UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Geração de testes a partir de máquinas de estados finitos estendidas extraídas de diagramas de sequência UML

Mauricio Rêgo Mota da Rocha

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP SÃO CARLOS

Mauricio Rêgo Mota da Rocha

Geração de testes a partir de máquinas de estados finitos estendidas extraídas de diagramas de sequência UML

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Adenilso da Silva Simão
Coorientador: Prof. Dr. Thiago Carvalho de Sousa

USP – São Carlos
Fevereiro de 2021

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

R672g

Rocha, Mauricio Rêgo Mota da
Geração de testes a partir de máquinas de estados finitos estendidas extraídas de diagramas de sequência UML / Mauricio Rêgo Mota da Rocha; orientador Adenilso da Silva Simão; coorientador Thiago Carvalho de Sousa. -- São Carlos, 2021.
141 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2021.

1. Teste Baseado em Modelo. 2. MDA. 3. Diagrama de Sequência. 4. Máquina de Estados Finitos Estendida. 5. ModelJUnit e JUnit. I. Simão, Adenilso da Silva, orient. II. Sousa, Thiago Carvalho de, coorient. III. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Mauricio Rêgo Mota da Rocha

Test generation by extended finite state machines extracted from UML sequence diagrams

Thesis submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Doctor in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Adenilso da Silva Simão
Co-advisor: Prof. Dr. Thiago Carvalho de Sousa

USP – São Carlos
February 2021

À Deus e a Nossa Senhora Aparecida.
Aos meus pais, Abimael e Amparo.
À minha esposa, Candice.
Às minhas filhas, Nina e Alice.

AGRADECIMENTOS

Agradeço em primeiro lugar à Deus, por sua imensa bondade e a Nossa Senhora Aparecida por ter ouvido minhas preces para conclusão dessa etapa da minha vida. Sem ajuda Divina nada seria possível.

A toda minha família, em especial aos meus pais, Abimael e Amparo, que não mediram esforços para me fornecer a melhor formação e pelo apoio em todos os momentos da minha vida.

À minha querida esposa Candice, pelo amor, compreensão e incentivo nos momentos mais difíceis desta jornada. Sua parceria foi fundamental para conclusão deste trabalho.

Às minhas queridas filhas Nina e Alice, pelo carinho e por compreenderem desde pequenas que o estudo é uma das coisas mais importantes da vida.

Ao meu orientador, professor Adenilso da Silva Simão, pela oportunidade, confiança e paciência na orientação deste trabalho. Seus conselhos e ensinamentos levarei por toda minha vida acadêmica. Muito obrigado por tudo!

Ao meu coorientador e amigo, professor Thiago Carvalho de Sousa, pela disponibilidade e ajuda em todas as etapas da realização deste trabalho. Serei sempre grato a você por tudo que fez por mim durante esta jornada.

À professora e colega da UESPI, Melissa Oda Souza, pela disponibilidade e ajuda na avaliação estatística do estudo experimental deste trabalho.

À servidora do Centro Cultural da USP, Ângela Cristina Pregnolato Giampedro, pelas excelentes revisões da língua inglesa nos artigos submetidos aos periódicos e conferências.

Ao primo Mateus e ao amigo Felipe, por sempre me receberem tão bem em São Carlos. A ajuda de vocês foi muito importante.

A todos os colegas do DINTER UESPI/USP, pela amizade e incentivo durante a realização do doutorado, em especial, Alessandro Wilk, Átila Lopes, Dario Calçada, Erasmo Artur, Lianna Duarte, Marcus Vinícius de Carvalho e Maria José Machado.

A todos os colegas do LABES/ICMC, pelo auxílio e troca de experiências, em especial, Carlos Diego Damasceno, Henrique Araújo, João Paulo Biazzoto, Jorge Cutigi, Kamila Lyra, Luiz Rodrigues, Misael Costa, Sidgley Andrade, Sofia Paiva, Stevão Andrade e Wilk Oliveira.

Aos professores do ICMC/USP, pela dedicação e ensinamentos em todas as disciplinas do doutorado. A todos os funcionários do ICMC/USP, pelo apoio durante toda a realização do curso.

À Universidade Estadual do Piauí (UESPI) pela oportunidade e apoio financeiro.

“Talvez não tenha conseguido fazer o melhor, mas lutei para que o melhor fosse feito. Não sou o que deveria ser, mas Graças a Deus, não sou o que era antes.”
(Marthin Luther King)

RESUMO

ROCHA, M. R. M. **Geração de testes a partir de máquinas de estados finitos estendidas extraídas de diagramas de sequência UML**. 2021. 141 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

A eficiência e eficácia do Teste Baseado em Modelo (TBM) se deve principalmente ao seu potencial para automação. Se o modelo for formal e baseado em Máquinas de Transição de Estados, os casos de teste podem ser derivados automaticamente. Uma das técnicas de modelagem formal mais utilizada é a interpretação de um sistema como uma Máquina de Estados Finitos Estendida (MEFE), porém, não é uma prática comum na indústria a utilização de modelos formais, já que os desenvolvedores tem dificuldades de lidar com os conceitos envolvidos. Por outro lado, a *Unified Modeling Language* (UML) tornou-se o padrão de fato para modelagem de software. No entanto, devido à falta de semântica formal da UML, seus diagramas podem gerar inconsistências e receber interpretações ambíguas. Dessa forma, os modelos UML não são adequados para automação de teste. Neste contexto, esta tese de doutorado apresenta um procedimento sistemático nomeado de *TestSd2Efsm* para geração de um ambiente de testes a partir de modelos UML que usa conceitos da MDA (*Model Driven Architecture*) para formalizar Diagramas de Sequência UML em Máquinas de Estados Finitos Estendidas e fornecer uma semântica precisa para esse modelo UML. O procedimento sistemático também aplica a biblioteca ModelJUnit para geração de casos de teste abstratos e um conjunto de métricas para avaliá-los. Além disso, utiliza o *framework* JUnit para concretizar os casos de teste na linguagem de programação Java. Foi realizado um estudo experimental para avaliar a eficiência, eficácia e qualidade do procedimento sistemático *TestSd2Efsm*. Os resultados do experimento mostraram que o custo medido em tempo para geração de um ambiente de teste utilizando o procedimento sistemático *TestSd2Efsm* é menor que o esforço utilizando a abordagem manual e os casos de teste gerados pelo procedimento sistemático *TestSd2Efsm* são mais eficazes que os gerados pela abordagem manual. Além disso, a percepção dos participantes do experimento é que a qualidade da geração do ambiente de teste pelo procedimento sistemático *TestSd2Efsm* é maior que a qualidade utilizando a abordagem manual.

Palavras-chave: Teste Baseado em Modelo, MDA, Diagrama de Sequência, Máquina de Estados Finitos Estendida, ModelJUnit, JUnit.

ABSTRACT

ROCHA, M. R. M. **Test generation by extended finite state machines extracted from UML sequence diagrams**. 2021. 141 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

The efficiency and effectiveness of the Model-Based Testing (MBT) derives mainly from its potential for automation. If the model is formal and based on State Transition Machines, test cases can be derived automatically. One of the widely used formal modeling techniques is the interpretation of a system as an Extended Finite State Machine (EFSM), however, it is not a common practice in the industry to use formal models, since developers have difficulties in dealing with the concepts involved. On the other hand, Unified Modeling Language (UML) has become the "de facto" standard for software modeling. Nevertheless, due to the lack of formal UML semantics, its diagrams can generate inconsistencies and receive ambiguous interpretations. For that reason, UML models are not suitable for test automation. In this context, this thesis presents a systematic procedure named *TestSd2Efsm* for generating a test environment from UML models that uses MDA (Model Driven Architecture) concepts to formalize UML Sequence Diagrams in Extended Finite State Machines and provide precise semantics for this UML model. The systematic procedure also applies the ModelJUnit library for generating abstract test cases and a set of metrics to evaluate them. In addition, it uses the JUnit framework to realize the test cases in the Java programming language. An experimental study was carried out to evaluate the efficiency, the effectiveness and the quality of the systematic procedure *TestSd2Efsm*. The results of the experiment have demonstrated that the cost measured in time for generating a test environment using the systematic procedure *TestSd2Efsm* is less than the effort using the manual approach and the test cases generated by the systematic procedure *TestSd2Efsm* are more effective than those generated by the manual approach. In addition, the perception of the experiment participants is that the quality of the generation of the test environment by the systematic procedure *TestSd2Efsm* is higher than the quality using the manual approach.

Keywords: Model-Based Testing, MDA, Sequence Diagram, Extended Finite State Machine, ModelJUnit, JUnit.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS-FONTE

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACM	Association for Computing Machinery
AMMA	*ATLAS Model Management Architecture*
ATL	*Atlas Transformation Language*
CEP	Comitê de Ética em Pesquisa
CNS	Conselho Nacional de Saúde
CONEP	Comissão Nacional de Ética em Pesquisa
DFS	*Depth First Search*
EMF	*Eclipse Modelling Framework*
ES	Engenharia de Software
GMF	*Graphical Modeling Framework*
LDAP	*Lightweight Directory Access Protocol*
LTS	*Labeled Transition Systems*
MDA	*Model Driven Architecture*
MDE	*Model Driven Engineering*
MEF	Máquina de Estados Finitos
MEFE	Máquina de Estados Finitos Estendida
MOF	*Meta Object Facility*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
P&D	Pesquisa e Desenvolvimento
PIM	*Platform Independent Model*
PSM	*Platform Specific Model*
QVT	*Query/View/Transformation*
SUT	*System Under Test*
TBM	Teste Baseado em Modelo
UBS	Unidade Básica de Saúde
UESPI	Universidade Estadual do Piauí
UFPI	Universidade Federal do Piauí
UML	*Unified Modeling Language*

SUMÁRIO

CAPÍTULO

1

INTRODUÇÃO

1.1 Contextualização

A Engenharia de Software (ES) visa disciplinar o desenvolvimento de software a fim de torná-lo viável economicamente. De acordo com Pressman (2006), o principal objetivo da ES é prover métodos, ferramentas e procedimentos que possibilitem o gerenciamento do processo de desenvolvimento de software e ofereça uma base para a construção de software de alta qualidade com grande produtividade.

A ES oferece diversas maneiras para se desenvolver software visando a qualidade e produtividade, em especial por meio de teste de software e modelagem formal. A atividade de teste de software é considerada um elemento crítico da garantia de qualidade de software e representa a última revisão de especificação, projeto e codificação (PRESSMAN, 2006). Por outro lado, a modelagem formal é usada nos estágios iniciais do processo de desenvolvimento para evitar ambiguidades nas especificações e minimizar falhas (SOMMERVILLE, 2007). Essas duas atividades se complementam, o que faz com que seu uso combinado melhore a qualidade do software em construção (SOUZA, 2010).

Uma prática comum na maioria dos processos de desenvolvimento de software é o uso de modelos abstratos, que representam as partes essenciais de um sistema e permitem que os engenheiros de software tenham uma visão conceitual de várias perspectivas diferentes do software (MATOS, 2012). A *Unified Modeling Language* (UML), uma alternativa amplamente utilizada para modelagem de software, permite a especificação de aspectos estáticos e estruturais, além de dinâmicos ou comportamentais (OMG, 2017). Segundo (PETRE, 2013), os diagramas UML mais utilizados são o Diagrama de Classes e o Diagrama de Sequência. O primeiro é utilizado para modelar aspectos estruturais e o segundo aspectos comportamentais.

Uma das formas principais de se representar os aspectos comportamentais é por meio de cenários, que são amplamente usados no desenvolvimento de software. Cenários de uso típicos,

comportamentos proibidos, casos de teste e muitos outros aspectos podem ser descritos com cenários gráficos (MICSKEI; WAESELYNCK, 2010). Na UML, os cenários são modelados com interações (OMG, 2017). O Diagrama de Sequência é a variação mais comum dos diagrama de interação da UML (OMG, 2017). A partir da versão 2.0 da UML, novos elementos foram adicionados ao diagrama, aumentando sua expressividade e consequentemente sua complexidade. Devido a tal incremento, tornou-se uma tarefa difícil prover uma única interpretação a um Diagrama de Sequência, sendo necessário se ter uma semântica formal precisa (MICSKEI; WAESELYNCK, 2010).

No contexto de teste de software, a criação de modelos pode aumentar a produtividade dessa atividade. De acordo com Utting, Pretschner e Legeard (2012), o Teste Baseado em Modelo (TBM) permite que testes sejam gerados automaticamente a partir de modelos e de outros artefatos de software, tornando possível a criação de testes para o software antes mesmo da sua codificação, reduzido assim o custo de desenvolvimento. A ideia central de TBM é, a partir de um modelo ou especificação, gerar sequências de entrada e suas saídas esperadas. As sequências de entrada são então aplicadas ao software em teste e as saídas do software são comparadas com as saídas do modelo. Isto implica que o modelo deve ser válido, isto é, representar fielmente os requisitos. Portanto, conforme Tretmans (2008), TBM é uma extensão dos métodos de verificação formal e eles têm objetivos complementares. A verificação formal tem a intenção de mostrar que o software tem algumas propriedades desejadas, provando que o modelo do software satisfaz estas propriedades. Já o TBM, começa com um modelo validado e, em seguida, tem a intenção de mostrar que uma implementação real do software se comporta em conformidade com o modelo.

Em TBM, recomenda-se a utilização de modelos formais, já que podem ser usadas como base para automatização do processo de testes, tornando-o mais eficiente e efetivo (HIERONS *et al.*, 2009). Outra vantagem dos modelos formais é que diminuem as ambiguidades que a linguagem natural pode gerar. Existem diversas técnicas de modelagem formal baseadas em Máquinas de Transição de Estados que podem ser utilizadas para especificar um modelo de teste. Essas técnicas diferenciam-se pelas características relativas ao modo como seus elementos são representados (SIMÃO, 2016). O modelo mais simples é a Máquina de Estados Finitos (MEF) (GILL, 1962). As MEFs tem sido amplamente utilizadas devido a sua simplicidade e capacidade para modelar alguns tipos de sistemas, tais como: protocolos, sistemas reativos e sistemas embarcados. Muitos métodos clássicos de geração de testes a partir de MEFs foram propostos ao longo dos anos, e muitos deles garantem que todas as falhas em um determinado domínio sejam cobertas, tais como: DS (GONENC, 1970), W (CHOW, 1978), Wp (FUJIWARA *et al.*, 1991), HSI (PETRENKO *et al.*, 1994; LUO; PETRENKO; BOCHMANN, 1995), H (DOROFEEVA R.; YEVTUSHENKO, 2005), SPY (SIMÃO; PETRENKO; YEVTUSHENKO, 2009) e P (SIMÃO; PETRENKO, 2010).

No entanto, para sistemas mais complexos que possuem muitas possíveis configurações

distintas e que necessitam modelar tanto fluxo de controle como também fluxo de dados, a utilização de uma MEF para representar esses sistemas não é adequada. Portanto, isto inspirou pesquisadores (LEE; YANNAKAKIS, 1996) a projetar um modelo estendido chamado de Máquina de Estados Finitos Estendida (MEFE). As MEFEs têm sido amplamente utilizadas pela comunidade de métodos formais, pois devido a sua expresividade, elas permitem a representação do fluxo de controle (ordem dos eventos de entrada e saída) e de dados (parâmetros dos eventos e variáveis de contexto) de sistemas complexos.

Os métodos de geração de casos de teste a partir de MEFEs seguem o mesmo roteiro de métodos de outros modelos formais, tais como as MEFs. Em uma visão de alto nível, esse roteiro leva em consideração dois aspectos: criação de um modelo do software em teste e condução do modelo para geração de casos de teste (YANG *et al.*, 2015). No entanto, os métodos de geração de testes a partir de MEFs não podem ser utilizados diretamente em MEFEs, pois eles são adequados apenas para testar a parte de controle das MEFEs (HIERONS *et al.*, 2009).

1.2 Definição do Problema e Justificativa para Pesquisa

UML é a linguagem de modelagem de software mais difundida na indústria. Cada elemento da UML possui uma sintaxe e uma semântica. De acordo com Bezerra (2007), a sintaxe corresponde à forma predefinida de desenhar o elemento e a semântica define o seu significado e com que objetivo ele deve ser utilizado. Porém, devido à falta de uma semântica formal, existem alguns problemas na utilização da UML, tais como: complexidade, inconsistência, problemas de sincronização entre diagramas e diferentes interpretações (PETRE, 2013). Segundo Lucas, Molina e Toval (2009), isso ocorre principalmente devido a existência de múltiplas visões (modelos) para o mesmo sistema, que podem potencialmente conter especificações contraditórias. Portanto, embora a UML seja a linguagem de modelagem mais popular, seus diagramas podem gerar inconsistências e interpretações diferentes.

Diversas definições de consistência de modelos e suas classificações são encontradas na literatura (ENGELS; KÜUSTER; GROENWEGEN, 2002; HUZAR *et al.*, 2005). Engels, Küuster e Groenwegen (2002) afirmam que diferentes submodelos são chamados consistentes se eles podem ser integrados em um único modelo com uma semântica adequada. Uma forma de amenizar o problema da natureza multivisão da UML e consequentemente da inconsistência entre seus diagramas seria a utilização de um único modelo UML para especificar o comportamento de funcionalidades dos sistemas. Neste contexto, o Diagrama de Sequência é o mais indicado, já que mesmo sendo um modelo comportamental, facilmente é possível extrair dos seus elementos aspectos estruturais do software.

Por outro lado, uma maneira de minimizar o problema da ausência de formalismo dos modelos UML é a utilização de modelos formais, já que eles têm uma semântica precisa para representar o comportamento de um sistema. No contexto de testes de software, os métodos de

geração baseados em modelos formais, tais como as MEFEs, podem ser tomados como referência, uma vez que são métodos bem estabelecidos (BOURHFIR; DSSOULI; ABOULHAMID, 1996; YANO; MARTINS; SOUSA, 2011; YANG *et al.*, 2015). Além disso, elas podem ser implementadas como um modelo de teste usando a biblioteca ModelJUnit (MODELJUNIT, 2010), projetada como uma extensão do JUnit (JUNIT, 2020) e escrita em Java, que é uma linguagem de programação muito popular. No entanto, na prática, os modelos formais raramente são usados na indústria, provavelmente devido à falta de treinamento e familiaridade dos desenvolvedores com a notação matemática.

Para contornar os problemas semânticos da UML e a baixa adesão dos modelos formais na indústria, pode-se utilizar mecanismos para prover semântica e refinamento formais para UML, através da transformação automática para modelos formais. Esses mecanismos de transformação de modelos fazem parte do contexto da *Model Driven Engineering* (MDE) (SCHMIDT, 2006), especialmente da proposta da OMG (OMG, 2014a), conhecida como *Model Driven Architecture* (MDA). No centro da MDE está a noção de que os modelos devem servir como artefatos principais no desenvolvimento de software, em vez de serem meramente recursos de suporte opcionais para análise e documentação (CICCOZZI; MALAVOLTA; SELIC, 2018). Além disso, a MDE pode promover a automatização durante o processo de desenvolvimento, facilitando a construção de ferramentas.

No contexto de teste baseados em modelos formais, a ideia é criar modelos de testes e através da simulação de execução de suas operações gerar casos de teste (ZANDER; SCHIEFERDECKER; MOSTERMAN, 2011). As sequências obtidas representam casos de teste abstratos. No entanto, esses casos de teste abstratos precisam ser concretizados para serem executados no software em teste. Um problema encontrado é que muitas abordagens que utilizam modelos formais não apresentam a concretização dos casos de teste em uma linguagem de programação. Normalmente, esses estudos apresentam abordagens de geração de casos de teste mais teóricas do que práticas. Portanto, é importante que os métodos de geração se preocupem com a geração de casos de teste concretos.

1.3 Objetivos do Trabalho

A UML é a mais popular linguagem de modelagem, mas devido à ausência de uma semântica formal seus diagramas podem gerar inconsistências e várias interpretações. Os modelos formais podem ajudar com esses problemas. No entanto, não é prática comum na indústria a utilização de modelos formais, e seus métodos de geração de testes normalmente se preocupam apenas com a geração de casos de teste abstratos. Mesmo com o crescente interesse da comunidade acadêmica na investigação dos tópicos de transformação de modelos UML em modelos formais e da concretização dos casos de teste gerados a partir de modelos formais em alguma linguagem de programação, mais contribuições podem ser alcançadas se esses processos forem

automatizados e unificados.

Neste contexto, esta tese investiga a seguinte questão de pesquisa: *É possível sistematizar e automatizar a geração de casos de teste abstratos e concretos a partir de modelos formais extraídos de diagramas UML?* Com base nesta questão de pesquisa, os seguintes objetivos específicos nortearam a execução da pesquisa realizada:

- Definir o meta-modelo simplificado do Diagrama de Sequência UML e o meta-modelo da MEFE.
- Definir as regras de transformação para mapeamento dos elementos do Diagrama de Sequência UML para Máquina de Estados Finitos Estendida.
- Formalizar o Diagrama de Sequência UML em termos de MEFEs, que são modelos semanticamente mais precisos.
- Definir um procedimento sistemático de geração de testes a partir de Diagrama de Sequência UML, com a concretização dos casos de teste na linguagem de programação Java.
- Implementar uma ferramenta para automatização de todas as etapas do procedimento sistemático de geração de testes.
- Executar o procedimento sistemático proposto em software real para verificar sua aplicabilidade.
- Realizar estudos empíricos para avaliar o procedimento sistemático de geração de testes proposto.

1.4 Sumário dos Resultados Obtidos

A principal contribuição deste trabalho é a definição do procedimento sistemático *TestSd2Efsm* para geração e execução de testes abstratos e concretos na linguagem de programação Java. Entretanto, existem outras contribuições, descritas a seguir:

- Definição de regras de transformação para mapeamento dos elementos do meta-modelo do Diagrama de Sequência UML para o meta-modelo da Máquina de Estados Finitos Estendida. Essas regras foram implementadas em *Atlas Transformation Language* (ATL) (BÉZIVIN; JOUAULT; TOUZET, 2005). Com a definição dessas regras, os Diagramas de Sequência são formalizados em MEFEs.
- Geração automática de código-fonte de classes do SUT (*System Under Test*) a partir dos Diagramas de Sequência UML utilizando a linguagem de transformação Acceleo (ACCELEO, 2018). Estas classes são usadas para execução dos testes abstratos e concretos gerados pelo procedimento sistemático.

- Geração automática de código-fonte dos testes abstratos na biblioteca ModelJUnit e os testes concretos na biblioteca JUnit.

- Criação de um protótipo que contemple todas as etapas do procedimento sistemático *TestSd2Efsm* para automatizar todo o processo de teste. O protótipo foi implementado utilizando o *Eclipse Modelling Framework* (EMF) (STEINBERG *et al.*, 2009).

- Execução do procedimento sistemático de geração de testes em um software real. A execução mostrou a aplicabilidade do procedimento proposto.

- Avaliação empírica realizada seguindo as orientações do processo experimental de Wohlin *et al.* (2012), na qual constatou-se que o procedimento sistemático *TestSd2Efsm* proposto reduz os custos de criação de ambiente de testes de software, melhorando a eficácia dos testes gerados e elevando a qualidade do processo de teste de software.

1.5 Organização da Tese

Além dessa introdução, em que foi apresentada a contextualização, definição do problema e justificativa, objetivos e sumário dos resultados obtidos, esta tese de doutorado está organizada em mais quatro capítulos.

O Capítulo 2 apresenta uma visão geral da fundamentação teórica, descrevendo os principais conceitos investigados no trabalho. Primeiramente, conceitos dos Diagramas de Sequência UML, explanando a sintaxe e semântica dos seus principais elementos. Em seguida, apresenta-se a transformação entre modelos, com seus conceitos chaves e o *framework* de desenvolvimento de software *Model Driven Architecture* (MDA). O capítulo também apresenta o Teste Baseado em Modelo, com uma visão geral dos métodos de geração de teste baseados em MEFs e MEFEs.

O Capítulo 3 apresenta o procedimento sistemático de geração de casos de teste a partir do Diagrama de Sequência UML com o seu conjunto de passos bem definidos. Neste capítulo são descritos os meta-modelos propostos para o Diagrama de Sequência e para a MEFE. Também são apresentadas as regras de transformação definidas para mapeamento dos elementos de cada meta-modelo e é realizada uma discussão de como a complexidade dos Diagramas de Sequência influenciam no tamanho da MEFE gerada. Além disso, são apresentadas as transformações Modelo-Texto (M2T) para geração do código-fonte do SUT e da geração dos casos de teste. Por fim, é apresentado um exemplo de aplicabilidade do procedimento sistemático *TestSd2Efsm*.

O Capítulo 4 apresenta o estudo empírico realizado para avaliação e validação do procedimento sistemático *TestSd2Efsm* proposto. O capítulo mostra o planejamento e condução de um estudo experimental para avaliar a eficiência, eficácia e qualidade do procedimento sistemático de geração de casos de teste proposto.

Por fim, no Capítulo 5 são descritas as conclusões do trabalho, destacando as principais contribuições, os trabalhos relacionados e as publicações resultantes. Além disso, são apresentadas as limitações e possíveis caminhos de investigações para trabalhos futuros.

CAPÍTULO

2

FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos abordados na pesquisa e que são necessários para o entendimento do trabalho. A Seção 2.1 discute o Diagrama de Sequência UML que é o ponto de partida do procedimento sistemático de geração de teste apresentado nesta tese de doutorado. A Seção 2.2, apresenta a transformação entre modelos que é um dos conceitos chaves da MDA. Por fim, na Seção 2.3 é apresentada uma visão geral do processo de Teste Baseado em Modelo (TBM), com destaque aos métodos de geração de testes baseados em MEFs e MEFEs.

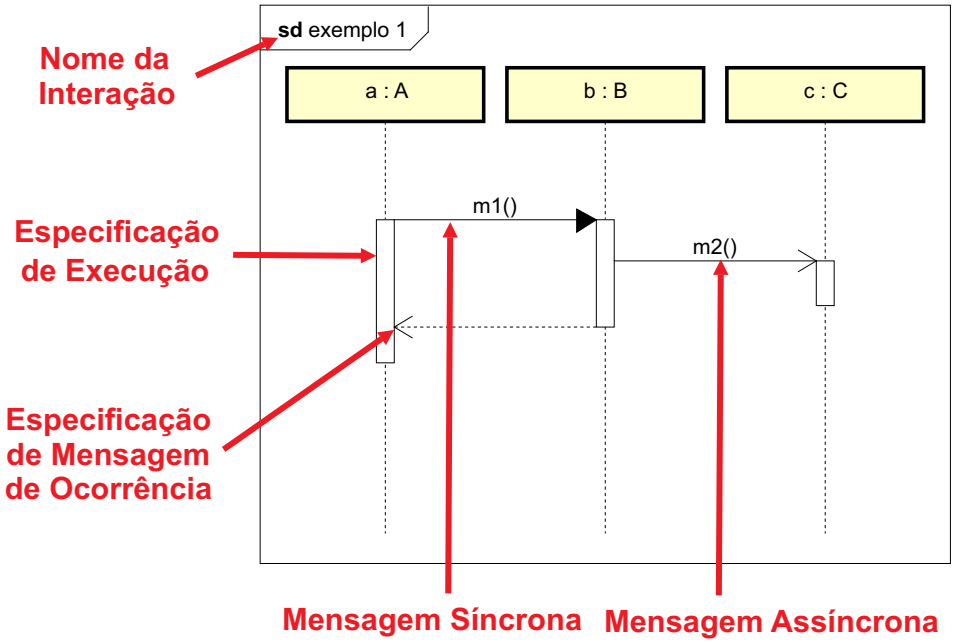
2.1 Diagrama de Sequência UML

A UML é uma linguagem visual que utiliza elementos gráficos que fornecem aos arquitetos, engenheiros e desenvolvedores de software, ferramentas para análise, projeto e implementação de sistemas de software, bem como para modelar processos de negócios (OMG, 2017). Através desses elementos gráficos pode-se construir diagramas que representam diversas perspectivas de um software. O aspecto comportamental dinâmico de um software orientado a objetos é definido através da interação entre os objetos e troca de mensagens entre eles (BEZERRA, 2007). Na UML essa perpectiva é tratata pelos diagramas de interação. A *Object Management Group* (OMG) [1] fornece os seguintes diagramas de interação: Diagrama de Sequência, Diagrama de Comunicação, Diagrama de Visão Geral da Interação e Diagrama de Tempo. O Diagrama de Sequência é a variação mais comum dos diagramas de interação da UML (OMG, 2017). Ele apresenta as interações entre objetos na ordem temporal em que elas acontecem (BEZERRA, 2007).

Para facilitar o entendimento da sintaxe do Diagrama de Sequência UML, os elementos das interações e suas notações podem ser divididas em dois tipos: interação básica e fragmento combinado (MICSKEI; WAESELYNCK, 2010). A Figura 1 apresenta os elementos de uma

[1] Consórcio internacional de empresas que define e ratifica padrões na área de orientação a objetos.

Figura 1 – Elementos de uma interação básica.



Fonte: Adaptada de Micskei e Waeselynck (2010).

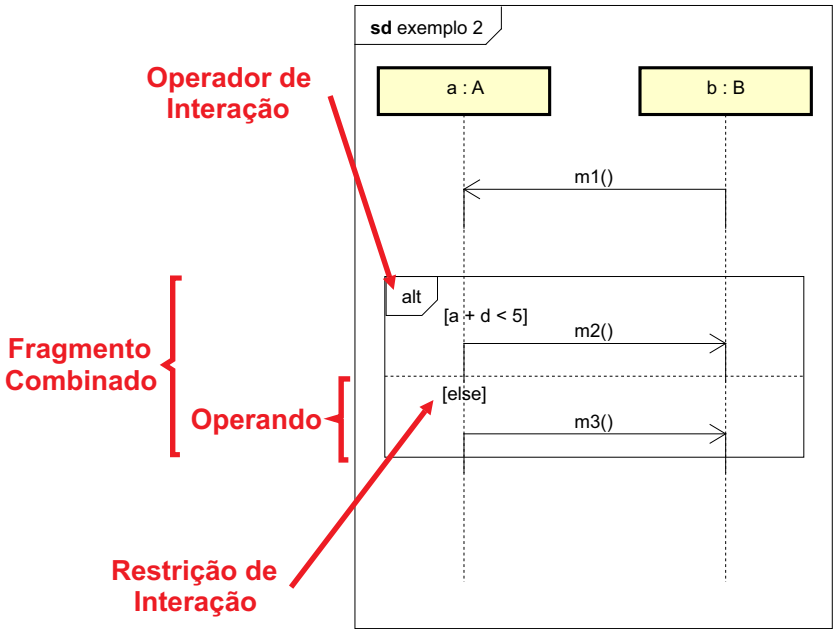
interação básica. Linhas de vida representam os participantes da interação que se comunicam através de mensagens. Essas mensagens podem ser síncronas ou assíncronas e podem corresponder a uma chamada de operação, envio de sinal ou mensagem de retorno. Se a assinatura da mensagem é uma operação, a operação é executada de acordo com os argumentos que correspondem aos seus parâmetros. As mensagens de retorno representam o retorno da chamada de operações síncronas. Se a assinatura da mensagem for um sinal, então a mensagem representa o envio assíncrono de uma instância do sinal. A especificação de execução é uma unidade de comportamento ou ação dentro de uma linha de vida e representa o tempo em que um objeto está ativo, ou seja, o momento em que ele realiza alguma operação. O envio e recebimento de mensagens são marcados com a especificação de ocorrência de mensagem.

A partir da versão 2.0 da UML, novos elementos foram adicionados ao Diagrama de Sequência, tornando-o mais expressivo e complexo. Estas interações mais complexas podem ser criadas através dos fragmentos combinados, que definem o fluxo de controle dos cenários modelados. Os fragmentos combinados são compostos de um ou mais operandos, zero ou mais restrições de interação e um operador de interação. Um operando corresponde a uma sequência de mensagens executadas apenas em circunstâncias específicas. As restrições de interação também são conhecidas como condições de guarda e representam uma expressão condicional. A Figura 2 apresenta os principais elementos desta construção.

A UML 2 define diversos operadores de interação, os três apresentados a seguir modelam as principais construções procedurais:

- ***alt***: construção do tipo *if-then-else*. Somente um operando é executado. Se nenhum dos

Figura 2 – Interações com fragmento combinado.



Fonte: Adaptada de Micskei e Waeselynck (2010).

operandos tiver uma condição de guarda avaliada como verdadeira, nenhum deles será executado;

- ***opt***: construção do tipo *if-then*. É muito semelhante ao operador *alt*, a principal diferença é que apenas um operando é definido, que pode ou não ser executado. Ele é semanticamente equivalente a um fragmento combinado alternativo;

- ***loop***: uma construção que representa um *loop*, em que o único operando é executado zero ou mais vezes. Quando a condição de guarda deste operador for avaliada como falsa, o *loop* é encerrado.

Outros operadores de interação definidos na UML 2 pela OMG (OMG, 2017) são descritos a seguir:

- ***seq***: operador que permite a alteração da ordem da sequência quando a especificação de ocorrência é executada em diferentes linhas de vida. Portanto, quando as mensagens não envolvem a mesma linha de vida, elas podem ser intercaladas em paralelo.

- ***break***: operador que representa um cenário de interrupção no fragmento de interação.Quando a condição de guarda é verdadeira, o restante do fragmento é ignorado. Caso contrário, o operando de interrupção é ignorado e o restante do fragmento é escolhido.

- ***par***: operador que permite o fragmento combinado representar uma mesclagem paralela entre o comportamento dos operandos. As especificações de ocorrência dos diferentes

operandos podem ser intercaladas de qualquer maneira, desde que a ordem imposta dentro de cada operando seja preservada.

- ***strict***: operador que permite o fragmento combinado representar um sequenciamento estrito entre o comportamento dos operandos. A semântica deste sequenciamento estrito define uma ordem de execução dos operandos.

- ***critical***: operador que permite o fragmento combinado representar uma região crítica. Uma região crítica significa que o operando não pode ser intercalado por outras especificações de ocorrências.

- ***neg***: operador utilizado para definir sequências inválidas. Todos os fragmentos de interação diferentes de *negativo* são considerados positivos, ou seja, descrevem sequências válidas e possíveis.

- ***assert***: operador de interação utilizado para especificar as únicas sequências válidas. Normalmente, este operador é utilizado em conjunto com os operadores *ignore* ou *consider*.

- ***ignore***: operador de interação utilizado para especificar alguns tipos de mensagens que serão ignoradas no fragmento combinado. Esses tipos de mensagens podem ser considerados insignificantes e são implicitamente ignoradas se aparecerem em uma execução correspondente.

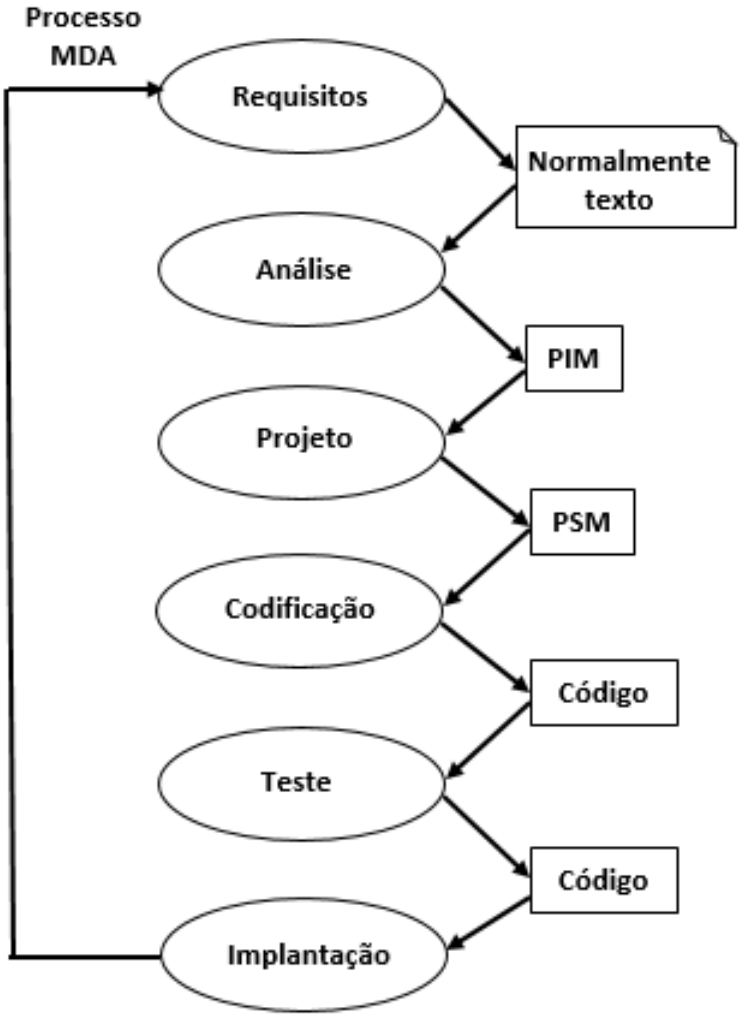
- ***consider***: operador de interação utilizado para especificar quais mensagens devem ser consideradas dentro do fragmento. Logo, todas as outras mensagens devem ser ignoradas.

2.2 MDA - Model Driven Architecture

A transformação de modelos é uma importante questão dentro do escopo da *Model Driven Engineering* (MDE). A MDE tem como objetivo apoiar o desenvolvimento de softwares complexos que envolvem diferentes tecnologias e domínios de aplicação, com foco em modelos e transformação de modelos (KENT, 2002; FAVRE, 2004; SCHMIDT, 2006). Uma proposta de MDE muito utilizada nos últimos anos é o *framework* de desenvolvimento de software definido pela OMG, *Model Driven Architecture* (MDA) (OMG, 2014a).

O conceito chave da MDA está relacionado com a importância dos modelos nos processos de desenvolvimento de software. Segundo Kleppe, Warmer e Bast (2003), o ciclo de vida de desenvolvimento MDA, que é ilustrado na Figura 3, não parece muito diferente do ciclo de vida de desenvolvimento tradicional, já que as mesmas fases são identificadas. Uma das principais diferenças reside na natureza dos artefatos que são gerados durante o processo de desenvolvimento. Esses artefatos são modelos que podem ser entendidos por computadores. Esses modelos podem ser de três tipos:

Figura 3 – Ciclo de vida do desenvolvimento de software MDA.



Fonte: Adaptada de Kleppe, Warmer e Bast (2003).

- ***Platform Independent Model* (PIM)**: modelo com um nível alto de abstração, que é independente de solução tecnológica.

- ***Platform Specific Model* (PSM)**: modelo utilizado para especificar o software, que é dependente de solução tecnológica específica. Um modelo PIM é transformado em um ou mais modelos PSMs.

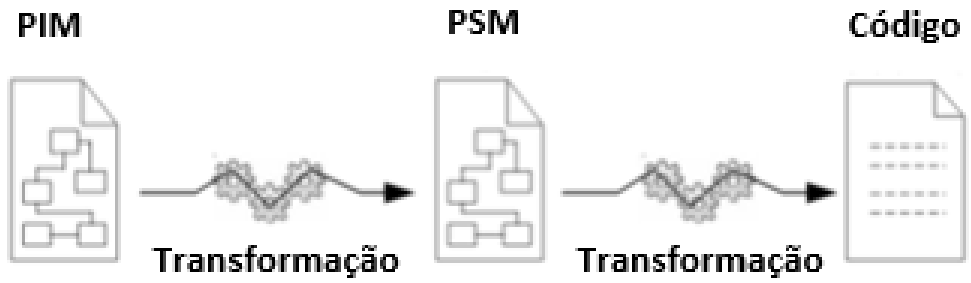
- **Código**: descrição da especificação do software em termos de código-fonte. Cada modelo PSM é transformado em código-fonte.

Em MDA, os modelos devem ser escritos em uma linguagem que tenha uma sintaxe bem definida. A linguagem UML, por ser a linguagem de modelagem padrão da OMG, é a mais utilizada no contexto de MDA. Outras linguagens também podem ser utilizadas, tais como as linguagens baseadas em máquinas de transições de estados. Portanto, a característica essencial

de um modelo em MDA é a linguagem na qual o modelo foi escrito, já que algumas linguagens são mais expressivas e mais adequadas para representar certos aspectos de um software.

Outra questão muito importante no contexto de MDA é a transformação entre os modelos. Conforme mostrado no processo ilustrado na Figura 3, vários modelos como PIM, PSM e código têm papel importante em MDA. Conforme ilustrado na Figura 4, o processo de MDA consiste basicamente na criação do PIM através de alguma linguagem de modelagem, na transformação do PIM em PSM a partir de regras de transformação, e por fim, a partir de outras regras de transformação, gerar o código-fonte correspondente.

Figura 4 – Processo MDA.



Fonte: Adaptada de David (2003).

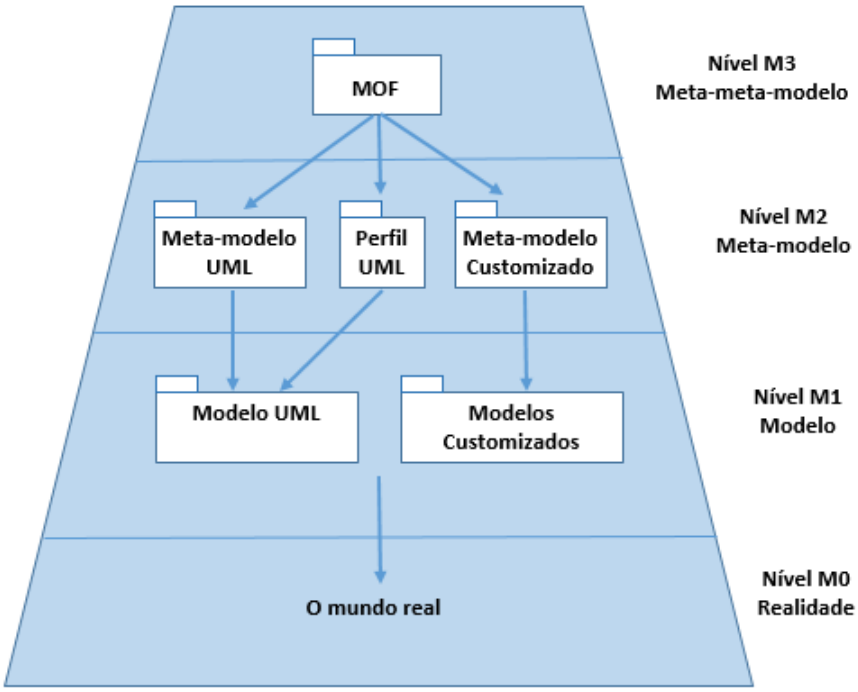
2.2.1 Meta-Modelagem

Assim como os modelos, os meta-modelos tem papel chave na MDA. Gasevic, Djuric e Devedzic (2009) definem meta-modelo como um modelo de especificação usado para validar modelos representados em uma linguagem específica, ou seja, um meta-modelo faz declarações sobre o que pode ser expresso nos modelos válidos de uma determinada linguagem de modelagem. As linguagens de modelagem usadas em MDA precisam ter definições formais para que as ferramentas de transformação possam converter automaticamente os modelos construídos nessas linguagens. A OMG criou uma linguagem especial chamada *Meta Object Facility* (MOF) (OMG, 2019), a qual é a meta-linguagem padrão para todas as linguagens de modelagem. Assim, cada linguagem é definida por meio de um meta-modelo utilizando a MOF.

A MDA é baseada em uma arquitetura de meta-modelo de quatro níveis conforme ilustrado na Figura 5. Os níveis podem ser descritos da seguinte forma:

- **Nível M3 - Meta-meta-modelo**: é o nível mais alto de abstração da arquitetura de meta-modelagem. Nesse nível, as linguagens de especificação de meta-modelos são definidas. A linguagem MOF é um exemplo de meta-meta-modelo.
- **Nível M2 - Meta-modelo**: é o nível onde as linguagens para especificar modelos são definidas. Todos os meta-modelos definidos nesse nível são instâncias dos meta-meta-modelos

Figura 5 – Arquitetura de meta-modelo da MDA.



Fonte: Adaptada de Gasevic, Djuric e Devedzic (2009).

do nível M3. Um exemplo são os meta-modelos da linguagem UML que descrevem estruturalmente como os modelos UML devem ser criados.

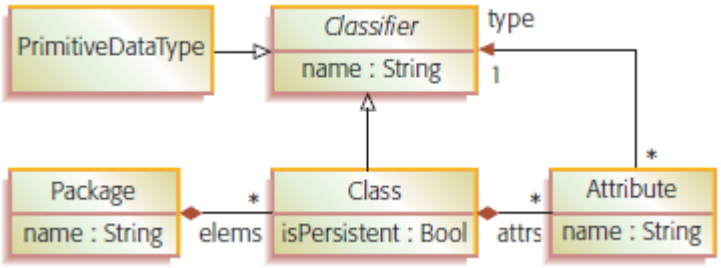
- **Nível M1 - Modelo**: é o nível onde as linguagens para descrever um sistema são definidas. Todos os modelos definidos nesse nível são instâncias dos meta-modelos do nível M2. Um exemplo são os modelos UML de um determinado software.

- **Nível M0 - Realidade**: os objetos que representam um determinado software do mundo real. Todos os objetos definidos nesse nível são instâncias dos modelos do nível M1. Um exemplo seria o código-fonte desse software.

Conforme apresentado na Figura 5, um perfil UML (UML *profile*) também pode ser criado no nível M2. O perfil UML é um conceito utilizado para estender os meta-modelos UML básicos a uma finalidade específica. Essencialmente, isto significa introduzir novos tipos de elementos de modelagem ao repertório de ferramentas do modelador.

Na Figura 6 é apresentado um exemplo de um meta-modelo (nível M2) simplificado do diagrama de classe da UML. O meta-modelo inclui o conceito abstrato de classificadores, que compreende classes e tipo de dados primitivos. Os pacotes contêm classes e as classes contêm atributos. Todos os elementos do modelo tem nomes e as classes podem ser marcadas como persistentes.

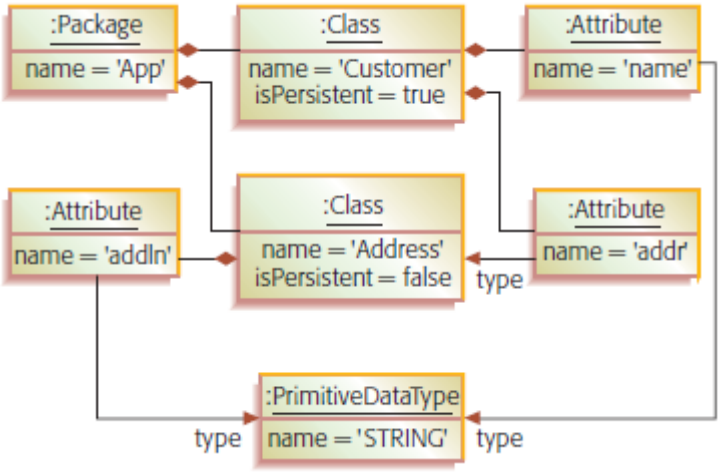
Figura 6 – Meta-modelo simplificado do diagrama de classe da UML.



Fonte: Czarnecki e Helsen (2006).

Um exemplo de um modelo de diagrama de classe da UML (nível M1) é ilustrado na Figura 7. O modelo contêm um pacote *App* com duas classes, *Customer* que é persistente e *Address* que não é persistente. A classe *Customer* tem o atributo *name* do tipo *String* e o atributo *addr* do tipo *Address* (associação). A classe *Address* tem o atributo *addIn*.

Figura 7 – Modelo de classes instanciado do meta-modelo da Figura 7.



Fonte: Czarnecki e Helsen (2006).

2.2.2 Transformação de Modelos

Conforme visto anteriormente, a transformação de modelos é uma questão chave no contexto da MDA. Essa transformação é a geração de um modelo destino a partir de um modelo origem. Esse processo de geração consiste em um conjunto de regras de transformação que descreve como os elementos do meta-modelo origem são mapeados em elementos do meta-modelo destino.

Kleppe, Warmer e Bast (2003) descreve algumas características desejáveis no processo de transformação, tais como:

- **Ajustável**: as regras de transformação podem ser ajustadas após a sua definição a fim de tornar o processo mais flexível.
- **Rastreável**: deve ser possível rastrear um elemento no modelo de destino de volta ao elemento no modelo de origem, a partir do qual ele foi gerado.
- **Consistência incremental**: quando informações específicas forem adicionadas no modelo de destino e houver a necessidade de gerar novamente esse modelo, as informações adicionadas devem persistir no modelo de destino ao final da nova geração.
- **Bidirecional**: significa que a transformação pode ser aplicada não só do modelo origem ao modelo destino, mas também de volta do modelo destino ao modelo origem.

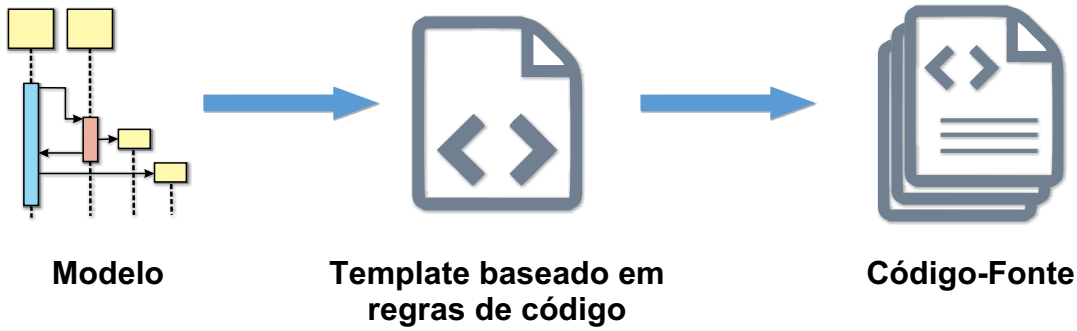
As transformações podem ser realizadas de duas formas: uma é o mapeamento Modelo-Modelo (M2M) e a outra é o mapeamento Modelo-Texto (M2T). A Figura 8 apresenta um esquema simples de transformação Modelo-Modelo. Ambos os modelos estão em conformidade com seus respectivos meta-modelos. As regras de transformação são definidas a partir do mapeamento dos elementos do meta-modelo origem e do meta-modelo destino. A transformação é executada em modelos concretos.

Figura 8 – Esquema de transformação Modelo-Modelo.

Fonte: Adaptada de Czarnecki e Helsen (2006).

O exemplo mais comum de transformação Modelo-Texto é a geração de código-fonte de linguagem de programação a partir de um modelo de origem. Neste tipo de transformação

Figura 9 – Um simples esquema de transformação Modelo-Texto.



Fonte: Adaptada de ACCELEO (2018).

pode-se usar uma tecnologia baseada em *template*. Neste contexto, um *template* consiste no texto de destino contendo o meta-código para acessar informações variáveis (CZARNECKI; HELSEN, 2006). A Figura 9 mostra um esquema simples de transformação Modelo-Texto.

2.2.2.1 Linguagens de Transformação de Modelos

Com o intuito de padronizar a transformação de modelos, a OMG criou a linguagem de transformação *Query/View/Transformation* (QVT) (OMG, 2016). A Linguagem QVT é baseada em três conceitos:

- **Consulta(*Query*)**: deve receber um modelo de entrada e alguns elementos específicos desse modelo são selecionados.
- **Visão(*View*)**: refere-se a modelos que são derivados de outros modelos, sendo resultado de uma consulta.
- **Transformação(*Transformation*)**: recebe um modelo de entrada e a saída é a atualização do modelo ou a criação de um novo modelo.

Outra linguagem muito utilizada no contexto da MDA é a linguagem de transformação de modelos *Atlas Transformaion Language* (ATL), um dos pacotes desenvolvidos na plataforma de MDE *ATLAS Model Management Architecture* (AMMA) (BÉZIVIN; JOUAULT; TOUZET, 2005). Um módulo ATL permite que se especifique como modelos de destinos são produzidos a partir de um conjunto de modelos de origem (JOUAULT; KURTEV, 2006). Estes módulos são compostos pelos seguintes elementos:

- **Seção Cabeçalho**: fornece o nome do módulo de transformação e declara os modelos de origem e destino.
- **Seção Importação**: permite declarar quais bibliotecas ATL devem ser importadas.
- **Conjunto de Auxiliares (*helpers*)**: este termo vem da especificação da *Object Constraint Language* (OCL) (OMG, 2014b) e podem ser de dois tipos: operação e atributos. Os auxiliares só podem ser especificados em um tipo OCL ou em um tipo de meta-modelo de origem, pois os modelos de destino não são navegáveis. O tipo operação pode ser definido no contexto de um elemento de modelo ou de um módulo. Ele podem ter parâmetros de entrada. Já o tipo atributo é utilizado para associar valores apenas de leitura aos elementos do modelo de origem.
- **Conjunto de regras**: construção básica usada para expressar a lógica de transformação. As regras de transformação implementadas em ATL podem ser especificadas em contruções declarativa (*Matched Rules*) ou imperativa (*Called Rules*). *Lazy Rules* são tipos de *Matched Rules* que são acionadas por outras regras.

No contexto de transformação Modelo-Texto, uma opção é a linguagem Acceleo. O Acceleo é o resultado de vários anos de Pesquisa e Desenvolvimento (P&D) iniciados na empresa Obeo[2]. Ele utiliza uma tecnologia baseada em *template*, incluindo ferramentas de autoria para criar geradores de código personalizados, permitindo a geração automática de qualquer tipo de código-fonte a partir de qualquer fonte de dados disponível no formato *Eclipse Modelling Framework* (EMF) (ACCELEO, 2018). A Linguagem Acceleo tem a sintaxe muito similar à sintaxe da OCL. Os principais tipos de estruturas nos módulos geradores são:

- ***Templates***: conjunto de instruções usadas para gerar texto. Dentro dessa estrutura, as *tags* de arquivo são usadas para informar ao mecanismo do Acceleo que ele deve gerar o conteúdo em um arquivo real.

- ***Queries***: usadas para extrair informações dos modelos. Elas retornam valores ou coleções de valores.

2.3 Teste Baseado em Modelo

Uma especificação de software é um documento que descreve o comportamento de um sistema. Normalmente, este comportamento é representado através de vários modelos, cada qual descrevendo uma perspectiva do software em construção. Teste baseado em modelo (TBM) surgiu com o intuito de tratar a atividade de teste de maneira sistemática. TBM consiste em uma técnica para geração automática de casos de teste a partir de modelos extraídos de especificações de software (BINDER, 2000). Essas especificações servem como entrada para a definição dos casos de teste e como a geração é feita automaticamente, qualquer modificação nas especificações do software será reproduzida nos casos de teste, permitindo que os testes e requisitos do software mantenham-se consistentes (DIAS-NETO; TRAVASSOS, 2010).

Uma questão importante no contexto de TBM é definir qual notação será utilizada para escrever os modelos. Existe uma grande variedade de notações que podem ser utilizadas no desenvolvimento de modelos de software. É desejável que a notação utilizada tenha uma semântica precisa, diminuindo ambiguidades e inconsistências. Portanto, o uso de especificações formais é uma boa opção, já que são técnicas que conseguem traduzir com exatidão o comportamento de um sistema eliminando ambiguidades e reduzindo a chance de erros serem introduzidos durante o desenvolvimento do software (HIERONS *et al.*, 2009). Existem diversas técnicas de modelagem formal baseadas em Máquinas de Transição de Estados utilizadas para especificação de software, tais como: Máquinas de Estados Finitos (MEFs) e as Máquinas de Estados Finitos Estendidas (MEFEs).

[2] Empresa francesa que oferece uma gama de soluções de código aberto para criar e transformar sistemas complexos. É um membro estratégico da *Eclipse Foundation*.

2.3.1 Processo e Terminologia de TBM

O teste de software tem como objetivo executar uma implementação do sistema em construção com dados de teste e verificar se seu comportamento operacional está conforme sua especificação (SOMMERVILLE, 2007). Esta implementação que está sendo testada é denominada de sistema sob teste (*System Under Test* (SUT)).

O TBM se baseia em modelos e os casos de teste são gerados a partir de um destes modelos ou da combinação de modelos, e em seguida, executados no SUT. Segundo Utting, Pretschner e Legeard (2012), o uso de modelos é motivado pela observação de que, tradicionalmente, o processo de testes é desestruturado, não reproduzível, não documentado e depende da criatividade dos Engenheiros de Software. A ideia é que artefatos utilizados na codificação do SUT possam ajudar a mitigar esses problemas.

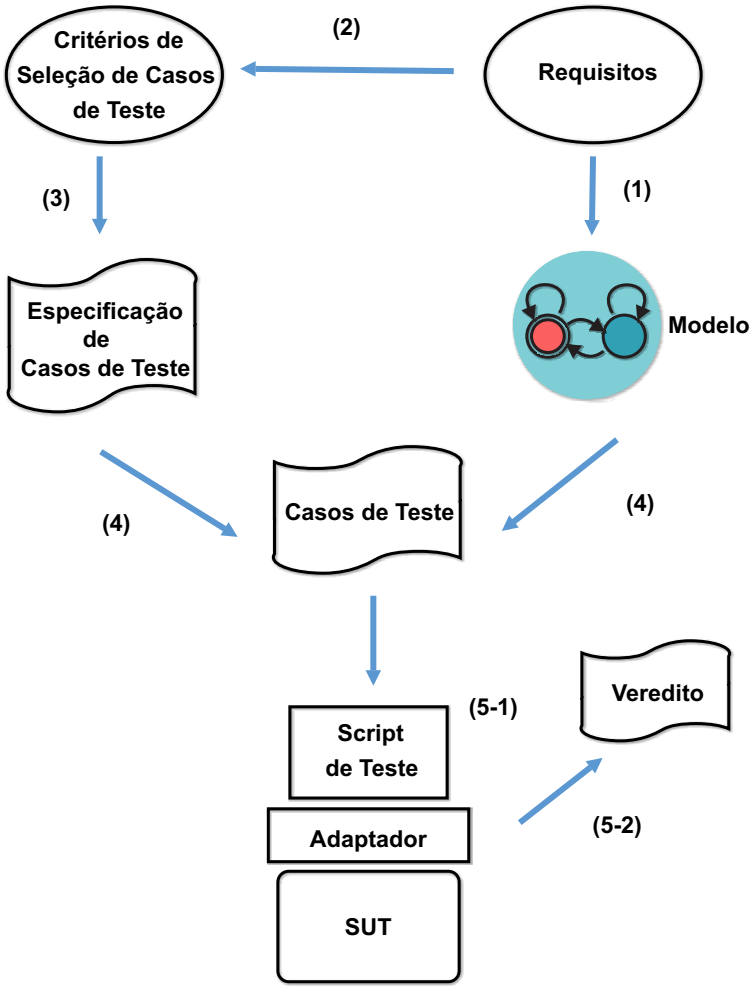
Em resumo, o TBM abrange os processos e técnicas para derivação automática de casos de teste a partir de modelos abstratos de software. Para a obtenção de sucesso nessa atividade, é necessário rigor nesse processo. Portanto, Utting, Pretschner e Legeard (2012) definem um processo genérico de TBM, ilustrado na Figura 10. O processo genérico é dividido em cinco etapas:

- **Etapa 1 - Modelo de Teste**: um modelo do SUT é criado a partir dos requisitos do software. Este modelo é chamado de modelo de teste, pois o nível de abstração e o foco do modelo estão diretamente ligados aos objetivos do teste. É importante a independência entre o modelo utilizado para geração de teste e qualquer modelo de desenvolvimento, para que os erros no desenvolvimento não sejam propagados para os testes gerados. O modelo de teste gerado deve ser suficientemente preciso para geração de casos de teste significativos. Para verificar a consistência do modelo pode-se utilizar técnicas de checagem de modelo (*model checking*). O resultado dessa etapa é um modelo preciso.

- **Etapa 2 - Critérios de Seleção de Casos de Teste**: são escolhidos e estabelecidos para guiar a geração automática de casos de teste a partir do modelo criado na etapa anterior. Estes critérios podem estar relacionados com uma funcionalidade do sistema (baseados em requisitos), a estrutura do modelo de teste (cobertura de estado, transição e fluxo de dados), a heurística de cobertura de dados, às propriedades do ambiente e também com um conjunto bem definido de falhas.

- **Etapa 3 - Especificações de Casos de Teste**: os critérios de seleção de testes são transformados em especificações de casos de teste através da formalização da notação desses critérios, tornando-os operacionais. Por exemplo, cobertura de estado de uma Máquina de Estados Finitos pode ser transformado em um conjunto de especificação de casos de teste, tais como: *alcançar S0*, *alcançar S1*, *alcançar S2*, ..., onde *S0*, *S1*, *S2*, ... são todos os estados da Máquina. Uma especificação de caso de teste é uma descrição de alto nível de um caso de teste desejado.

Figura 10 – O Processo Genérico de TBM.



Fonte: Adaptada de Utting, Pretschner e Legeard (2012).

- **Etapa 4 - Geração de Casos de Teste**: uma vez que o modelo e as especificações dos casos de teste são definidos, um conjunto de casos de teste abstratos é gerado, com o objetivo de satisfazer as especificações dos casos de teste.

- **Etapa 5 - Execução dos Casos de Teste**: uma vez que o conjunto de testes abstratos foi gerado, os casos de teste são concretizados e executados no SUT. A execução pode ser manual ou automatizada e os resultados dos testes são gerados.

Resumidamente, o TBM envolve as seguintes atividades principais: construção do modelo de teste, definição dos critérios de seleção de teste, transformação dos critérios em especificações de casos de teste, geração de casos de teste (abstratos e concretos), execução dos casos de teste concretos no SUT e geração dos resultados da execução (UTTING; PRETSCHNER; LEGEARD, 2012).

Um ponto importante no processo de TBM é a escolha do formato em que o modelo será representado. Um formato muito utilizado são os modelos formais baseados em Máquinas de

Transições de Estados, tais como: MEFs e MEFEs. Nas próximas seções, essas duas técnicas são apresentadas, com seus principais conceitos e diferenças.

2.3.2 Máquinas de Estados Finitos

Uma Máquina de Estados Finitos (MEF) é um modelo bem estabelecido para representar o comportamento de sistemas e muito utilizado para geração automática de casos de teste. Gill (1962) define uma MEF como um sistema síncrono e determinístico, composto por entradas, saídas, estados e um par de funções de caracterização representadas pelas transições. Cada transição liga dois estados, que podem ser distintos ou não. Como é uma máquina determinística, a cada instante, ela pode estar em apenas um de seus estados. A cada entrada, a máquina gera uma saída e executa uma transição.

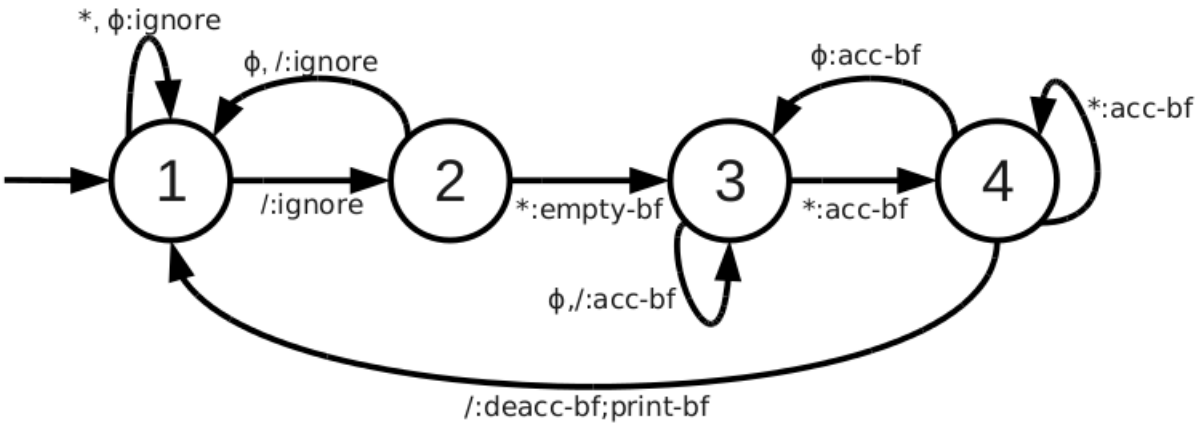
Existem dois tipos de MEF: Máquina de *Mealy* e Máquina de *Moore* (LEE; YANNAKAKIS, 1996). A forma de representar suas saídas é a principal diferença entre elas. Na máquina de *Mealy*, as saídas estão associadas às transições, pois ao receber uma entrada, o evento de saída é produzido durante a mesma transição. Na máquina de *Moore*, as saídas estão associadas aos estados, pois ao receber uma entrada, o evento de saída é produzido no estado destino da transição.

De acordo com Simão (2016), uma MEF pode ser representada formalmente como uma tupla $M = (X; Z; S; s_0; f_z; f_s)$, sendo que:

- X é um conjunto finito não-vazio de símbolos de entrada;
- Z é um conjunto finito de símbolos de saída;
- S é um conjunto finito não-vazio de estados;
- s_0 ε S é o estado inicial;
- f_z: $(S \times X) \rightarrow Z$ é a função de saída;
- f_s: $(S \times X) \rightarrow S$ é a função de próximo estado.

Para exemplificar, na Figura 11 é apresentada uma MEF mínimal correspondente a uma especificação informal de um sistema que imprime comentários de um código escrito na linguagem C. Um comentário é uma cadeia de caractere entre '$/*$' e '$*/$'. A MEF possui o alfabeto de entrada $X = \{*, /, \phi\}$, onde 'ϕ' representa qualquer caractere diferente de $*$ e $/$. O conjunto de variáveis de saída é representado pelo alfabeto $Z = \{ignore, empty\text{-}bf, acc\text{-}bf, deacc\text{-}bf, print\}$ e o conjunto de estados é representado por $S = \{1,2,3,4\}$. Além disso, possui a função $f_z(1,/) = ignore$ e a função $f_s(1,/) = 2$ que representam respectivamente, uma função de saída e uma função de próximo estado para MEF. No exemplo da Figura 11 existem

Figura 11 – MEF de um sistema extrator de comentários.



Fonte: Chow (1978).

algumas ações que são executadas de acordo com as operações de saída conforme apresentadas na Tabela 1.

Tabela 1 – Operações de saída da MEF da Figura 11.

Operação	Ação
ignore	Ação nula.
empty-bf	buffer:= <>.
acc-bf	buffer:= buffer concatenado com o caractere corrente.
deacc-bf	buffer:= buffer com o caractere mais à direita truncado.
print	imprime o conteúdo do buffer.

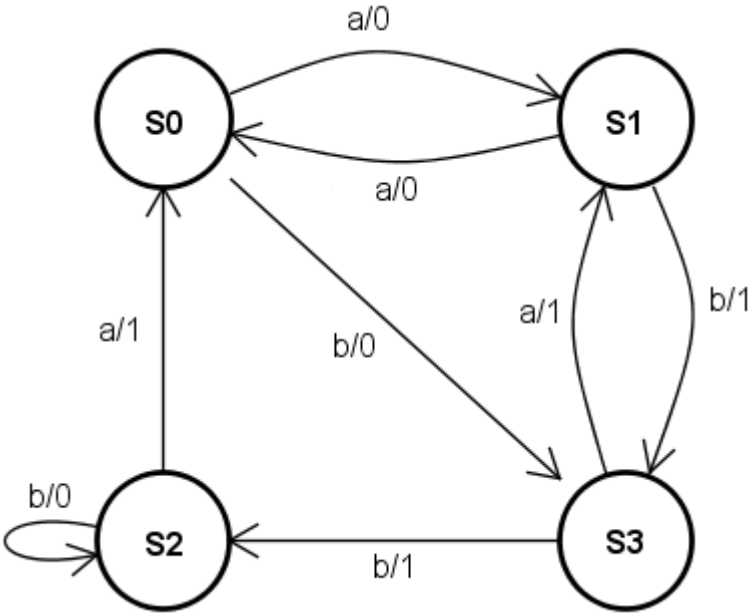
Fonte: Adaptada de Chow (1978).

Outras propriedades importantes que uma MEF pode satisfazer são descritas a seguir (SIMÃO, 2016):

- Uma MEF é completamente especificada se todas as entradas são tratadas em todos os estados. Caso contrário, ela é parcialmente especificada.
- Uma MEF é fortemente conexa se, para cada par de estados (s_i, s_j), existir um caminho por transições que vai de s_i a s_j. Ela é inicialmente conexa se, a partir do estado inicial, for possível atingir todos os demais estados da MEF.
- Uma MEF é mínimal ou reduzida se não existirem quaisquer dois estados equivalentes. Dois estados são equivalentes se produzem as mesmas sequências de saídas para as mesmas sequências de entrada.
- Uma MEF é determinística quando, para qualquer estado e para dada entrada, a MEF permite uma única transição para o próximo estado. Caso contrário, é não determinística.

Para exemplificar essas propriedades, a Figura 12 apresenta uma MEF completa, fortemente conexa, minimal e determinística.

Figura 12 – MEF completa, fortemente conectada, minimal e determinística.



Fonte: Elaborada pelo autor.

Já a Figura 13 apresenta uma MEF com as seguintes propriedades: (1) parcial, pois a entrada a não é tratada no estado $S1$; (2) desconexa, pois não é possível alcançar o estado $S1$ a partir de nenhum dos estados; (3) não-determinística, dado que o estado $S1$ possui duas transições para o mesmo evento de entrada b; e (4) não-minimal, pois os estados $S2$ e $S3$ são equivalentes.

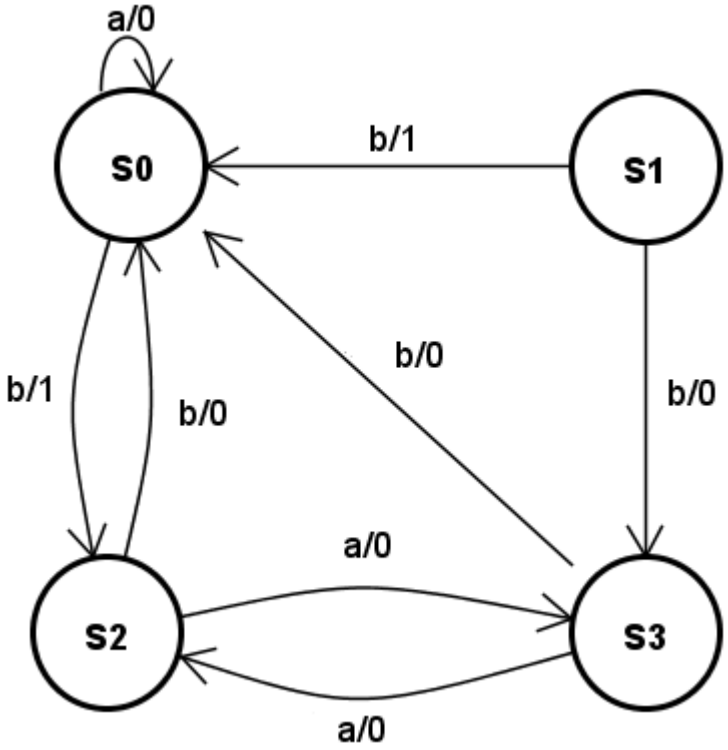
2.3.2.1 Métodos de Geração de Testes Baseados em MEFs

Testes gerados a partir de uma MEF consistem em um conjunto de sequências de teste. Uma sequência de teste é uma sequência de símbolos de entrada com sua respectiva saída. O objetivo do teste é submeter esse conjunto de sequências de teste a uma implementação e verificar se a saída gerada pela implementação está de acordo com a saída especificada na sequência de teste.

De acordo com Chow (1978), os erros de implementação são classificados em:

- **Erros de saída**: quando a saída da implementação difere da saída especificada na sequência de teste. Portanto, a função de saída da implementação deverá ser modificada.
- **Erros de transferência**: quando a transição da implementação atinge um estado incorreto. Portanto, a função de transição da implementação deverá ser modificada.

Figura 13 – MEF parcial, desconexa, não-minimal e não-determinística.



Fonte: Elaborada pelo autor.

- **Estados ausentes**: a implementação possui um número de estados menor que a especificação. Portanto, o número de estados da implementação deverá ser aumentado.
- **Estados extras**: a implementação possui um número de estados maior que a especificação. Portanto, o número de estados da implementação deverá ser diminuído.

O tamanho de uma sequência de teste é calculado pela quantidade de símbolos de entrada somados a quantidade de operações *resets*. A operação especial *reset* é denotada pela letra r e leva a MEF de qualquer estado ao seu estado inicial. A saída dessa operação é nula.

Os métodos de geração de sequências de teste são baseados em algumas sequências básicas, as quais são utilizadas para se obter resultados parciais importantes (SIMÃO, 2016). Essas sequências possuem características interessantes que são utilizadas pelos métodos de geração de teste. Algumas definem um caminho que percorrem todos os estados ou que executem todas as transições. Outras têm a finalidade de assegurar que, ao serem aplicadas à MEF, as saídas sejam diferentes para cada um dos estados. As sequências básicas mais utilizadas pelos métodos de geração de testes são:

- **Cobertura de Estado (*state cover*)**: conjunto de sequências de símbolos de entrada que leva a MEF do estado inicial para cada um dos seus estados, incluindo a sequência vazia ε. A MEF da Figura 12 possui o conjunto *state cover* $Q = \{\varepsilon, a, bb, ab\}$.

- **Cobertura de Transição (*transition cover*)**: conjunto de sequências de símbolos de entrada que exercita todas as transições da MEF, incluindo a sequência vazia ε. A MEF da Figura 12 possui o conjunto *transition cover* $P = \{\varepsilon, a, b, aa, ab, ba, bb, bbb, bba\}$.

- **Sequências de sincronização**: é uma sequência de símbolos de entrada que leva a MEF ao mesmo estado, independente do estado original da MEF. Esse sequência é utilizada quando se deseja que a MEF vá para um determinado estado. Na MEF da Figura 12 uma sequência de sincronização para o estado $S3$ é $\{ab\}$.

- **Sequências de distinção**: sequência de símbolos de entrada que produz saídas diferentes em cada estado da MEF. Na MEF da Figura 12 uma sequência de distinção é DS=$\{ab\}$. A Tabela 2 exibe as saídas quando se aplica a sequência para cada estado.

Tabela 2 – Saídas obtidas para sequência DS.

Estados	ab
S0	01
S1	00
S2	10
S3	11

Fonte: Elaborada pelo autor.

- **Sequências UIO**: sequência de símbolos de entrada que produz saídas únicas para um estado determinado da MEF. A sequência de símbolos de entrada de uma UIO de um estado pode ser igual à sequência de símbolos de entrada da UIO de outro estado. Com a aplicação da sequência UIO é possível distinguir um estado dos demais, pois a saída é única do estado. Uma sequência DS é UIO para todos os estados. A sequência de entrada $\{ab\}$ gera as saídas $\{0,0\}$ para o estado S1 e as saídas $\{1,1\}$ para o estado S3 da MEF da Figura 12. Como em nenhum outro estado desta MEF retorna estas saídas, a sequência é uma UIO para os estados 1 e 3.

- **Conjunto de caracterização**: também conhecido como conjunto W, é formado por sequências de símbolos de entrada que produzem saídas diferentes em cada estado da MEF. A união de todas sequências UIO formam um conjunto W, assim como toda DS representa um conjunto W unitário, e toda MEF, desde que seja minima contêm o conjunto W. O conjunto $\{a, bb\}$ é um conjunto de caracterização para MEF da Figura 12.

- **Conjunto de identificação** W_i: é um subconjunto do conjunto W que identifica individualmente cada estado S_i. A MEF da Figura 12 possui os seguintes conjuntos de identificação: $W_0 = \{\text{a, bb}\}$, $W_1 = \{\text{bb}\}$, $W_2 = \{\text{bb}\}$ e $W_3 = \{\text{ab}\}$.

- **Conjunto de identificadores harmonizados** H_i: é um conjunto de identificadores de estados, tal que para dois estados diferentes quaisquer S_i e S_j, $\exists \beta \in H_i$, $\gamma \in H_j$ que tem um

prefixo α em comum que $f_z(S_i, \alpha) \neq f_z(S_j, \alpha)$. A MEF da Figura 12 possui os seguintes conjuntos de identificadores harmonizados H_i: $H_0 = \{a, b\}$, $H_1 = \{bb\}$, $H_2 = \{a, b\}$ e $H_3 = \{bb\}$.

Existem diversos métodos propostos para geração de sequências de testes a partir de MEF. Todos esses métodos definem procedimentos para geração de testes. O que os diferenciam é o custo da geração dessas sequências e a efetividade na detecção de defeitos. O objetivo principal é encontrar o maior número de defeitos em uma implementação considerando o tamanho do conjunto de teste, pois a quantidade de testes pode inviabilizar a sua aplicação prática. A seguir é apresentada uma breve descrição dos principais métodos de geração de testes baseados em MEFs:

- *Método DS* (GONENC, 1970): utiliza a sequência de distinção para gerar as sequências de teste. Portanto, a utilização desse método fica condicionada à existência dessa sequência na MEF. Inicialmente o método seleciona a menor sequência de distinção e a aplica aos estados para gerar um grafo X_d. A partir do grafo X_d duas sequências são geradas para verificar, respectivamente, os estados e transições da MEF. Os casos de teste são resultado da concatenação dessas duas sequências.

- *Método W* (CHOW, 1978): esse método pode ser considerado o mais difundido para geração de sequências de testes. Sua aplicabilidade está restrita as MEFs completas, inicialmente conexas, minimais e determinísticas. Ele é baseado na geração de duas sequências básicas: conjunto de caracterização W e *transition cover* P. A partir do conjunto W, é gerada uma sequência Z que verifica se as transições levarão aos estados corretos. Dessa forma, é necessário considerar que a quantidade de estados da implementação pode ser maior ou igual a quantidade de estados da especificação. Os casos de teste são resultado da concatenação das sequências Z e P.

- *Método* W_p (FUJIWARA *et al.*, 1991): proposto como melhoria do método W. A principal vantagem deste método em relação ao método W está no tamanho dos conjuntos de testes gerados, já que são menores e com a mesma garantia da cobertura de defeitos. Assim como o método W, para utilização do método W_p, a MEF deve ser completa, inicialmente conexa, minimal e determinística. Além dos conjuntos de caracterização W e *transition cover* P, o método W_p utiliza também o conjunto *state cover* Q e o conjunto de identificação W_i que é um subconjunto do conjunto W e que identifica individualmente cada estado S_i.

- *Método HSI* (PETRENKO *et al.*, 1994; LUO; PETRENKO; BOCHMANN, 1995): também foi proposto como uma melhoria para o método W. Sua aplicabilidade é maior que os outros métodos, já que pode ser aplicado também às MEFs parciais, garantindo cobertura completa dos defeitos. Os conjuntos de identificação H_i são utilizados para fazer as distinção dos estados. O método consiste basicamente de duas fases. Na primeira fase

utiliza-se o conjunto *state cover* Q e o conjunto de identificação H e na segunda fase utiliza-se o conjunto *transition cover* P com o conjunto de identificação H.

- *Método H*: inicialmente proposto por Koufareva e Dorofeeva (2002) para MEFs completas e determinísticas. Dorofeeva R. e Yevtushenko (2005) estenderam o método para aplicar em máquinas parciais determinísticas e conseguiram obter conjuntos de testes completos. Diferentemente do método HSI, nesse método estendido, o conjunto de testes é construído sequência por sequência, evitando-se incluir sequências desnecessárias. Para identificar os estados, as sequências do conjunto *state cover* são aplicadas ao conjunto de identificação H_i. Para verificar as transições, as sequências do conjunto *transition cover* P são concatenadas com as sequências do conjunto *state cover* Q dos estados de destino alcançados na aplicação do conjunto *transition cover* P.

- *Método SPY* (SIMÃO; PETRENKO; YEVTUSHENKO, 2009): obtém conjunto de testes m-completos em MEFs com implementações de estados extras e pode ser aplicada em MEFs parciais determinísticas. O método distribui as sequências dos conjuntos de travessia (*traversal set*) a fim de reduzir as ramificações de teste e gerar suíte de testes menores. Utiliza o conjunto *state cover* Q e o conjunto de identificação H_i. Além disso, seleciona o conjunto *transition cover* para que todas as transições sejam cobertas e novamente o conjunto H_i é adicionado ao final dos estados atingidos para confirmá-los.

2.3.3 Máquinas de Estados Finitos Estendidas

Embora as MEFs sejam simples e amplamente utilizadas para especificar modelos de testes, existem limitações para modelar o comportamento de sistemas complexos que necessitem representar tanto fluxo de controle como também fluxo de dados (YANG *et al.*, 2015). Para estes tipos de sistemas, é mais adequado a utilização de um modelo estendido chamado de Máquina de Estados Finitos Estendida (MEFE) (LEE; YANNAKAKIS, 1996). As MEFE consistem de estados, predicados e atribuições relacionadas a variáveis entre transições, de tal forma que pode-se representar o fluxo de controle e de dados de sistemas complexos.

Segundo Yang *et al.* (2015), uma MEFE pode ser representada formalmente como uma 6-tupla $M = (s_0, S, V, I, O, T)$, sendo que:

- $s_0 \in S$ é o estado inicial;
- S é um conjunto finito de estados;
- V é um conjunto finito de variáveis de contexto;
- I é um conjunto de entradas de transições;
- O é um conjunto de saídas de transições;

- T é um conjunto finito de transições;

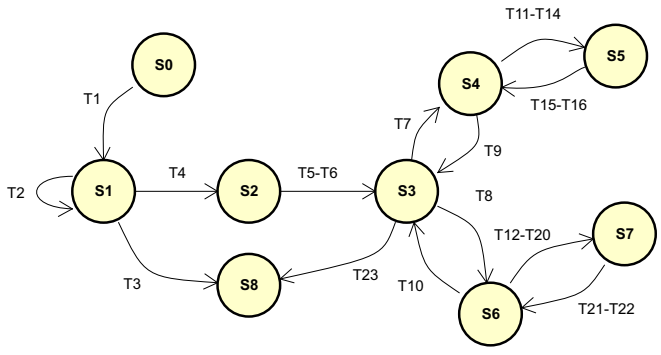
Cada transição $t_x \in T$ também pode ser representada formalmente por uma tupla $t_x = (s_i, s_j, P_{t_x}, A_{t_x}, i_{t_x}, o_{t_x})$, onde s_i e s_j são os estados de origem e de destino da transição t_x, e $i_{t_x} \in I$ representam os parâmetros de entrada do início da transição de estado t_x, como os eventos, e $o_{t_x} \in O$ representam os resultados de saída do final da transição t_x. Além disso, P_{t_x} representa as condições de predicado com suas respectivas variáveis de contexto e A_{t_x} os operadores com suas respectivas variáveis atuais. Alguns autores chamam P_{t_x} de guardas e A_{t_x} de ações.

Assim como nas MEFs, os diagramas de transições de estados também são utilizados nas MEFEs para a visualização das informações. Esse tipo de diagrama pode ser representado por um grafo dirigido $G = (V, E)$, onde:

- V é um conjunto de vértices que representam os estados da MEFE.
- E é um conjunto de arcos dirigidos que representam as transições da MEFE.

A Figura 14 (a) representa um grafo dirigido com a simplificação de uma MEFE de um Sistema de Caixa Eletrônico (*Automated Teller Machine* - ATM). A Figura 14 (b) apresenta os detalhes das transições do ATM com seus eventos, guardas e ações. Inicialmente, a MEFE está no estado

Figura 14 – MEFE simplificada de um Sistema de Caixa Eletrônico (*Automated Teller Machine* - ATM).



(a)

Nome da Transição	Transição de Estado	Evento	Guarda	Ação
T1	S0 → S1	card(pin,sb,cb)		attemps=0
T2	S1 → S1	pim(p)	p!=pin && attemps<3	
T3	S1 → S8	pim(p)	p!=pin && attemps==3	
T4	S1 → S2	pim(p)	p==pin	
T5	S2 → S3	english()		l=e
T6	S2 → S3	chinese()		
T7	S3 → S4	current()		
T8	S3 → S6	saving()		
T9	S4 → S3	done()		
T10	S6 → S3	done()		
T11	S4 → S5	balance(l)	l==c	print_c(cb)
T12	S4 → S5	balance(l)	l==c	print_e(cb)
T13	S4 → S5	deposit(d)		cb=cb+d
T14	S4 → S5	withdraw(w)	w>0 && w<cb	cb=cb-w
T15	S5 → S4	display(e)	l==e	receipt(e)
T16	S5 → S4	display(c)	l==c	receipt(c)
T17	S6 → S7	withdraw(w)	w>0 && w<sb	sb=sb-w
T18	S6 → S7	deposit(d)	d>0	sb=sb+d
T19	S6 → S7	balance(l)	l==e	print_e(sb)
T20	S6 → S7	balance(l)	l==c	print_c(sb)
T21	S7 → S6	receipt(l)	l==e	receipt(e)
T22	S7 → S6	receipt(l)	l==c	receipt(c)
T23	S3 → S8	exit()		

(b)

Fonte: Adaptada de Yang *et al.* (2015).

inicial s_0 associados com os valores iniciais das variáveis. A transição t_1 irá ocorrer, se os valores atuais das variáveis ou os parâmetros de entrada i_{t_1} forem válidos para o predicado condicional P_{t_1} associado a esta transição. Nesse processo, a ação A_{t_1} associada à essa transição é então executada, podendo alterar as variáveis ou produzir resultados de saídas o_{t_1}, e a MEFE alcançar o próximo estado. Este processo ocorre várias vezes até que a MEFE fique em um determinado

estado s_i. Esta sequência de estados constitui o Caminho de Estado (*State Path* - SP), enquanto a sequência de transições constitui o Caminho de Transição (*Transition Path* - TP).

Conforme foi visto, a MEFE é um modelo avançado que se estende das MEFs. Se todos os predicados sempre forem verdadeiros e o conjunto de variáveis vazio, a MEFE é uma MEF. Portanto, A MEF pode ser vista como um subconjunto de MEFE (YANG *et al.*, 2015). Em uma MEFE, algumas transições podem estar associadas a condições de predicados que dificilmente serão satisfeitas. Conforme Hedley e Hennell (1985), um caminho é considerado inviável quando os predicados são contraditórios e precisam ser satisfeitos para um caminho ser executado. Portanto, a detecção de um caminho inviável é geralmente indecidível. A existência de caminhos inviáveis é um desafio da geração automática de casos de teste para MEFE (KALAJI; HIERONS; SWIFT, 2010; ZHAO; HARMAN; LI, 2010; ASOUDEH; LABICHE, 2014).

2.3.4 Frameworks de Geração de Testes

Como visto anteriormente, no processo de TBM são gerados casos de teste abstratos e concretos. Os testes abstratos são conjuntos de sequências de operações gerados a partir do modelo criado e conforme os critérios de testes definidos no processo. Eles são executáveis apenas no modelo. Já os testes concretos são resultados da transformação dos casos de teste abstratos em *scripts* executáveis em alguma linguagem de programação e são executados no SUT. Neste trabalho foram utilizados os *frameworks* ModelJUnit (UTTING; LEGEARD, 2006) e JUnit (BECK; GAMMA, 1998) para geração de testes abstratos e concretos, respectivamente.

ModelJUnit é um *framework* de código aberto que possibilita a implementação de TBM utilizando modelos formais tais como MEFs e MEFEs. Esses modelos são escritos na linguagem de programação Java e a biblioteca fornece uma variedade de algoritmos para geração de teste, recursos de visualização de modelos e estatísticas de cobertura de modelos (ZANDER; SCHIEFERDECKER; MOSTERMAN, 2011). Essas funcionalidades são disponibilizadas através da implementação de três classes:

- **Modelo (*Model*)**: para criação desta classe que representa o modelo, a biblioteca disponibiliza a interface *FsmModel* e todos os estados do modelo são definidos através da variável do tipo enumeração (*enum*) *State*. As variáveis de contexto são definidas na classe e para todas as entradas das transições, um método (*@Action*) é definido para vincular os estados de origem e destino da transição. Além disso, são definidos na classe o método *getState* que retorna o estado atual e o método *reset* que leva a máquina ao estado inicial.

- **Adaptador (*Adapter*)**: classe que permite que o modelo se comunique com o SUT. Para cada método (*@Action*) da classe modelo, um método com nome semelhante é adicionado na classe adaptador que aciona um evento do SUT. Além disso, é necessário instanciar um objeto no adaptador para cada classe do SUT.

- **Teste (*Test*)**: nesta classe é necessário instanciar a classe do modelo e escolher a estratégia de teste a ser usada. A biblioteca ModelJUnit oferece diversos algoritmos, tais como: *GreedyTester*, *LookaheadTester* e *RandomTester*. O *LookaheadTester* é um algoritmo mais sofisticado e pode cobrir todos os estados e transições rapidamente (UTTING, 2012). Ele é mais eficiente do que as outras duas estratégias mencionadas, pois não executa etapas aleatórias como *RandomTester* e, embora seja similar ao *GreedyTester*, ele oferece opções mais refinadas que permite alcançar transições e estados não visitados (CHERUKURI; GUPTA, 2010). Por fim, pode-se usar o método *buildGraph* para construir o grafo e gerar os testes. Este grafo também é usado para calcular métricas de cobertura para transições, estados e ações.

JUnit é um *framework* de código aberto, criado por Beck e Gamma (1998), que permite a criação de testes automatizados com suporte a linguagem de programação Java. A biblioteca do JUnit fornece um conjunto de classes para criar testes, executá-los e verificar se o SUT funciona da forma esperada. Um teste JUnit é um método contido em uma classe com a anotação *@Test*. Para verificar se o resultado do teste está em conformidade com o resultado esperado, o JUnit fornece um conjunto de métodos *assert*, tais como: *assertTrue*, *assertFalse*, *assertEquals*, *assertNull*, *assertNotNull*, etc.

2.4 Considerações Finais

Este capítulo apresentou os principais temas utilizados neste trabalho. Inicialmente, o Diagrama de Sequência UML que é o ponto de partida do trabalho foi abordado. A partir da versão 2.0 da UML, este diagrama tornou-se mais expressivo para modelagem de cenários e do fluxo de controle de sistemas. Como um dos objetivos do trabalho é a formalização do Diagrama de Sequência UML, os conceitos da MDA foram descritos, com destaque a meta-modelagem e transformação de modelos. Por fim, o Teste Baseado em Modelo foi abordado com ênfase em modelos formais. Os métodos de geração de testes baseados em MEF foram discutidos e as MEFEs que são modelos formais mais expressivos e amplamente utilizados para modelar sistemas mais complexos foram apresentadas. Além disso, as bibliotecas de geração de testes ModelJUnit e JUnit utilizadas para geração de casos de teste abstratos e concretos foram abordadas.

No próximo capítulo é apresentado o procedimento sistemático de geração de testes chamado de *TestSd2Efsm* que utiliza todos os conceitos abordados nesta fundamentação teórica.

CAPÍTULO

3

PROCEDIMENTO SISTEMÁTICO *TESTSD2EFSM*

Este capítulo descreve o procedimento sistemático *TestSd2Efsm* para geração de casos de teste a partir de MEFEs extraídas de Diagramas de Sequência UML. Na Seção 3.1, uma breve explanação das etapas do procedimento é apresentada. A Seção 3.2 define os meta-modelos simplificados do Diagrama de Sequência UML e da MEFE utilizados na transformação entre modelos. Na Seção 3.3, as regras de transformação são descritas, exemplos de cada regra são ilustrados e a complexidade da transformação é discutida em termos do tamanho da MEFE gerada. A Seção 3.4 apresenta a transformação Modelo-Texto para geração das classes do SUT. A Seção 3.5 apresenta a geração automática dos testes abstratos e concretos nas bibliotecas ModelJUnit e JUnit, respectivamente. Na Seção 3.6 é apresentada a execução dos casos de teste e as métricas de coberturas geradas pelo procedimento. Por fim, na Seção 3.7 é apresentado um exemplo da aplicação do procedimento sistemático *TestSd2Efsm* em um software real.

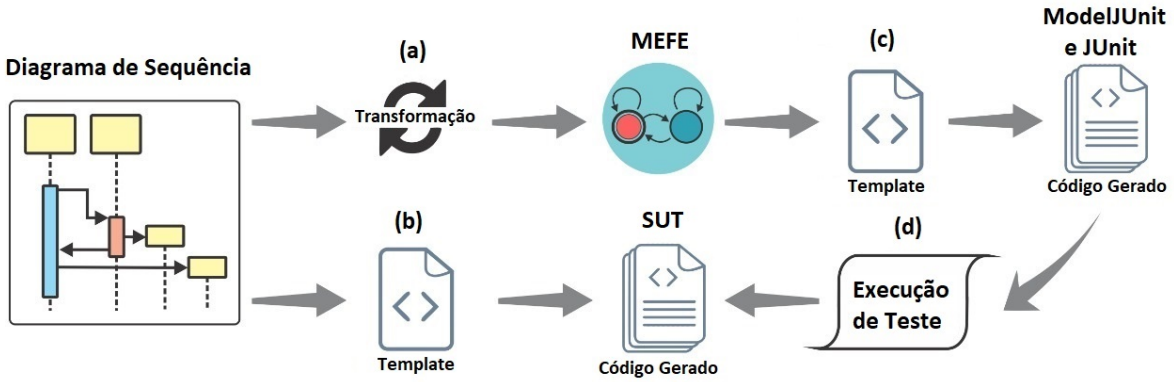
3.1 Etapas do Procedimento

O procedimento sistemático *TestSd2Efsm* ilustrado na Figura 15 é dividido em quatro principais etapas que são descritas a seguir:

(a) **Transformação entre Modelos**: cenários são especificados através do Diagrama de Sequência UML, que é transformado em uma MEFE a partir do mapeamento entre os elementos dos seus respectivos meta-modelos usando *Atlas Transformation Language* (ATL). O resultado é um modelo formal do software representado por uma MEFE.

(b) **Geração do SUT**: o código-fonte do SUT é gerado a partir do Diagrama de Sequência. Assim, uma transformação Modelo-Texto é realizada utilizando o Acceleo. O resultado desta etapa são classes do SUT na linguagem de programação Java.

Figura 15 – Procedimento sistemático *TestSd2Efsm* para geração de casos de teste.



Fonte: Elaborada pelo autor.

(c) **Geração de Casos de Teste**: os casos de teste são gerados pelos métodos de geração de teste baseados em MEFEs e pelas bibliotecas ModelJUnit e JUnit, a partir de um modelo do software representado por uma MEFE. Uma transformação Modelo-Texto com quatro módulos geradores é realizada utilizando o Acceleo. O resultado desta epata são os casos de teste abstratos no ModelJUnit e os casos de teste concretos em JUnit.

(d) **Execução dos Casos de Teste**: os casos de teste abstratos gerados pela biblioteca ModelJUnit são executados no SUT e as métricas de cobertura de estado, transição e ação são geradas automaticamente. Em seguida, os testes concretos gerados na biblioteca JUnit são executados no SUT e como resultado tem-se os vereditos dos testes.

Para todas as etapas do procedimento, um projeto foi criado e o código-fonte do protótipo está disponível no repositório[1] desta tese. No repositório, há um arquivo *README.md* explicando como executar cada etapa do procedimento.

3.2 Meta-Modelos

Os meta-modelos do Diagrama de Sequência (origem) e da MEFE (destino) definidos neste trabalho foram implementados em *Ecore* através do *Eclipse Modeling Framework* (EMF) (STEINBERG *et al.*, 2009).

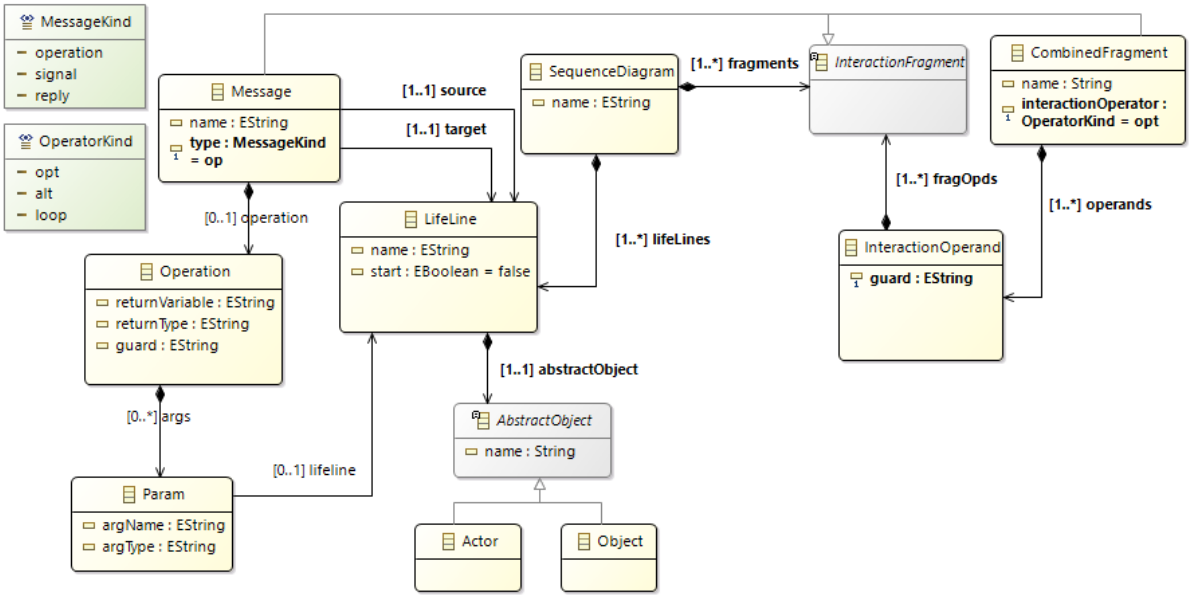
A especificação oficial completa da UML (OMG, 2017) é complexa, pois a sintaxe abstrata é representada em vários diagramas separados, o que dificulta a visualização de todas as conexões entre os elementos importantes. Além disso, a especificação usa os chamados pontos de variação semântica, ou seja, parte da semântica não é totalmente especificada para permitir o uso de UML em vários domínios. Portanto, o meta-modelo oficial da UML tem sido fortemente

[1] https://github.com/TESTSD2EFSM/TESE

criticado, pois muitos de seus elementos raramente são utilizados na prática (SEN *et al.*, 2009; FONDEMENT *et al.*, 2013).

O meta-modelo do Diagrama de Sequência mostrado na Figura 16 é mais simples que o especificado pela OMG e não possui construções que raramente são usadas na prática. A aplicação de meta-modelos simplificados tem sido amplamente relatada na literatura (GRØNMO; MØLLER-PEDERSEN, 2010; MICSKEI; WAESELYNCK, 2010; SEO *et al.*, 2016). O meta-

Figura 16 – Meta-Modelo do Diagrama de Sequência.



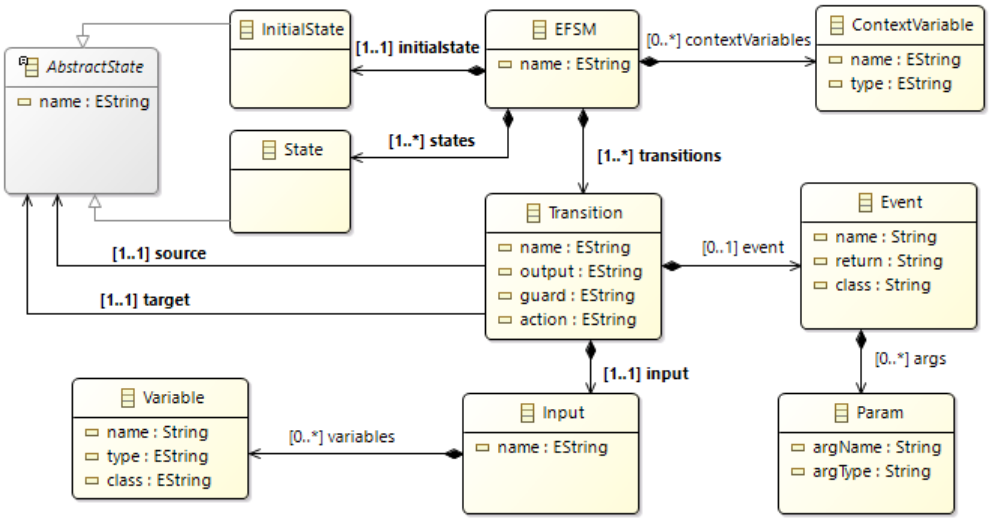
Fonte: Elaborada pelo autor.

modelo contém 11 meta-classes, dentre as quais *SequenceDiagram* representa um modelo de Diagrama de Sequência UML e fornece várias linhas de vida (*LifeLine*) e fragmentos (*InteractionFragment*). Uma linha de vida é representada por um objeto abstrato (*AbstractObject*), que pode ser um ator ou um objeto, e possui um atributo *start*, que indica a linha de vida que inicia o processo. Apenas uma linha de vida pode ter o atributo *start* = *true*. Uma mensagem (*Message*) ou um fragmento combinado (*CombinedFragment*) pode ser representado a partir da meta-classe abstrata *InteractionFragment*. A meta-classe *CombinedFragment* possui um atributo (*InteractionOperator*) que define seu tipo (por exemplo, *alt*, *opt* e *loop*). Um fragmento combinado é composto por um ou mais operandos, representados pela classe *InteractionOperand*. Cada operando tem uma condição de guarda e pode compreender fragmentos representados por mensagens ou fragmentos combinados aninhados. Uma mensagem é acionada entre linhas de vida e pode ser de três tipos: operação, sinal e retorno.

O meta-modelo proposto para a MEFE (Figura 17) é baseado na definição formal Yang *et al.* (2015), explicado na Subseção 2.3.3. É composto por dez meta-classes, e a entidade *EFSM* representa um modelo de uma MEFE. Ela é constituída por estados (*State*), transições (*Transition*) e variáveis de contexto (*ContextVariable*). As transições são compostas por entradas,

saídas, guardas, ações e podem disparar eventos (*Event*). Os eventos podem conter ou não argumentos (*Param*).

Figura 17 – Meta-Modelo da Máquina de Estados Finitos Estendida.



Fonte: Elaborada pelo autor.

Esses meta-modelos implementados em *Ecore* estão disponíveis nas pastas *SEQUENCE-DIAGRAM/model* e *EFSM/model* do repositório[1] desta tese.

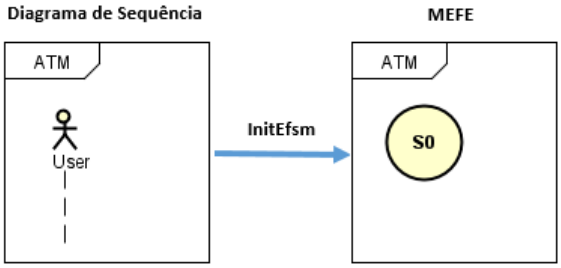
3.3 Regras de Transformação

Após definir o meta-modelo que representa a linguagem do Diagrama de Sequência ao qual a semântica formal é atribuída, bem como o meta-modelo da MEFE, a próxima etapa é definir as regras de transformação que mapeiam os elementos desses meta-modelos. As regras terão um modelo de Diagrama de Sequência como entrada e um modelo de MEFE como saída. Estas regras de transformação foram implementamos em *Atlas Transformation Language* (ATL) (BÉZIVIN; JOUAULT; TOUZET, 2005) e são utilizadas na Etapa (a) do procedimento.

Antes das regras de transformação serem abordadas especificamente, é importante observar que essas regras adicionam novos estados à MEFE de forma iterativa e os conectam aos estados adicionados anteriormente. Para esse propósito, as regras usam três variáveis definidas no módulo de transformação *SD2EFSM*: a variável *order* que representa a ordem dos estados, a variável *preState* que descreve o estado anterior e a variável *curState* que expressa o estado atual.

Para facilitar o entendimento das regras de transformação definidas neste trabalho, utilizou-se partes do Diagrama de Sequência de um *Automated Teller Machine* (ATM) (ROCHA *et al.*, 2019). A seguir, é apresentada a descrição destas regras de transformação:

Figura 18 – Regra *InitEfsm*.



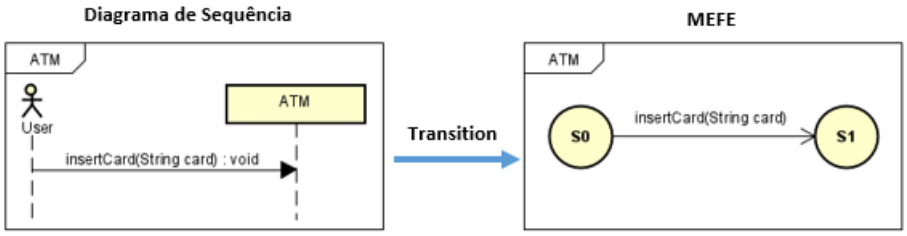
Fonte: Elaborada pelo autor.

- ***InitEfsm***: uma instância da meta-classe *SequenceDiagram* é mapeada diretamente para uma instância da meta-classe *EFSM* e recebe o mesmo nome. Além disso, quando uma linha de vida com o atributo *start* igual a *true* é encontrada, o estado inicial *S0* da MEFE é criado. Esta regra pode ser aplicada apenas uma vez. A Figura 18 apresenta um exemplo desta regra de transformação.

- ***Transition***: para todas as instâncias da meta-classe *Message* do tipo sinal (*type* = *si*) ou operação (*type* = *op*), um estado é criado e adicionado a MEFE, uma transição que conecta o estado anterior ao estado adicionado na MEFE é criada e uma instância da meta-classe *Input* é criada na transição da MEFE e rotulada com o nome da mensagem. Se o tipo da mensagem for operação, as instâncias das variáveis de entrada (*Variable*) serão criadas com o nome, o tipo do argumento da operação e a classe igual à classe da linha de vida que enviou a mensagem. Se houver um retorno em uma operação, a saída, a guarda e a ação dessa transição serão rotuladas com o retorno da operação. O evento é rotulado com o nome da operação, seu retorno, argumentos e classe iguais à linha de vida que recebe a mensagem. A Figura 19 e a Figura 20 mostram exemplos desta regra de transformação com operação sem retorno e operação com retorno, respectivamente.

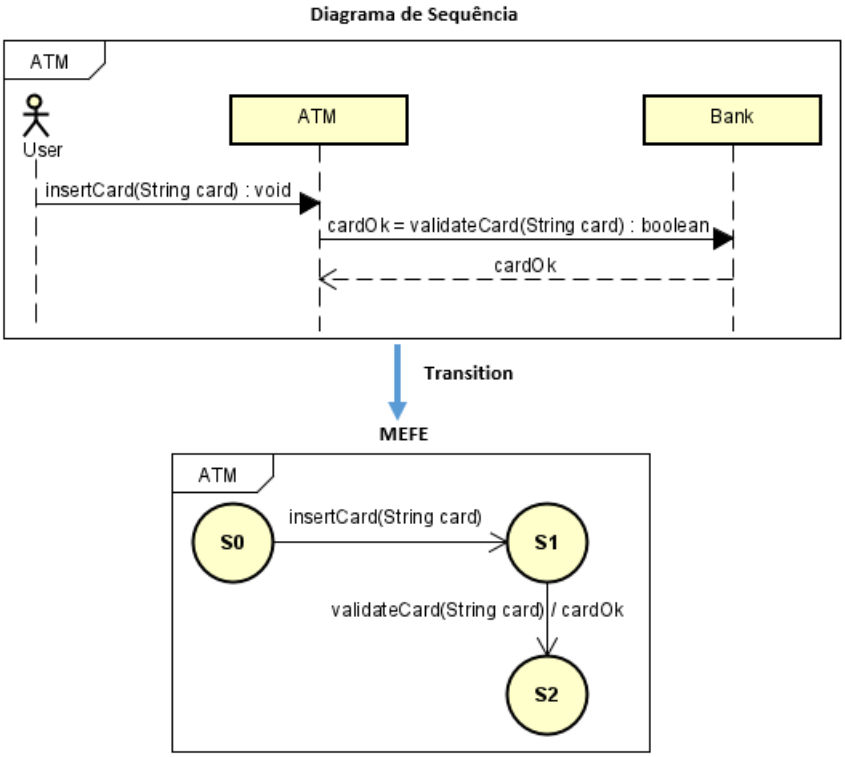
- ***ContextVariable***: para todas as instâncias da meta-classe *Message* do tipo operação (*type* = *op*) com retorno diferente de *void*, uma variável de contexto é criada na MEFE com o mesmo nome e retorno da operação da mensagem. Na Figura 20, a variável de contexto

Figura 19 – Regra *Transition* com operação sem retorno.



Fonte: Elaborada pelo autor.

Figura 20 – Regra *Transition* com operação com retorno.



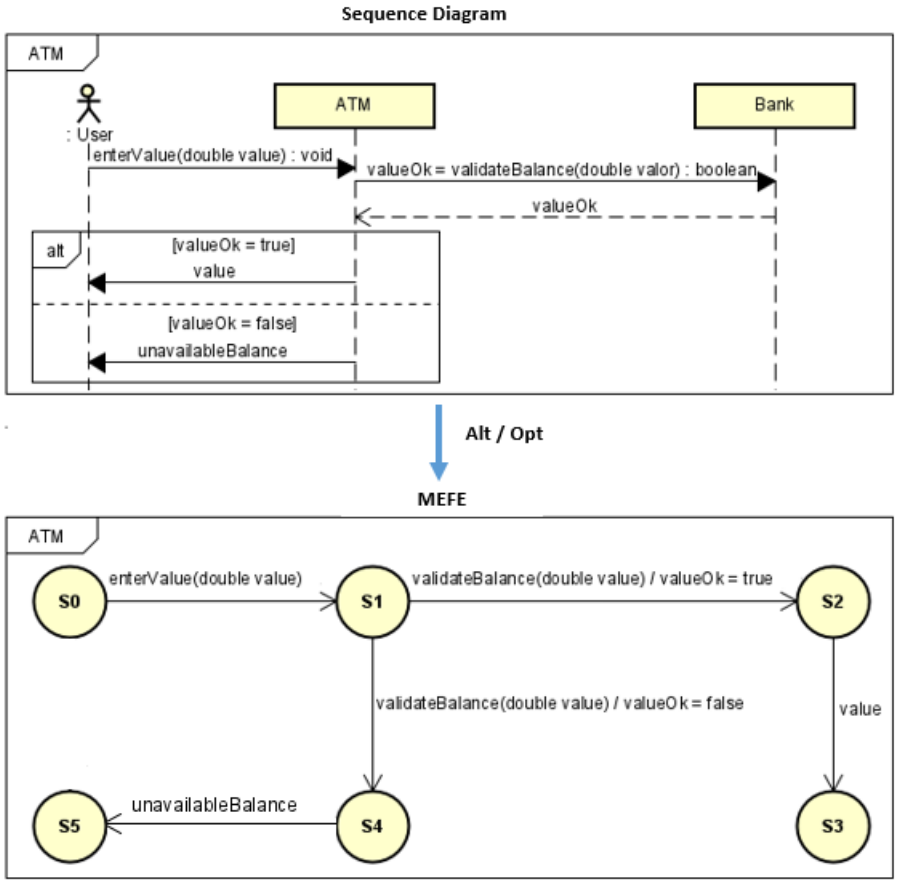
Fonte: Elaborada pelo autor.

cardOk com tipo *boolean* foi criada na MEFE.

- ***Alt*** e ***Opt***: para todas as instâncias da meta-classe *CombinedFragment* com operador de interação *alt* ou *opt*, um novo estado e uma nova transição para cada operando (com exceção ao primeiro) são criados na MEFE. As novas transições vinculam o estado atual com os novos estados. A entrada das transições será rotulada com o nome da mensagem (anterior ao fragmento combinado) e as variáveis de entrada são criadas com o nome e o tipo do argumento da operação e a classe igual à classe da linha de vida que enviou a mensagem. As saídas, guardas e ações das transições são rotuladas com a guarda do respectivo operando. O evento é rotulado com o nome da operação, seu retorno, argumentos e classe iguais à linha de vida que recebe a mensagem. A Figura 21 mostra um exemplo desta regra de transformação.

- ***Loop***: para todas as instâncias da meta-classe *CombinedFragment* com operador de interação *loop*, uma mensagem de resposta deve ser definida com o nome igual a variável utilizada na condição de guarda do operador. Quando o processo encontra uma instância dessa mensagem de resposta, um novo estado e uma nova transição que conectam o estado atual ao estado adicionado são criados na MEFE. A entrada da transição será rotulada com o nome da mensagem e a saída com a negação da guarda do operador. Outra transição é criada conectando o estado anterior ao último estado criado antes do fragmento. A entrada

Figura 21 – Regra *Alt* e *Opt*.



Fonte: Elaborada pelo autor.

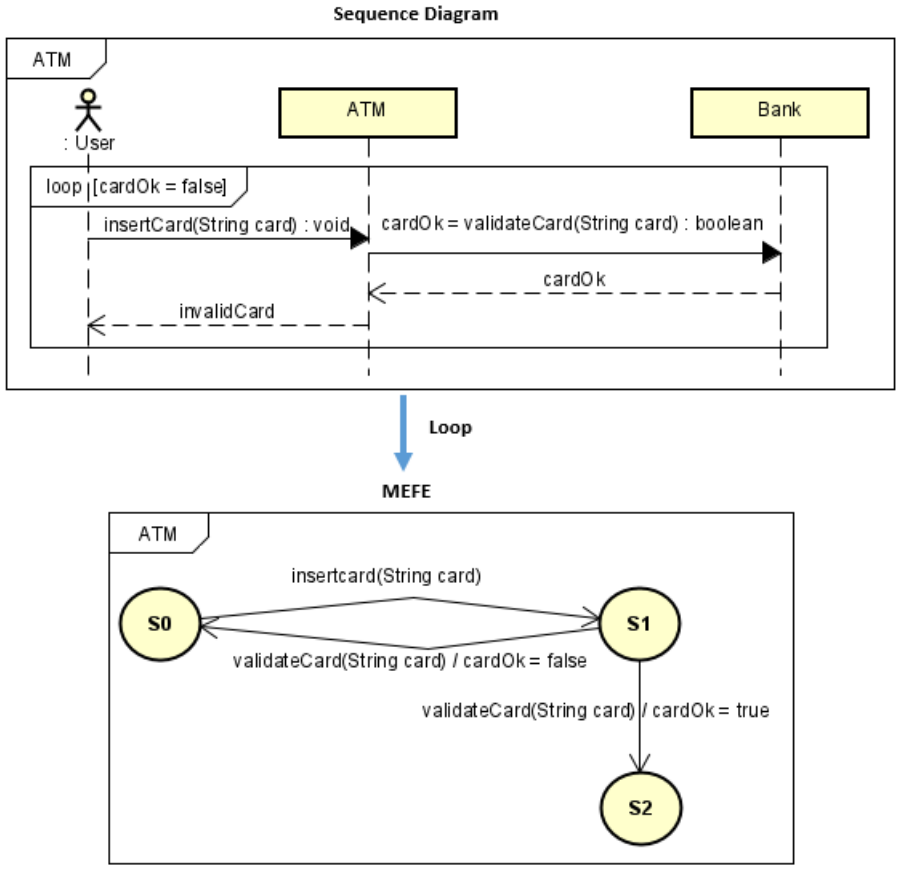
dessa transição será rotulada com o nome da mensagem e a saída com a guarda do operador. A Figura 22 mostra um exemplo desta regra de transformação.

Todas estas regras foram implementadas dentro de um regra de transformação principal chamada *SequenceDiagram2EFSM*. Esta regra recebe como entradas um modelo de Diagrama de Sequência e tem como saída um modelo de MEFE. A regra possibilita a utilização de fragmentos combinados aninhados. Quando uma instância da meta-classe *InteractionFragment* é encontrada e este é um fragmento combinado, a regra verifica se o próximo fragmento também é um fragmento combinado. Se isso ocorrer todas as regras de transformação descritas são consideradas novamente. Esta verificação foi implementada para permitir a utilização de fragmentos combinados aninhados.

Para viabilizar as regras de transformação apresentadas anteriormente, algumas regras que são acionadas por outras regras, chamadas de *Lazy Rules*, foram implementadas. A seguir, estas regras são descritas:

- ***LrInitialState***: cria o estado inicial *S0*, incrementa a ordem dos estados (variável *order*) e altera os estados anterior (variável *preState*) e atual (variável *curState*) com o estado

Figura 22 – Regra *Loop*.



Fonte: Elaborada pelo autor.

inicial *S0* criado. As variáveis *order*, *preState* e *curState* são definidas no módulo de transformação *SD2EFSM*. O Código-fonte 1 apresenta esta *Lazy Rules* em ATL.

Código-fonte 1 – Regra *LrInitialState*

```
1:  lazy rule LrInitialState {
2:    from
3:      l : SequenceDiagram!LifeLine
4:    to
5:      i : EFSM!InitialState(name <- 'S0')
6:    do {
7:      thisModule.order <- thisModule.order + 1;
8:      thisModule.preState <- i;
9:      thisModule.curState <- i;
10:   }
11: }
```

- ***LrState***: cria um novo estado, incrementa a ordem dos estados (variável *order*), o estado

anterior (variável *preState*) é alterado para o estado atual (variável *curState*) e o estado atual é alterado para o novo estado criado. As variáveis *order*, *preState* e *curState* são definidas no módulo de transformação *SD2EFSM*. O Código-fonte 2 apresenta esta *Lazy Rules* em ATL.

Código-fonte 2 – Regra *LrState*

```
1:  lazy rule LrState {
2:    from
3:      m : SequenceDiagram!Message
4:    to
5:      i : EFSM!State(name <- 'S'+thisModule.order.toString())
6:    do {
7:      thisModule.order <- thisModule.order + 1;
8:      thisModule.preState <- thisModule.curState;
9:      thisModule.curState <- i;
10:   }
11: }
```

- ***LrTransition***: cria uma transição que conecta o estado de origem (variável *source*) ao estado de destino (variável *target*). A saída, guarda e ação da transição podem ser nulas e dependem do operador e do tipo de mensagem do Diagrama de Sequência. O nome da transição é rotulado com a concatenação do estado de origem, símbolo "→"e estado destino. As variáveis *source*, *target*, *output*, *guard* e *action* são definidas no módulo de transformação *SD2EFSM*. O Código-fonte 3 apresenta esta *Lazy Rules* em ATL.

Código-fonte 3 – Regra *LrTransition*

```
1:  lazy rule LrTransition {
2:    from
3:      m : SequenceDiagram!Message
4:    to
5:      t : EFSM!Transition(output <- thisModule.output,
6:                  source <- thisModule.source,
7:                  target <- thisModule.target,
8:                  name <- thisModule.source.name+'->'
9:                  +thisModule.target.name,
10:                 guard <- thisModule.guard,
11:                 action <- thisModule.action)
12: }
```

- ***LrTransitionInput***: cria as entradas das transições rotuladas com o nome das mensagens. A variável *inputName* é definida no módulo de transformação *SD2EFSM*. O Código-fonte 4 apresenta esta *Lazy Rules* em ATL.

Código-fonte 4 – Regra *LrTransitionInput*

```
1:  lazy rule LrTransitionInput {
2:    from
3:      t : EFSM!Transition
4:    to
5:      i : EFSM!Input(name <- thisModule.inputName)
6:  }
```

- ***LrTransitionInputVar***: cria variáveis de entrada com o nome, tipo e classe dos argumentos da operação da mensagem. As variáveis *inputVarName*, *inputVarType* e *inputVarClass* são definidas no módulo de transformação *SD2EFSM*. O Código-fonte 5 apresenta esta *Lazy Rules* em ATL.

Código-fonte 5 – Regra *LrTransitionInputVar*

```
1:  lazy rule LrTransitionInputVar {
2:    from
3:      t : EFSM!Input
4:    to
5:      i : EFSM!Variable(name <- thisModule.inputVarName,
6:              type <- thisModule.inputVarType,
7:              class <- thisModule.inputVarClass)
8:  }
```

- ***LrTransitionEvent***: cria um evento com o nome e retorno da mensagem. Já a classe é igual à classe da linha de vida que recebe a mensagem. As variáveis *eventName*, *eventReturn* e *eventClass* são definidas no módulo de transformação *SD2EFSM*. O Código-fonte 6 apresenta esta *Lazy Rules* em ATL.

Código-fonte 6 – Regra *LrTransitionEvent*

```
1:  lazy rule LrTransitionEvent {
2:    from
3:      t : EFSM!Transition
```

```
4:     to
5:       i : EFSM!Event(name <- thisModule.eventName,
6:                 return <- thisModule.eventReturn,
7:                 class <- thisModule.eventClass)
8:   }
```

- ***LrTransitionEventArg***: cria os parâmetros dos eventos com nome e tipo dos argumentos das operações das mensagens. As variáveis *argName* e *argType* são definidas no módulo de transformação *SD2EFSM*. O Código-fonte 7 apresenta esta *Lazy Rules* em ATL.

Código-fonte 7 – Regra *LrTransitionEventArg*

```
1:   lazy rule LrTransitionEventArg {
2:     from
3:       t : EFSM!Event
4:     to
5:       i : EFSM!Param(argName <- thisModule.argName,
6:             argType <- thisModule.argType)
7:   }
```

- ***LrContextVariable***: cria uma variável de contexto com o nome e o tipo rotulados com a variável de retorno da operação e o tipo da operação, respectivamente. As variáveis *returnVariable* e *returnType* são definidas no módulo de transformação *SD2EFSM*. O Código-fonte 8 apresenta esta *Lazy Rules* em ATL.

Código-fonte 8 – Regra *LrContextVariable*

```
1:   lazy rule LrContextVariable {
2:     from
3:       o : SequenceDiagram!Operation
4:     to
5:       v : EFSM!ContextVariable(name <- o.returnVariable,
6:                   type <- o.returnType)
7:   }
```

Todas essas regras implementadas em ATL estão disponíveis no arquivo *SD2EFSM/SequenceDiagram2EFSM.atl* do repositório[1] desta tese.

3.3.1 Complexidade da Transformação

Nesta seção discute-se como o tamanho dos Diagramas de Sequência influencia o tamanho das MEFEs geradas. O tamanho das MEFEs é medido em termos de quantidade de estados e transições. Essa contagem será realizada comparando os elementos dos Diagramas de Sequência em relação à criação de estados e transições nas MEFEs. A Tabela 3 mostra a relação entre os elementos dos Diagramas de Sequência e a quantidade de estados e transições gerados nas MEFEs. Observa-se que em duas linhas da tabela, a quantidade de estados e transições não é representado diretamente por um número. Esses casos são detalhados a seguir:

Tabela 3 – Impacto dos elementos dos Diagramas de Sequência no tamanho das MEFEs.

Diagrama de Sequência	Quantidade de Estados	Quantidade de Transições
Linha de Vida (*start = true*)	1	0
Mensagem (*type = op / si*)	1	1
Fragmento Combinado (Operador *opt*)	0	Quantidade de Operandos no último nível
Fragmento Combinado (Operador *alt*)	Quantidade de Operandos - 1	Quantidade de Operandos - 1 + Quantidade de Operandos no último nível
Fragmento Combinado (Operador *loop*)	0	1
Total	Soma dos Valores	Soma dos Valores

Fonte: Elaborada pelo autor.

- Se o elemento do Diagrama de Sequência for um fragmento combinado com operador *opt* e houver uma mensagem após o fragmento combinado, a quantidade de transições correspondente da MEFE será igual a quantidade de operandos no último nível.

- Se o elemento do Diagrama de Sequência for um fragmento combinado com um operador *alt*, a quantidade de transições correspondente da MEFE será igual a quantidade de operandos menos 1. Se houver uma mensagem após o fragmento combinado, será adicionado na quantidade de transições, a quantidade de operandos no último nível. A quantidade de estados é igual ao número de operandos menos 1.

- A quantidade total de estados e transições da MEFE é obtida através da soma do resultado da coluna "Quantidade de Estados"e "Quantidade de Transições"da Tabela 3, respectivamente.

3.4 Geração do SUT

O procedimento sistemático *TestSd2Efsm* gera o código-fonte do SUT automaticamente para uma automatização completa do processo de teste. Nesta Etapa (b), todas as classes do SUT com seus respectivos atributos e operações são geradas a partir do Diagrama de Sequência.

O SUT é gerado automaticamente pela transformação Modelo-Texto usando Acceleo (ACCELEO, 2018). Foi implementado o módulo gerador *generateSUT*, cuja entrada é um modelo de Diagrama de Sequência criado pelo editor implementado em EMF e consequentemente em conformidade com o meta-modelo abordado na Seção 3.2. Logo, o mesmo modelo do Diagrama de Sequência implementado na Etapa (a) do procedimento será utilizado nesta etapa. A criação de cada elemento das classes do SUT é explicada a seguir:

- **Classes**: uma classe do SUT é criada para cada instância da meta-classe *LifeLine* do Diagrama de Sequência. O nome da classe é o nome da linha da vida.
- **Atributos**: os atributos das classes são obtidos dos parâmetros das operações das mensagens com retorno diferente de *void*. Os parâmetros são selecionados em duas situações:

  1. Nome da linha de vida do parâmetro (que indica qual linha de vida insere valores de argumento de parâmetro - passagem de bastão) igual ao nome da classe criada;
  2. Nome da linha de vida de destino da mensagem igual ao nome da classe criada e o parâmetro seja referenciado na guarda da operação.

  Nesses dois casos, o nome do atributo e seu tipo são obtidos, respectivamente, dos tipos e nomes dos argumentos da operação da mensagem do Diagrama de Sequência.

  Outros atributos são obtidos de mensagens com retornos diferentes de nulo, que são do tipo GET (nome começando com *get*) e cujo nome da linha de vida de destino é o mesmo da classe criada. Nesse cenário, o nome do atributo e seu tipo são obtidos, respectivamente, da variável de retorno e do tipo de retorno da operação de mensagem do Diagrama de Sequência.

- **Operações**: as operações das classes do SUT são obtidas de três formas:

  1. para cada atributo criado na classe, uma operação GET é criada sob as mesmas condições da criação dos atributos da classe. O corpo da operação será o retorno do atributo definido na classe criada.
  2. Outras operações também são obtidas de mensagens com retornos diferentes de nulo, que são do tipo GET (nome começando com *get*), e o nome da linha de vida de destino é o mesmo que a classe criada. Nesse cenário, o nome dessa operação é igual à mensagem, o retorno da operação é do mesmo tipo que o retorno da operação da mensagem e os parâmetros da operação são os mesmos que os dos argumentos da

mensagem Operação. O corpo da operação será o retorno da variável de retorno da operação de mensagem.

3. Para todas as mensagens cujo nome da linha de vida de destino é igual ao nome da classe, uma nova operação é criada. O nome desta operação é o mesmo da mensagem, o retorno da operação é do mesmo tipo que a operação da mensagem e os parâmetros da operação são os mesmos que os argumentos da operação da mensagem. Se a guarda da operação da mensagem estiver preenchida, o corpo da operação será criado com o teste condicional da guarda da operação da mensagem. Se o retorno da operação da mensagem for booleano, o teste condicional retornará verdadeiro e falso. Caso contrário, o retorno do teste condicional será a variável de retorno.

O código do gerador *generateSUT.mtl* implementado em Acceleo está disponível na pasta *Sd2SUT/src/ Sd2SUT/main/* do repositório[1] desta tese.

3.5 Geração de Casos de Teste

O procedimento usa as bibliotecas ModelJUnit e JUnit para a geração de casos de teste, uma vez que são bibliotecas de código-aberto e seus usos são simples para programadores Java. Além disso, o ModelJUnit permite a implementação de modelos formais amplamente usados (por exemplo, MEFE) em TBM e fornece uma variedade de algoritmos de geração de casos de teste, recursos de visualização de modelo, estatísticas de cobertura de modelo e outros recursos (UTTING, 2012).

A Etapa (c) do procedimento que implementa um ambiente de TBM usando ModelJUnit e JUnit é realizada em quatro passos:

1. **Criação do Modelo**: inicialmente, implementa-se a interface *FsmModel* que especifica a MEFE gerada na biblioteca ModelJUnit e define todos os estados da máquina utilizando a variável do tipo enumeração *State*. Para cada variável de contexto da MEFE, um atributo com nome e tipo da variável é definido na classe. As entradas das transições da MEFE são definidas por métodos com anotação *@Action* que representam as transições que ligam os estados. Além disso, são definidos os métodos *getState* que retorna o estado atual e *reset* que leva a máquina ao estado inicial.

2. **Criação do Adaptador**: a classe *adapter* permite que o modelo da MEFE se comunique e controle o SUT. Para cada entrada da MEFE gerada, um método com o mesmo nome é adicionado na classe do adaptador. Estes métodos retornam a chamada dos métodos originais das classes do SUT. Portanto, na classe do adaptador é necessário instanciar um objeto para cada classe do SUT.

3. **Geração de testes**: inicialmente, instancia-se o modelo da MEFE definido no Passo 1 desta etapa do procedimento. Em seguida, a estratégia de teste é escolhida. Conforme explanado na Subseção 2.3.4, a biblioteca ModelJUnit oferece várias estratégias de teste. Neste procedimento, utilizou-se *LookaheadTester*, uma vez que é um algoritmo sofisticado, que pode cobrir todos os estados e transições da MEFE rapidamente (UTTING, 2012). Depois da instanciação do objeto da classe *LookaheadTester*, utiliza-se o método *buildGraph* para geração das sequências de teste e das métricas de cobertura de estados, transições e ações.

4. **Concretização de testes**: os casos de teste são concretizados na linguagem de programação Java, na biblioteca JUnit. Para cada transição que aciona um evento com retorno diferente de *void*, é criado um caso de teste com valores válidos para condição de guarda. Como a MEFE gerada na Etapa (a) do procedimento tem uma transição para cada condição de guarda que especifica estados válidos ou inválidos para as entradas do evento, todo o domínio de entrada é coberto pelos casos de teste gerados. Dessa forma, o objetivo é gerar casos de teste concretos para todos os caminhos da MEFE.

Esses quatro passos são gerados automaticamente pela transformação Modelo-Texto implementada em Acceleo. Foram implementados os módulos geradores *generateClassModel*, *generateClassAdapter*, *generateClassTest* e *generateClassJUnit*. A entrada dos módulos é a MEFE gerada na Etapa (a) do procedimento. Os geradores de código implementados em Acceleo estão disponíveis na pasta *Efsm2/ ModelJUnit/src/Common/* do repositório[1] desta tese.

3.6 Execução dos Casos de Teste

Nesta Etapa (d), usando o Eclipse Modeling Framework (EMF), os casos de teste são executados a partir das classes *ClassTest* e *ClassJUnit* descritas nos Passos 3 (geração de testes abstratos) e 4 (geração de testes concretos) da seção anterior.

Na execução dos casos de teste abstratos, além das sequências de teste, a biblioteca ModelJUnit também gera métricas de cobertura de estado, transição e ação. A seguir, uma breve explicação dessas métricas de cobertura:

- **Cobertura de Estado**: mostra a comparação entre o número de estados cobertos pelo número de estados definidos no modelo.

- **Cobertura de Transição**: mostra a comparação entre o número de transições executadas pelo número total de transições definidas no modelo.

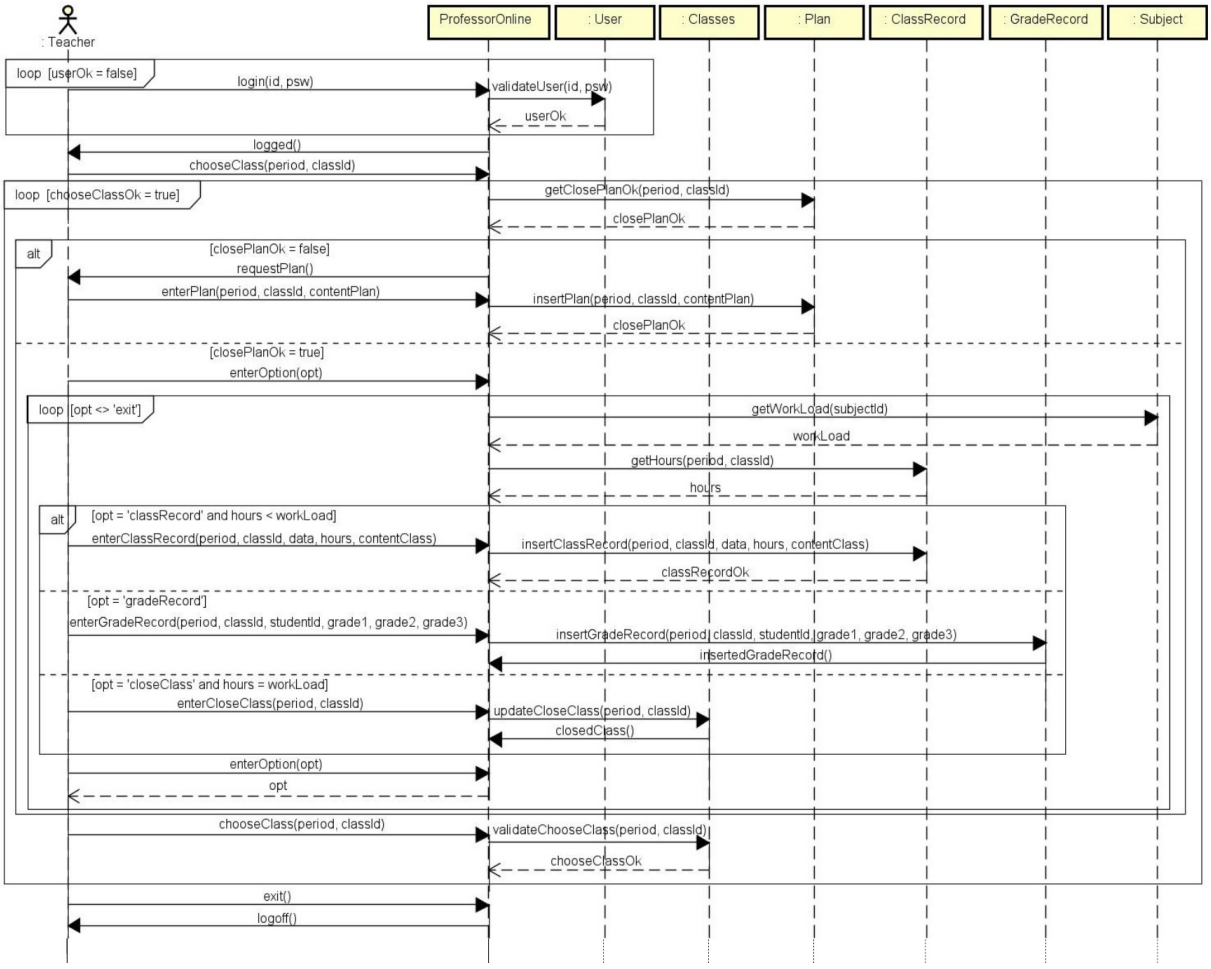
- **Cobertura de Ação**: mostra a comparação entre o número de ações executadas pelo número total de ações definidas no modelo.

Na execução dos casos de teste concretos, a biblioteca JUnit gera os vereditos da execução dos testes, mostrando os casos de teste que encontraram falhas.

3.7 Exemplo de Aplicabilidade do Procedimento

Esta seção descreve a execução do procedimento em um software real de Registro de Diário *Online* do Professor da Universidade Estadual do Piauí (UESPI) denominado Professor Online[2] para verificar a aplicabilidade do mesmo. Este sistema tem as seguintes funcionalidades:

Figura 23 – Diagrama de Sequência UML do sistema Professor Online.



Fonte: Elaborada pelo autor.

- **Validação do Usuário**: quando o professor acessa o sistema são solicitadas suas credenciais (*id*, *psw*). Após validar as credenciais, o professor pode acessar as demais funcionalidades do sistema.

[2] http://sistemas4.uespi.br/profonline/

- **Minhas Turmas**: permite ao professor escolher a turma a ser atualizada. Após a escolha, todos os demais recursos são relacionados a esta turma.

- **Plano de Disciplina**: permite que o professor preencha o plano de disciplina e depois feche-o. Todos os outros recursos estarão disponíveis ao professor somente após o fechamento do plano de disciplina.

- **Lançamento de Aulas**: permite ao professor inserir suas aulas até que a carga horária da disciplina seja concluída.

- **Lançamento de Notas**: permite ao professor digitar as notas das avaliações dos alunos.

- **Fechamento do Diário**: o professor só pode fechar o diário se todas as informações tiverem sido postadas. Se o diário for fechado, todos os outros recursos estarão indisponíveis.

O Diagrama de Sequência UML da Figura 23 apresenta os cenários e interações do sistema Professor Online.

3.7.1 Transformação entre Modelos

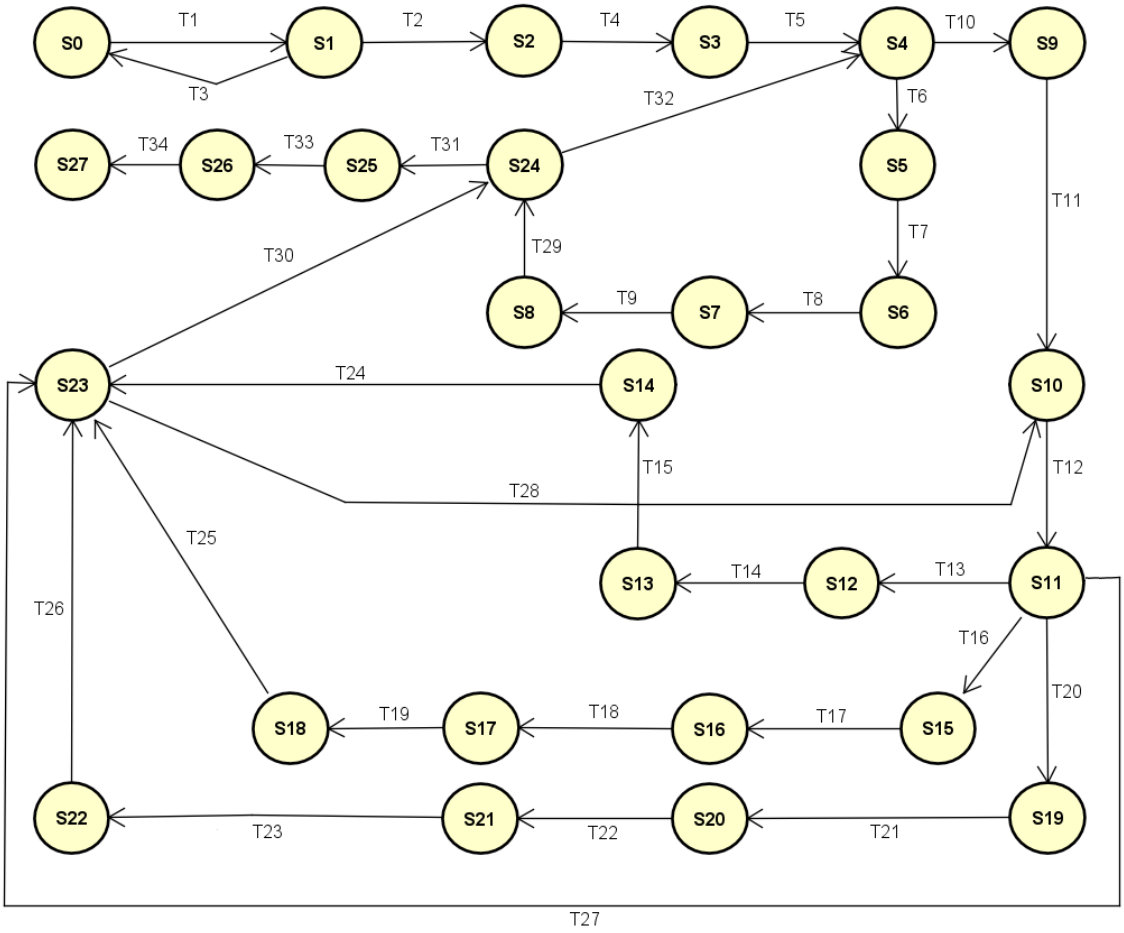
Inicialmente, foi criado o modelo do Diagrama de Sequência ilustrado na Figura 23 usando o editor de Diagrama de Sequência implementado no EMF. Esse modelo representa a especificação dos cenários das funcionalidades do sistema Professor Online descritas no início desta seção. Em seguida, usando as regras de transformação implementadas em ATL descritas na Seção 3.3, o Diagrama de Sequência UML foi convertido em uma Máquina de Estados Finitos Estendida. Ao final da execução das regras de transformação foi gerada uma MEFE conforme mostrada na Figura 24. Para facilitar a visualização e entendimento das transições, foi apresentado na Tabela 5 o detalhamento das informações das transições da MEFE. Esta transformação do modelo corresponde à Etapa (a) do procedimento.

Conforme discutido na Subseção 3.3.1, o tamanho do Diagrama de Sequência influencia no tamanho da MEFE gerada. Seguindo as orientações contidas na Tabela 3, ao qual relaciona os elementos do Diagrama de Sequência com a quantidade de estados e transições da MEFE, a Tabela 4 apresenta essa relação entre o Diagrama de Sequência e a MEFE do sistema Professor Online.

Os modelos referentes ao Diagrama de Sequência gerado utilizando o editor de modelos e a MEFE gerada no processo de transformação estão disponíveis na pasta *SD2EFSM* do repositório[3] de uma publicação resultante desta tese de doutorado.

[3] https://github.com/TESTSD2EFSM/SQJO

Figura 24 – Máquina de Estados Finitos Estendida do sistema Professor Online.



Fonte: Elaborada pelo autor.

Tabela 4 – Tamanho da MEFE do sistema Professor Online

Diagrama de Sequência	Quantidade de Estados	Quantidade de Transições
Linha de Vida (*start = true*)	1	0
Mensagem (*type = op / si*)	24	24
Fragmento Combinado (Operador *alt*)	3	7
Fragmento Combinado (Operador *loop*)	0	3
Total	28	34

Fonte: Elaborada pelo autor.

Tabela 5 – Detalhes das transições da MEFE do sistema Professor Online.

Id	Nome	Entrada	Saída
T1	S0→S1	login(id, psw)	
T2	S1→S2	validateUser(id, psw)	not (userOk = false)
T3	S1→S0	validateUser(id, psw)	userOk = false
T4	S2→S3	logged	
T5	S3→S4	chooseClass(period, classId)	
T6	S4→S5	getClosePlanOk(period, classId)	closePlanOk = false
T7	S5→S6	requestPlan	
T8	S6→S7	enterPlan(period, classId, contentPlan)	
T9	S7→S8	insertPlan(period, classId, contentPlan)	closePlanOk
T10	S4→S9	getClosePlanOk(period, classId)	closePlanOk = true
T11	S9→S10	enterOption(opt)	
T12	S10→S11	getWorkLoad(subjectId)	workLoad
T13	S11→S12	getHours(period, classId)	opt = “classRecord” and hours < workLoad
T14	S12→S13	enterClassRecord(period, classId, date, hours, contentClass)	
T15	S13→S14	insertClassRecord(period, classId, date, hours, contentClass)	classRecordOk
T16	S11→S15	getHours(period, classId)	opt = “gradeRecord”
T17	S15→S16	enterGradeRecord(period, classId, studentId, grade1, grade2,grade3)	
T18	S16→S17	insertGradeRecord(period, classId, studentId, grade1, grade2,grade3)	
T19	S17→S18	insertedGradeRecord	
T20	S11→S19	getHours(period, classId)	opt = “closeClass” and hours = workLoad
T21	S19→S20	insertCloseClass(period, classId)	
T22	S20→S21	updateCloseClass(period, classId)	
T23	S21→S22	closedClass	
T24	S14→S23	enterOption(opt)	not (opt <> “exit”)
T25	S18→S23	enterOption(opt)	not (opt <> “exit”)
T26	S22→S23	enterOption(opt)	not (opt <> “exit”)
T27	S11→S23	enterOption(opt)	not (opt <> “exit”)
T28	S23→S10	enterOption(opt)	opt <> “exit”
T29	S8→S24	chooseClass(period, classId)	
T30	S23→S24	chooseClass(period, classId)	
T31	S24→S25	validateChooseClass(period, classId)	not (chooseClassOk = true)
T32	S24→S4	validateChooseClass(period, classId)	chooseClassOk = true
T33	S25→S26	exit	
T34	S26→S27	loggof	

3.7.2 Geração do SUT

Na Etapa (b), as classes (*Teacher*, *ProfessorOnline*, *User*, *Class*, *Plan*, *ClassRecord*, *GradeRecord* e *Subject*) com seus respectivos atributos e operações são geradas automaticamente

a partir do modelo do Diagrama de Sequência ilustrado Figura 23 pelo módulo gerador *generate-SUT* implementado em Acceleo. A seguir, como exemplo das classes geradas, é apresentado o Código-fonte 9 da classe *User*.

Código-fonte 9 – Classe *User* do SUT

```
 1: public class User {
 2:   private String id = "111";
 3:   private String psw = "123";
 4:
 5:   public String getId() {
 6:     return id;
 7:   }
 8:
 9:   public String getPsw() {
10:     return psw;
11:   }
12:
13:   public boolean validateUser(String id, String psw) {
14:     if (this.id.equals(id) && this.psw.equals(psw)){
15:       return true;
16:     }
17:     else{
18:       return false;
19:     }
20:   }
21: }
```

Todos os códigos-fonte das classes do SUT gerados na linguagem de programação Java estão disponíveis na pasta *Sd2SUT/Files* do repositório[3] de uma publicação resultante desta tese de doutorado.

3.7.3 Geração de Casos de Teste

Na Etapa (c) do procedimento, os casos de teste são gerados a partir da MEFE ilustrada na Figura 24 extraída na Etapa (a). As classes (*ProfessorOnlineModel*, *ProfessorOnlineAdapter*, *ProfessorOnlineTest* e *ProfessorOnlineJUnit*) foram geradas automaticamente pelos módulos geradores *generateClassModel*, *generateClassAdapter*, *generateClassTest* e *generateClassJUnit* implementados em Acceleo.

A classe *ProfessorOnlineModel* é uma implementação da interface *FsmModel*. Um objeto chamado *adapter*, instanciado da classe *ProfessorOnlineAdapter*, uma variável enumeração *State*

que representa todos os estados (S0, ..., S27) da MEFE, todas as variáveis de contexto da MEFE, um método *getState*, um método de *reset* e métodos com anotação *@Action* foram definidos neste classe. A seguir é apresentado o Código-fonte 10 que representa um trecho de código da classe *ProfessorOnlineModel* gerado automaticamente.

Código-fonte 10 – Trecho de código da classe *ProfessorOnlineModel*

```
1: import nz.ac.waikato.modeljunit.Action;
2: import nz.ac.waikato.modeljunit.FsmModel;
3:
4: public class ProfessorOnlineModel implements FsmModel {
5:   private ProfessorOnlineAdapter adapter = new
      ProfessorOnlineAdapter();
6:
7:   private boolean userOk;
8:   private boolean closePlanOk;
9:   private int workLoad;
10:  private int hours;
11:  private boolean classRecordOk;
12:  private String opt;
13:  private boolean chooseClassOk;
14:
15:  private enum State{S0, S1, S2, S3, S4, S5, S6, S7, S8, S9,
      S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21,
      S22, S23, S24, S25, S26, S27}
16:  private State state = State.S0;
17:
18:  public Object getState() {
19:    return state;
20:  }
21:
22:  public void reset(boolean arg0) {
23:    state = State.S0;
24:  }
25:
26:  @Action
27:  public void login() {
28:    if (state == State.S0) {
29:      adapter.login();
30:      state = State.S1;
31:    }
32:  }
```

```
33:    ...
34: }
```

Na classe *ProfessorOnlineAdapter* foram instanciados objetos do tipo *Teacher*, *ProfessorOnline*, *User*, *Plan*, *Subject*, *ClassRecord*, *GradeRecord* e *Classes* que pertencem ao SUT. Também foi criado um método para cada evento acionado nas transições da MEFE. Estes métodos promovem a comunicação do modelo com o SUT. A seguir é apresentado o Código-fonte 11 que representa um trecho de código da classe *ProfessorOnlineAdapter* gerado automaticamente.

Código-fonte 11 – Trecho de código da classe *ProfessorOnlineAdapter*

```
 1: public class ProfessorOnlineAdapter {
 2:    private Teacher teacher = new Teacher();
 3:    private ProfessorOnline professorOnline = new ProfessorOnline
       ();
 4:    private User user = new User();
 5:    private Plan plan = new Plan();
 6:    private Subject subject = new Subject();
 7:    private ClassRecord classRecord = new ClassRecord();
 8:    private GradeRecord gradeRecord = new GradeRecord();
 9:    private Classes classes = new Classes();
10:
11:    public void login(){
12:    }
13:
14:    public boolean validateUser(){
15:      return user.validateUser(teacher.getId(),teacher.getPsw());
16:    }
17:
18:    ...
19: }
```

Na classe *ProfessorOnlineTest* foram instanciados objetos do tipo ProfessorOnlineModel e LookaheadTester que representam o modelo da MEFE e a estratégia utilizada nos testes, respectivamente. Para percorrer todas as transições do modelo, o algoritmo foi configurado para gerar uma sequência de 70 etapas de teste. Além disso, os métodos de geração das métricas de coberturas de estado, transição e ações foram acionados. A seguir é apresentado o Código-fonte 12 da classe *ProfessorOnlineTest* gerado automaticamente.

Código-fonte 12 – Classe *ProfessorOnlineTest*

```
1: import org.junit.Test;
```

```
 2:
 3: import nz.ac.waikato.modeljunit.LookaheadTester;
 4: import nz.ac.waikato.modeljunit.StopOnFailureListener;
 5: import nz.ac.waikato.modeljunit.Tester;
 6: import nz.ac.waikato.modeljunit.VerboseListener;
 7: import nz.ac.waikato.modeljunit.coverage.ActionCoverage;
 8: import nz.ac.waikato.modeljunit.coverage.StateCoverage;
 9: import nz.ac.waikato.modeljunit.coverage.TransitionCoverage;
10:
11: public class ProfessorOnlineTest {
12:   @Test
13:   public void testProfessorOnline() throws Exception{
14:     ProfessorOnlineModel ProfessorOnlineModel = new
      ProfessorOnlineModel();
15:     Tester tester = new LookaheadTester(ProfessorOnlineModel);
16:
17:     tester.buildGraph();
18:     tester.addListener(new VerboseListener());
19:     tester.addListener(new StopOnFailureListener());
20:     tester.addCoverageMetric(new TransitionCoverage());
21:     tester.addCoverageMetric(new StateCoverage());
22:     tester.addCoverageMetric(new ActionCoverage());
23:
24:     tester.generate(70);
25:     tester.printCoverage();
26:   }
27: }
```

Para geração e execução dos testes é necessário configurar os atributos das classes do SUT. Os seguintes valores foram atribuídos aos atributos das classes:

- ***User***: *id* = “111” e *psw* = “123”.
- ***Classes***: *period* = “20192” e *classId* = “1”.
- ***Plan***: *period* = “20192”, *classId* = “1” e *closePlanOk* = *false* ou *closePlanOk* = *true*.
- ***Subject***: *subjectId* = “10” e *workLoad* = 60.
- ***ClassRecord***: *period* = “20192”, *classId* = “1” e *hours* = 30 ou *hours* = 60.

O procedimento gerou casos de teste para exercitar todos os caminhos da MEFE, conforme mostrado na Tabela 6. Um caso de teste foi gerado para cada domínio. Por exemplo, os

Tabela 6 – Casos de teste gerados

T	id	psw	period	class Id	close PlanOk	cont. Plan	subj. Id	opt	hours	cont. Class
1	111	123								
2	222	246								
3	111	123	20192	1	true					
4	111	123	20192	2	false					
5	111	123	20192	1	false	aaa				
6	111	123	20192	1	false	null				
7	111	123	20192	1	true		10		60	
8	111	123	20192	1	true		10		30	
9	111	123	20192	1	true		10	class Record	60	
10	111	123	20192	1	true		10	grade Record	60	
11	111	123	20192	1	true		10	close Class	60	
12	111	123	20192	1	true		10	exit	60	
13	111	123	20192	1	true		10	class Record	30	aaa
14	111	123	20192	1	true		10	class Record	30	null
15	111	123	20192	1				exit		
16	111	123	20192	2				exit		

atributos *id* e *psw* da classe *User* foram configurados com os valores *111* e *123*, respectivamente. Em seguida, o procedimento gerou um caso de teste com valores *id = 111* e *psw = 123* e outro caso de teste com os valores *id = 222* e *psw* = 246, e ambos os cenários serão testados. Em seguida é apresentado o Código-fonte 13 que representa um trecho de código da classe *ProfessorOnlineJUnit* gerada automaticamente com a concretização dos casos de teste na linguagem de programação Java, no framework JUnit.

Código-fonte 13 – Trecho de código da classe *ProfessorOnlineJUnit*

```
1: import static org.junit.Assert.*;
2:
3: import java.util.Date;
4:
5: import org.junit.Test;
6:
7: public class ProfessorOnlineJUnit {
8:
9:   @Test
10:   public void testValidateUser01() {
```

```
11:     User user = new User();
12:     boolean output = user.validateUser("111","123");
13:     assertTrue(output);
14:   }
15:
16:   @Test
17:   public void testValidateUser02() {
18:     User user = new User();
19:     boolean output = user.validateUser("222","246");
20:     assertFalse(output);
21:   }
22:
23:   ...
24:
25: }
```

Essas classes Java (*ProfessorOnlineModel*, *ProfessorOnlineAdapter*, *ProfessorOnlineTest* e *ProfessorOnlineJUnit*) com o código-fonte completo estão disponíveis na pasta *Efsm2Model JUnit/Files* do repositório[3] de uma publicação resultante desta tese de doutorado.

3.7.4 Execução dos Casos de Teste

Na Etapa (d) do procedimento, usando o *Eclipse Modeling Framework* (EMF), os casos de teste são executados usando a classe *ProfessorOnlineTest* e a classe *ProfessorOnlineJUnit*.

O algoritmo *LookaheadTester* testa todas as ações possíveis executando a classe *ProfessorOnlineTest*. Para cada caso de teste, são geradas métricas de cobertura de estados, transições e ações, conforme mostrado na Tabela 7. As métricas de cobertura de ações correspondem ao número de ações realizadas. Como o algoritmo usado testa todas as possibilidades, a cobertura de ação é igual ao número de métodos com anotação *@Action*. A métrica de cobertura de estado corresponde ao número de estados visitados e a métrica de cobertura de transição indica o número de transições acionadas. Com a execução de todos os casos de teste, todos os estados e transições da MEFE foram cobertos.

A execução de *ProfessorOnlineJUnit* produziu os resultados esperados para cada subconjunto do domínio de entrada, e nenhuma falha foi encontrada no SUT, conforme ilustrado na Figura 25. Em seguida, para testar a eficácia dos casos de teste concretos gerados, uma nova versão modificada do SUT foi criada, inserindo erros manualmente nas classes *User*, *Plan*, *Classes* e *ClassRecord*, através da alteração dos testes condicionais de operações das referidas classes. Novamente a classe *ProfessorOnlineJUnit* foi executada e produziu os resultados esperados, e as falhas foram identificadas no resultado de seis casos de teste, conforme ilustrado na Figura 26.

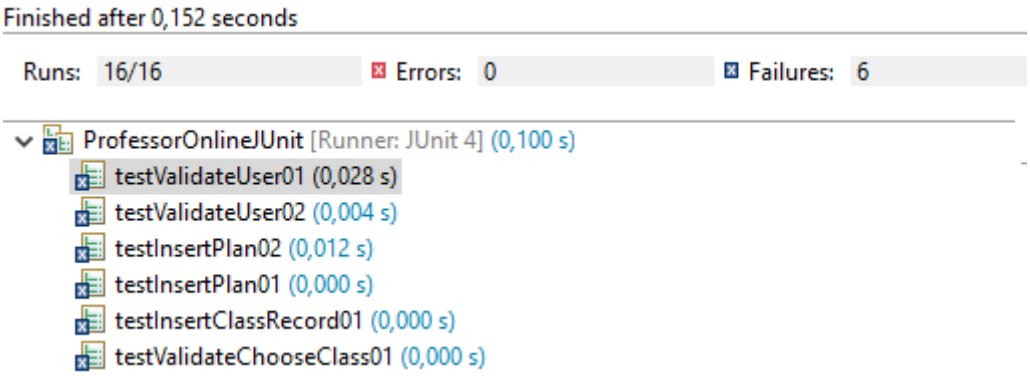
Tabela 7 – Métricas de cobertura geradas

T	Action Coverage	State Coverage	Transition Coverage
1	22/22	2/28	2/34
2	22/22	4/28	3/34
3	22/22	6/28	5/34
4	22/22	7/28	6/34
5	22/22	9/28	8/34
6	22/22	9/28	8/34
7	22/22	8/28	7/34
8	22/22	8/28	7/34
9	22/22	9/28	8/34
10	22/22	9/28	8/34
11	22/22	9/28	8/34
12	22/22	14/28	13/34
13	22/22	11/28	10/34
14	22/22	11/28	10/34
15	22/22	11/28	10/34
16	22/22	13/28	12/34
Total	22/22	28/28	34/34

Figura 25 – Resultado da execução dos casos de teste no SUT gerado na Etapa (b) do procedimento.

Fonte: Elaborada pelo autor.

Figura 26 – Resultado da execução dos casos de teste na versão do SUT com defeitos.



Fonte: Elaborada pelo autor.

Para execução dos casos de teste, foi criado um projeto na pasta *MODELJUNIT* do

repositório[3] de uma publicação resultante desta tese de doutorado.

3.8 Considerações Finais

Este capítulo descreveu o procedimento sistemático *TestSd2Efsm* para geração de casos de teste abstratos e concretos a partir de MEFEs extraídas de Diagramas de Sequência UML. Como o procedimento proposto utiliza transformação de modelos, os meta-modelos simplificados para o Diagrama de Sequência e MEFE foram definidos no EMF. Para o mapeamento dos elementos dos modelos de origem e destino envolvidos na transformação, regras de transformação foram definidas e implementadas na linguagem ATL. Além disso, o procedimento apresentou uma transformação Modelo-Texto para geração do código-fonte do SUT a partir do Diagrama de Sequência e outra para geração dos casos de teste abstratos e concretos a partir da MEFE nas bibliotecas ModelJUnit e JUnit, respectivamente.

O capítulo também apresentou a execução do procedimento sistemático *TestSd2Efsm* em um software real para verificar a aplicabilidade do mesmo. Esta execução foi descrita e mostrou detalhadamente a aplicação de todas as etapas do procedimento sistemático *TestSd2Efsm* proposto. O exemplo de aplicabilidade apresentado mostrou a viabilidade da utilização do procedimento sistemático de geração de testes em softwares reais. Ainda que o exemplo apresentado tenha mostrado sua viabilidade, mais resultados são necessários para avaliar o procedimento sistemático proposto. Dessa forma, no próximo capítulo é apresentado um estudo experimental para avaliar o procedimento sistemático *TestSd2Efsm*.

CAPÍTULO

4

AVALIAÇÃO DO PROCEDIMENTO *TESTSD2EFSM*

Estudos empíricos são instrumentos utilizados para comparar as teorias e observações usando dados da vida real para análise. Sua condução permite reunir evidências que podem ser usadas para apoiar as alegações de eficiência de uma dada técnica ou tecnologia. Assim, os estudos empíricos desempenham um papel importante na área de Engenharia de Software, pois auxiliam na construção de conhecimento para que os processos e produtos sejam aprimorados resultando em softwares de alta qualidade (MALHOTRA, 2015).

No contexto de teste de software, os estudos experimentais mostram-se muito importantes, uma vez que obter uma prova teórica da eficácia é uma tarefa muito difícil, ou em alguns casos impossível (PAIVA, 2016). O custo e a eficácia são as métricas mais utilizadas nos estudos experimentais na área de teste de software e envolvem normalmente duas dimensões: esforço humano e custo computacional, embora a primeira geralmente seja a mais crucial (BRIAND, 2007). No entanto, conforme González, Juzgado e Vegas (2014), o que se vê na prática é que a maioria dos experimentos na área de teste não levam em consideração a influência do fator humano no comportamento da técnica.

Neste capítulo é apresentado um estudo experimental para avaliar o procedimento sistemático *TestSd2Efsm*. Este estudo segue o processo experimental proposto por Wohlin *et al.* (2012), já que é um guia com etapas bem definidas para que se realize um experimento de forma apropriada. Nas próximas seções as fases do estudo experimental são detalhadas. Na Seção 4.1, o Escopo do Experimento é descrito através dos objetivos, questões de pesquisa e métricas. A Seção 4.2 apresenta o Planejamento do Experimento a partir da seleção de contexto, definição das hipóteses, seleção de variáveis, seleção de sujeitos, definição do desenho instrumental, instrumentação e avaliação da validade dos resultados. Na Seção 4.3, a Operação do Experimento é descrita. Por fim, na Seção 4.4 são apresentadas as Análises dos Dados e Interpretação dos Resultados obtidos.

4.1 Escopo do Estudo Experimental

Nesta etapa, a base do estudo experimental é determinada através da definição dos seus objetivos (WOHLIN *et al.*, 2012).

4.1.1 Definição dos Objetivos

O objetivo deste experimento é analisar a geração de ambiente de teste de software (SUT + Casos de Teste) nas abordagens manual e utilizando o procedimento *TestSd2Efsm*. O propósito do experimento é avaliar a eficiência (esforço em minutos), eficácia (quantidade de falhas detectadas) e a qualidade (percepção dos participantes) na utilização dessas duas abordagens, no contexto de discentes, docentes e desenvolvedores de software na geração de ambiente de teste a partir das especificações de sistemas reais.

4.1.2 Questões de Pesquisa e Métricas

Para guiar a análise da geração do ambiente de testes foram utilizadas as seguintes questões de pesquisas e métricas:

- **QP1**: a abordagem proposta neste trabalho é mais eficiente que a abordagem manual para geração de ambiente de teste de software? O tempo gasto em minutos foi utilizado como métrica para avaliar o esforço despendido na geração do ambiente de teste de software. Quanto maior o tempo, maior o esforço e menor a eficiência da abordagem.
- **QP2**: a abordagem proposta neste trabalho é mais eficaz que a abordagem manual para geração de testes de software? Utilizou-se a medida de quantidade de falhas encontradas como métrica para avaliar a eficácia dos casos de teste gerados. Quanto maior a quantidade de falhas encontradas, maior a eficácia da abordagem.
- **QP3**: a percepção da qualidade do ambiente de teste de software gerado pela abordagem proposta neste trabalho é maior que o gerado pela abordagem manual? Para mensurar a qualidade do ambiente de teste gerado percebida pelos participantes foi aplicado um questionário pós-experimento.

4.2 Planejamento do Estudo Experimental

Conforme definido por Wohlin *et al.* (2012), o Escopo determina a base para experimentação, ou seja, a motivação para a condução do experimento, enquanto o Planejamento determina como o estudo experimental é conduzido. Como em todas as atividades de engenharia, o experimento deve ser planejado e os planos acompanhados, pois os resultados obtidos podem ser alterados ou mesmo invalidados se não houver um planejamento adequado (WOHLIN *et al.*, 2012).

4.2.1 Seleção de Contexto

O contexto de um experimento pode ser caracterizado conforme quatro dimensões (WOHLIN *et al.*, 2012):

- **Processo de execução**: on-line/off-line;
- **Participantes**: estudantes/profissionais;
- **Problema**: real/brinquedo;
- **Generalidade**: específico/geral.

Este estudo experimental foi realizado em um processo de execução *off-line*, pois foi executado em laboratório, contando com a participação de discentes, docentes e profissionais de equipes de desenvolvimento, na geração de ambiente de teste a partir de especificações de sistemas reais utilizando duas abordagens (manual e procedimento *TestSd2Efsm*) em um contexto específico.

4.2.2 Definição das Hipóteses

Uma hipótese é formulada em um experimento com o propósito de rejeitá-la ou refutá-la, ou seja, se o objetivo do experimento é decidir se uma abordagem é melhor que outra, formula-se a hipótese que as duas abordagens são similares. Estas são chamadas de hipóteses nulas e são denotadas como H_0. Qualquer hipótese diferente dessas será chamada de hipótese alternativa. Uma hipótese alternativa à hipótese nula será denotada como H_1 (JURISTO; MORENO, 2010). Nesse sentido, Pfleeger (1995) afirma que testar a hipótese de um experimento significa determinar se os dados são convincentes o suficiente para rejeitar a hipótese nula e aceitar a hipótese alternativa como verdadeira.

Para análise estatística deste experimento, as seguintes hipóteses foram formuladas:

- **Hipótese nula,** H_0 ***Eficiência***: o esforço medido em tempo (minutos) para geração de ambiente de teste utilizando a abordagem manual é similar ao esforço da geração utilizando o procedimento *TestSd2Efsm*. H_0 *Eficiência*: Tempo (Abordagem manual) = Tempo (Procedimento *TestSd2Efsm)*.
- **Hipótese alternativa,** H_1 ***Eficiência***: o esforço medido em tempo (minutos) para geração de ambiente de teste utilizando a abordagem manual é diferente do esforço da geração utilizando o procedimento *TestSd2Efsm*. H_1 *Eficiência*: Tempo (Abordagem manual) $\neq$ Tempo (Procedimento *TestSd2Efsm*).
- **Hipótese nula,** H_0 ***Eficácia***: a eficácia medida em quantidade de falhas encontradas utilizando a abordagem manual é similar à eficácia dos casos de teste gerados utilizando o

procedimento *TestSd2Efsm*. H_0 *Eficácia*: Quantidade de Falhas (Abordagem manual) = Quantidade de Falhas (Procedimento *TestSd2Efsm*).

- **Hipótese alternativa:** H_1 ***Eficácia***: a eficácia medida em quantidade de falhas encontradas utilizando a abordagem manual é diferente da eficácia dos casos de teste gerados utilizando o procedimento *TestSd2Efsm*. H_1 *Eficácia*: Quantidade de Falhas (Abordagem manual) $\neq$ Quantidade de Falhas (Procedimento *TestSd2Efsm*).

- **Hipótese nula:** H_0 ***Qualidade***: medida pela percepção dos participantes sobre a qualidade do ambiente de teste gerado utilizando a abordagem manual é similar a qualidade do ambiente de teste gerado utilizando o procedimento *TestSd2Efsm*. H_0 *Qualidade*: Qualidade do Ambiente de Teste (Abordagem manual) = Qualidade do Ambiente de Teste (Procedimento *TestSd2Efsm*).

- **Hipótese alternativa:** H_1 ***Qualidade***: medida pela percepção dos participantes sobre a qualidade do ambiente de teste gerado utilizando a abordagem manual é diferente da qualidade do ambiente de teste gerado utilizando o procedimento *TestSd2Efsm*. H_1 *Qualidade*: Qualidade do Ambiente de Teste (Abordagem manual) $\neq$ Qualidade do Ambiente de Teste (Procedimento *TestSd2Efsm*).

4.2.3 Seleção de Variáveis

Nesta seção são definidas as variáveis independentes e as variáveis dependentes. Conforme definido por Wohlin *et al.* (2012), as variáveis independentes podem ser controladas e modificadas durante a execução do estudo experimental. Por outro lado, o efeito dos tratamentos é medido pelas variáveis dependentes.

Neste estudo experimental, as seguintes variáveis foram selecionas:

- **Independentes**: biblioteca de geração de testes JUnit e o procedimento *TestSd2Efsm*.

- **Dependentes**: o esforço medido em tempo, a eficácia medida em quantidade de falhas encontradas e a qualidade medida pela percepção dos participantes do experimento.

4.2.4 Seleção de Sujeitos (participantes)

A seleção de sujeitos também chamada de amostra da população é uma importante etapa da fase de planejamento do experimento. Além disso, o tamanho da amostra está diretamente ligada à generalização dos resultados do experimento. Conforme definido por Wohlin *et al.* (2012), a amostra pode ser probabilística ou não probabilística. A principal diferença está na forma de seleção dos sujeitos. Na primeira essa forma é conhecida e na segunda não. O cenário ideal para escolha de sujeitos de um experimento é através da amostragem probabilística, em especial a seleção aleatória de participantes. No entanto, em muitos experimentos, o pesquisador

tende a utilizar os sujeitos disponíveis. Este tipo de experimento é chamado de *quasi-experiment*, pois os sujeitos não são selecionados aleatoriamente (WOHLIN *et al.*, 2012). Independentemente da forma de seleção, é importante caracterizar os sujeitos selecionados para ajudar na avaliação da validade dos resultados.

Os sujeitos selecionados para realização deste experimento são discentes de cursos superiores da área de Computação que tenham cursado a disciplina de Engenharia de Software, docentes de cursos superiores da área de Computação e profissionais de equipes de desenvolvimento de software. Antes da execução do experimento, os participantes responderam um questionário para caracterizá-los de forma que seja possível agrupá-los quanto aos níveis de conhecimento dos assuntos abordados no experimento. A Seção A.1 do Apêndice A apresenta o questionário de caracterização dos participantes.

4.2.5 Desenho Experimental

Um experimento consiste em uma série de testes dos tratamentos. O desenho experimental descreve como os testes dos tratamentos serão organizados e executados. Na definição dos testes é importante considerar os seguintes princípios (WOHLIN *et al.*, 2012):

- **Aleatoriedade**: todos os métodos estatísticos utilizados para análise de dados estabelecem que as variáveis sejam selecionadas de forma aleatória. Este princípio se aplica à seleção dos objetos, sujeitos e a ordem que os testes são realizados.

- **Bloqueio**: em alguns casos tem-se fatores que afetam o resultado dos tratamentos. Para tanto, pode-se utilizar o bloqueio, que elimina o efeito indesejado na comparação entre tratamentos através do agrupamento de participantes e de atribuição de tratamentos. Este princípio aumenta a precisão do experimento.

- **Balanceamento**: para que os experimentos tenham tratamentos equilibrados, é importante que o número de sujeitos e atribuições de tratamentos sejam similares. O balanceamento é desejável, já que simplifica e fortalece a análise estatística dos dados do experimento.

Neste estudo experimental foi adotado o tipo de desenho com um fator e dois tratamentos. O fator é a geração de um ambiente de teste e os dois tratamentos são o procedimento *TestSd2Efsm* e a abordagem manual com o JUnit utilizados na geração do ambiente de teste. O objetivo é a comparação dos dois tratamentos entre si na aplicação em dois objetos distintos. Para melhorar a precisão do experimento, o desenho escolhido é a comparação emparelhada, onde cada sujeito usa os dois tratamentos nos mesmos objetos. Para minimizar o efeito da ordem em que os sujeitos aplicam os tratamentos, essa ordem foi definida aleatoriamente a cada sujeito, conforme mostrado na Tabela 8.

Tabela 8 – Desenho experimental escolhido.

Sujeitos	Objeto 1		Objeto 2	
	Abordagem Manual	Procedimento *TestSd2Efsm*	Abordagem Manual	Procedimento *TestSd2Efsm*
1	2	1	1	2
2	1	2	2	1
3	2	1	1	2
4	1	2	2	1
5	1	2	2	1
6	2	1	1	2
7	1	2	2	1
8	2	1	1	2
9	1	2	2	1
10	2	1	1	2
11	1	2	2	1
12	2	1	1	2
13	2	1	1	2
14	1	2	2	1
15	2	1	1	2
16	1	2	2	1
17	1	2	2	1
18	2	1	1	2
19	1	2	2	1
20	2	1	1	2
21	1	2	2	1

Fonte: Elaborada pelo autor.

4.2.6 Instrumentação

A instrumentação de um experimento é composta por três tipos de instrumentos: objetos, diretrizes (instruções e manuais) e instrumentos de medição (WOHLIN *et al.*, 2012). No planejamento do experimento, antes da sua execução, os instrumentos foram desenvolvidos para o experimento em questão.

Neste experimento utilizou-se dois objetos com suas respectivas especificações e código-fonte com defeitos para verificar a eficiência e eficácia das duas abordagens que foram testadas. Estes objetos referem-se aos sistemas reais descritos a seguir:

- **Sistema de Acesso**: sistema de criação de senha única para os sistemas utilizados pela comunidade acadêmica (discentes, docentes e técnicos) e e-mail institucional para docentes e técnicos. A especificação do Sistema de Acesso é detalhada na Seção A.5 do Apêndice A.

- **Sistema de Unidade Básica de Saúde (UBS)**: sistema de controle de atendimento e marcação de atividades nas Unidades Básicas de Saúde. A especificação do Sistema de

UBS é detalhada na Seção A.6 do Apêndice A.

Além destes dois objetos, também foi desenvolvido mais um objeto (Sistema de Triângulo) para treinamento dos participantes nos dois tratamentos (abordagem manual e procedimento *TestSd2Efsm*) que foram avaliados. A especificação do Sistema de Triângulo é detalhada na Seção A.7 do Apêndice A.

Os dados das medições do experimento foram coletados através de questionários no Google Forms, formulários em editor de texto e planilhas eletrônicas. Antes da execução do experimento, os participantes responderam um questionário para caracterização dos sujeitos, conforme mostrado na Seção A.1 do Apêndice A. Na execução dos tratamentos, as métricas de tempo e quantidade de falhas foram coletados em formulários entregues aos participantes, conforme apresentado na Seção A.2 do Apêndice A e depois armazenados em planilhas eletrônicas. Após a execução do experimento, os participantes responderam um questionário pós-experimento, para avaliar o *feedback* sobre a execução do experimento e qualidade do procedimento *TestSd2Efsm*, conforme apresentado na Seção A.3 do Apêndice A.

Além destes instrumentos, também foi utilizado um Termo de Consentimento Livre e Esclarecido de participação voluntária dos participantes no experimento, no qual todas as instruções, objetivos, justificativa e benefícios do estudo experimental, assim como os riscos de sua participação foram explicados. Este termo é apresentado na Seção A.4 do Apêndice A.

4.2.7 Avaliação de Validade

Uma questão relevante na fase de planejamento do experimento, é a validade dos resultados obtidos. Primeiramente, os resultados devem ser válidos para a população da qual a amostra é retirada. Além disso, pode ser importante generalizar os resultados para uma população mais ampla. Para contornar problemas que ameacem a validade dos resultados, é importante tratar as ameaças a validade dos resultados experimentais (WOHLIN *et al.*, 2012).

Existem diferentes esquemas de classificação para diferentes tipos de ameaças à validade de um experimento. No entanto, uma classificação muito utilizada na literatura e que é facilmente mapeada para as etapas de um experimento divide as ameaças em quatro tipos: validade de conclusão, validade interna, validade de construção e validade externa (COOK; CAMPBELL, 1979).

4.2.7.1 Validade de Conclusão

Tipo de validade que considera questões que afetam à conclusão correta sobre as relações entre tratamentos e o resultado do experimento. Como exemplos desse tipo de validade, (WOHLIN *et al.*, 2012) cita: definição do teste estatístico, tamanho da amostra e atenção na execução do experimento. Em seguida são discutidas as ameaças a esse tipo de validade:

- **Teste com baixo poder estatístico**: essa ameaça diz respeito a capacidade do teste estatístico revelar um padrão verdadeiro nos dados, ou seja, se o teste tem um baixo poder estatístico, há um alto risco do resultado do experimento ter uma conclusão errônea. Para minimizar essa ameaça, a definição do teste estatístico a ser utilizado foi compatível com o desenho experimental escolhido. Portanto, como o desenho adotado foi a comparação emparelhada, onde cada sujeito usa os dois tratamentos nos mesmos objetos, o teste estatístico utilizado foi o *t-teste pareado* ou o *teste pareado de Wilcoxon*.
- **Violação de premissas de testes estatísticos**: essa ameaça diz respeito à violação de premissas que devem ser consideradas em determinados testes estatísticos, por exemplo, amostras distribuídas e independentes. Para evitar esse problema, os dados têm que ser analisados para garantir que estão distribuídos e independentes. Neste experimento, selecionou-se participantes distribuídos em três segmentos: 7 docentes, 7 discentes e 7 desenvolvedores. Além disso, a definição de qual teste deve ser aplicado depende da análise dos dados, assim calcula-se as diferenças entre as variáveis dos tratamentos. Se as diferenças apresentarem uma distribuição normal pelo *teste de Shapiro-Wilk* (SHAPIRO; WILK, 1965), prossegue-se com o *t-teste pareado*, caso contrário aplica-se o *teste pareado de Wilcoxon* (WILCOXON, 1945).
- **Confiabilidade das medidas**: essa ameaça diz respeito à utilização de medidas confiáveis, as quais quando usadas para medir um fenômeno mais de uma vez, os resultados encontrados devem ser os mesmos. Dessa forma, as medidas objetivas, que não envolvem julgamentos humanos, são mais fáceis de serem obtidas e consequentemente mais confiáveis que as medidas subjetivas. Neste experimento foram utilizadas as seguintes medidas objetivas: tempo em minutos para medir o esforço para construção de um ambiente de teste e a quantidade de falhas encontradas para verificar a eficácia dos casos de teste gerados. Além disso, foi definida uma medida subjetiva para medir a qualidade do procedimento proposto através da percepção dos participantes do experimento.
- **Confiabilidade da implementação dos tratamentos**: essa ameaça diz respeito ao risco de implementação de tratamentos diferentes aos sujeitos. Para minimizar esse risco, este experimento aplicará os mesmos objetos em cada tratamento e cada sujeito participará dos dois tratamentos. A ordem de execução dos tratamentos pelos sujeitos será definida aleatoriamente.

4.2.7.2 Validade Interna

Tipo de validade que tenta garantir que a relação observada entre tratamento e o resultado do experimento não seja resultante de um fator não controlado ou medido. Dessa forma, ameaças à validade interna dizem respeito à conclusão sobre a relação causal entre o tratamento e o resultado. Como exemplos de fatores que influenciam a validade interna, tem-se: ameaças de grupo único, ameaças de vários grupos e ameaças sociais (WOHLIN *et al.*, 2012).

Em seguida são discutidas algumas ameaças de grupo único:

- **História**: essa ameaça diz respeito ao risco de diferentes tratamentos que utilizam o mesmo objeto sejam aplicados em momentos distintos. Portanto, os resultados do experimento podem ser influenciados pela história, uma vez que as circunstâncias não são as mesmas nas duas ocasiões. Para evitar esse risco, neste experimento cada participante aplicou os tratamentos (manual e procedimento *TestSd2Efsm*) na mesma seção de execução. Além disso, todas as seções de execução do experimento ocorreram em circunstâncias similares.
- **Maturação**: essa ameaça está relacionada com os efeitos de como os sujeitos reagem de maneira diferente com o passar do tempo. Esses efeitos podem afetar negativamente (cansados ou entediados) ou positivamente (aprendendo) os sujeitos durante o experimento. Para minimizar esses efeitos, a ordem de aplicação dos tratamentos foram definidas aleatoriamente e para evitar tarefas cansativas e longas, ao final de cada tratamento era concedido um intervalo para descanso aos participantes.
- **Instrumentação**: essa ameaça está relacionada aos efeitos causados pelos artefatos da instrumentação. Caso esses artefatos sejam mal projetados, o experimento é afetado negativamente. Para minimizar essa ameaça, todos os artefatos foram analisados e verificados em um experimento piloto executado pelo pesquisador.
- **Mortalidade**: essa ameaça está relacionada aos diferentes tipos de participantes que abandonam o experimento. Neste experimento, as desistências foram caracterizadas para verificar se são representativas na amostra total.

As ameaças relacionadas a experimentos com vários grupos, onde diferentes estudos são realizados nos diferentes grupos, não foram consideradas neste experimento, pois mesmo os participantes sendo de segmentos diferentes (docentes, discentes e desenvolvedores), o estudo experimental os considerou em um único grupo.

Para finalizar, foram consideradas as seguintes ameaças sociais:

- **Equalização compensatória de tratamentos**: esta ameaça está relacionada a participantes que recebem algum tipo de compensação por terem mais ou menos conhecimento nos tratamentos. Para evitar essa ameaça, todos os participantes foram treinados igualmente nos dois tratamentos antes da execução do experimento.
- **Rivalidade ou Ressentimento**: esta ameaça está relacionada à aptidão dos participantes com os tratamentos aplicados. Isso pode motivá-los ou desmotivá-los a mostrar que um tratamento é melhor que o outro. Para evitar essa ameaça, na apresentação do experimento, os objetivos do estudo foram abordados e foi explicado que os participantes não seriam avaliados individualmente. Além disso, os resultados de cada participante são confidenciais e foram utilizados para fins acadêmicos e científicos.

4.2.7.3 Validade de Construção

Tipo de validade que leva em consideração a relação entre teoria e observação. Essa validade preocupa-se com a generalização do resultado do experimento para o conceito ou teoria estudada. As principais ameaças referem-se ao desenho experimental e a fatores sociais (WOHLIN *et al.*, 2012). A seguir são discutidas algumas ameaças a esse tipo de validade:

- **Viés de operação única**: essa ameaça leva em consideração experimentos que não representam completamente a construção e, portanto, não fornece uma imagem completa da teoria. Neste experimento, para minimizar essa ameaça, foram utilizadas três variáveis dependentes e dois objetos nos tratamentos.

- **Adivinhação de hipóteses**: quando as pessoas participam de um experimento, elas podem tentar descobrir qual é o propósito e o resultado pretendido do experimento. Portanto, os participantes poderão basear seu comportamento em palpites sobre as hipóteses, seja positiva ou negativamente, dependendo de sua atitude em relação à hipótese antecipada. Neste experimento, para amenizar essa ameaça, na apresentação no início da execução do mesmo, o pesquisador não deixou claro quais eram exatamente as hipóteses envolvidas no estudo.

- **Apreensão dos sujeitos sob avaliação**: algumas pessoas têm medo de serem avaliadas e tentam parecer melhores ao serem avaliadas. Para minimizar essa ameaça, antes do início da execução do experimento, o pesquisador apresentou os objetivos do estudo, enfatizando que os participantes não seriam avaliados individualmente.

4.2.7.4 Validade Externa

Tipo de validade que preocupa-se com a generalização dos resultados do experimento fora do ambiente do experimento. Essa validade é afetada por três tipos de interação com o tratamento: pessoas, lugar e tempo (WOHLIN *et al.*, 2012). A seguir são discutidas algumas ameaças a esse tipo de validade:

- **Seleção dos participantes**: essa ameaça ocorre quando os participantes selecionados não representam a população a qual deseja-se generalizar. Neste estudo, para ampliar as possibilidades de generalização dos resultados, foram selecionados discentes, docentes e profissionais de equipes de desenvolvimento de software.

- **Instrumentação inadequada**: essa ameaça ocorre quando a instrumentação escolhida é inadequada ou não utilizada na prática. Neste estudo, todas as ferramentas selecionadas no experimento, tanto na abordagem manual como no procedimento *TestSd2Efsm*, são bastante utilizadas na prática acadêmica e industrial.

- **História**: essa ameaça ocorre quando o dia de execução do experimento pode afetar os resultados do mesmo. Um exemplo seria a execução do experimento após uma avaliação prática de Testes de Software utilizando JUnit. Os participantes tendem a se comportar de forma diferente, caso o experimento fosse executado antes ou em algumas semanas depois. Portanto, o período de execução deste experimento foi em datas que não influenciaram os resultados.

4.2.8 Comitê de Ética em Pesquisa

Conforme resolução número 466/2012 do Conselho Nacional de Saúde (CNS), pesquisas envolvendo serem humanos devem ser submetidas à apreciação do Sistema CEP/CONEP . Dessa forma, este planejamento do estudo experimental foi submetido ao Comitê de Ética em Pesquisa (CEP) da Universidade Federal do Piauí (UFPI) e obteve parecer (número 4.387.996) favorável à sua execução.

4.3 Operação do Estudo Experimental

Conforme (WOHLIN *et al.*, 2012), a fase de operação de um experimento consiste em três etapas:

- **Preparação**: etapa onde os participantes são escolhidos e caracterizados.
- **Execução**: etapa onde os participantes realizam suas tarefas de acordo com diferentes tratamentos e os dados que serão analisados são coletados.
- **Validação do dados**: etapa onde os dados coletados são validados.

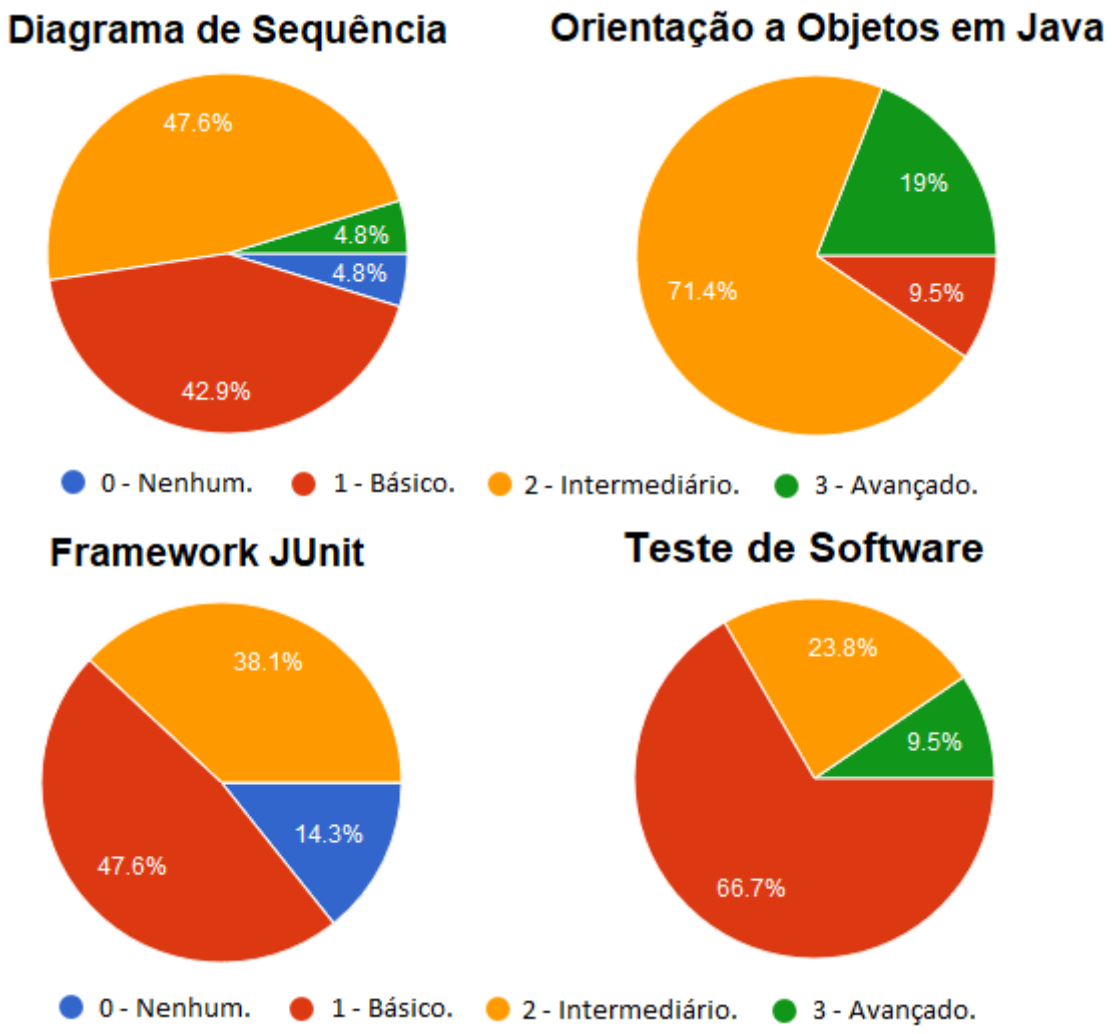
4.3.1 Preparação

Antes da execução do experimento, os participantes escolhidos responderam o questionário de caracterização, conforme apresentado na Seção A.1 do Apêndice A. O estudo foi realizado com 21 participantes: 7 discentes, 7 docentes e 7 desenvolvedores de software. A Figura 27 apresenta o conhecimento dos participantes nos assuntos abordados no experimento. Importante observar que todos dos participantes responderam que tem pelo menos conhecimento básico em Orientação a Objetos na linguagem Java e em Teste de Software, conhecimentos necessários para participar do experimento.

4.3.2 Execução

A execução do experimento foi realizada em cinco turmas presenciais no laboratório de informática do curso de Ciência da Computação da UESPI do Campus Torquato Neto. As datas

Figura 27 – Conhecimento dos participantes nos assuntos abordados no experimento.



Fonte: Elaborada pelo autor.

e horários destas turmas foram disponibilizadas aos participantes com antecedência para que escolhessem a data de execução que mais se adequava às suas atividades. Portanto, a execução do experimento foi realizada em quatro turmas com 4 participantes e uma turma com cinco participantes.

Em cada sessão de execução, antes do início do experimento, o pesquisador fez uma breve apresentação sobre os tratamentos que seriam trabalhados e as atividades que seriam realizadas pelos participantes. Foi informado a todos os participantes que os dados dos estudos seriam mantidos em sigilo e como seriam utilizados. Importante ressaltar que as hipóteses envolvidas no estudo experimental não foram abordadas para evitar que os participantes fossem influenciados pelo propósito e resultados pretendidos no experimento. Após a apresentação, foi entregue aos participantes o Termo de Consentimento Livre e Esclarecido, conforme Seção A.4 do Apêndice A que após lido foi assinado por todos os participantes. Em seguida, foi realizado

um treinamento nos tratamentos (abordagem manual e procedimento *TestSd2Efsm*), onde cada participante realizou as atividades no objeto Sistema de Triângulo, conforme especificado na Seção A.7 do Apêndice A. Neste treinamento, os participantes tiveram total liberdade para consultar o pesquisador para esclarecer qualquer dúvida sobre as atividades que foram realizadas.

Antes da execução dos tratamentos nos objetos que foram medidos, a ordem de execução foi sorteada da seguinte forma. Foi realizado o sorteio da ordem de execução do tratamento para o primeiro participante e essa ordem se alternava para os outros participantes. Além disso, a ordem se alternava também para o participante na execução dos tratamentos no segundo objeto. Dessa forma, se um participante iniciasse com o tratamento da abordagem manual no objeto Sistema de Acesso, no objeto Sistema de UBS ele iniciaria com o tratamento da abordagem do procedimento *TestSd2Efsm*. Com isso, a execução dos tratamentos nos objetos manteve-se equilibrada.

No início da execução de cada tratamento em cada objeto, o participante anotava o horário de início, o tratamento e objeto que estava trabalhando. Ao final de cada tratamento, o participante anotava o horário final da execução. Ao finalizar os tratamentos nos dois objetos, o participante juntamente com o pesquisador, executava os casos de teste gerados em cada tratamento em uma implementação defeituosa para verificar a quantidade de falhas encontradas.

Para finalizar o experimento, foi enviado para cada participante, após a execução dos tratamentos, um questionário pós-experimento para avaliar a percepção dos participantes sobre a qualidade da execução do experimento e do procedimento *TestSd2Efsm*, conforme mostrado na Seção A.3 do Apêndice A.

Dos 21 participantes que iniciaram o experimento, todos conseguiram concluir todas as etapas do estudo. Os dados coletados para as métricas eficiência (custo em minutos) e eficácia (quantidade de falhas) foram tabulados, conforme mostrado na Tabela 9 e na Tabela 10, respectivamente.

4.3.3 Validação dos Dados

Antes de analisar e interpretar os resultados do experimento, o pesquisador verificou se os dados coletados são razoáveis e se foram coletados corretamente. Observando a Tabela 9 e a Tabela 10, verifica-se que o custo em minutos e a quantidade de falhas encontradas foram maiores no objeto Sistema de UBS. Isto justifica-se pelo tamanho e complexidade desse objeto em relação ao objeto Sistema de Acesso. Em relação a quantidade de falhas inseridas nas implementações defeituosas, tem-se: 7 falhas para o Sistema de Acesso e 14 falhas para o Sistema de UBS. Observando a Tabela 10 verifica-se que a quantidade de falhas está dentro do esperado.

Portanto, os dados coletados na execução do experimento podem ser considerados válidos e estão prontos para serem utilizados na análise estatística e interpretação dos resultados do experimento. Na próxima seção, a análise e discussão dos resultados são apresentados.

Tabela 9 – Resultados da execução do experimento quanto ao custo medido em minutos para o desenvolvimento do ambiente de teste.

Sujeitos	Sistema de Acesso		Sistema de UBS	
	Abordagem Manual	Procedimento *TestSd2Efsm*	Abordagem Manual	Procedimento *TestSd2Efsm*
1	51	36	55	39
2	56	38	48	37
3	55	41	59	42
4	54	43	59	42
5	71	40	78	59
6	47	35	78	61
7	69	43	73	45
8	57	29	59	39
9	53	29	55	42
10	47	35	57	40
11	80	40	60	44
12	69	37	57	47
13	50	39	60	45
14	83	51	93	66
15	70	42	70	55
16	68	37	69	55
17	45	31	50	43
18	46	31	53	38
19	57	41	65	51
20	56	45	62	50
21	52	33	57	45

Fonte: Elaborada pelo autor.

4.4 Análise e Interpretação dos Resultados

Depois de coletar os dados na fase de operação, para se tirar conclusões válidas, os dados do experimento devem ser interpretados. De acordo com Wohlin *et al.* (2012), a interpretação quantitativa pode ser realizada em três etapas:

- **Estatística descritiva**: etapa onde os dados coletados são caracterizados por meio da visualização de sua dispersão, variação, distribuição, *quartis* e *outliers*.

- **Redução do conjunto de dados**: etapa onde os dados anormais e falsos são excluídos, reduzindo o conjuntos de dados a um conjunto de dados válidos.

- **Teste de hipóteses**: etapa onde as hipóteses do experimento são avaliadas estatisticamente, em um determinado nível de significância.

Tabela 10 – Resultados da execução do experimento quanto a quantidade de falhas encontradas nas implementações defeituosas pelos casos de teste gerados.

Sujeitos	Sistema de Acesso		Sistema de UBS	
	Abordageml Manual	Procedimento *TestSd2Efsm*	Abordagem Manual	Procedimento *TestSd2Efsm*
1	6	7	9	13
2	6	7	9	14
3	7	7	13	14
4	7	7	14	14
5	6	7	12	14
6	2	7	12	14
7	6	7	9	14
8	7	7	13	14
9	5	7	13	14
10	7	7	14	14
11	7	7	12	14
12	7	7	11	14
13	7	7	13	14
14	4	7	7	14
15	6	7	12	14
16	6	8	14	14
17	7	7	14	14
18	7	7	13	14
19	4	7	5	14
20	3	7	5	14
21	6	7	11	14

Fonte: Elaborada pelo autor.

4.4.1 Estatística Descritiva

Os dados coletados foram visualizados utilizando o diagrama de caixa (*box-plot*). A simplicidade do *box-plot* o torna um método padrão para a apresentação de dados científicos (POTTER, 2006). Com a construção desses diagramas foi possível visualizar a variabilidade e distribuição dos dados, os *quartis* e *outliers* (dados anormais ou falsos). A Figura 28 e a Figura 29 apresentam os diagramas, respectivamente, para as métricas de eficiência (tempo em minutos) e eficácia (quantidade de falhas encontradas), para os dois tratamentos nos Sistemas de Acesso e UBS.

Ao se analisar o diagrama da Figura 28, encontrou-se três *outliers* para os dados de eficiência medidos em tempo. Estes valores foram coletados na execução dos tratamentos do participante 14. No questionário de caracterização, o mesmo respondeu que tinha conhecimento intermediário em Orientação Objetos em Java, em Diagrama de Sequência e em JUnit. No entanto, o que se observou na prática é que o participante tinha baixo domínio destes três tópicos. Além disso, o participante relatou a dificuldade de usabilidade do editor do Diagrama

de Sequência. Mas isto não prejudicou os resultados, pois o participante teve dificuldade nos dois tratamentos e o tempo gasto foi o maior em todas as execuções.

Ao se analisar o diagrama da Figura 29, encontrou-se cinco *outliers* para os dados de eficácia medidos em quantidade de falhas. No tratamento manual, observa-se três ocorrências de *outliers* e todas são de participantes caracterizados sem nenhum conhecimento ou com conhecimento básico no framework JUnit. Já no tratamento do procedimento *TestSd2Efsm*, como os participantes não escrevem nenhuma linha de código e o procedimento percorre todos os caminhos possíveis da implementação, apenas dois valores foram diferentes da quantidade máxima de defeitos inseridos na implementação. Portanto, estes dois *outliers* ocorreram devido à inserção de condição de guarda nos Diagramas de Sequência diferentes das especificações.

Em relação ao esforço gasto na implementação do ambiente de teste nos dois objetos tratados no experimento, foi possível perceber no diagrama da Figura 28 que o tempo gasto utilizando o procedimento *TestSd2Efsm* é sempre menor que a implementação utilizando a abordagem manual. Este resultado era esperado, pois o procedimento *TestSd2Efsm* encapsula toda complexidade relacionada à interpretação da especificação do software pelo desenvolvedor, assim como também a análise das funcionalidades que deverão ser testadas e os caminhos que serão percorridos para geração dos casos de teste. Além disso, observa-se no diagrama de caixa que a variabilidade das amostras obtidas no tratamento utilizando o procedimento *TestSd2Efsm* é menor que o tratamento da abordagem manual nos dois objetos.

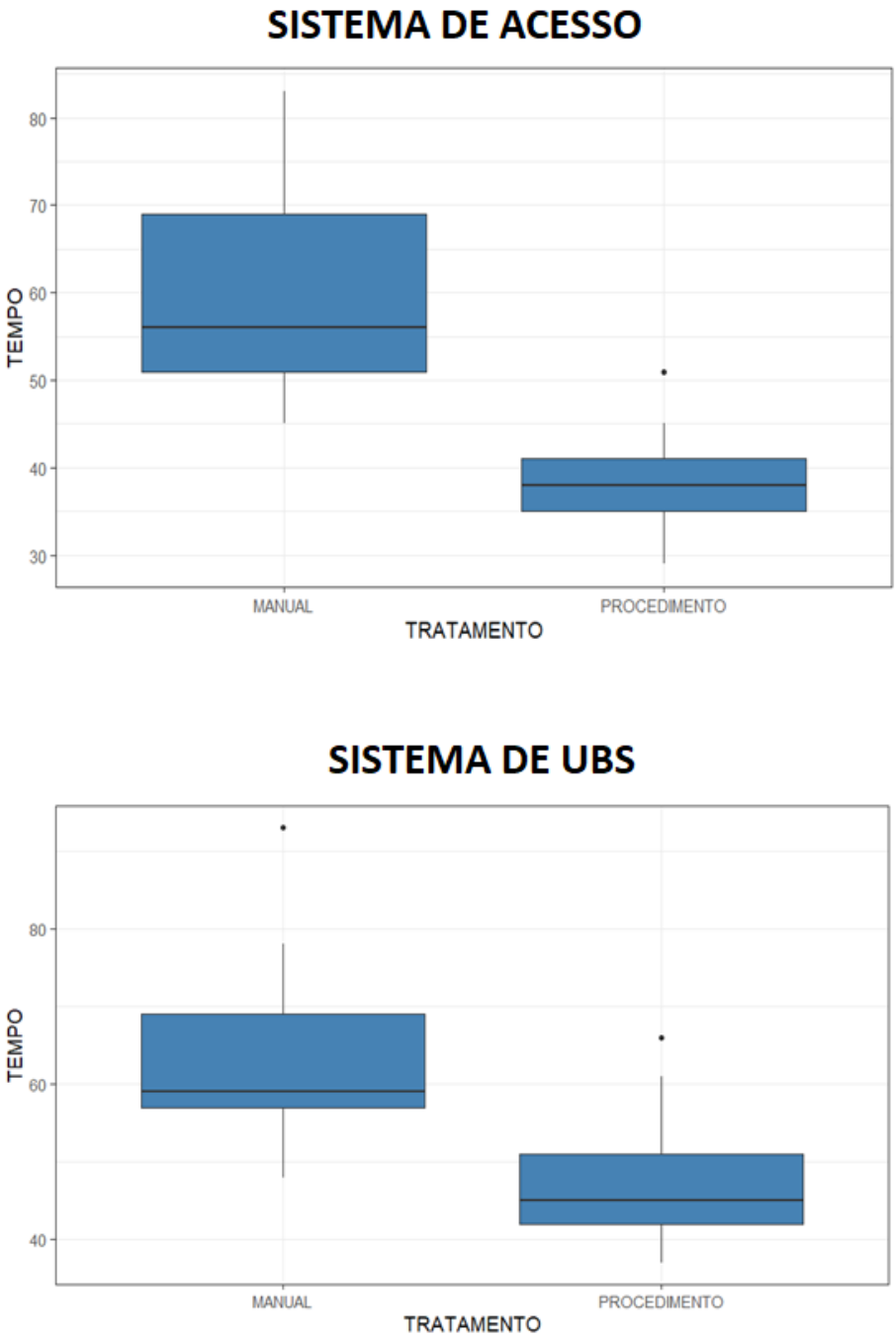
No que se refere à eficácia dos casos de teste gerados nos tratamentos do experimento, observa-se no diagrama da Figura 29 que a quantidade de falhas encontradas nas implementações defeituosas foram maiores no tratamento utilizando o procedimento *TestSd2Efsm*. Isso evidencia que a implementação manual dos casos de teste pode não ser satisfatória, já que depende muito do conhecimento do desenvolvedor. Por outro lado, utilizando o procedimento *TestSd2Efsm* que formaliza o processo de geração de testes, os resultados são mais confiáveis, aumentando a eficácia dos casos de teste. Importante observar no diagrama de caixa, que com exceção dos *outliers*, as amostras não apresentam variabilidade e estão na mediana.

Mesmo verificando na análise dos dados por meio dos diagramas de caixa que os resultados dos tratamentos executados pelo procedimento *TestSd2Efsm* foram promissores, é necessário a utilização de testes estatísticos para verificação das hipóteses do experimento. Esses testes, que foram realizados através do software de computação estatística R (R CORE TEAM, 2018), são apresentados e discutidos na Subseção 4.4.3.

4.4.2 Redução do Conjunto de Dados

Conforme visto na seção anterior, alguns *outliers* foram encontrados no conjunto de dados capturados na operação do experimento. No entanto, é necessário observar se estes *outliers* são dados completamente errados ou se é ocorrência de um evento raro que pode ocorrer novamente

Figura 28 – Diagrama de caixa (*box-plot*) do tempo em minutos para implementação do ambiente de teste.

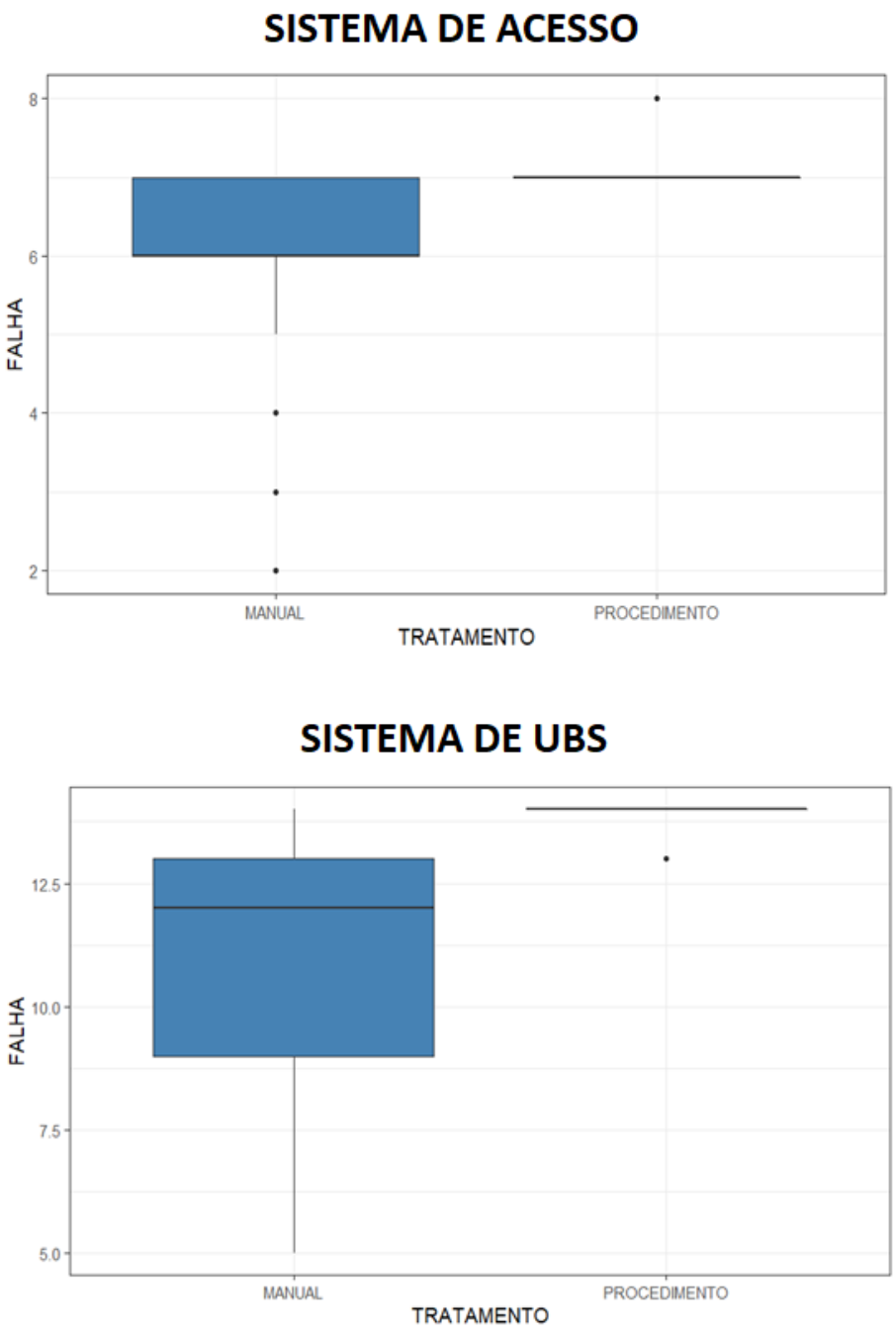


Fonte: Elaborada pelo autor.

(WOHLIN *et al.*, 2012). Embora os valores discrepantes sejam frequentemente considerados como um erro ou ruído, eles podem conter informações importantes (BEN-GAL, 2005).

Portanto, analisando os *outliers* do diagrama de caixa da Figura 28, verificou-se que os dados não comprometeram os resultados do experimento, já que estes dados discrepantes não ocorreram por problemas na condução ou medição do experimento e sim pelo baixo domínio das

Figura 29 – Diagrama de caixa (*box-plot*) da quantidade de falhas encontradas pelos casos de teste.



Fonte: Elaborada pelo autor.

ferramentas pelo participantes. Além disso, o participante teve dificuldade tanto no tratamento manual como no tratamento pelo procedimento *TestSd2Efsm*.

Por outro lado, analisando os *outliers* do diagrama de caixa da Figura 29, verificou-se a necessidade de exclusão dos dados dos dois tratamentos no objeto Sistema de Acesso antes dos testes estatísticos, pois o participante 16 encontrou um número de falhas maior que a quantidade máxima de falhas na implementação defeituosa. Logo, a utilização desse *outlier* pode contribuir

para conclusões errôneas do estudo.

Após a redução do conjunto de dados, a Tabela 11 apresenta as estatísticas descritivas dos dados do experimento.

Tabela 11 – Estatísticas descritivas do experimento.

Estatísticas	Tempo (minutos)				Quantidade de falhas			
	S. de Acesso		S. de UBS		S. de Acesso		S. de UBS	
	Manual	Proced.	Manual	Proced.	Manual	Proced.	Manual	Proced.
Mínimo	45,00	29,00	48,00	37,00	2,00	7,00	5,00	13,00
1ºQuartil	51,00	35,00	57,00	42,00	5,75	7,00	9,00	14,00
Mediana	56,00	38,00	59,00	45,00	6,00	7,00	12,00	14,00
Média	58,86	37,90	62,71	46,90	5,85	7,00	11,90	13,95
Desvio Padrão	11,21	5,57	10,80	8,14	1,50	0,00	2,82	0,22
3ºQuartil	69,00	41,00	69,00	51,00	7,00	7,00	13,00	14,00
Máximo	83,00	51,00	93,00	66,00	7,00	7,00	14,00	14,00

Fonte: Elaborada pelo autor.

4.4.3 Teste de Hipóteses

Os dados resultantes foram analisados pelo Teste Estatístico *t* pareado ou pelo Teste pareado de *Wilcoxon*. A definição de qual teste deve ser aplicado depende da análise dos dados. Assim, calculou-se as diferenças entre as variáveis (tempo e falha) dos tratamentos (manual e procedimento *TesdSd2Efsm*). Caso as diferenças apresentem uma distribuição normal pelo teste de *Shapiro-Wilk*, prossegue-se com o Teste *t* pareado, caso contrário, aplica-se o Teste de *Wilcoxon*.

Para verificar se os dados apresentam distribuição normal, eles foram submetidos ao teste de normalidade *Shapiro-Wilk*. A hipótese nula deste teste sugere que a amostra possui distribuição normal. Portanto, um valor de $p < 0.05$ indica que a hipótese nula foi rejeitada, ou seja, os dados não possuem distribuição normal. Analisando os resultados do teste de normalidade apresentado na Tabela 12, verificou-se que as variáveis Tempo e Quantidade de Falhas nos dois objetos (Sistema de Acesso e Sistema de UBS) não apresentaram distribuição normal, pois o valor de *p* em todos os cenários é menor que 0.05. Dessa forma, para avaliar as hipóteses definidas no experimento, foi utilizado o teste estatístico de *Wilcoxon* que é um método não-paramétrico para comparação de duas amostras pareadas.

Após a execução do teste de *Wilcoxon* nos dados das variáveis (Tempo e Quantidade de Falhas) capturados na aplicação dos dois tratamentos (Abordagem Manual e Procedimento *TestSd2Efsm*) nos dois objetos (Sistema de Acesso e Sistema de UBS), os resultados da estatística *W* e do *p-value* do teste são mostrados na Tabela 13.

Tabela 12 – Resultados do teste de normalidade de *Shapiro-Wilk*.

Variáveis	Sistema de Acesso		Sistema de UBS	
	Abordageml Manual	Procedimento *TestSd2Efsm*	Abordagem Manual	Procedimento *TestSd2Efsm*
Tempo	$W = 0,93592$ $p = 0,02062$		$W = 0,94354$ $p = 0,03805$	
Falha	$W = 0,55627$ $p = 0,0000000008295$		$W = 0,65256$ $p = 0,00000001018$	

Fonte: Elaborada pelo autor.

Analisando os dados da tabela Tabela 13, pode-se afirmar que o esforço medido em minutos para construção do ambiente de teste nos dois objetos (Sistema de Acesso e Sistema de UBS) são estatisticamente diferentes, pois o valor de $p < 0,001$ em ambos os casos. Dessa forma, a hipótese H_0 *Eficiência* foi refutada e a hipótese alternativa H_1 *Eficiência* aceita. Portanto, a questão de pesquisa QP1 pode ser respondida da seguinte forma: *conforme os resultados deste estudo experimental, o esforço para construção de um ambiente de teste utilizando o procedimento TestSd2Efsm é menor que o esforço utilizando a abordagem manual*.

Tabela 13 – Resultados do teste de *Wilcoxon* com amostras pareadas.

Variáveis	Sistema de Acesso		Sistema de UBS	
	Abordagem Manual	Procedimento *TestSd2Efsm*	Abordagem Manual	Procedimento *TestSd2Efsm*
Tempo	$W = 435$ $p = 0,00000007198$		$W = 392$ $p = 0,00001648$	
Falha	$W = 90$ $p = 0,0001602$		$W = 46,5$ $p = 0,000001229$	

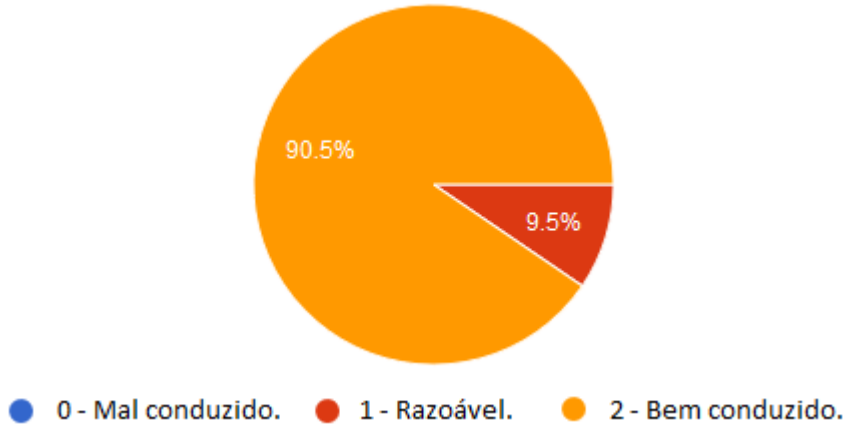
Fonte: Elaborada pelo autor.

Em relação à quantidade de falhas encontradas nas implementações defeituosas pelo conjunto de casos de teste gerados nas abordagens, pode-se afirmar que nos dois objetos (Sistema de Acesso e Sistema de UBS) são estatisticamente diferentes, já que o valor de p também é menor que $0,001$, conforme mostrado na Tabela 13. Dessa forma, a hipótese H_0 *Eficácia* foi rejeitada e a hipótese alternativa H_1 *Eficácia* aceita. Portanto, a segunda questão de pesquisa QP2 pode ser respondida da seguinte forma: *conforme os resultados deste experimento, a eficácia dos casos de teste gerados utilizando o procedimento TestSd2Efsm é maior que a eficácia utilizando a abordagem manual*.

A terceira avaliação realizada foi em relação à percepção dos participantes sobre a qualidade e a possibilidade de utilização do procedimento *TestSd2Efsm* na geração de testes. Para esta avaliação foi utilizado as respostas dos participantes no questionário pós-experimento, apresentado na Seção A.3 do Apêndice A. Além disso, também foi avaliado a condução do experimento e sugestões de melhorias do procedimento *TestSd2Efsm*. A seguir, as respostas dos participantes foram detalhadas.

Os resultados do questionário pós-experimento mostram que mais de 90% dos participantes consideraram que o experimento foi bem executado e não tiveram problemas durante a aplicação dos dois tratamentos. A Figura 30 apresenta o gráfico com as respostas sobre a execução do experimento.

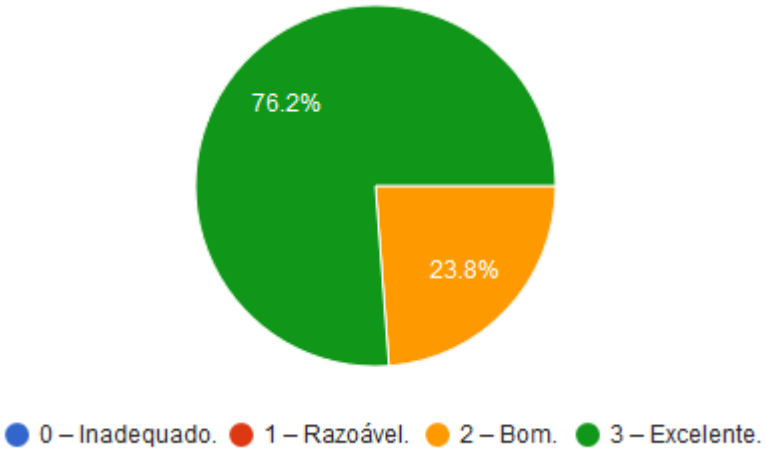
Figura 30 – Percepção dos participantes sobre a execução do experimento.



Fonte: Elaborada pelo autor.

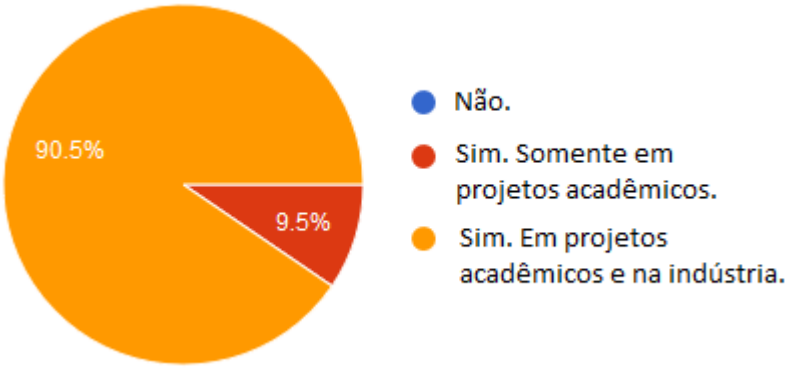
Em relação à qualidade do procedimento *TestSd2Efsm*, 76,2% dos participantes responderam que o procedimento é excelente e mais de 90% dos participantes responderam que utilizariam o procedimento em ambiente acadêmico e industrial. A Figura 31 e a Figura 32 apresentam os gráficos com as respostas sobre o procedimento *TestSd2Efsm*. As respostas sobre melhorias do procedimento foram utilizadas na seção de trabalhos futuros do capítulo de conclusão deste trabalho.

Figura 31 – Percepção dos participantes sobre a qualidade do procedimento *TestSd2Efsm*.



Fonte: Elaborada pelo autor.

Figura 32 – Percepção dos participantes sobre a utilização do procedimento *TestSd2Efsm*.



Fonte: Elaborada pelo autor.

Portanto, a hipótese H_0 *Qualidade* foi rejeitada pelo pesquisador e a hipótese alternativa H_1 *Qualidade* aceita. Dessa forma, a terceira questão de pesquisa QP3 pode ser respondida da seguinte forma: *conforme este estudo experimental e as respostas dos participantes no questionário pós-experimento, a qualidade da geração do ambiente de teste de software utilizando o procedimento TestSd2Efsm é maior que a qualidade utilizando a abordagem manual.*

4.4.4 Considerações Finais

Este capítulo apresentou um estudo empírico conforme o processo experimental proposto por Wohlin *et al.* (2012), para avaliar a eficiência, eficácia e qualidade do procedimento sistemático *TestSd2Efsm* em relação a abordagem manual. O estudo apontou que o procedimento proposto é mais eficiente para geração de ambiente de teste e os casos de teste gerados são mais eficazes para identificar defeitos nas implementações defeituosas. Além disso, a percepção dos participantes do experimento é que a qualidade do SUT e dos casos de teste gerados pelo procedimento sistemático *TestSd2Efsm* é maior que a qualidade utilizando a abordagem manual. Importante ressaltar que a generalização dos resultados não pode ser feita, pois a amostra utilizada foi pequena e os resultados estão restritos a essa amostra da população.

No próximo capítulo, as conclusões desta tese são descritas, abordando as suas principais contribuições, trabalhos relacionados, publicações resultantes, limitações e novas oportunidades de trabalhos futuros e pesquisas.

CAPÍTULO

5

CONCLUSÕES

Apesar dos diversos trabalhos existentes na literatura para geração de testes a partir de modelos UML, muitos desafios ainda são encontrados, principalmente no que diz respeito às inconsistências e várias interpretações que os modelos UML podem gerar. Uma opção para minimizar esses problemas é a utilização de modelos formais, já que possuem uma semântica precisa para representar com exatidão o comportamento de sistemas. Além disso, os métodos de geração de teste de software a partir de modelos formais, tais como as MEFs, podem ser tomados como referência, já que são métodos bem estabelecidos. Porém, o que se observa na prática é que os métodos formais são pouco utilizados na indústria, provavelmente devido à ausência de treinamento e familiaridade com a notação matemática por parte dos desenvolvedores.

Para sistemas mais complexos que necessitam modelar tanto fluxo de controle como fluxo de dados, a utilização de MEFs pode não ser adequada. Com isso, muitos estudos estão trazendo os conceitos de métodos de geração de testes baseados em MEFs para serem utilizados em outros modelos baseados em Máquinas de Transição de Estados, tais como as MEFEs. Porém, essa não é uma tarefa trivial, já que esses modelos possuem comportamentos diferentes. Além disso, muitos trabalhos geram apenas casos de teste abstratos e esses não são concretizados em alguma linguagem de programação.

Neste contexto, este trabalho apresentou um procedimento sistemático chamado de *TestSd2Efsm* para a geração de casos de teste a partir de Diagramas de Sequência UML, que usa conceitos de MDE para formalizar Diagramas de Sequência UML em Máquinas de Estados Finitos Estendidas e as bibliotecas ModelJUnit e JUnit para a geração automática de casos de teste abstratos e concretos. Dessa forma, os resultados aqui apresentados apontam que é possível solucionar o problema da questão de pesquisa introduzida na Seção 1.3, isto é, *é possível sistematizar e automatizar a geração de casos de teste abstratos e concretos a partir de modelos formais extraídos de diagramas UML.*

O restante deste capítulo está organizado da seguinte forma: as contribuições para abordar

o problema de pesquisa deste doutorado são revisitadas na Seção 5.1. A Seção 5.2 apresenta os trabalhos relacionados com esta tese que foram selecionados através de uma busca sistemática na literatura. Na Seção 5.3 são apresentadas as publicações resultantes do trabalho. Na Seção 5.4 são discutidas as limitações desta tese de doutorado. Por fim, na Seção 5.5, os trabalhos futuros e possíveis extensões deste trabalho são discutidos.

5.1 Contribuições

A apresentação do procedimento *TestSd2Efsm*, detalhado no Capítulo 3, representa a primeira contribuição desta Tese. Inicialmente foram definidos meta-modelos simplificados para o Diagrama de Sequência e a MEFE. Em seguida foram definidas regras de transformação para mapeamento entre os elementos do Diagrama de Sequência e MEFE. Estas regras viabilizaram a formalização do Diagrama de Sequência em MEFE. A definição dos meta-modelos e das regras de transformação possibilitaram a implementação da Etapa (a) do procedimento sistemático. A partir do meta-modelo do Diagrama de Sequência foi implementada uma transformação Modelo-Texto (M2T) que possibilitou a geração do SUT, descrita na Etapa (b) do procedimento sistemático. A partir do meta-modelo da MEFE foram implementadas outras transformações Modelo-Texto (M2T) para geração do ambiente de teste ModelJUnit e dos casos de teste concretos no *framework* JUnit. Estas transformações foram detalhadas na Etapa (c) do procedimento sistemático. Por fim, a execução dos casos de teste abstratos e concretos são realizadas pelas bibliotecas ModelJUnit e JUnit, conforme descrito na Etapa (d) do procedimento sistemático *TestSd2Efsm*.

A segunda contribuição desta tese foi a implementação de um protótipo para apoiar o procedimento sistemático *TestSd2Efsm* utilizando *Eclipse Modeling Framework* (EMF). Os meta-modelos simplificados do Diagrama de Sequência UML e da MEFE foram implementados no padrão Ecore. As regras de transformação entre modelos da Etapa (a) foram implementadas na linguagem ATL. As regras Modelo-Texto das Etapas (b) e (c) foram implementadas na linguagem Acceleo. Dessa forma, a geração dos artefatos das etapas do procedimento sistemático *TestSd2Efsm* foram todas automatizadas. A implementação do protótipo possibilitou que as próximas duas contribuições fossem realizadas com sucesso. O código-fonte do protótipo pode ser obtido no repositório[1] criado para este trabalho.

A terceira contribuição, detalhada na Seção 3.7 do Capítulo 3, apresenta a aplicabilidade do procedimento sistemático *TestSd2Efsm* em um software real de Registro de Diário *Online* de Professores. O Diagrama de Sequência usado tem construções complexas com fragmentos combinados aninhados em vários níveis de profundidade. Os cenários e interações deste sistema exercitaram todas as regras de transformações definidas e a MEFE gerada contém 28 estados e 37 transições. O SUT gerado tem 8 classes e todos os atributos e operações foram gerados corretamente. Na geração dos casos de teste, as classes do *framework* ModelJUnit foram geradas corretamente, possibilitando a geração dos casos de teste abstratos e várias métricas de cobertura.

Foram gerados 16 casos de teste concretos no *framework* JUnit. Os casos de teste concretos foram executados em uma implementação defeituosa do SUT e todas as falhas foram encontradas. O código-fonte da aplicação do procedimento sistemático *TestSd2Efsm* neste software real pode ser obtido no repositório[3] de uma publicação resultante desta tese de doutorado.

Por fim, a quarta contribuição, detalhada no Capítulo 4, foi a avaliação empírica do procedimento sistemático *TestSd2Efsm* através de um experimento seguindo o processo experimental proposto por Wohlin *et al.* (2012). O objetivo do estudo experimental foi avaliar a eficiência (esforço medido em minutos), eficácia (quantidade de falhas detectadas) e a qualidade (percepção dos participantes) do procedimento sistemático *TestSd2Efsm*. Neste estudo observou-se que a utilização do procedimento sistemático *TestSd2Efsm* reduz o tempo para construção de um ambiente de teste e aumenta a eficácia dos casos de teste gerados. Além disso, o experimento mostrou que a percepção de qualidade em relação ao procedimento sistemático *TestSd2Efsm* pelos participantes é elevada.

5.2 Trabalhos Relacionados

Um dos pontos fortes deste trabalho é a transformação automática de modelos. Como foi desenvolvido um protótipo para dar suporte ao procedimento *TestSd2Efsm*, essa tarefa foi facilitada pelo uso dos recursos da MDA. Outra vantagem é a proposição de um modelo UML em modelo formal, uma vez que a UML possui problemas semânticos e os modelos formais fornecem um conjunto de técnicas baseadas em notação precisa que podem representar fielmente o comportamento de um sistema. Como o objetivo principal é a geração de testes, a abordagem usa a biblioteca JUnit para concretizar os casos de teste na linguagem de programação Java. Por outro lado, uma limitação que pode ser considerada é o uso de apenas um diagrama UML. Para efeitos de comparação, foi realizada uma busca sistemática de trabalhos relacionados à geração de testes a partir de modelos UML. Além disso, outros campos de pesquisa também foram considerados na busca sistemática, tais como: *Model Driven Architecture* e modelos formais. O protocolo da busca sistemática de trabalhos relacionados é apresentado no Apêndice B.

5.2.1 Descrição dos Estudos Incluídos na Seleção Final

De acordo com a abordagem de Teste Baseado em Modelo (TBM) de Cartaxo, Neto e Machado (2007), casos de teste são gerados a partir de modelos de *Labeled Transition Systems* (LTS) traduzidos a partir de Diagramas de Sequência UML. A abordagem utiliza uma estratégia de transformação entre os modelos com seis passos bem definidos. No entanto, os autores não deixam claro como essa transformação é realizada. Para geração dos casos de teste, todos os caminhos do LTS são identificados utilizando o algoritmo *Depth First Search* (DFS). Embora o trabalho se relacione com esta tese de doutorado, ou seja, emprega um modelo formal extraído do Diagrama de Sequência para gerar casos de teste, ele não utiliza os recursos da MDA para a

transformação entre os modelos e não tem suporte a ferramentas. Além disso, a abordagem não concretiza os casos de teste em uma linguagem de programação.

O trabalho de Lamancha *et al.* (2009) apresenta uma abordagem de Teste Baseado em Modelo (TBM) que utiliza como entrada cenários de casos de uso especificados em modelos de Diagramas de Sequência e como saída um modelo de teste dividido em duas partes: arquitetura de teste e comportamento de casos de teste. A transformação entre modelos é realizada utilizando a linguagem QVT (*Query/View/Transformation*) e o meta-modelo da UML *Testing Profile* (UTP, 2016) da OMG. Portanto, o trabalho se relaciona com esta tese de doutorado, pois utiliza o Diagrama de Sequência para geração de casos de teste e recursos da MDA para transformação entre modelos utilizando a linguagem QVT. No entanto, a abordagem não utiliza modelo formal e a concretização dos casos de teste em alguma linguagem de programação está descrita como um trabalho futuro.

O artigo de Shirole, Suthar e Kumar (2011) apresenta uma abordagem de Teste Baseado em Modelo (TBM) que utiliza transformação entre modelos e algoritmo genético para geração de casos de teste. A proposta utiliza o Diagrama de Transição de Estados da UML como entrada, realiza sua transformação em Máquina de Estados Finitos Estendida (MEFE), que depois é transformada em um grafo de Fluxo de Controle Estendido e como saída tem-se as sequências de teste que são geradas através de um Algoritmo Genético. Além disso, os casos de teste são selecionados de acordo com a sua cobertura de caminhos a partir da técnica de fluxo de dados. Essa abordagem é muito semelhante a esta tese de doutorado, pois utiliza um modelo UML para geração de casos de teste e recursos da MDA para transformação em modelo formal. No entanto, não utiliza transformação Modelo-Texto (M2T) para geração do SUT e não concretiza os casos de teste em alguma linguagem de programação.

O trabalho de Panthi e Mohapatra (2013) descreve um método sistemático de geração de casos de teste a partir do Diagrama de Sequência UML. A abordagem de Teste Baseado em Modelo (TBM) converte o Diagrama de Sequência em um grafo que é percorrido pelo algoritmo *Depth First Search* (DFS) para selecionar as funções de predicado. Em seguida, os predicados são transformados em uma Máquina de Estados Finitos Estendida (MEFE), a partir da qual os casos de teste são gerados de acordo com a cobertura de estados, transições e ações. A abordagem é muito semelhante a esta tese de Doutorado, no entanto, não utiliza recursos da MDA para geração automática da MEFE e algumas importantes construções do Diagrama de Sequência (por exemplo, fragmento combinado) não são utilizadas. Além disso, a abordagem utiliza o ModelJUnit para geração de casos de teste abstratos, mas os mesmos não são concretizados em alguma linguagem de programação.

O trabalho introduzido por Tripathy e Mitra (2013) gera casos de teste usando o Diagrama de Atividade e o Diagrama de Sequência UML. A abordagem consiste em transformar o Diagrama de Sequência em um Grafo de Sequência e transformar o Diagrama de Atividades em Grafo de Atividade. O grafo do software é formado integrando os dois grafos, que são percorridos

para gerar o conjunto de testes. A proposta usa modelos UML para gerar testes, mas difere desta tese de doutorado, pois não apresenta as regras de transformação e não utiliza modelos formais na geração de casos de teste. Além disso, os casos de teste não são concretizados em uma linguagem de programação.

Faria e Paiva (2014) desenvolveram uma abordagem para testar automaticamente a conformidade de implementações de sistemas de software em relação a modelos comportamentais construídos com Diagramas de Sequência UML. Os casos de teste são gerados automaticamente através de modelos do Diagrama de Sequência e são concretizados no *framework* JUnit. A verificação de conformidade é baseada na tradução do Diagrama de Sequência para Redes de Petri. O conjunto de ferramentas foi avaliado com sucesso através de vários estudos de casos, um dos quais foi demostrado no trabalho. Várias partes do trabalho se relacionam com esta tese de doutorado, já que utiliza o Diagrama de Sequência para gerar testes automaticamente. Além disso, utiliza um modelo formal e concretiza os testes na linguagem de programação Java através do JUnit. No entanto, a abordagem não utiliza recursos da MDA para as transformações.

O trabalho de Bahrin e Mohamad (2015) propõe um meta-modelo para extrair informações do Diagrama de Sequência UML e um algoritmo chamado de Gerador de Casos de Teste (TCG). A abordagem utiliza o software *Generic Modeling Environment* (GME) (LEDECZI *et al.*, 2001) para especificar o meta-modelo e os modelos de Diagrama de Sequência. Para a geração dos casos de teste foram definidas dez regras de transformação para traduzir o Diagrama de Sequência no meta-modelo do algoritmo TCG. Por fim, o algoritmo gera os casos de teste que atendem aos requisitos de cobertura de caminhos do Diagrama de Sequência. A abordagem se relaciona com esta tese de doutorado, pois gera casos de teste a partir do Diagrama de Sequência e utiliza recursos da MDA para as transformações. No entanto, o artigo não utiliza modelos formais e nem concretiza casos de teste em alguma linguagem de programação.

A abordagem de Elallaoui, Nafil e Touahni (2016) propõe a geração de testes no processo ágil Scrum. A abordagem utiliza o meta-modelo da UML *Testing Profile* (UTP, 2016) e o meta-modelo do Diagrama de Sequência UML. Neste trabalho são realizadas duas transformações: Modelo-Modelo (M2M) para transformar o Diagrama de Sequência em UML *Testing Profile* (UTP) e uma transformação Modelo-Texto (M2T) para gerar os casos de uso a partir do UTP. Essas transformações são efetuadas utilizando o *framework* AndroMDA (ANDROMDA, 2014). A proposta se relaciona com esta tese de doutorado, já que gera casos de teste a partir de um modelo UML. Além disso, utiliza recursos da MDA para concretizar os casos de teste em algumas linguagens de programação. Porém, a abordagem não formaliza o diagrama UML através de um modelo formal.

Seo *et al.* (2016) desenvolveram um método que gera casos de teste a partir de Diagramas de Sequência. Esse método sugere a geração de casos de teste após a realização de uma transformação intermediária de um Diagrama de Sequência em um Diagrama de Atividades. A proposta se relaciona com esta tese de doutorado, pois usa transformação de modelo, no entanto, não

usa um modelo formal para gerar casos de teste. Além disso, não identificou-se na abordagem como a transformação entre os modelos foi realizada, já que não descreve os meta-modelos manipulados no processo. A abordagem não concretiza casos de teste em uma linguagem de programação.

No trabalho de Strüber, Rieger e Taentzer (2017) foi proposta uma abordagem para criação de modelo de teste de unidade que fornece uma notação textual e gráfica ao mesmo tempo. Esta notação chamada de *VisiText* pode ser especificada através de modelos de entrada (origem) e de saída (destino). Dessa forma, é estabelecida a visibilidade direta e a rastreabilidade de todos os modelos envolvidos nos casos de teste. O modelo de entrada pode ser qualquer diagrama UML e a saída pode ser outro modelo ou código-fonte. Para avaliar a abordagem foram realizados estudos de casos para verificar a utilidade em modelos reais. A avaliação preliminar indica que a abordagem é útil para modelos de tamanho pequeno. A proposta se relaciona com essa tese de doutorado, pois utiliza tanto a notação gráfica como a notação texto para especificar o modelo de teste. Além disso, concretiza os casos de teste na linguagem de programação Java. A diferença encontrada diz respeito a não utilização de modelos formais no trabalho aqui descrito.

Meiliana *et al.* (2017) apresentam uma abordagem que utiliza o Diagrama de Atividade e o Diagrama de Sequência UML para gerar casos de teste. O primeiro passo da abordagem converte o Diagrama de Atividade em um Grafo de Atividade e o Diagrama de Sequência em um Grafo de Sequência. O próximo passo é formar o Grafo de Teste do Sistema com a combinação dos dois grafos criados no primeiro passo. Por fim, os casos de teste são gerados utilizando uma versão modificada do algoritmo *Deep First Search* (DFS). A entrada do algoritmo são os três grafos gerados nos passos anteriores da abordagem. O trabalho se relaciona com esta tese de doutorado, pois o objetivo principal é a geração automática de casos de teste a partir de modelos UML. Além disso, a abordagem utiliza transformação de modelos para construção do Grafo de Teste do sistema. No entanto, a abordagem não utiliza modelos formais e nem concretiza os casos de teste em alguma linguagem de programação.

O trabalho de Pradhan, Ray e Swain (2019) apresenta um conjunto de algoritmos para gerar casos de teste a partir do Diagrama de Estados UML com base em vários critérios de cobertura. A abordagem inicia com a construção do Diagrama de Estados usando a ferramenta *StarUML* (MKLABS, 2014). O Diagrama de Estados é então convertido em um grafo intermediário que depois é percorrido para gerar os caminhos de teste. O algoritmo utilizado na geração dos caminhos depende do critério de cobertura escolhido. Por fim, os casos de teste são gerados por um algoritmo que testa todos os caminhos possíveis e concretiza-os no *framework* JUnit. Para avaliação da abordagem dois estudos de casos são executados. O artigo se relaciona com esta tese de doutorado, pois gera automaticamente casos de teste a partir de um modelo UML, utiliza transformação de modelos e concretiza os testes na linguagem de programação Java no *framework* JUnit. A principal diferença é que a abordagem proposta no trabalho não utiliza modelos formais.

Além dos trabalhos selecionados nas bibliotecas digitais listadas no Apêndice B, foi incluído o trabalho de Vaandrager (2020). Neste trabalho é apresentado um método de geração de teste a partir do Diagrama de Sequência UML. A abordagem transforma o Diagrama de Sequência em termos do modelo formal *Labelled Transition Systems* (LTS). Para geração de testes, o método utiliza a ferramenta *TorXakis* (TNO, 2017) e os concretiza na linguagem de programação Python. Este trabalho é semelhante a esta tese de Doutorado. No entanto, ela não utiliza recursos da MDA para transformação de modelos e não apresenta formalmente as regras de transformação entre o Diagrama de Sequência e o modelo formal LTS. Além disso, não foi possível observar se todas as etapas do método são automatizadas.

A Tabela 14 apresenta o relacionamento entre esta tese de doutorado e os trabalhos relacionados. A comparação é realizada levando em consideração cinco aspectos: uso de outros diagramas UML, utilização de recursos da MDA, uso de modelos formais, suporte a ferramenta e concretização de casos de teste em uma linguagem de programação.

Tabela 14 – Comparação com trabalhos relacionados.

Aspectos para Comparação	Outros Diagramas UML	Recursos da MDA	Modelos Formais	Suporte a Ferramentas	Concretização de Testes
(CARTAXO; NETO; MACHADO, 2007)	✗	✗	✔	✗	✗
(LAMANCHA *et al.*, 2009)	✔	✔	✗	✔	✗
(SHIROLE; SUTHAR; KUMAR, 2011)	✔	✔	✔	✔	✗
(PANTHI; MOHAPATRA, 2013)	✗	✗	✔	✔	✗
(TRIPATHY; MITRA, 2013)	✔	✔	✗	✗	✗
(FARIA; PAIVA, 2014)	✗	✗	✔	✔	✔
(BAHRIN; MOHAMAD, 2015)	✗	✔	✗	✔	✗
(ELALLAOUI; NAFIL; TOUAHNI, 2016)	✗	✔	✗	✔	✔
(SEO *et al.*, 2016)	✔	✔	✗	✗	✗
(STRüBER; RIEGER; TAENTZER, 2017)	✔	✔	✗	✔	✔
(MEILIANA *et al.*, 2017)	✔	✔	✗	✗	✗
(PRADHAN; RAY; SWAIN, 2019)	✔	✔	✗	✔	✔
(VAANDRAGER, 2020)	✔	✗	✔	✔	✔
Tese de Doutorado	✗	✔	✔	✔	✔

5.3 Publicações Resultantes

O desenvolvimento deste trabalho resultou em algumas publicações que são apresentadas em ordem cronológica a seguir:

1. ROCHA, M.; SIMÃO, A.; SOUSA, T.; BATISTA, M. ***Test Case Generation by EFSM Extracted from UML Sequence Diagrams***. *In: The 31 International Conference on Software Engineering & Knowledge Engineering*. p. 135–140. DOI: 10.18293/SEKE2019-133, 2019 (ROCHA *et al.*, 2019).

2. ROCHA, M.; SIMÃO, A.; SOUSA, T.; ***Model-Based Test Case Generation from UML Sequence Diagrams using Extended Finite State Machines***. DOI: 10.1007/s11219-020-09531-0. *In: Software Quality Journal*, 2020. (Aceito para publicação).

Além desses artigos, pretende-se publicar um trabalho com os resultados do estudo experimental que foi descrito no Capítulo 4 desta tese de doutorado.

5.4 Limitações

Mesmo com todos os resultados obtidos, este trabalho apresenta algumas limitações que podem ser resolvidas com a continuidade desta pesquisa. Umas das limitações identificadas pelo pesquisador e pelos participantes no questionário pós-experimento, foi a necessidade de uma melhor usabilidade no protótipo que apoia o procedimento sistemático *TesSd2Efsm*. Outra limitação do protótipo é que, para sua utilização, é necessária a instalação do *Eclipse Modeling Framework* (EMF). Isto pode ser eliminado implementando uma ferramenta com interface gráfica que contemple todas as etapas do procedimento sistemático *TesSd2Efsm* através do *Graphical Modeling Framework* (GMF) (GMF, 2014).

Outra limitação do trabalho que pode ser considerada é a utilização de apenas um diagrama UML. A adoção de outros diagramas no procedimento sistemático *TesSd2Efsm* tornaria as especificações do software mais expressivas, permitindo que outros elementos sejam considerados nas transformações. Em relação à geração do SUT, com a utilização do Diagrama de Classe UML, essa transformação poderia ganhar novos elementos, tornando as classes do SUT mais expressivas e completas. Além disso, no meta-modelo simplificado do Diagrama de Sequência utilizou-se apenas três operadores de interação (*alt*, *opt* e *loop*) nos fragmentos combinados. Dessa forma, a utilização apenas desses operadores de interação não é adequada a sistemas concorrentes. Essa é outra limitação identificada na definição do modelo UML utilizado no trabalho.

A avaliação do procedimento sistemático também pode ser considerada como uma limitação do trabalho, pois não foi possível realizar um estudo de caso em ambiente de desenvolvimento de software de uma empresa seguindo o protocolo de estudo de caso proposto por Wohlin *et al.* (2012). A aplicabilidade do procedimento sistemático *TesSd2Efsm* foi analisada em um software real, mas foi executada pelo próprio pesquisador. Em relação ao estudo experimental realizado, as limitações identificadas dizem respeito à quantidade de participantes que não é uma amostra significativa e também a forma de seleção, pois não foi possível uma escolha aleatória

dos participantes. Além disso, a comparação realizada no experimento foi com a abordagem manual utilizando o *framework* JUnit. Este é um fator limitante do trabalho, já que não foi feita a comparação do procedimento sistemático *TestSd2Efsm* com outra técnica de TBM encontrada na literatura.

Importante ressaltar que as limitações identificadas neste trabalho mostram que existem possibilidades de melhorias e abrem espaço para novas oportunidades de pesquisas no contexto de TBM. Na próxima seção são apresentadas as possibilidades de trabalhos futuros desta tese de doutorado.

5.5 Trabalhos Futuros

O desenvolvimento desta pesquisa e as limitações descritas na seção anterior abrem diversas possibilidades de extensões e trabalhos futuros. A seguir são descritas algumas pesquisas que podem ser desenvolvidas a partir desta tese de doutorado:

- Desenvolvimento de uma ferramenta com interface gráfica para especificação do Diagrama de Sequência. Isso pode ser alcançado utilizando o *Graphical Modeling Framework* (GMF). Além disso, a ferramenta deve contemplar todas as etapas do procedimento sistemático *TestSd2Efsm*.

- Verificar a possibilidade da incorporação de outros modelos UML ao procedimento. Com isso, novos elementos poderiam ser utilizados nas transformações, tornando o SUT e os casos de teste mais expressivos.

- Inclusão de novos operadores de interação da OMG no meta-modelo simplificado do Diagrama de Sequência. Além disso, criar novas regras de transformação para esses operadores. Dessa forma, novas interações e tipos de sistemas poderiam ser especificados no procedimento sistemático *TestSd2Efsm*.

- Criação de um estudo de caso seguindo o protocolo proposto por Wohlin *et al.* (2012), para avaliar a aplicabilidade do procedimento sistemático *TestSd2Efsm* em um ambiente de projeto em uma empresa de desenvolvimento de software.

- Planejamento e execução de outro estudo experimental com uma amostra mais significativa. Verificar a possibilidade da comparação ser realizada em relação à outras técnicas encontradas na literatura.

REFERÊNCIAS

ACCELEO, E. M. F. **Acceleo**. 2018. Disponível em: <https://www.eclipse.org/acceleo/>. Citado nas páginas 31, 43, 45 e 71.

ANDROMDA, A. M. D. A. F. **AndroMDA**. 2014. Disponível em: <http://andromda.sourceforge.net/>. Citado na página 113.

ASOUDEH, N.; LABICHE, Y. Multi-objective construction of an entire adequate test suite for an efsm. In: IEEE. **2014 IEEE 25th International Symposium on Software Reliability Engineering**. [S.l.], 2014. p. 288–299. Citado na página 56.

BAHRIN, N. K.; MOHAMAD, R. Tcg algorithm approach for uml sequence diagram. In: **2015 9th Malaysian Software Engineering Conference (MySEC)**. [S.l.: s.n.], 2015. p. 43–48. Citado nas páginas 113 e 115.

BECK, K.; GAMMA, E. Junit test infected: Programmers love writing tests. In: . [S.l.: s.n.], 1998. Citado nas páginas 56 e 57.

BEN-GAL, I. Outlier detection. In: ____. **Data Mining and Knowledge Discovery Handbook**. Boston, MA: Springer US, 2005. p. 131–146. ISBN 978-0-387-25465-4. Disponível em: <https://doi.org/10.1007/0-387-25465-X_7>. Citado na página 103.

BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. 2 edition. ed. Rio de Janeiro: Elsevier, 2007. Citado nas páginas 29 e 35.

BÉZIVIN, J.; JOUAULT, F.; TOUZET, D. An introduction to the atlas model management architecture. 03 2005. Citado nas páginas 31, 44 e 62.

BINDER, R. Testing object-oriented systems: Models, patterns, and tools. **Addison-Wesley**, 2000. Citado na página 45.

BOURHFIR, C.; DSSOULI, R.; ABOULHAMID, E. M. Automatic test generation for efsm-based systems. In: . [S.l.: s.n.], 1996. Citado na página 30.

BRIAND, L. C. A critical analysis of empirical research in software testing. In: **First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)**. [S.l.: s.n.], 2007. p. 1–8. Citado na página 87.

CARTAXO, E. G.; NETO, F. G. O.; MACHADO, P. D. L. Test case generation by means of uml sequence diagrams and labeled transition systems. In: **2007 IEEE International Conference on Systems, Man and Cybernetics**. [S.l.: s.n.], 2007. p. 1292–1297. ISSN 1062-922X. Citado nas páginas 111 e 115.

CHERUKURI, V. K.; GUPTA, P. **Model Based Testing for Non-Functional Requirements**. 2010. 72 p. Citado na página 57.

CHOW, T. S. Testing software design modeled by finite-state machines. **IEEE transactions on software engineering**, IEEE Computer Society, v. 4, n. 3, p. 178, 1978. Citado nas páginas 28, 49, 50 e 53.

CICCOZZI, F.; MALAVOLTA, I.; SELIC, B. Execution of uml models: a systematic review of research and practice. **Software & Systems Modeling**, 04 2018. Citado na página 30.

COOK, T.; CAMPBELL, D. **Quasi-Experimentation: Design and Analysis Issues for Field Settings**. [S.l.]: Houghton Mifflin, 1979. Citado na página 93.

CZARNECKI, K.; HELSEN, S. Feature-based survey of model transformation approaches. **IBM Systems Journal**, IBM, v. 45, n. 3, p. 621–645, 2006. Citado nas páginas 42, 43 e 44.

DAVID, F. S. **Model driven architecture: applying MDA to enterprise computing**. [S.l.]: Wiley publishing, Inc. USA, 2003. Citado na página 40.

DIAS-NETO, A. C.; TRAVASSOS, G. H. A picture from the model-based testing area: Concepts, techniques, and challenges. In: ZELKOWITZ, M. V. (Ed.). **Advances in Computers**. Elsevier, 2010, (Advances in Computers, v. 80). p. 45 – 120. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0065245810800026>. Citado na página 45.

DOROFEEVA R., E.-F. K.; YEVTUSHENKO, N. An improved conformance testing method. **Formal Techniques for Networked and Distributed Systems - FORTE 2005**, p. 204–218, 2005. Citado nas páginas 28 e 54.

ELALLAOUI, M.; NAFIL, K.; TOUAHNI, R. Automatic generation of testng tests cases from uml sequence diagrams in scrum process. In: **2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)**. [S.l.: s.n.], 2016. p. 65–70. Citado nas páginas 113 e 115.

ENGELS, G.; KÜUSTER, J. M.; GROENWEGEN, L. Consistent interaction of software components. **Journal of Integrated Design and Process Science**, IOS Press, v. 6, n. 4, p. 2–22, 2002. Citado na página 29.

FARIA, J.; PAIVA, A. A toolset for conformance testing against uml sequence diagrams based on event-driven colored petri nets. **International Journal on Software Tools for Technology Transfer**, v. 18, p. 285–304, 2014. Citado nas páginas 113 e 115.

FAVRE, J.-M. Towards a basic theory to model model driven engineering. In: CITESEER. **3rd Workshop in Software Model Engineering, WiSME**. [S.l.], 2004. p. 262–271. Citado na página 38.

FONDEMENT, F.; MULLER, P.-A.; THIRY, L.; WITTMANN, B.; FORESTIER, G. Big metamodels are evil. In: MOREIRA, A.; SCHÄTZ, B.; GRAY, J.; VALLECILLO, A.; CLARKE, P. (Ed.). **Model-Driven Engineering Languages and Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 138–153. ISBN 978-3-642-41533-3. Citado na página 61.

FUJIWARA, S.; BOCHMANN, G. v.; KHENDEK, F.; AMALOU, M.; GHEDAMSI, A. Test selection based on finite state models. **IEEE Transactions on Software Engineering**, v. 17, n. 6, p. 591–603, Jun 1991. ISSN 0098-5589. Citado nas páginas 28 e 53.

GASEVIC, D.; DJURIC, D.; DEVEDZIC, V. **Model Driven Engineering and Ontology Development**. [S.l.: s.n.], 2009. ISBN 978-3-642-00281-6. Citado nas páginas 40 e 41.

GILL, A. Introduction to the theory of finite-state machine. 1962. Citado nas páginas 28 e 48.

GMF, G. M. F. **GMF - Graphical Modeling Framework**. 2014. Disponível em: <https://www.eclipse.org/gmf-tooling/>. Citado na página 116.

GONENC, G. A method for the design of fault detection experiments. **IEEE Transactions on Computers**, C-19, n. 6, p. 551–558, June 1970. ISSN 0018-9340. Citado nas páginas 28 e 53.

GONZÁLEZ, J. E.; JUZGADO, N. J.; VEGAS, S. A systematic mapping study on testing technique experiments: has the situation changed since 2000? In: **ESEM '14**. [S.l.: s.n.], 2014. Citado na página 87.

GRØNMO, R.; MØLLER-PEDERSEN, B. From sequence diagrams to state machines by graph transformation. In: TRATT, L.; GOGOLLA, M. (Ed.). **Theory and Practice of Model Transformations**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 93–107. ISBN 978-3-642-13688-7. Citado na página 61.

HEDLEY, D.; HENNELL, M. A. The causes and effects of infeasible paths in computer programs. In: **Proceedings of the 8th International Conference on Software Engineering**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1985. (ICSE '85), p. 259–266. ISBN 0-8186-0620-7. Disponível em: <http://dl.acm.org/citation.cfm?id=319568.319648>. Citado na página 56.

HIERONS, R. M.; BOGDANOV, K.; BOWEN, J. P.; CLEAVELAND, R.; DERRICK, J.; DICK, J.; GHEORGHE, M.; HARMAN, M.; KAPOOR, K.; KRAUSE, P.; LüTTGEN, G.; SIMONS, A. J. H.; VILKOMIR, S.; WOODWARD, M. R.; ZEDAN, H. Using formal specifications to support testing. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 41, n. 2, p. 9:1–9:76, fev. 2009. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1459352.1459354>. Citado nas páginas 28, 29 e 45.

HUZAR, Z.; KUZNIARZ, L.; REGGIO, G.; SOURROUILLE, J. L. Consistency problems in uml-based software development. In: NUNES, N. J.; SELIC, B.; SILVA, A. Rodrigues da; ALVAREZ, A. T. (Ed.). **UML Modeling Languages and Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 1–12. ISBN 978-3-540-31797-5. Citado na página 29.

JOUAULT, F.; KURTEV, I. Transforming models with atl. In: BRUEL, J.-M. (Ed.). **Satellite Events at the MoDELS 2005 Conference**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 128–138. ISBN 978-3-540-31781-4. Citado na página 44.

JUNIT. **JUnit 5**. 2020. Disponível em: <https://junit.org/junit5/>. Citado na página 30.

JURISTO, N.; MORENO, A. M. **Basics of Software Engineering Experimentation**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441950117. Citado na página 89.

KALAJI, A. S.; HIERONS, R. M.; SWIFT, S. Generating feasible transition paths for testing from an extended finite state machine (efsm) with the counter problem. In: IEEE. **Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on**. [S.l.], 2010. p. 232–235. Citado na página 56.

KENT, S. Model driven engineering. In: SPRINGER. **International Conference on Integrated Formal Methods**. [S.l.], 2002. p. 286–298. Citado na página 38.

KLEPPE, A. G.; WARMER, J. B.; BAST, W. **MDA explained: the model driven architecture: practice and promise**. [S.l.]: Addison-Wesley Professional, 2003. Citado nas páginas 38, 39 e 42.

KOUFAREVA, I.; DOROFEEVA, M. A novel modification of w-method. **Joint Bulletin of the Novosibirsk computing center and AP Ershov institute of informatics systems. Series: Computing science**, n. 18, p. 69–81, 2002. Citado na página 54.

LAMANCHA, B. P.; MATEO, P. R.; GUZMáN, I. R. de; USAOLA, M. P.; VELTHIUS, M. P. Automated model-based testing using the uml testing profile and qvt. In: **Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation**. New York, NY, USA: Association for Computing Machinery, 2009. (MoDeVVa '09). ISBN 9781605588766. Disponível em: <https://doi.org/10.1145/1656485.1656491>. Citado nas páginas 112 e 115.

LEDECZI, A.; MAROTI, M.; BAKAY, A.; KARSAI, G.; GARRETT, J.; THOMASON, C.; NORDSTROM, G.; SPRINKLE, J.; VöLGYESI, P. The generic modeling environment. **Workshop on Intelligent Signal Processing, Budapest, Hungary**, v. 17, 01 2001. Citado na página 113.

LEE, D.; YANNAKAKIS, M. Principles and methods of testing finite state machines-a survey. **Proceedings of the IEEE**, v. 84, n. 8, p. 1090–1123, Aug 1996. ISSN 0018-9219. Citado nas páginas 29, 48 e 54.

LUCAS, F. J.; MOLINA, F.; TOVAL, A. A systematic review of {UML} model consistency management. **Information and Software Technology**, v. 51, n. 12, p. 1631 – 1645, 2009. ISSN 0950-5849. Quality of {UML} Models. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584909000433>. Citado na página 29.

LUO, G.; PETRENKO, A.; BOCHMANN, G. v. Selecting test sequences for partially-specified nondeterministic finite state machines. In: ____. **Protocol Test Systems: 7th workshop 7th IFIP WG 6.1 international workshop on protocol text systems**. Boston, MA: Springer US, 1995. p. 95–110. ISBN 978-0-387-34883-4. Disponível em: <http://dx.doi.org/10.1007/978-0-387-34883-4_6>. Citado nas páginas 28 e 53.

MALHOTRA, R. **Empirical Research in Software Engineering: Concepts, Analysis, and Applications**. [S.l.]: Chapman & Hall/CRC, 2015. ISBN 1498719724. Citado na página 87.

MATOS, E. C. B. d. **BETA: Uma ferramenta para geração de testes de unidade a partir de especificações B**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2012. Citado na página 27.

MEILIANA; SEPTIAN, I.; ALIANTO, R. S.; DANIEL; GAOL, F. L. Automated test case generation from uml activity diagram and sequence diagram using depth first search algorithm. **Procedia Computer Science**, v. 116, p. 629 – 637, 2017. ISSN 1877-0509. Discovery and innovation of computer science technology in artificial intelligence era: The 2nd International Conference on Computer Science and Computational Intelligence (ICCSCI 2017). Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050917320732>. Citado nas páginas 114 e 115.

MICSKEI, Z.; WAESELYNCK, H. The many meanings of UML 2 sequence diagrams: A survey. **Software & Systems Modeling**, v. 10, n. 4, p. 489–514, 10 2010. Disponível em: <https://doi.org/10.1007/s10270-010-0157-9>. Citado nas páginas 28, 35, 36, 37 e 61.

MKLABS. **StarUML**. 2014. Disponível em: <https://staruml.io/>. Citado na página 114.

MODELJUNIT. **The Model-Based Testing Tool**. 2010. Disponível em: <https://sourceforge.net/projects/modeljunit/>. Citado na página 30.

OMG, O. M. G. **MDA - Model Driven Architecture Guide rev. 2.0**. 2014. Disponível em: <http://www.omg.org/mda/>. Citado nas páginas 30 e 38.

____. **OCL - Object Constraint Language**. 2014. Disponível em: <http://www.omg.org/spec/OCL/2.4/>. Citado na página 44.

____. **QVT - Query/View/Transformation, version 1.3**. 2016. Disponível em: <http://www.omg.org/spec/QVT/1.3/>. Citado na página 44.

____. **Unified Modeling Language 2.5.1**. 2017. Disponível em: <https://www.omg.org/spec/UML/2.5.1/>. Citado nas páginas 27, 28, 35, 37 e 60.

____. **MOF - Meta Object Facility Core Specification 2.5.1**. 2019. Disponível em: <http://www.omg.org/spec/MOF/>. Citado na página 40.

PAIVA, S. L. C. **Aplicação de modelos de defeitos na geração de conjuntos de teste completos a partir de Sistemas de Transição com Entrada/Saída**. Tese de Doutorado, 2016. Citado na página 87.

PANTHI, V.; MOHAPATRA, D. Automatic test case generation using sequence diagram. **Advances in Intelligent Systems and Computing**, v. 174 AISC, p. 277–284, 2013. Cited By 9. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84871311363&doi=10.1007%2f978-81-322-0740-5_33&partnerID=40&md5=c49b2d431fcc77912753c1cc053ce700>. Citado nas páginas 112 e 115.

PETRE, M. UML in practice. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 722–731. ISBN 978-1-4673-3076-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2486883>. Citado nas páginas 27 e 29.

PETRENKO, A.; YEVTUSHENKO, N.; LEBEDEV, A.; DAS, A. Nondeterministic state machines in protocol conformance testing. In: **Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems VI**. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1994. p. 363–378. ISBN 0-444-81697-6. Disponível em: <http://dl.acm.org.ez187.periodicos.capes.gov.br/citation.cfm?id=648128.761244>. Citado nas páginas 28 e 53.

PFLEEGER, S. L. Experimental design and analysis in software engineering: Part 2: How to set up and experiment. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, New York, NY, USA, v. 20, n. 1, p. 22–26, jan. 1995. ISSN 0163-5948. Disponível em: <https://doi.org/10.1145/225907.225910>. Citado na página 89.

POTTER, K. Methods for presenting statistical information: The box plot. 01 2006. Citado na página 101.

PRADHAN, S.; RAY, M.; SWAIN, S. K. Transition coverage based test case generation from state chart diagram. **Journal of King Saud University - Computer and Information Sciences**, 2019. ISSN 1319-1578. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1319157818311078>. Citado nas páginas 114 e 115.

PRESSMAN, R. S. **Engenharia de Software**. 6 edition. ed. Rio de Janeiro: Mcgraw-Hill Interamericana, 2006. Citado na página 27.

R CORE TEAM. **R: A Language and Environment for Statistical Computing**. Vienna, Austria, 2018. Disponível em: <https://www.R-project.org/>. Citado na página 102.

ROCHA, M.; SIMãO, A.; SOUSA, T.; BATISTA, M. Test case generation by EFSM extracted from UML sequence diagrams. In: **The 31 International Conference on Software Engineering & Knowledge Engineering**. [s.n.], 2019. p. 135–140. Disponível em: <https://doi.org/10.18293/SEKE2019-133>. Citado nas páginas 62 e 116.

SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 39, n. 2, p. 25–31, fev. 2006. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2006.58>. Citado nas páginas 30 e 38.

SEN, S.; MOHA, N.; BAUDRY, B.; JÉZÉQUEL, J.-M. Meta-model pruning. In: SCHÜRR, A.; SELIC, B. (Ed.). **Model Driven Engineering Languages and Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 32–46. ISBN 978-3-642-04425-0. Citado na página 61.

SEO, Y.; CHEON, E. Y.; KIM, J. A.; KIM, H. S. Techniques to generate utp-based test cases from sequence diagrams using m2m (model-to-model) transformation. In: **2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)**. [S.l.: s.n.], 2016. p. 1–6. Citado nas páginas 61, 113 e 115.

SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples). **Biometrika**, [Oxford University Press, Biometrika Trust], v. 52, n. 3/4, p. 591–611, 1965. ISSN 00063444. Disponível em: <http://www.jstor.org/stable/2333709>. Citado na página 94.

SHIROLE, M.; SUTHAR, A.; KUMAR, R. Generation of improved test cases from uml state diagram using genetic algorithm. In: **Proceedings of the 4th India Software Engineering Conference**. New York, NY, USA: Association for Computing Machinery, 2011. (ISEC '11), p. 125–134. ISBN 9781450305594. Disponível em: <https://doi.org/10.1145/1953355.1953374>. Citado nas páginas 112 e 115.

SIMÃO, A.; PETRENKO, A. Fault coverage-driven incremental test generation. **Comput. J.**, v. 53, p. 1508–1522, 11 2010. Citado na página 28.

SIMÃO, A.; PETRENKO, A.; YEVTUSHENKO, N. Generating reduced tests for fsms with extra states. In: ____. **Testing of Software and Communication Systems: 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 129–145. ISBN 978-3-642-05031-2. Disponível em: <http://dx.doi.org/10.1007/978-3-642-05031-2_9>. Citado nas páginas 28 e 54.

SIMÃO, A. S. Teste baseados em modelos. In: DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. (Ed.). **Introducao ao Teste de Software**. [S.l.]: Elsevier Editora Ltd, 2016. cap. 3, p. 39–57. Citado nas páginas 28, 48, 49 e 51.

SOMMERVILLE, I. **Engenharia de Software**. [S.l.]: Pearson Brasil, 2007. Citado nas páginas 27 e 46.

SOUZA, F. M. d. **Geração de Casos de Teste a partir de Especificações B**. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2010. Citado na página 27.

STEINBERG, D.; BUDINSKY, F.; PATERNOSTRO, M.; MERKS, E. **EMF: Eclipse Modeling Framework 2.0**. 2nd. ed. [S.l.]: Addison-Wesley Professional, 2009. ISBN 0321331885. Citado nas páginas 32 e 60.

STRüBER, D.; RIEGER, F.; TAENTZER, G. A text-based visual notation for the unit testing of model-driven tools. **Computer Languages, Systems & Structures**, v. 49, p. 196 – 215, 2017. ISSN 1477-8424. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1477842416300276>. Citado nas páginas 114 e 115.

TNO, R. U. **TorXakis2017**. 2017. Disponível em: <https://readthedocs.org/projects/torxakis/>. Citado na página 115.

TRETMANS, J. Model based testing with labelled transition systems. In: ____. **Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 1–38. ISBN 978-3-540-78917-8. Disponível em: <https://doi.org/10.1007/978-3-540-78917-8_1>. Citado na página 28.

TRIPATHY, A.; MITRA, A. Test case generation using activity diagram and sequence diagram. **Advances in Intelligent Systems and Computing**, v. 174 AISC, p. 121–129, 2013. Cited By 11. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84871288863&doi=10.1007%2f978-81-322-0740-5_16&partnerID=40&md5=042722c3d21d3dfdd8b2c8f9f9e6d224>. Citado nas páginas 112 e 115.

UTP, O. M. G. **UML Testing Profile**. 2016. Disponível em: <http://utp.omg.org/>. Citado nas páginas 112 e 113.

UTTING, M. How to design extended finite state machine test models in java. In: ZANDER, J.; SCHIEFERDECKER, I.; MOSTERMAN, P. J. (Ed.). **Model-Based Testing for Embedded Systems**. Boca Raton, FL: CRC Press/Taylor and Francis Group, 2012. p. 147–170. Disponível em: <https://eprints.qut.edu.au/56821/>. Citado nas páginas 57, 72 e 73.

UTTING, M.; LEGEARD, B. **Practical Model-Based Testing: A Tools Approach**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN 0123725011. Citado na página 56.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. A taxonomy of model-based testing approaches. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., Chichester, UK, v. 22, n. 5, p. 297–312, ago. 2012. ISSN 0960-0833. Disponível em: <http://dx.doi.org/10.1002/stvr.456>. Citado nas páginas 28, 46 e 47.

VAANDRAGER, F. W. Automated testing: from sequence diagrams to model-based testing. In: . [S.l.: s.n.], 2020. Citado na página 115.

WILCOXON, F. Individual comparisons by ranking methods. **Biometrics Bulletin**, [International Biometric Society, Wiley], v. 1, n. 6, p. 80–83, 1945. ISSN 00994987. Disponível em: <http://www.jstor.org/stable/3001968>. Citado na página 94.

WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. **Experimentation in Software Engineering**. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434. Citado nas páginas 32, 87, 88, 89, 90, 91, 92, 93, 94, 96, 97, 100, 103, 108, 111, 116 e 117.

YANG, R.; CHEN, Z.; ZHANG, Z.; XU, B. Efsm-based test case generation: Sequence, data, and oracle. **International Journal of Software Engineering and Knowledge Engineering**, v. 25, n. 04, p. 633–667, 2015. Citado nas páginas 29, 30, 54, 55, 56 e 61.

YANO, T.; MARTINS, E.; SOUSA, F. L. de. Most: a multi-objective search-based testing from efsm. In: IEEE. **Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on**. [S.l.], 2011. p. 164–173. Citado na página 30.

ZANDER, J.; SCHIEFERDECKER, I.; MOSTERMAN, P. J. **Model-Based Testing for Embedded Systems**. 1st. ed. USA: CRC Press, Inc., 2011. ISBN 1439818452. Citado nas páginas 30 e 56.

ZHAO, R.; HARMAN, M.; LI, Z. Empirical study on the efficiency of search based test generation for efsm models. In: **2010 Third International Conference on Software Testing, Verification, and Validation Workshops**. [S.l.: s.n.], 2010. p. 222–231. Citado na página 56.

APÊNDICE

A

INSTRUMENTAÇÃO DO EXPERIMENTO

A.1 Questionário de Caracterização dos Participantes

Prezado(a) participante,

Este questionário de caracterização dos participantes visa facilitar a análise dos resultados do experimento. Dessa forma, responda corretamente todas as questões.

DADOS GERAIS

1. **Em qual segmento você atua?**

   [ ] Docente [ ] Discente [ ] Desenvolvedor

2. **Qual o seu nível de formação?**

   [ ] Graduando [ ] Graduado

   [ ] Mestrando [ ] Mestre

   [ ] Doutorando [ ] Doutor

3. **Qual o nome da sua instituição ou empresa?**

   _____

4. **Quanto tempo de experiência você tem no segmento que você atua?**

   _____

CONHECIMENTO EM DIAGRAMA DE SEQUÊNCIA

5. **Como você definiria seus conhecimentos com relação a modelagem utilizando o Diagrama de Sequência da UML?**

   [ ] 0 - Nunca especifiquei software utilizando o Diagrama de Sequência.

   [ ] 1 - Conhecimento básico. O Diagrama de Sequência foi utilizado somente no contexto acadêmico.

   [ ] 2 - Conhecimento intermediário. O Diagrama de Sequência foi utilizado em projetos reais individuais.

   [ ] 3 – Conhecimento avançado. O Diagrama de Sequência foi utilizado em projetos reais na indústria.

   Comentários adicionais):

   ---

6. **Como você definiria seus conhecimentos com relação aos operadores (loop, alt e opt) dos Fragmentos Combinados do Diagrama de Sequência da UML?**

   [ ] 0 - Nenhum conhecimento.

   [ ] 1 - Conhecimento básico. Os operadores foram utilizados somente no contexto acadêmico.

   [ ] 2 - Conhecimento intermediário. Os operadores foram utilizados em projetos reais individuais.

   [ ] 3 – Conhecimento avançado. Os operadores foram utilizados em projetos reais na indústria.

   Comentários adicionais:

CONHECIMENTO NO FRAMEWORK JUNIT

7. **Qual o seu conhecimento em orientação a objetos na linguagem de programação Java?**

   [ ] 0 - Nenhum conhecimento.

   [ ] 1 - Conhecimento básico. A linguagem Java foi utilizada somente no contexto acadêmico.

   [ ] 2 - Conhecimento intermediário. A linguagem Java foi utilizada em projetos reais individuais.

[ ] 3 – Conhecimento avançado. A linguagem Java foi utilizada em projetos reais na indústria.

Comentários adicionais:

---

8. **Qual o seu conhecimento no framework JUnit?**

   [ ] 0 - Nunca utilizei o JUnit.

   [ ] 1 - Conhecimento básico. O framework JUnit foi utilizada somente no contexto acadêmico.

   [ ] 2 - Conhecimento intermediário. O framework JUnit foi utilizada em projetos reais individuais.

   [ ] 3 – Conhecimento avançado. O framework JUnit foi utilizada em projetos reais na indústria.

   Comentários adicionais:

CONHECIMENTOS EM TESTES DE SOFTWARE

9. **Qual o seu conhecimento em Teste de Software?**

   [ ] 0 – Não tenho nenhum conhecimento.

   [ ] 1 - Conhecimento básico. Teste de Software realizado somente no contexto acadêmico.

   [ ] 2 - Conhecimento intermediário. Teste de Software realizado em projetos reais individuais.

   [ ] 3 – Conhecimento avançado. Teste de Software realizado em projetos reais na indústria.

   Comentários adicionais:

---

10. **Como você definiria seus conhecimentos nas seguintes ferramentas de Teste de Software?**

    Selenium [ ]

    Appium [ ]

    Robotium [ ]

    Legenda:

0 – Nenhum conhecimento.

1 – Conhecimento básico. Ferramenta(s) de Teste de Software utilizada(s) somente no contexto acadêmico.

2 – Conhecimento intermediário. Ferramenta(s) de Teste de Software utilizada(s) em projetos reais individuais.

3 – Conhecimento avançado. Ferramenta(s) de Teste de Software utilizada(s) em projetos reais na indústria.

Comentários adicionais:

A.2 Formulário de Execução do Experimento

Data da Execução: ————————-

Objeto: SISTEMA DE ACESSO

Tratamento: Manual

Início: ————————-

Fim: ————————-

Quantidade de Falhas: ————————-

Tratamento: Procedimento *TestSd2Efsm*

Início: ————————-

Fim: ————————-

Quantidade de Falhas: ————————-

Objeto: SISTEMA DE ATENDIMENTO UBS

Tratamento: Manual

Início: ————————-

Fim: ————————-

Quantidade de Falhas: ————————-

Tratamento: Procedimento *TestSd2Efsm*

Início: ————————-

Fim: ————-

Quantidade de Falhas: ————-

_____

PARTICIPANTE

A.3 Questionário Pós-Experimento

Prezado(a) participante,

Este questionário pós-experimento tem o intuito de coletar informações sobre a operação do estudo experimental e procedimento sistemático *TestSd2Efsm* avaliado. Todas as respostas são importantes para a conclusão do estudo. Dessa forma, responda corretamente todas as questões.

1. **Qual sua nota para condução do estudo experimental?**

   [ ] 0 – Mal conduzido. Os objetivos não foram abordados e o experimento foi mal executado.

   [ ] 1 – Razoável. Os objetivos foram abordados, porém aconteceram muitos problemas no decorrer da condução do experimento.

   [ ] 2 – Bem conduzido. Os objetivos foram abordados e não aconteceram problemas no decorrer da execução do estudo.

   Comentários adicionais:

   _____

2. **O que você achou do procedimento sistemático *TestSd2Efsm?***

   [ ] 0 – Inadequado. O procedimento não é adequado para apoiar a geração de um ambiente de teste.

   [ ] 1 - Razoável. O procedimento apresenta ganhos quanto à eficiência na geração de um ambiente de teste, entretanto ainda apresenta muitas falhas e é muito complexo.

   [ ] 2 – Bom. O procedimento é adequado para apoiar a geração de um ambiente de teste, apresenta poucas falhas e é fácil de usar, entretanto não é adequado para utilização em projetos reais na indústria.

   [ ] 3 – Excelente. O procedimento é adequado para apoiar a geração de um ambiente de teste, apresenta poucas falhas, o seu uso é simples e pode ser utilizada em projetos reais na indústria.

   Comentários adicionais:

3. **Quais são as vantagens e limitações da utilização do procedimento *TestSd2Efsm*?**

4. **Quais as suas sugestões de melhorias e extensões para trabalhos futuros?**

5. **Você utilizaria o procedimento *TestSd2Efsm* caso precisasse gerar casos de teste abstratos e concretos?**

   [ ] Não.

   [ ] Sim. Somente em projetos acadêmicos.

   [ ] Sim. Em projetos acadêmicos e na indústria.

   Comentários adicionais:

A.4 Termo de Consentimento Livre e Esclarecido

Prezado(a) Senhor(a)

Você está sendo convidado(a) a participar como voluntário(a) de uma pesquisa denominada "Geração de Testes a partir de Máquinas de Estados Finitos Estendidas Extraídas de Diagramas de Sequência UML". Esta pesquisa está sob a responsabilidade do pesquisador Mauricio Rêgo Mota da Rocha, professor da Universidade Estadual do Piauí, e tem como objetivo avaliar a geração de um ambiente de teste utilizando o procedimento *TestSd2Efsm* e manualmente no framework JUnit. Esta pesquisa tem por finalidade a validação de uma nova abordagem de geração de testes de software que poderá trazer benefícios ao processo de desenvolvimento de software tanto na academia quanto na indústria. Outro benefício deste estudo experimental é a de possibilitar a disponibilização de uma ferramenta que facilite o processo de geração de testes de software.

Neste sentido, solicitamos sua colaboração mediante a assinatura desse termo. Este documento, chamado Termo de Consentimento Livre e Esclarecido (TCLE), visa assegurar seus direitos como participante. Após seu consentimento, assine todas as páginas e ao final desse documento que está em duas vias. O mesmo, também será assinado pelo pesquisador em todas as páginas, ficando uma via com você participante da pesquisa e outra com o pesquisador. Por

favor, leia com atenção e calma, aproveite para esclarecer todas as suas dúvidas. Se houver perguntas antes ou mesmo depois de indicar sua concordância, você poderá esclarecê-las com o pesquisador Mauricio Rêgo Mota da Rocha, responsável pela pesquisa através do telefone (86)998009471 e e-mail mauricio@ctu.uespi.br. Se mesmo assim, as dúvidas ainda persistirem você pode entrar em contato com o Comitê de Ética em Pesquisa da UFPI, que acompanha e analisa as pesquisas científicas que envolvem seres humanos, no Campus Universitário Ministro Petrônio Portella, bairro Ininga, Teresina –PI, telefone (86) 3237-2332, e-mail cep.ufpi@ufpi.br, no horário de atendimento ao público, segunda a sexta, manhã: 08h00 às 12h00 e a tarde: 14h00 às 18h00. Se preferir, pode levar este Termo para casa e consultar seus familiares ou outras pessoas antes de decidir participar. Esclarecemos mais uma vez que sua participação é voluntaria, caso decida não participar ou retirar seu consentimento a qualquer momento da pesquisa, não haverá nenhum tipo de penalização ou prejuízo e o pesquisador estará a sua disposição para qualquer esclarecimento.

A pesquisa tem como justificativa a validação de um novo procedimento de geração de ambiente de teste de software e para sua realização serão utilizados os seguintes procedimentos para a coleta de dados: questionário de caracterização dos participantes, aplicação de dois tratamentos em dois objetos distintos e questionário pós-procedimento. A aplicação dos questionários será de forma online pelo *google forms* e a execução do experimento será realizada presencialmente no laboratório de informática do Curso de Computação no Campus Poeta Torquato Neto em Teresina-PI.

A execução deste estudo experimental pode acarretar em alguns riscos ao participante. O primeiro risco identificado foi o problema da pandemia do novo coronavírus. Para contornar esse risco, a execução do experimento será realizada em turmas de no máximo 4 participantes. Além disso, será disponibilizado um kit com máscara e álcool em gel para cada participante e os mesmos devem utilizar a máscara durante toda a execução do experimento. Outro risco identificado é a postura sentada do participante em frente ao computador. Para minimizar esse risco, o pesquisador vai observar e orientar a ergonomia. Por fim, identificamos também que a participação dos sujeitos pode acarretar risco de algum constrangimento, caso o mesmo não consiga realizar as atividades dos tratamentos. Para minimizar esse risco, haverá um treinamento antes do início da execução do experimento e o participante pode desistir a qualquer momento sem que os outros participantes tomem conhecimento.

Os resultados obtidos nesta pesquisa serão utilizados para fins acadêmico-científicos (divulgação em revistas e em eventos científicos) e os pesquisadores se comprometem a manter o sigilo e identidade anônima, como estabelecem as Resoluções do Conselho Nacional de Saúde nº. 466/2012 e 510/2016 e a Norma Operacional 01 de 2013 do Conselho Nacional de Saúde, que tratam de normas regulamentadoras de pesquisas que envolvem seres humanos. E você terá livre acesso as todas as informações e esclarecimentos adicionais sobre o estudo, bem como lhe é garantido acesso a seus resultados.

Esclareço ainda que você não terá nenhum custo com a pesquisa, e caso haja por qualquer motivo, asseguramos que você será devidamente ressarcido. Não haverá nenhum tipo de pagamento por sua participação, ela é voluntária. Caso ocorra algum dano comprovadamente decorrente de sua participação neste estudo você poderá ser indenizado conforme determina a Resolução 466/12 do Conselho Nacional de Saúde, bem como lhe será garantido a assistência integral.

Após os devidos esclarecimentos e estando ciente de acordo com os que me foi exposto, Eu, ———————————————————- declaro que aceito participar desta pesquisa, dando pleno consentimento para uso das informações por mim prestadas. Para tanto, assino este consentimento em duas vias, rubrico todas as páginas e fico com a posse de uma delas.

Local e data:

Assinatura do Participante:

Assinatura do Pesquisador Responsável:

A.5 Especificação do Sistema de Acesso

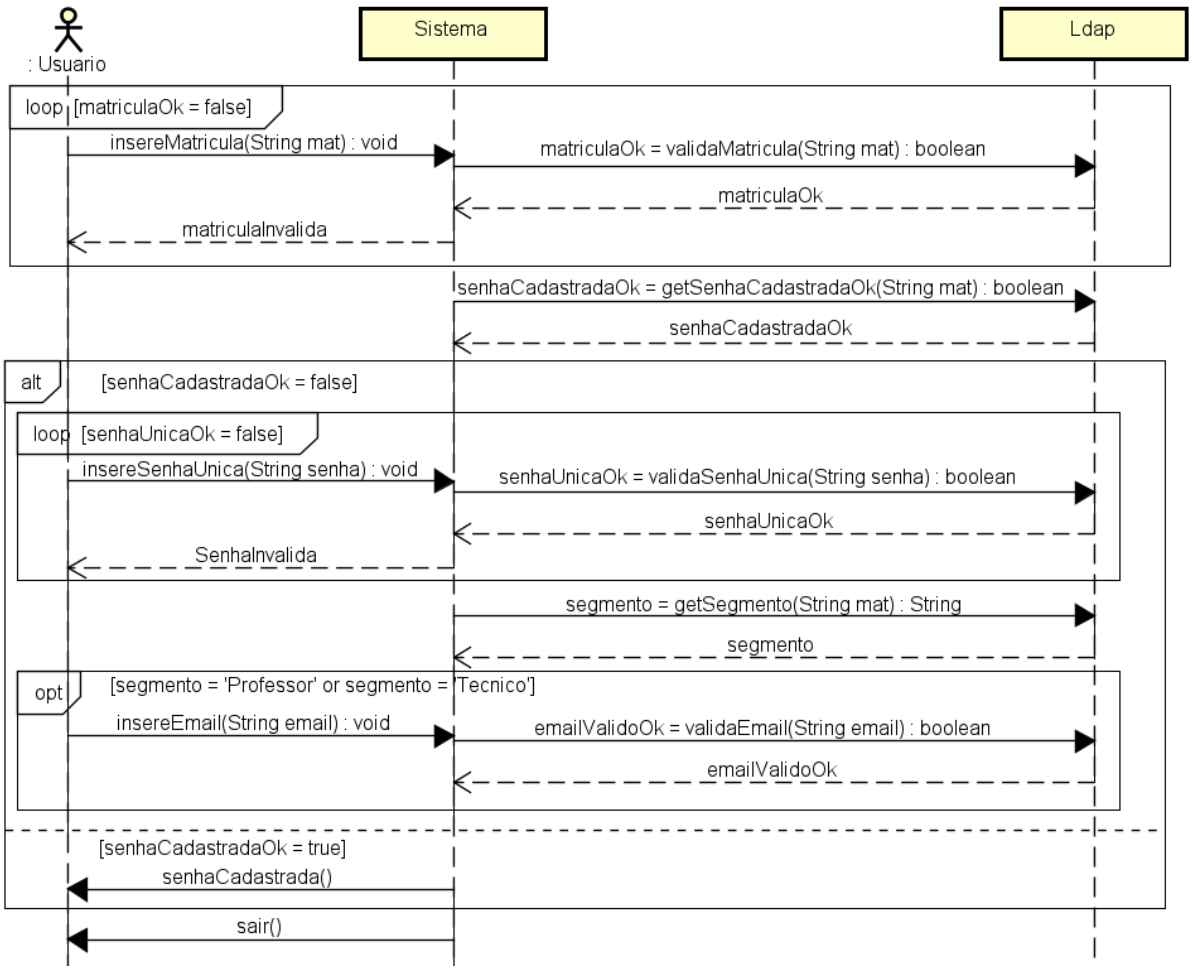
A.5.1 Descrição do Cenário

O usuário admitido ou aprovado acessa o *Sistema de Acesso* e digita sua matrícula. O sistema verifica na base *Lightweight Directory Access Protocol* (LDAP) se a matrícula existe. Caso a matrícula não seja validada, o sistema exibe uma mensagem de matrícula inválida. Caso a matrícula seja válida, o sistema verifica se o usuário tem a senha única cadastrada. Caso não exista uma senha única cadastrada, o sistema solicitará que o usuário digite uma senha e o sistema validará senha. A senha será válida se tiver no mínimo 6 caracteres. Após esse procedimento, o sistema verifica o segmento do usuário. Caso segmento seja diferente de “Aluno”, ou seja, igual a “Professor” ou “Técnico”, o sistema irá solicitar que o usuário cadastre um e-mail. O sistema valida o e-mail que deve conter "@uespi.br"e o processo é finalizado. Caso a senha única já tenha sido cadastrada, o sistema emite uma mensagem informando ao usuário e o processo termina.

A.5.2 Diagrama de Sequência

A Tabela 15 apresenta as condições de guardas das operações do Diagrama de Sequência do Sistema de Acesso da Figura 33.

Figura 33 – Diagrama de Sequência do Sistema de Acesso.



Fonte: Elaborada pelo autor.

Tabela 15 – Condições de Guarda das Operações do Sistema de Acesso.

Operação	Guarda
validaMatricula	this.mat.equals(mat)
getSenhaCadastradaOk	this.mat.equals(mat)
validaSenhaUnica	senha.length() >= 6
getSegmento	this.mat.equals(mat)
validaEmail	email.contains('@uespi.br')

Fonte: Elaborada pelo autor.

A.6 Especificação do Sistema de Unidade Básica de Saúde

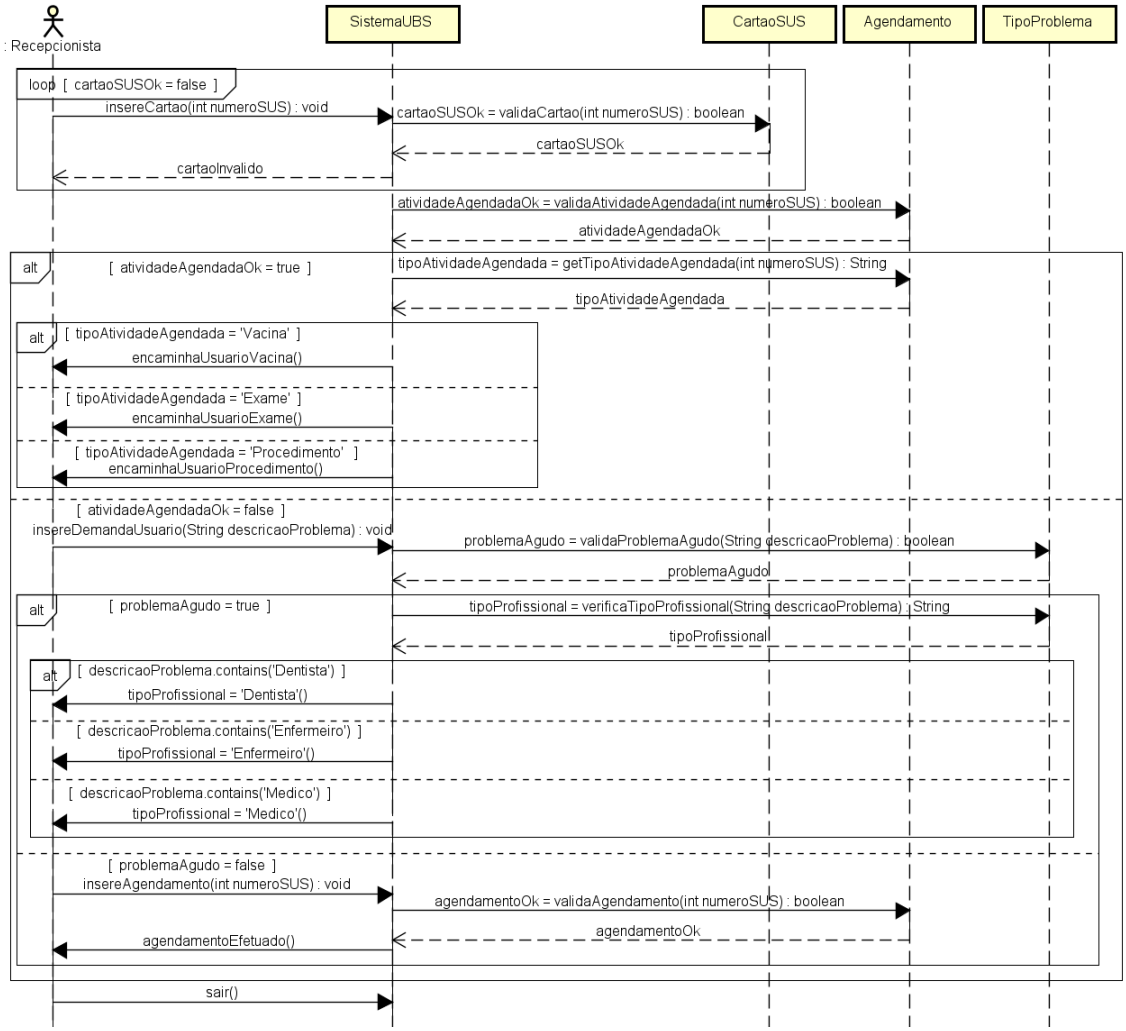
A.6.1 Descrição do Cenário

O usuário chega a Unidade Básica de Saúde e a recepcionista insere o número do cartão do SUS no Sistema de UBS. O sistema valida o número do cartão e verifica se o cartão é válido. Se o cartão for válido, o sistema verifica se há alguma atividade agendada para o usuário. Se há

uma atividade agendada, o sistema verifica pelo número do cartão do SUS, qual atividade está agendada. Esta atividade pode ser Vacina, Exame ou Procedimento. O usuário será encaminhado para a atividade agendada. Caso não haja atividade agendada, a recepcionista insere a demanda do usuário, informando a descrição do problema. O sistema valida se o problema é agudo ou não. Se for um problema agudo, o sistema busca o tipo de profissional pela descrição do problema. O profissional pode ser Dentista, Enfermeiro ou Médico. Caso o problema não seja agudo, a recepcionista irá agendar um atendimento através do número de cartão do SUS. Após esses procedimentos a recepcionista pode sair do sistema.

A.6.2 Diagrama de Sequência

Figura 34 – Diagrama de Sequência do Sistema de UBS.



Fonte: Elaborada pelo autor.

A Tabela 16 apresenta as condições de guardas das operações do Diagrama de Sequência do Sistema de Unidade Básica de Saúde da Figura 34.

Tabela 16 – Condições de Guarda das Operações do Sistema de UBS.

Operação	Guarda
validaCartao	this.numeroSUS = numeroSUS
validaAtividadeAgendada	this.numeroSUS = numeroSUS
getTipoAtividadeAgendada	this.numeroSUS = numeroSUS
validaProblemaAgudo	descricaoProblema.contains('agudo')
validaAgendamento	this.numeroSUS = numeroSUS

Fonte: Elaborada pelo autor.

A.7 Especificação do Sistema de Triângulo

A.7.1 Descrição do Cenário

O Sistema de Triangulo determina se um triângulo é válido ou não. Um triângulo válido não pode ter um lado maior ou igual a soma dos outros lados. Assim como também, os lados não podem ter tamanho menor ou igual a zero. Além disso, caso o triângulo seja válido, o sistema retorna o seu tipo, conforme a seguir:

- Equilátero, se todos os lados são iguais,
- Isósceles, se apenas dois lados são iguais.
- Escaleno, se todos os lados são diferentes.

A.7.2 Diagrama de Sequência

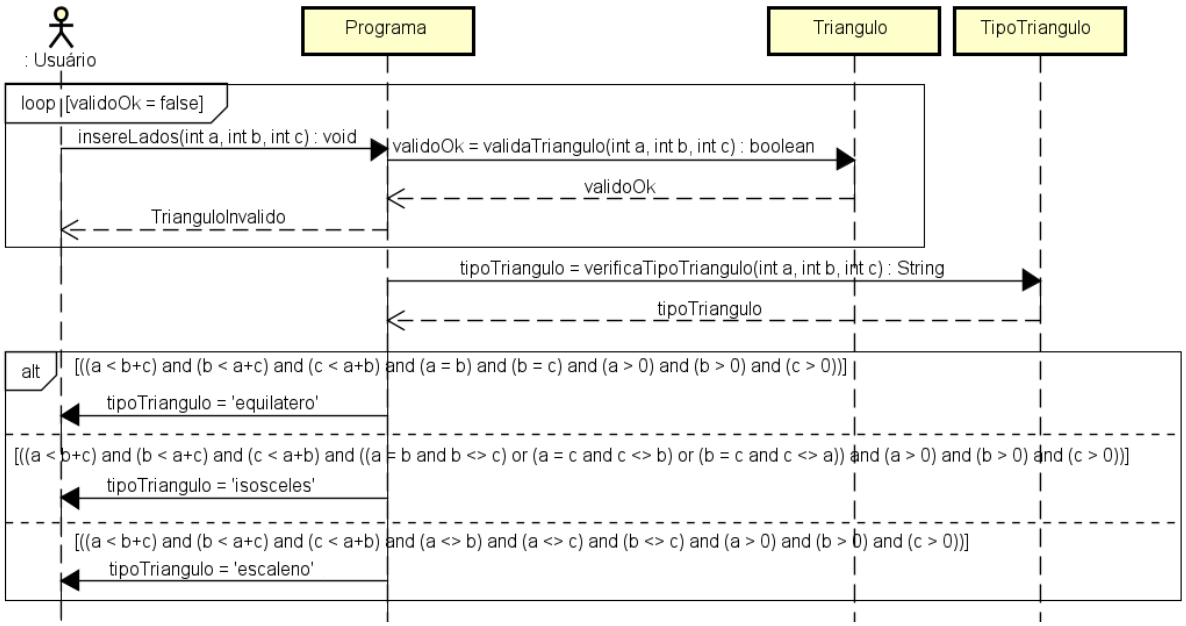
A Tabela 17 e a Tabela 18 apresentam respectivamente as condições de guardas das operações e dos operandos do Diagrama de Sequência do Sistema de Triângulo da Figura 35.

Tabela 17 – Condições de Guarda das Operações do Sistema de Triângulo.

Operação	Guarda
validaTriangulo	(a < b+c) and (b < a+c) and (c < a+b) and (a > 0) and (b > 0) and (c > 0)

Fonte: Elaborada pelo autor.

Figura 35 – Diagrama de Sequência do Sistema de Triângulo.



Fonte: Elaborada pelo autor.

Tabela 18 – Condições de Guarda dos Operandos do Sistema de Triângulo.

alt	Guarda
equilatero	(a < b+c) and (b < a+c) and (c < a+b) and (a > 0) and (b > 0) and (c > 0) and (a = b) and (b = c)
Isósceles	(a < b+c) and (b < a+c) and (c < a+b) and (a > 0) and (b > 0) and (c > 0) and ((a = b and b <> c) or (a = c and c <> b) or (b = c and c <> a))
escaleno	(a < b+c) and (b < a+c) and (c < a+b) and (a > 0) and (b > 0) and (c > 0) and (a <> c) and (a <> c) and (b <> c)

Fonte: Elaborada pelo autor.

APÊNDICE

B

BUSCA SISTEMÁTICA DE TRABALHOS RELACIONADOS

Para realização de uma busca sistemática na literatura mais genérica foi construída a seguinte *string* de busca: *("sequence diagram"OR "sequence diagrams"OR "UML diagram"OR "UML diagrams"OR "UML model"OR "UML models") AND ("test case generation"OR "test generation"OR "test suit generation"OR "MBT"OR "model-based testing"OR "test case"OR "test cases")*. Os outros temas tratados neste trabalho foram considerados nos critérios de inclusão e exclusão de trabalhos, pois quando adicionados na *string* de busca, poucos trabalhos retornaram.

B.1 Metodologia de Busca

O processo de busca sistemática foi realizado em cinco passos:

1. Seleção dos estudos através da string de busca.
2. Exclusão dos estudos duplicados e que não foram escritos em inglês.
3. Exclusão de estudos com base na leitura do título, resumo e palavras-chaves.
4. Exclusão de estudos com base na leitura dinâmica do texto completo.
5. Escrever uma breve descrição dos estudos incluídos, comparando-os com este trabalho.

No Passo 1, a busca dos trabalhos foi realizada através da seleção em anais de conferências ou periódicos das bibliotecas digitais a seguir:

- IEEE Xplore (http://ieeexplore.ieee.org/)
- Springer (https://link.springer.com/)

- Scopus (http://www.scopus.com/)
- Association for Computing Machinery (ACM) (https://dl.acm.org/)
- Elsevier-Science Direct (http://www.sciencedirect.com/)

A *string* de busca foi adaptada e executada em cada biblioteca digital. Importante relatar que na bliblioteca digital *Elsevier-Science Direct*, por ter uma limitação de no máximo oito termos na busca, utilizou-se a *string* de busca: *("sequence diagram"OR "UML diagram"OR "UML model") AND ("test case generation"OR "test generation"OR "test suit generation"OR "model-based testing"OR "test case")*. O resultado da quantidade de estudos retornados em cada base são apresentados na Tabela 19.

Tabela 19 – Estudos retornados no Passo 1 da busca sistemática de trabalhos relacionados.

Biblioteca Digital	Quantidade
IEEE Xplore	150
Springer	2.534
Scopus	405
ACM	1.004
Elsevier-Science Direct	949
Total	5.042

Fonte: Elaborada pelo autor.

No Passo 2, os trabalhos duplicados e que não foram escritos em inglês foram excluídos. Além disso, na biblioteca digital *Springer* foram selecionados neste passo os 1.000 trabalhos mais relevantes, pois a ferramenta de busca só permite exportar 1000 itens. A Tabela 20 apresenta a quantidade de estudos retornados no Passo 2.

Tabela 20 – Estudos retornados no Passo 2 da busca sistemática de trabalhos relacionados.

Biblioteca Digital	Quantidade
IEEE Xplore	79
Springer	905
Scopus	311
ACM	939
Elsevier-Science Direct	680
Total	2.914

Fonte: Elaborada pelo autor.

No Passo 3 foram lidos os títulos, resumos e palavras-chaves de todos os trabalhos retornados no Passo 2 e excluídos os trabalhos que não abordam a geração de testes a partir

Tabela 21 – Estudos selecionados no Passo 3 da busca sistemática de trabalhos relacionados.

Biblioteca Digital	Quantidade
IEEE Xplore	39
Springer	42
Scopus	106
ACM	55
Elsevier-Science Direct	32
Total	274

Fonte: Elaborada pelo autor.

de modelos UML e que não utilizam a MDA ou modelos formais. A Tabela 21 apresenta a quantidade de estudos selecionados no Passo 3.

No Passo 4 realizou-se a leitura dinâmica no texto completo de todos os estudos do Passo 3 e foram selecionados os trabalhos que mais se relacionam com esta tese de doutorado. A Tabela 22 apresenta a quantidade de estudos selecionados no Passo 4.

Tabela 22 – Estudos selecionados no Passo 4 da busca sistemática de trabalhos relacionados.

Biblioteca Digital	Quantidade
IEEE Xplore	3
Springer	1
Scopus	3
ACM	2
Elsevier-Science Direct	3
Total	12

Fonte: Elaborada pelo autor.

Por fim, no Passo 5 foi realizada uma breve descrição dos trabalhos incluídos na seleção final, apresentando as similaridades e diferenças com esta tese de doutorado. A descrição destes estudos foi apresentada na Subseção 5.2.1 do Capítulo 5.


ICMC
USP
SÃO CARLOS