmo em casos de pessoas sem conhecimento profundo de técnica de teste de software e verificação de modelos.

# REFERÊNCIAS

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. Cambridge: Cambridge University Press, 2008.

ARTS, T. *et al.* Testing telecoms software with quviq QuickCheck. In: **Erlang'06 - Proceedings of the ACM SIGPLAN 2006 Erlang Workshop**, Sept. 2006, p. 2–10, 2006.

BAIER, C.; KATOEN, J.-P. **Principles of model checking**. Cambridge: MIT Press, 2008.

BEYER, D. et al. Genereting tests from counterexamples. In: **Proceedings. 26th International Conference on Software Engineering**, Edinburgh, May, p. 326-335, 2004.

BEYER, D.; LEMBERGER, T. Software Verification: Testing vs. Model Checking A Comparative Evaluation of the State of the Art. **Hardware and Software: Verification and Testing**, *[S. l.]*, p. 99–114, 2017.

BURR, K.; YOUNG, W. Combinatorial test techniques: Table-based automation, test generation and code coverage. In: **Proc. of the Intl. Conf. on Software Testing Analysis & Review**. 1998.

CAMPOS, J.; PANICHELLA, A.; FRASER, G. EvoSuite at the SBST 2019 tool competition. **2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST).** IEEE, 2019. p. 29-32.

CHEN, J.; ZHOU, H.; BRUDA, S. D. Combining model checking and testing for software analysis. **2008 International Conference on Computer Science and Software Engineering**. IEEE, 2008. p. 206-209.

CLARKE, E. M.; GRUMBERG, O.; KROENING, D.; PELED, D. A.; VEITH, H. **Model Checking.** Cambridge: The MIT Press, 2018.

CORDY, J. R. et al. Source transformation in software engineering using the TXL transformation system. **Information and Software Technology**, [S. l.], v. 44, n. 13, p. 827–837, Oct. 2002.

DUARTE, L. Integrating Software Testing and Model Checking Through Model Extraction. In: **14th Brazilian Symposium on Formal Methods**: short papers, São Paulo, p. 73–78, 2011.

DUARTE, L. M. **Behaviour Model Extraction using Context Information**. 2007. 245 f. Tese (Doutorado em Computing) - Imperial College London, University of London, Londres, 2007.

DUARTE, L. M.; KRAMER, J.; UCHITEL, S. Using contexts to extract models from code. **Software and Systems Modeling**, [S. l.], v. 16, n. 2, p. 523–557, May. 2017.

FRASER, G. A tutorial on using and extending the EvoSuite search-based test generator. In: **International Symposium on Search Based Software Engineering**. Springer, Cham, Alemanha 2018. p. 106-130.

GARGANTINI, A. Using Model Checking to Generate Fault Detecting Tests. In: MEYER B.; GUREVICH, Y. (Ed.). **Tests and Proofs**. Berlin, Heidelberg: Springer, Berlin, Heidelberg, 2007. p. 189–206.

GROZ, R. et al. Modular System Verification by Inference, Testing and Reachability Analysis. In: SUZUKI, K. et al. (eds) **Testing of Software and Communicating Systems**. Berlin, Heidelberg: Springer, Berlin, Heidelberg, 2007. p. 216–233.

HOLZMANN, G. J.; SMITH, M. H. Practical method for verifying event-driven software. *In*: 1999, New York, New York, USA. **Proceedings - International Conference on Software Engineering**. New York: ACM Press, 1999. p. 597–607.

JIANGUO, C.; HANGXIA, Z.; BRUDA, S. D. Combining model checking and testing for software analysis. **Proceedings - International Conference on Computer Science and Software Engineering, CSSE 2008**, Hubei, v. 2, p. 206–209, 2008.

KELLER, R. M. Formal Verification of Parallel Programs. **Communications of the ACM**, [S. l.], v. 19, n. 7, p. 371–384, July 1976.

LAMELA SEIJAS, P.; THOMPSON, S.; FRANCISCO, M. Á. Model extraction and test generation from JUnit test suites. **Software Quality Journal**, [S. l.], v. 26, n. 4, p. 1519–1552, 2018.

LANGR, J.; HUNT, A.; THOMAS, D. **Pragmatic Unit Testing in Java 8 with JUnit**. CIDADEDAEDITORA: Pragmatic Bookshelf, 2015.

MAGEE, J.; KRAMER, J. Concurrency: State Models and Java Programs. **Concurrency**: State Models Java Programs, CIDADEDAEDITORA: Wiley, 2006.

MAJERKOWSKI FILHO, L. F. N. **Integração entre verificação de modelos e teste de software para melhoria da detecção de erros em sistemas computacionais.** 2012.73 f. Monografia (Curso de Ciências da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2012.

MYERS, G.J.; BADGETT, T.; SANDLER, C. **The Art of Software Testing**. Chichester: John Wiley & Sons, 2012.

TILLMANN, N.; DE HALLEUX, J.; XIE, T. Parameterized unit testing: Theory and practice. **2010 ACM/IEEE 32nd International Conference on Software Engineering. IEEE**, 2010. p. 483-484.

UCHITEL, S.; KRAMER, J.; MAGEE, J. Behaviour model elaboration using partial labelled transition systems. *In*: 2003, New York, New York, USA. IN: **Proceedings of the 9th European software engineering conference held jointly with 10th ACM**

**SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE '03**. New York, New York, USA: ACM Press, 2003. p. 19.

VILLANI, E. et al. Integrating model checking and model based testing for industrial software development. **Computers in Industry**, *[S. l.]*, v. 104, p. 88–102, 2019. 19.

WALKINSHAW, N.; DERRICK, J.; GUO, Q. Iterative refinement of reverse-engineered models by model-based testing. In: CAVALCANTI, A.; DAMS, D. (Eds.). **FM 2009**: Formal Methods. Lecture Notes in Computer Science, vol 5850. Springer, Berlin, Heidelberg, 2009. p. 305–320.