UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE ELETRICIDADE
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE ELETRICIDADE

GERSON LOBATO RABELO FILHO

UMA ABORDAGEM BASEADA EM ENGENHARIA DIRIGIDA POR MODELOS E COMPUTAÇÃO EM NUVEM PARA SUPORTAR O TESTE DE MODELOS SAAS DE CÓDIGO ABERTO

São Luís – MA
2015

GERSON LOBATO RABELO FILHO

UMA ABORDAGEM BASEADA EM ENGENHARIA DIRIGIDA POR MODELOS E COMPUTAÇÃO EM NUVEM PARA SUPORTAR O TESTE DE MODELOS SAAS DE CÓDIGO ABERTO

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão para obtenção do título de Mestre em Engenharia de Eletricidade, área de Concentração Ciência da Computação.

Orientador: Prof. Ph.D. Zair Abdelouahab
Coorientador: Prof. Dr. Denivaldo Cicero Pavão Lopes

São Luís – MA
2015

Rabelo Filho, Gerson Lobato
Uma abordagem em engenharia dirigida por modelos e computação em nuvem para suportar o teste de modelos SAAS de código aberto / Gerson Lobato Rabelo Filho. — São Luís, 2015.
115 f.

Orientador**:** Prof. Ph. D. Zair Abdelouahab.
Dissertação (Mestrado) – Universidade Federal do Maranhão, Programa de Pós-Graduação em Engenharia de Eletricidade, 2015.

1. Engenharia de programas. 2. Open SaaS. 3. Computação em nuvem. 4. Software livre. I. Título.

CDU 004.27

Aos meus amados avós João e Eulinda.

AGRADECIMENTO

Agradeço em primeiro lugar a Deus, Senhor e Pai, por toda a sabedoria, força, paz e luz concedidos para superar as dificuldades no percurso e alcançar todas as vitórias.

Aos meus pais, Gerson Lobato e Maria Santana, por toda as lições de vida ensinadas, todo o apoio e amor empreendidos durante estes anos.

Aos meus avós João Câncio e Eulinda, por toda a alegria proporcionada em vida.

À minha namorada Cintia, pela força e estímulo concedidos, além de todo o amor manifestado desde o início de tudo.

A todos os meus familiares que me acompanharam durante a realização do mestrado.

Ao meu orientador, professor Ph. D. Zair Abdelhouahab, por todo o suporte, aulas e conselhos repassados durante a pesquisa, assim como a confiança depositada.

Ao meu coorientador, professor Dr. Denivaldo Lopes, pela orientação, compreensão e sabedoria repassada durante a pesquisa.

Aos meus colegas de laboratório Pablo, Amanda e Wesley e todos os que estiveram no LESERC, por todos os momentos vividos e sorrisos compartilhados.

Aos meus colegas Claudio, Thiago, Rayane, Raquel, Antonio Oseias, Dhileane, Messias, Paulo e todos que estiveram juntos durante as disciplinas do mestrado, durante as provas e trabalhos desenvolvidos e durante os momentos de descontração.

A CAPES pelo financiamento da pesquisa.

A todos que direta ou indiretamente contribuíram para a realização deste trabalho.

“Tudo posso Naquele que me fortalece.”
(Filipenses 4, 13)

RESUMO

A Computação em Nuvem é um paradigma computacional que surgiu com a ideia de serviços, recursos e funcionalidades baseados em nuvem e que são ofertados por empresas para usuários finais através de modelos de entregas de serviços. Os modelos principais são *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) e *Software as a Service* (SaaS). Os serviços e recursos ofertados são constantemente testados pelas equipes de manutenção e teste com o objetivo de detectarem e eliminarem as falhas presentes. Em relação aos modelos SaaS, os tipos de teste mais usados são os funcionais, de performance, de escalabilidade, de componentes e os testes baseados nos inquilinos dos modelos SaaS. No entanto, testes como os de confiabilidade, de usabilidade, de disponibilidade e de aceitação, mais orientados ao cliente, são pouco realizados. No campo da Engenharia Dirigida por Modelos (*Model Driven Engineering* - MDE), os testes em modelos SaaS, PaaS e IaaS são feitos através de abordagens como a transformação de modelos e a geração de casos de teste de acordo com os tipos de teste, permitindo que os serviços e recursos sejam testados com maior qualidade e eficiência. Esta dissertação apresenta uma abordagem baseada em Engenharia Dirigida por Modelos para suportar a geração de casos de teste de disponibilidade, confiabilidade, usabilidade (após o usuário utilizar o modelo SaaS de código aberto e responder um questionário sobre seu perfil e sobre o modelo) e aceitação para modelos SaaS de código aberto (*Open Software as a Service* – Open SaaS). Um *framework* e metamodelos são propostos para este fim, além de métricas quantitativas com o propósito de analisar estes critérios para o modelo Open SaaS proposto.

Palavras-chave: Computação em Nuvem, Engenharia Dirigida por Modelos, *Open SaaS*, Testes.

ABSTRACT

Cloud Computing is a computational paradigm which came with the idea of cloud-based services, resources and functionalities offered by enterprises to end users through service delivery models. The main models are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Offered services and resources are commonly tested by testing and maintenance teams with the purpose of detecting and eliminating present failures. In relation to SaaS models, the most common test types used are functional, performance, scalability, component-based and tenant-based ones. However, tests like reliability, availability, usability and acceptance ones, more oriented to users, are less performed. In the field of Model Driven Engineering (MDE), the tests in SaaS, PaaS and IaaS models are performed through approaches like transformation of models and generation of test cases according to test types, allowing services and resources being tested with great quality and efficiency. This dissertation shows an approach based on Model Driven Engineering to support generation of availability, reliability, usability (after user uses the open-source SaaS model and answer a questionnaire about his/her profile and about the model) and acceptance test cases for SaaS models with open source code (Open Software as a Service – Open SaaS). A framework and metamodels are proposed to this end, besides quantitatives metrics with the purpose of analize these criteria for the proposed Open SaaS model.

Keyword: Cloud Computing, Model Driven Engineering, Open SaaS, tests.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE SIGLAS

API Application Programming Interface

AST Abstract Testing Suite

ATL Atlas Transformation Language

CADP CAESAR/ALDEBARAN Development Package

CFT Coverage of Fault Tolerance

CFR Coverage of Failure Recovery

CRAM Computing Resource Allocation Meter

CRUM Computing Resource Utilization Meter

CWM Common Warehouse Metamodel

EBNF Extended Backus-Naur Form

EC2 Amazon Elastic Compute Cloud

EMF Eclipse Modeling Framework

ESR Effective Scalable Range

ESS Effective System Scalability

FCTSaaS Framework gerador de Casos de Teste para SaaS

GMF Graphical Modelling Framework

HPC High Performance Computing

IF intermediary Format

IaaS Infrastructure as a Service

M2C Model 2 Code

M2M Model 2 Model

M2T Model 2 Text

MACT4SaaS Metamodel for Abstract Cloud Testing for Software as a Service

MDA Model-Driven Architecture

MDD Model Driven Development

MDE Model-Driven Engineering

MDT Model-Driven Testing

MOF Meta Object Facility

MOF2QVT Meta Object Facility 2 Query/View/Transformation

MPIT Metamodel for Platform Independent Testing

MTBE Model Transformation By-Example

NIST National Institute of Standards and Technology

open-SaaS open-source Software as a Service

PaaS Platform as a Service

PC Performance Change

PDF Portable Document Format

PHP PHP Hypertext Preprocessor

PIM Platform Independent Model

PRR Performance Resource Ratio

PSM Platform Specific Model

QoS Quality of Service

QVT Query/View/Transformation

RA Resource Allocated

ROS Robustness of Service

RU Resource Utilization

SA Service Accuracy

SaaS Software as a Service

SCM System Capacity Meter

SEC System Effective Capacity

SL System Load

SLA Service Level Agreement

SLM System Load Meter

SP System Performance

TaaS Testing as a Service

UML Unified Modelling Language

UPM Unified Process Model

UTP UML Testing Profile

XMI XML Metadata Interchange

SUMÁRIO

1 Introdução

Este capítulo descreve a contextualização em que a dissertação foi desenvolvida, a problemática, a solução proposta e os objetivos, a metodologia empregada e por último a apresentação dos demais capítulos da dissertação.

1.1 Contexto

Atualmente, a tecnologia segue a tendência da Computação em Nuvem. Este paradigma é baseado na ideia de recursos e serviços oferecidos através de uma nuvem, gerenciada por provedores de nuvem. As maiores empresas em tecnologia atualmente (*Amazon*, *Google*, *Microsoft*, *Salesforce*, *SugarCRM* e *Wordpress*) possuem nuvens as quais através delas ofertam seus recursos para milhares de usuários. Suas características particulares [21] explicam o motivo do rápido crescimento deste paradigma: escalabilidade, orientação a serviço, virtualização e a facilidade de uso por parte do usuário.

Os tipos de nuvens mais conhecidos são *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) e *Software as a Service* (SaaS). Cada uma destas nuvens é responsável por ofertar determinado tipo de recurso, como memória e servidores (caso do IaaS), sistemas operacionais e banco de dados (caso do PaaS), até mesmo aplicações inteiras (SaaS). A baixo custo, o usuário pode aproveitar a funcionalidade de cada um destes recursos para realizarem suas tarefas.

Outro paradigma que tem tido destaque no cenário internacional é o da Engenharia Dirigida por Modelos (*Model Driven Engineering* – MDE) [31], que se caracteriza por gerar código a partir de modelos e transformações de modelos. Duas abordagens MDE conhecidas são a Arquitetura Dirigida por Modelos (*Model Driven Architecture* – MDA) [35] e Teste Dirigido por Modelos (*Model Driven Testing* – MDT) [25]. A abordagem MDA realiza geração de código com a intenção de capacitar a interoperabilidade e a portabilidade dos sistemas de software através de modelos [64]. A abordagem MDT tem como objetivo reduzir os esforços na criação de casos de teste de software, oferecendo geração automática a partir de modelos [64]. As ferramentas construídas pra suportar MDT permitem, entre outros beneficios, realização de tarefas de criação e edição de modelos dos componentes de software e geração de conjuntos de testes para os componentes de software [25].

No campo da Computação em Nuvem, casos de teste podem ser gerados para as nuvens IaaS, PaaS e SaaS, permitindo maior confiabilidade, produtividade e qualidade no desenvolvimento [50].

1.2 Problemática

Durante o estudo realizado dentro do campo da Computação em Nuvem, mais particularmente sobre nuvens SaaS de código aberto, e os testes realizados nestas nuvens, notou-se que os testes mais realizados nesta nuvem são os funcionais, de performance, de escalabilidade, de componentes e os testes baseados nos inquilinos [50] [18] [15] [16] [17] [71] [26] [67]. O fato destes testes serem realizados pelos provedores de nuvem faz com que testes como os de disponibilidade, confiabilidade, usabilidade e aceitação sejam pouco realizados ou até mesmo ignorados, o que constitui um problema, pois estes testes estão relacionados diretamente aos usuários da nuvem. Como consequência negativa, eles se afastam das nuvens SaaS implantadas por estas empresas, seja por causa da baixa disponibilidade, confiabilidade, usabilidade e/ou aceitação.

Sendo assim, uma solução que permita a geração dos casos de testes de disponibilidade, de confiabilidade, de usabilidade e de aceitação para nuvens SaaS de código aberto é necessária, pois assim confere-se a este tipo de nuvem maior qualidade por meio destes testes.

Outro ponto observado durante o estudo é a ausência da utilização das métricas de disponibilidade, confiabilidade, usabilidade e aceitação para medir quantitativamente a qualidade de nuvens SaaS de código aberto. Os valores destas métricas estão localizados em uma faixa de valores que vai de 0 a 1 [38], de forma que uma nuvem SaaS de código aberto apresenta melhor disponibilidade, confiabilidade, usabilidade e aceitação do que outra nuvem SaaS quando estas métricas estão mais próximas da unidade. Por exemplo, uma nuvem SaaS cujas métricas de disponibilidade, confiabilidade, usabilidade e aceitação resultaram em 0,9 e outra nuvem SaaS cujas mesmas métricas resultam em 0,5 permite a conclusão de que a primeira nuvem é uma boa nuvem SaaS enquanto que a segunda é uma nuvem SaaS ruim quanto à disponibilidade, confiabilidade, usabilidade e aceitação. Assim, é necessário que a solução para a geração de casos de teste utilize estas métricas, pois dessa forma a solução diferencia-se das demais

soluções existentes, que se preocupam em medir a *performance* e a escalabilidade das nuvens SaaS de código aberto, entre outros critérios.

1.3 Solução proposta

Este trabalho apresenta uma solução para a geração de casos de teste utilizando MDE a fim de serem aplicados em uma nuvem SaaS de código aberto.

Um *framework* é construído com o propósito de gerar casos de teste de disponibilidade, confiabilidade, usabilidade e aceitação a partir de metamodelos e transformação de modelos. A utilização desta solução permitirá a geração dos casos de teste de confiabilidade, disponibilidade, usabilidade e aceitação para nuvens SaaS de código aberto, permitindo assim uma verificação mais abrangente desta nuvem.

Esta solução também utilizará métricas quantitativas de confiabilidade, usabilidade, disponibilidade e aceitação para nuvens SaaS de código aberto, permitindo assim que estes requisitos sejam verificados objetivamente. As fórmulas das métricas de disponibilidade e de confiabilidade foram redefinidas a partir das fórmulas destas métricas apresentadas em [38], ao passo que a fórmula da métrica de usabilidade foi criada a partir de três critérios definidos em [32], [46] e [27] e a fórmula da métrica de aceitação é definida de acordo com a taxa de cenário de uso, o nível de segurança e as métricas de disponibilidade, confiabilidade e usabilidade.

1.4 Objetivo

O objetivo geral do trabalho é fornecer uma solução viável para suportar os testes de usabilidade, confiabilidade, disponibilidade e aceitação para nuvens SaaS de código aberto, e que realize a medição destes requisitos através das métricas acima propostas. Esta solução compõe-se de metamodelos de teste (alterações de alguns metamodelos de teste presentes em [50]), modelos conformes aos metamodelos aqui propostos e de definições de transformação.

1.4.1 Objetivos específicos

Os objetivos específicos são os seguintes:

- Utilização da MDE a fim de construir um *framework* e que seja a extensão do *framework* definido em [50];

- Aplicação do *framework* para a geração dos casos de teste de confiabilidade, usabilidade, disponibilidade e aceitação para a execução em nuvens SaaS de código aberto;

- Utilização deste *framework* a fim de gerar os casos de teste propostos para a nuvem SaaS do *Wordpress* [70] (software com licença *open-source* e livre) e verificar a eficiência dos casos de teste através das métricas de disponibilidade, confiabilidade, usabilidade e aceitação inseridas neles.

1.5 Metodologia

A metodologia utilizada para o desenvolvimento deste trabalho, é apresentada a seguir:

1. Pesquisa bibliográfica para coletar informações sobre os tópicos a seguir:
   a. Engenharia Dirigida por Modelos, incluindo linguagens de modelagem, transformação entre modelos e Teste Dirigido por Modelos;
   b. Computação em Nuvem, incluindo nuvens IaaS, PaaS e SaaS, assim como a realização de testes nestes modelos.
   c. Teste de software e seus tipos.
2. Proposição de um *framework* baseado em MDE que possibilite a geração de casos de teste de confiabilidade, usabilidade, disponibilidade e aceitação para os modelos SaaS de código aberto.
3. Utilização do *framework* para a análise da nuvem SaaS de código aberto do *Wordpress* como experiência prática.

1.6 Apresentação do trabalho

Este trabalho está estruturado em 7 (sete) capítulos descritos como a seguir:

O primeiro Capítulo apresenta o contexto no qual o trabalho está inserido, a problemática identificada a partir de estudo bibliográfico e a solução proposta para a resolução do problema. Neste capítulo, os objetivos e a metodologia adotada no trabalho foram apresentados.

O segundo Capítulo apresenta a fundamentação teórica na qual a solução se baseia, com o propósito de situar o leitor. Fala-se sobre o que é a Computação em Nuvem, suas características, os tipos de nuvens mais conhecidos e exemplos famosos, dando maior foco em nuvens SaaS. Fala-se sobre o que é Engenharia Dirigida por Modelos e suas abordagens, MDA e MDT, dando maior ênfase a esta última abordagem, uma vez que a solução está baseada nela.

O terceiro Capítulo apresenta o estado da arte dos testes em nuvens SaaS, com o objetivo de apresentar onde os estudos relacionados a este assunto estão atualmente. Mostra-se trabalhos relacionados e uma análise comparativa é também apresentada.

O quarto Capítulo introduz a solução proposta por este trabalho, isto é, a abordagem baseada em MDE para testes em nuvens SaaS de código aberto. Mostra-se o *framework* proposto com todos os seus componentes, assim como as métricas que farão a análise objetiva dos casos de teste gerados.

O quinto Capítulo apresenta a implementação do protótipo para o *framework* definido neste trabalho, a qual constitui uma ferramenta que realiza a geração dos casos de teste de disponibilidade, confiabilidade, usabilidade e aceitação para nuvens SaaS de código aberto.

O sexto Capítulo apresenta um exemplo ilustrativo que mostra a utilização do protótipo, os testes realizados e a análise dos resultados.

O sétimo Capítulo apresenta a conclusão extraída com o trabalho realizado, os objetivos alcançados, os trabalhos futuros, as limitações e as contribuições que este trabalho oferece.

Finalizando a dissertação, os anexos que correspondem aos códigos gerados durante a construção da solução são apresentados.

2 Fundamentação Teórica

Este capítulo tem como objetivo apresentar os fundamentos teóricos e tecnologias nos quais o *framework* proposto por este trabalho é baseado.

Os principais conceitos sobre teste de *software*, Engenharia Dirigida por Modelos, Teste Dirigido por Modelos e Computação em Nuvem serão apresentados neste capítulo.

2.1 Teste de *software*

A etapa do teste de *software* é uma parte integral de um ciclo de vida de desenvolvimento de *software* que se espalha sobre todas as fases de desenvolvimento. Esta etapa é importante porque garante a qualidade do *software*, e um *software* que passa por boas práticas de teste tende a ter um bom nível de qualidade, ao mesmo tempo que se um teste não é bem executado, o *software* tende a ter uma qualidade ruim. Um dos seus principais desafios é desenvolver e manter uma plataforma de teste desde o início do projeto. A tecnologia da virtualização tem sido uma solução para este desafio desde o começo dos anos 1960 [26].

Vários tipos de teste de *software* foram criados com o objetivo de testar a configuração, os requisitos funcionais, a segurança, a usabilidade, etc. Dependendo do alvo do teste, os principais tipos de *software* são:

- Teste de configuração: testa se o *software* funciona no *hardware* a ser instalado;
- Teste de instalação: testa se o *software* funciona como desejado, em diferentes *hardwares* e sob diferentes condições, como pouco espaço de memória, interrupções de rede, interrupções de instalação, etc;
- Teste de integridade: testa a resistência do *software* a falhas (robustez);
- Teste de segurança: testa se o sistema e os dados são acessados de maneira segura, apenas pelo autor das ações;
- Teste de unidade: testa um componente isolado ou uma classe do sistema;
- Teste de integração: testa se um ou mais componentes combinados funcionam de maneira satisfatória.

- Teste de *performance*: este tipo de teste divide-se em três tipos:
    - Teste de carga: testa o *software* sob as condições normais de uso. Ex.: tempo de resposta, número de transações por minuto, usuários simultâneos, etc;
    - Teste de *stress*: testa o *software* sob condições extremas de uso: grande volume de transações e usuários simultâneos, e picos excessivos de carga em curtos períodos de tempo;
    - Teste de estabilidade: testa se o sistema se mantém funcionando de maneira satisfatória após um período de uso;
- Teste de usabilidade: teste focado na experiência do usuário, consiste de *interface, layout,* acesso às funcionalidade, etc;
- Testes de caixa branca e caixa preta: o teste de caixa branca realiza o teste de código do *software*, enquanto que o teste de caixa preta testa a saída dos dados usando entradas de vários tipos. Estas entradas não são escolhidas conforme a estrutura do *software*;
- Teste de regressão: consiste na aplicação de versões mais recente do *software*, para garantir que não surgiram novos defeitos em componentes já analisados;
- Teste funcional: testa os requisitos funcionais do *software*, as funções e os seus casos de uso.

Os testes dos requisitos não-funcionais também é outro tipo de teste realizado atualmente. O teste de *performance* e de segurança são dois tipos de testes não funcionais.

A forma de execução destes tipos de teste é realizada através de casos de teste, que podem ser agrupados ou não em suítes de teste. Junto com o resultado da execução, os relatórios de execução são criados. Desta forma, os defeitos podem ser identificados e corrigidos. Em sistemas de grande escala, a quantidade de casos de teste varia de centenas a milhares, solicitando recursos computacionais significativos e longos tempos de execução.

A abordagem para o teste de *software* pode ser baseada nas especificações do *software*, na sua implementação, ou em ambas. Durante o teste baseado na especificação, os requisitos guiam o processo de seleção do caso de teste, e oferece um meio de avaliar a adequação do teste. O teste baseado na

implementação foca em exercitar o programa sob teste, isto é, testar todas as declarações e caminhos que o programa realiza [34].

O teste tradicional de *software* envolve algumas deficiências [72]:

- Difícil obtenção de um poder computacional de larga escala: algumas vezes, milhõe de usuários virtuais precisam ser gerados no teste de *performance*, o que necessita de grande *hardware*. Com isso, é quase impossível terminar o teste especialmente para pequenas empresas ou indivíduos;
- Difícil obtenção dos *software* de teste e *hardware* desejados para criar o ambiente de teste: os *softwares* de teste tendem a ser caros, além de possuírem rápida atualização. Dessa forma, é um desperfício obter o *software* e o *hardware* para um certo teste;
- Configuração complexa do ambiente de teste: a combinação de configurações (configuração de rede, do *firewall* de segurança, do registro, o relacionamento com outras aplicações e muito mais) tende a crescer via parâmetros geométricos, além de serem extremamente difíceis, dependendo do *software* de teste escolhido;
- Difícil construção dos casos de teste: por causa da ausência de acúmulos, o teste tradicional tem que construir diferentes casos de teste para cada teste, especialmente no campo dos testes *fuzzy*, de segurança e de *performance*, sendo que este necessita da experiência de um especialista.

Devido estas dificuldades, um esforço para migrar o teste de *software* para a nuvem vem sendo discutido nos últimos anos [58]. Embora a migração do teste de *software* para a nuvem não seja necessariamente a melhor solução, nem é um processo automático, muito menos fácil, este recurso diminui a exigência de grande poder computacional em larga escala e diminui as demais deficiências mostradas acima.

2.2 Engenharia Dirigida por Modelos

A Engenharia Dirigida por Modelos (*Model Driven Engineering* - MDE) é um paradigma que tem como base a utilização de modelos, responsáveis por gerenciarem a complexidade no desenvolvimento, manutenção e evolução do software. Os exemplos mais famosos deste paradigma são a Arquitetura Dirigida por Modelos (MDA) [35]; o *Eclipse Modeling Framework* (EMF), do Projeto Eclipse [9]; e

o Teste Dirigido por Modelo (MDT) [25] [64]. A utilização da abordagem MDE tem como vantagens a harmonização entre diferentes tecnologias envolvidas, eliminando o problema da incompatibilidade entre elas e o estabelecimento de uma abordagem para definir metodologias claras, para desenvolver sistemas em qualquer nível de abstração e para organizar e automatizar as atividades de teste e validação. Outro benefício no uso da MDE é a elevação do nível de abstração, uma vez que os modelos estão mais próximos do domínio do problema [1].

A abordagem MDE é proposta através de modelos, e tais modelos estão estruturados em quatro níveis, demonstrados na Figura 1 a seguir.

A Figura 1 exibe a arquitetura em quatro níveis, os quais são:

- M0: representa um uso particular do modelo definido no nível M1;
- M1: representa o modelo propriamente dito e que é definido por meio de uma linguagem de modelagem, por exemplo, a UML;
- M2: o nível dos metamodelos, que contem a especificação dos modelos definidos no nível M1. Exemplos de metamodelos neste nível são o metamodelo do *Unified Process Model* (UPM) [52], o metamodelo da *Unified Modelling Language* (UML) [55] e o metamodelo do *Common Warehouse Metamodel* (CWM) [53];
- M3: o nível dos metametamodelos. Este nível contem a especificação dos metamodelos definidos no nível M2. O metametamodelo da linguagem *Meta Object Facility* (MOF) [54], que especifica os metamodelos do nível M2, é o exemplo mais conhecido.

MDE foca na exploração assim como na criação de modelos específicos de domínio que capturam informações sobre o problema a um nível adequado de abstração, ao invés de focar em conceitos algorítmicos ou de computação em baixo nível. Ela se baseia no gerenciamento automatizado de modelos e em particular no suporte automatizado para transformação de modelo. Uma transformação pode ser de modelo para modelo (*Model 2 Model* – M2M) ou modelo para texto (*Model 2 Text* – M2T). Uma transformação M2M consiste em um modelo de entrada ser transformado para outro modelo – provavelmente expresso em uma linguagem de modelagem diferente -, e uma transformação M2T consiste em um modelo de entrada ser transformado em artefatos textuais como código, documentação e relatórios lidos por pessoas [47].

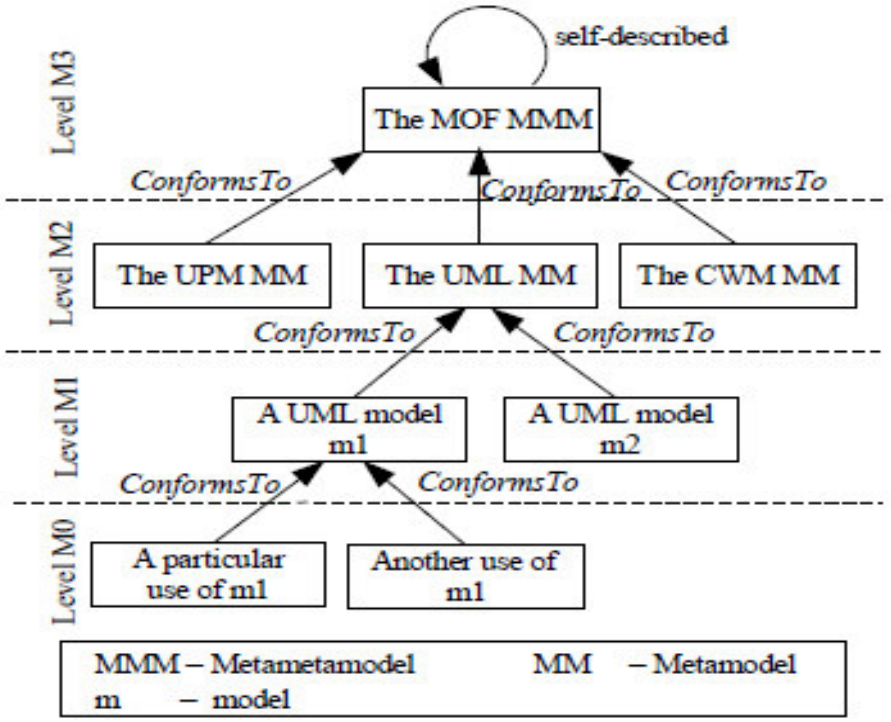

Figura 1. Arquitetura que exibe a organização dos modelos [51]

A Figura 2 exibe o *framework* básico do MDA. Nele, é visto como os principais conceitos da MDE estão estruturados.

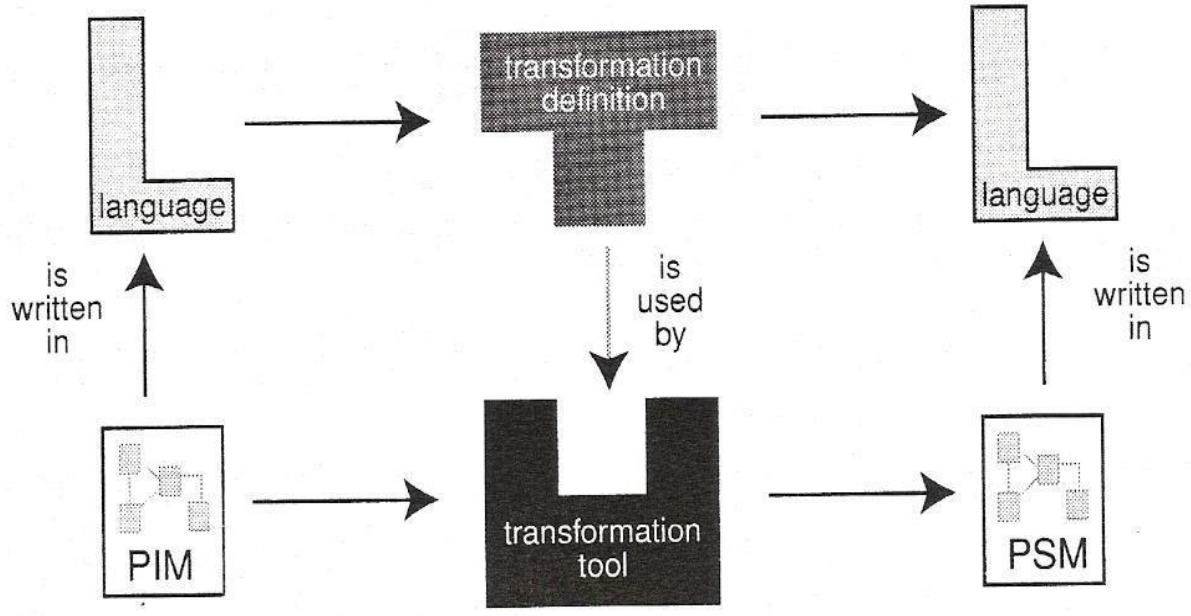

Figura 2. Visão geral do *framework* básico do MDA [51]

O modelo independente de plataforma (*Platform Independent Model* – PIM) reflete a lógica do negócio. Ele é independente da plataforma de implementação final, sendo construído através uma linguagem de modelagem, por exemplo, o UML.

O modelo específico de plataforma (*Platform Specific Model* – PSM), por sua vez, descreve um sistema com pleno conhecimento sobre a plataforma final, sendo construído também por uma linguagem de modelagem, por exemplo, o UML. Uma definição de transformação é um conjunto de regras de transformação que descreve como um modelo em uma linguagem fonte (metamodelo fonte) pode ser transformado em um modelo em uma linguagem alvo (metamodelo alvo). Uma definição de transformação é usada por uma ferramenta de transformação, a qual transforma um PIM para um PSM, no caso de uma transformação M2M, e um PSM para texto, no caso de uma transformação M2T. Algumas ferramentas de transformação conhecidas são ATL [30], Epsilon [36] ou QVT [56].

O paradigma da Engenharia Dirgida por Modelos já foi utilizado para resolver problemas em vários campos de estudos, como por exemplo: na distribuição de larga escala de programas em TV digital [38], no desenvolvimento de sistemas de informação médica [45], e no gerenciamento de dados em Sistemas de Detecção de Intrusão [62].

2.2.1 Teste Dirigido por Modelos

O Teste Dirigido por Modelos (*Model Driven Testing* – MDT) é um exemplo do MDE voltado para as atividades de teste de *software*. A sua aplicação resulta na geração de casos de teste, seja para sistemas sob desenvolvimento, seja para códigos gerados por meio do MDA, assumindo dessa forma um papel importante no sucesso desta abordagem MDE.

Algumas vantagens trazidas pelo MDT foram: redução de esforço e de tempo na geração de casos de teste, além de uma geração automática a partir de modelos; desenvolvimento de software mais produtivo; e manutenção de altos padrões de qualidade [25] [64].

Existem diferentes técnicas que são dirigidas por modelos. Elas diferem na linguagem de modelagem usada, no mecanismo automatizado ou não de geração de casos de teste, no alvo do teste (modelos de projeto ou implementação), e no suporte através de ferramenta [48].

Quanto à linguagem de modelagem, por exemplo, Cavarra et al [6] realizam a geração de casos de teste através da extensão da linguagem UML usando perfis UML [13]. Instâncias desses perfis podiam ser construídas usando qualquer

ferramenta CASE capaz de de exportar o modelo como formato XMI (*XML Metadata Interchange*) [57]. Uma ferramenta de transformação foi desenvolvida para mapear os perfis UML construídos para um formato intermediário.

Por sua vez, Javed et al. [28] apresentam um processo de geração de casos de teste unitários automaticamente através da abordagem MDA. Eles utilizam ferramentas de transformação como Tefkat [37] e MOFScript [49] para esta geração. Eles também definem um processo de mapeamento prático de sua abordagem para o padrão UTP (*UML Testing Profile*).

Quanto à geração automática de teste, o mecanismo utilizado por Javed et al. [28] baseia-se em diagramas de sequências para realizar a geração de casos de teste unitários a partir de modelos independentes de plataforma do sistema. Eles iniciam com a modelagem do comportamento do sistema usando diagramas de sequência, os quais são então automaticamente transformados para um modelo geral de caso de teste unitário usando regras de transformação M2M. O modelo de caso de teste resultante é posteriormente transformado para casos de teste concretos e executáveis, usando regras de transformação adicionais. Cavarra et al. [6] utiliza um mecanismo no qual diagramas de classe, de objeto e de estado em UML são usados para definir o modelo do sistema. Diagramas de objeto e de estado são usadas para introduzir diretivas de teste, outro componente do mecanismo, que são estruturas compostas de restrições de teste, critérios de cobertura e propósitos de teste. O modelo do sistema é compilado para uma coleção de máquinas de estado extendidas, escritas na linguagem Formato Intermediário (*Intermediary Format* – IF) [5]. A saída da geração de teste é um suíte de teste abstrato (*abstract test* suite – AST) que contem a sequência de simulações e observações que uma máquina de teste deve realizar para executar o teste sobre a implementação do sistema. Figura 3 apresenta o mecanismo completo.

Quanto ao alvo do teste, Javed et al. [28] propõem uma forma de testar a implementação do sistema usando *xUnit*. Cavarra et al. [6] abordam o modelo UML do sistema a fim de detectar defeitos de projeto.

Por fim, quanto ao suporte de ferramenta, Cavarra et al. [6] compilam os modelos escritos para a linguagem IF. Esta nova representação pode ser verificada usando as ferramentas do CADP (*CAESAR/ALDEBARAN Development Package*) [19]. Javed et al. [28] utiliza como ferramenta uma máquina de transformação baseada em EMF (*Eclipse Modeling Framework*) [10] para gerar casos de teste.

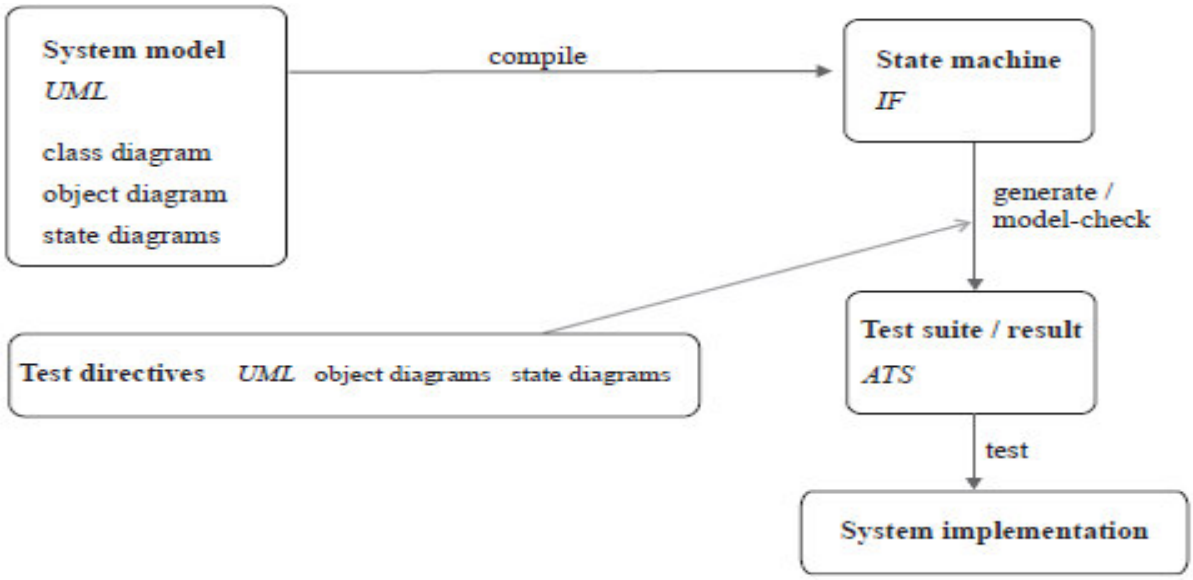

Figura 3. Mecanismo de geração de teste automática em Cavarra et al. [6]

2.2.2 Definição de transformação e regras de transformação

A definição de transformação é o componente da Engenharia Dirigida por Modelos (MDE) que permite que modelos sejam transformados até a geração de código, no caso da abordagem da Arquitetura Dirigida por Modelos (MDA) [35] [50], ou até a geração de casos de teste, no caso da abordagem do Teste Dirigido por Modelos (MDT) [64]. Por exemplo, na abordagem MDA, uma definição de transformação entre dois metamodelos realiza a geração de um PSM a partir de um PIM.

Uma definição de transformação é executada por uma máquina de transformação, como foi demonstrado na Figura 2. Uma definição de transformação contem regras de transformação que, por sua vez, são a descrição de como um ou mais elementos em uma linguagem fonte pode ser transformado em um ou mais elementos de uma linguagem alvo. No entanto, em [41] é demonstrado que uma definição de transformação é antecedida por uma especificação de mapeamento. Uma especificação de mapeamento é o conceito responsável por definir as correspondências entre metamodelos (por exemplo, um metamodelo para construir PIM e outro para construir PSM) [1] [42]. Ambos fazem parte de um processo chamado processo de transformação de modelos.

Os primeiros passos dados para gerar definições e regras de transformação de maneira semi-automática no contexto do MDA estão presentes em [1] e [41]. Em [41] é apresentada uma maneira de gerar definições de transformação a partir de especificações de mapeamento. Para tal propósito, os autores propuseram uma ferramenta baseada em Eclipse que permite a edição de mapeamentos e a geração de definição de transformação a partir destas mapeamentos, tendo como linguagens participantes o UML e o WSDL [68]. A Figura 4 exibe a ferramenta baseada em Eclipse.

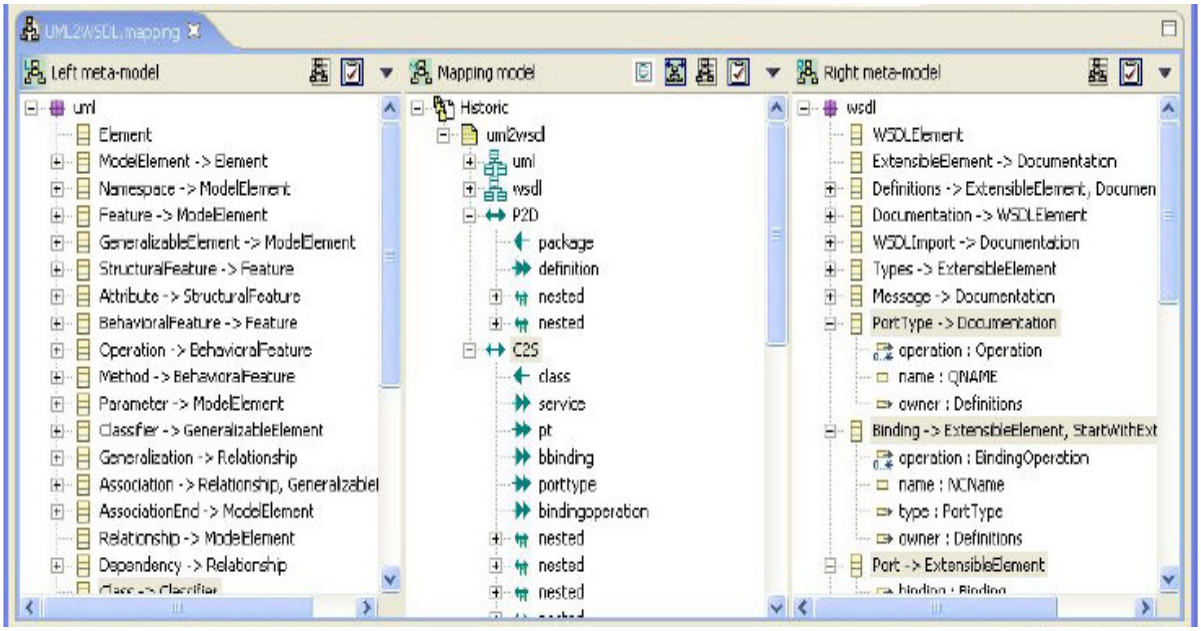

Figura 4. Ferramenta de edição de mapeamento para geração de definição de transformação [41]

Em [1] os esforços concentram-se na geração semi-automática de regras de transformação dentro do contexto da Arquitetura Dirigida por Modelos. Os motivos para estes esforços se baseiam em tornar a transformação de modelos mais fácil, mais rápida e em reduzir o processo de custo da geração das regras. Além de introduzir os conceitos de mapamento e correspondência de metamodelos como entidades de primeira classe, são adicionadas também duas novas operações: a de adaptação e a de derivação, que permitem justamente a geração semi-automática das regras de transformação de modelos. A adaptação consiste na responsabilidade do usuário especialista que deve aceitar, discordar ou modificar os mapeamentos obtidos, e de, além disso, especificar as correspondências que o correspondente foi incapaz de encontrar [1], enquanto que a derivação consiste na definição completa do modelo de mapeamento gerado na operação anterior, permitindo a geração

automática do modelo de transformação [1]. A Figura 5 exibe o processo de transformação semi-automático proposto.

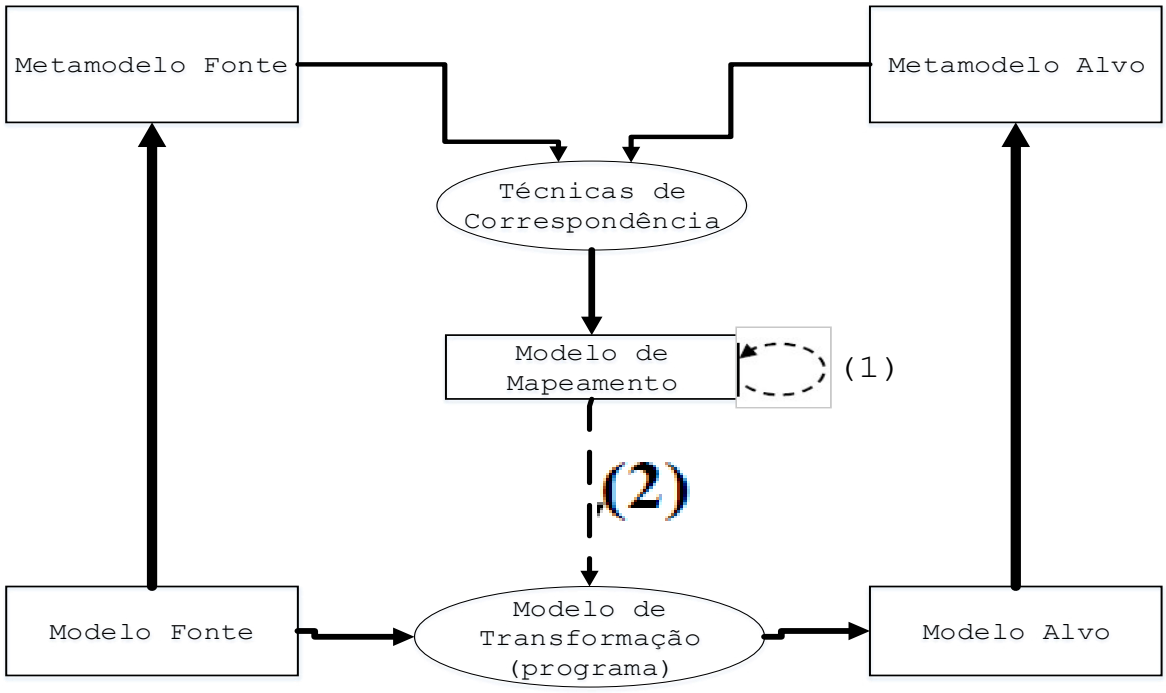

Figura 5. Processo de transformação semi-automático (proposto em [1])

Na Figura 5, os números (1) e (2) correspondem às operações de adaptação e derivação, respectivamente. Observa-se que a adaptação é realizada no modelo de mapeamento, gerada após técnicas de correspondências serem aplicadas sobre os metamodelos fonte e alvo. Após esta etapa, é gerado semi-automaticamente o modelo de transformação, isto é, a definição de transformação que contém as regras de transformação e que serão aplicadas sobre os modelos fonte e alvo.

Wimmer et al. [69] propõem a geração de definições de transformação por meio de exemplos (*Model Transformation By-Example* – MTBE). Seu argumento se fundamenta na afirmação de que o foco específico da implementação baseada em metamodelos torna difícil aos modeladores desenvolver transformações de modelo, porque os metamodelos não necessariamente definem todos os conceitos de linguagens explicitamente e que estão disponíveis para propósito de notação. Esta forma de definir transformações de modelo através de exemplos tem o objetivo de definir mapeamentos inter-modelo que representam correspondências semânticas entre modelos concretos de domínio e, em seguida, especificar diretamente as regras de transformação de modelo ou mapeamentos baseados na sintaxe abstrata.

Eles afirmam também que aplicando esta abordagem no EMF (*Eclipse Modeling Framework*) [9] e no GMF (*Graphical Modelling Framework*) [11] é possível reusar as restrições já disponíveis para derivar regras de transformação expressas em ATL (*Atlas Transformation Language*) [30] entre as linguagens de modelagem fora dos mapeamentos na sintaxe concreta. O conhecimento do usuário sobre a notação da linguagem de modelagem (por exemplo, o UML [55]) é suficiente para a definição de transformações de modelo considerando as correspondências semânticas. Portanto, eles concluem que nenhum detalhe sobre metamodelos é solicitada. Entretanto, eles fazem uma observação de que é necessário alinhar dois modelos, as quais representam o mesmo domínio do problema, para automaticamente derivar as regras de transformação.

Em [40] um metamodelo para definições de transformação é proposto como uma forma de superar as deficiências presentes na abordagem baseada em transformações gráficas, principalmente realizadas por meio de diagramas de classe em UML, as quais são a insuficiência no suporte direto para os diagramas de classe e a definição formal explícita. O metamodelo proposto utiliza a técnica de correspondência de *template* de regras, a qual é baseada em dois algoritmos: o algoritmo de iteração de regras e o algoritmo de escalonamento de regras, que formam o modelo computacional, conforme a este metamodelo. A Figura 6 exibe este metamodelo.

Por sua vez, Cuadrado et al. [7] definem um modelo componente para transformações de modelos. Eles observaram que enquanto outros paradigmas de desenvolvimento tornam as técnicas disponíveis para aumentar a produtividade através da reutilização, há poucas propostas para o reuso de transformações de modelos entre linguagens de modelagem. Como consequência, quase sempre as definições de transformação tem que ser escritas desde o rascunho mesmo que outras similares existam. Para resolver este problema, eles propuseram uma técnica que permite a reutilização flexível de transformações de modelo. Esta técnica é baseada em programação genérica e em desenvolvimento por meio de componentes, e tem os objetivos de definir e instanciar *templates* de transformação, além de encapsular e compor estes *templates*. Este modelo componente é suportado por um implementação que abordou atualmente a linguagem ATL [30]. A Figura 7 exibe um excerto do metamodelo que descreve o modelo componente proposto em [8].

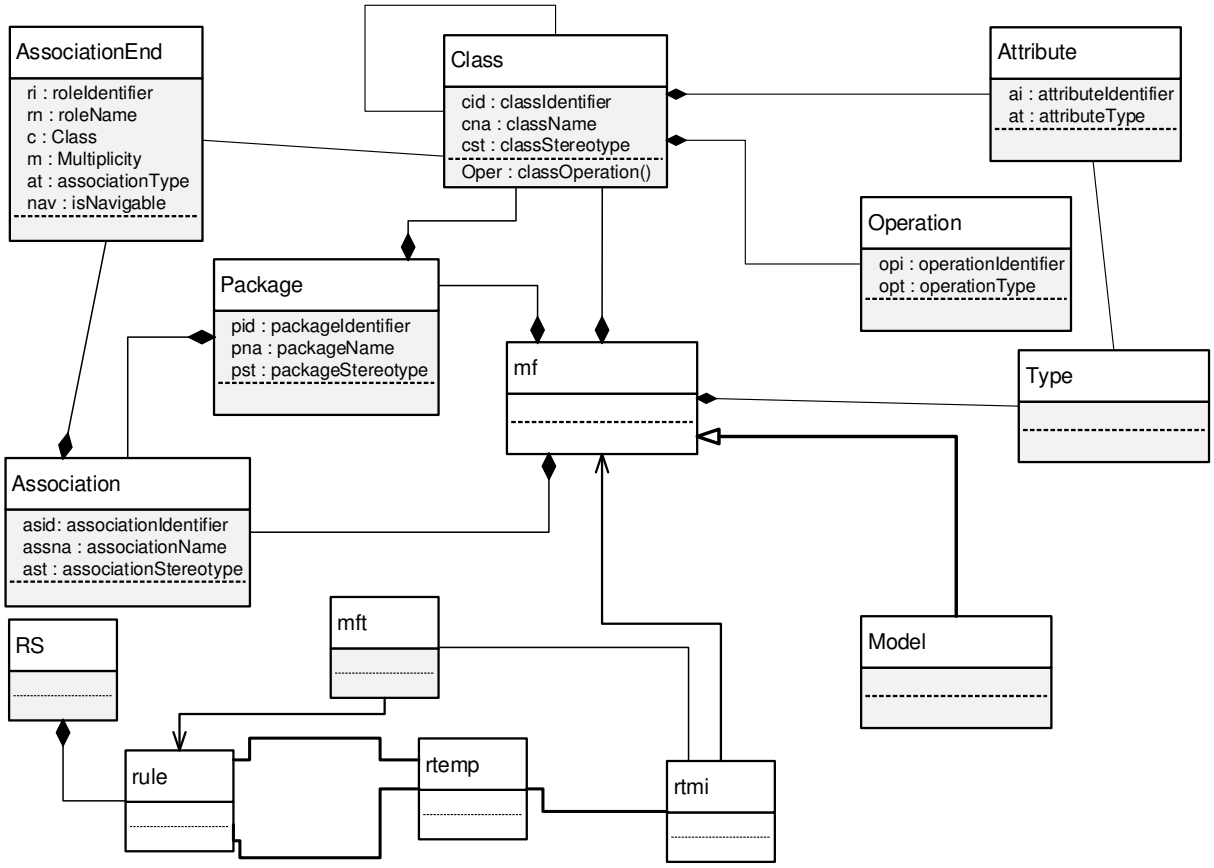

Figura 6. Metamodelo de Definição de Transformação definido em [40]

Alguns elementos presentes na Figura 7 são:

- *Component*: principal classe deste metamodelo, representa uma entidade abstrata que realiza uma operação sobre um conjunto de modelos fonte e alvo.
- *Meta-Model*: classe que manipula os modelos que participam da transformação. Ela possui também portas que são fornecidas pela classe *Component* que definem a interface componente.
- *Concept*: tipo especial de metamodelo que admite conexões (*bindings*) com outros metamodelos.
- *TransformationComponent*: classe que representa um único componente que encapsula um *template* de transformação, criada usando alguma linguagem de modelagem, por exemplo, o ATL [30].
- *CompositeComponent*: classe que é composta de vários componentes, simples ou compostos.

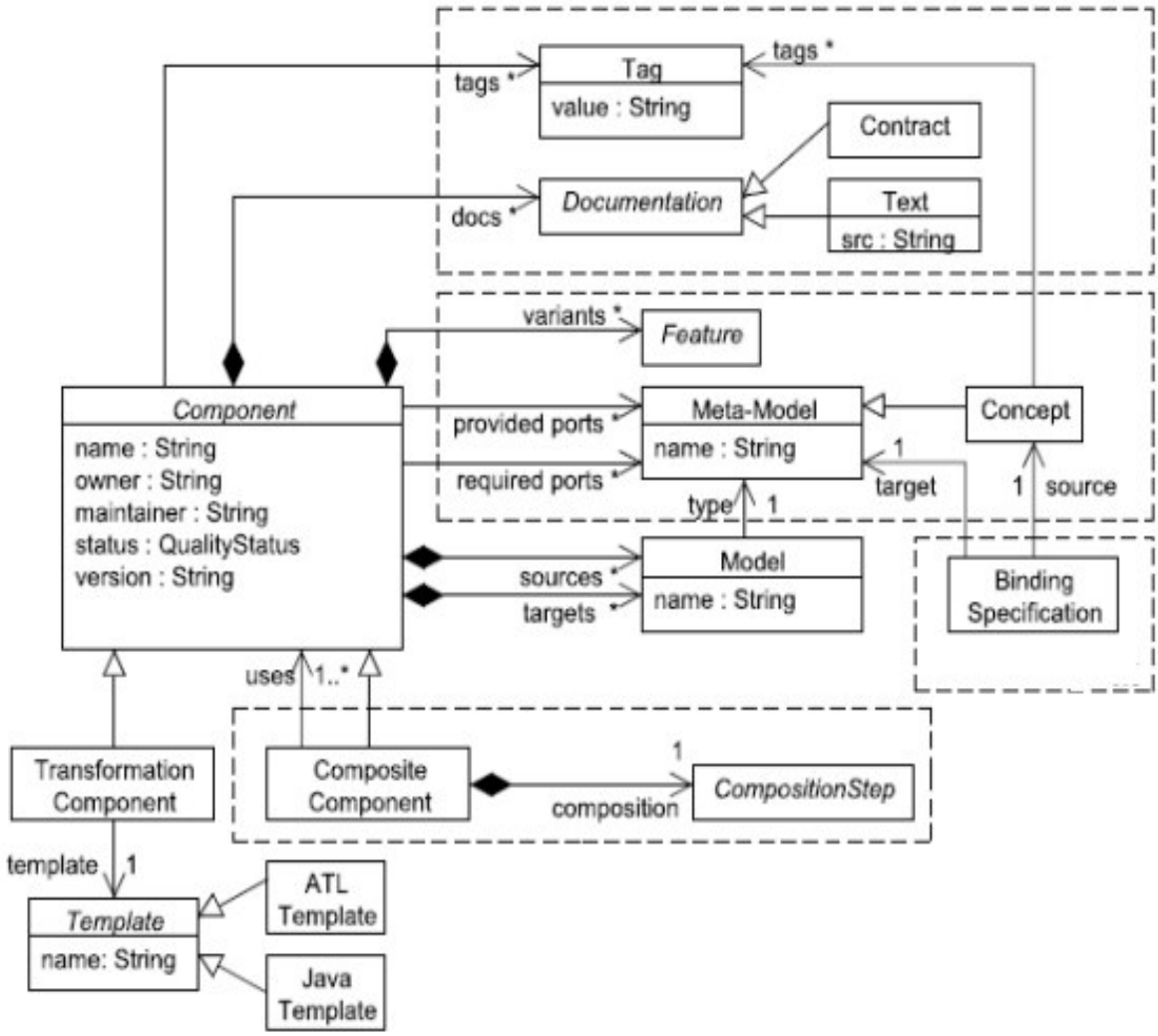

Figura 7. Excerto do metamodelo representativo do modelo componente para transformação de modelos [7]

2.2.3 Linguagens de Transformação de Modelos

Com a crescente aplicação do Desenvolvimento Dirigido por Modelo (*Model Driven Development* – MDD), as abordagens e ferramentas de transformação de modelos se tornaram um campo de pesquisa emergente. Um aspecto importante a ser considerado é a linguagem de transformação de modelo escolhida para facilitar a definição de transformações de modelo. É essencial investigar a expressividade da linguagem, isto é, seu nível de auto-explicação, que tipo de construções de linguagem ele oferece, e o nível de efetividade em que esta definição de transformação pode ser processada [12].

Algumas das linguagens criadas nos últimos anos são ATL [30], Epsilon [36], e *MOF2QVT* [56]. Estas linguagens serão brevemente explanadas:

- **ATL**: esta linguagem híbrida permite tanto construções declarativas quanto imperativas. Esta linguagem de transformação de modelos possui um conjunto de ferramentas como editor, compilador, máquina virtual, e um depurador [30]. Além destas ferramentas, ela possui um ambiente de desenvolvimento integrado que é baseado em Eclipse.
- **Epsilon**: outro tipo de linguagem de transformação de modelos híbrida, ela foi construída sobre a infraestrutura fornecida pela plataforma de gerenciamento de modelos Epsilon [36]. Esta plataforma é um componente do projeto Eclipse GMT (*Generative Modeling Technology*) [10], e que fornece uma infraestrutura para implementar linguagens de gerenciamento de modelo interoperáveis e uniformes.
- **MOF2QVT (Meta Object Facility 2 *Query/View/Transformation*)**: a linguagem de transformação de modelos *MOF2QVT* possui uma natureza tanto declarativa quanto imperativa. Sua parte declarativa está organizada em dois metamodelos, o de relações (*Relations*) e o central (*Core*). Sua parte imperativa compõe-se de dois mecanismos de implementação que atuam juntamente com estes metamodelos: o mecanismo de mapeamentos operacionais (*Operational Mappings*), isto é, a linguagem padrão; e o mecanismo de operações *MOF* em caixa-preta (*Black-Box MOF Operation*) [56].

Para este projeto, a linguagem de transformação de modelos escolhida foi a ATL, devido à sua facilidade de acesso e todas as ferramentas que ela possui dentro do ambiente de programação Eclipse, por meio de seu *plugin* [8].

2.3 Computação em Nuvem

A Computação em Nuvem é um paradigma que vem ganhando destaque nos últimos anos na comunidade acadêmica, devido seu modo de operação, isto é, fornecer serviços e recursos sob demanda, suas características e suas vantagens.

Gong et al. [21], em um intuito de diferenciá-la das outras áreas de pesquisa, salientam que as principais características da Computação em Nuvem são a orientação a serviço, baixo acoplamento, forte tolerância a falhas, modelo de negócio baseado no conceito *pay-as-you-use* (pague enquanto usa), requer forte segurança, e facilidade de uso por parte do usuário. Além destas, outras

características são a base via TCP/IP e a virtualização. Todas estas características, de acordo com eles, permitem diferenciá-la da computação em grade e da computação de alta performance (*high performance computing* – HPC).

O conceito deste paradigma também foi motivo de inúmeras discussões. Por exemplo, Kherajani e Shrivastava em [33] definem nuvem como um tipo de sistema de computação paralela e distribuída que consiste de uma coleção de computadores interconectados e virtualizados que são dinamicamente ofertados e apresentados como um ou mais recursos computacionais unificados baseados em acordos de nível de serviço (*Service Level Agreement* – SLA) estabelecidos através da negociação entre o provedor do serviço e os consumidores. Por outro lado, o conceito elaborado pelo Instituto Nacional de Padrões e Tecnologias (*National Institute of Standards and Technology* – NIST) [43] dos Estados Unidos afirma que

> Computação em Nuvem é um modelo que habilita o acesso conveniente em rede e sob demanda para um agrupamento compartilhado de recursos computacionais configuráveis (e.g., redes, servidores, memória, aplicações, e serviços) que podem ser rapidamente ofertadas e lançadas com o mínimo de esforço de gerenciamento ou interação de provedores de serviços [44].

Além destes conceitos, vários outros especialistas deram seus próprios conceitos acerca deste paradigma [20].

2.3.1 Classificação

Uma nuvem pode ser classificada quanto ao tipo de implantação e ao tipo de serviço.

2.3.1.1 Quanto ao tipo de implantação

Quanto ao tipo de implantação, uma nuvem pode ser pública, privada, comunitária ou híbrida [29].

Nuvem pública é o tipo de nuvem que permite aos usuários acesso gratuito aos recursos e funcionalidades dessa nuvem. Seu acesso ocorre através da

interface de um navegador web (*Internet Explorer*, *Mozilla Firefox*, *Google Chrome*). O usuário só paga apenas pelo tempo que gasta para utilizar esta nuvem. Esta nuvem possui um nível de segurança menor, uma vez que seus recursos são acessados gratuitamente por qualquer usuário dessa nuvem. Seu gerenciamento de acesso é responsabilidade do provedor da nuvem.

Nuvem privada é o tipo de nuvem que possui uma política mais rígida quanto ao acesso de recursos. Ela funciona dentro de um centro de dados interno de uma organização, e o acesso aos recursos da nuvem é feito somente pelos funcionários da organização detentora da nuvem. Seu funcionamento é semelhante à intranet [29]. O gerenciamento de acesso aos recursos na nuvem privada é responsabilidade da organização que a mantem.

Nuvem comunitária é o tipo de nuvem que é mantida por várias organizações e somente os usuários dessas organizações tem acesso aos recursos. A infraestrutura dessa nuvem pode ser gerenciada por uma das organizações que utilizam esta nuvem ou por um provedor de fora [29].

Nuvem híbrida é a combinação das nuvens privada e pública. Esta nuvem pode estar ligada a um ou mais serviços externos. É uma maneira mais segura de controlar as aplicações e dados (característica da nuvem privada) e permite ao mesmo tempo acesso a informações através da internet (característica de nuvem pública).

2.3.1.2 Quanto ao tipo de serviço

Quanto ao tipo de serviço, uma nuvem pode ser classificada em *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), *Software as a Service* (SaaS).

2.3.1.2.1 *Infrastructure as a Service* (IaaS)

Este tipo de nuvem é responsável por oferecer recursos como servidores, memória, máquina virtual e outros tipos de *hardware* aos seus usuários. Exemplos conhecidos deste tipo de nuvem são o *Amazon Elastic Compute Cloud* (EC2) [2] e *Eucalyptus* [24].

Um IaaS é um ambiente completo e gerenciado cujo principal objetivo é facilitar a oferta de recursos. Ele é baseado na virtualização de recursos

computacionais que podem ser dinamicamente escaláveis para aumentar ou diminuir de acordo com a demanda de usuários. Tal escalonamento é realizado de maneira transparente.

Como vantagens, um IaaS reduz o investimento em hardware físico assim, elimina custos de segurança e manutenção, libera espaço físico do computador, e possui flexibilidade para aumentar ou diminuir o poder de processamento ou armazenamento [4].

2.3.1.2.2 *Platform as a Service* (PaaS)

Este tipo de nuvem consiste de uma plataforma computacional a qual está assentada sobre um IaaS. Esta camada da arquitetura de uma nuvem oferece recursos como banco de dados, sistemas operacionais, serviços de mensagem e serviço de armazenamento de dados, além de oferecer um ambiente de desenvolvimento de software, facilitando a implantação de aplicações SaaS (*Software as a Service*) sem os custos e complexidades relacionados à compra e gerenciamento de hardware e software adjacente [4].

As vantagens de se usar uma plataforma como um serviço são: baixo investimento inicial, reduzindo menor risco de negócio; manutenção sob resposabilidade do provedor; suporte e atualização mais agéis; foco aplicado no negócio; disponibilidade e segurança de dados aumentadas [4].

Um exemplos de plataforma como serviço é o *Google AppEngine* [22].

2.3.1.2.3 Software as a Service (SaaS)

A última camada da arquitetura de uma nuvem consiste em uma aplicação inteira executando na internet e acessível através de um navegador web. O SaaS é uma forma de entregar o software através da internet, em uma única instância e passível de compartilhamento com os demais usuários [4].

Uma aplicação em SaaS possui as seguintes características importantes [43]: acesso baseado em rede e gerenciamento de *software* disponível comercialmente que são controlados em lugares centralizados, que permitem os consumidores acessarem-na remotamente através da internet.

As vantagens de se usar *software* como um serviço são [4]: custo inicial baixo porque não há taxas de licença; atualizações transparentes, sem existência de arquivos de download ou para serem instalados; melhor gerenciamento da aplicação dada a centralidade do ambiente; alto nível de disponibilidade; escalabilidade infinita da infraestrutura para satisfazer a demanda dos usuários.

Exemplos de provedores SaaS são *Salesforce* [60] e *Wordpress* [70].

2.4 Síntese

Neste capítulo, apresentaram-se os principais conceitos das tecnologias envolvidas na solução proposta.

Iniciou-se demonstrando os principais conceitos ligados à teste de software e Engenharia Dirigida por Modelos (MDE). Este paradigma apresenta o modelo como artífício primário de alto nível para o desenvolvimento e manutenção de sistemas computacionais. Exemplos conhecidos da abordagem MDE são o MDA, o EMF e o MDT. Estes exemplos tem como objetivos gerar código, gerar caso de teste, construir metamodelos de maneira facilitada e com boa qualidade.

Em seguida, os fundamentos sobre o Teste Dirigido por Modelos, definição de transformação e regras de transformação foram apresentados, sendo explicado que este tipo de abordagem MDE permite a geração de casos de teste, muitas vezes de maneira automatizada e com boa qualidade durante ao geração.

Por fim, falou-se sobre Computação em Nuvem, seu conceito e principais aspectos. A Computação em Nuvem é um paradigma computacional de crescente ascenção entre as empresas de tecnologia, possuindo características que a diferem de outros paradigmas (como a computação em grade e a computação de alta performance), além de apresentar como entidade representante a nuvem. Uma nuvem pode ser classificada quanto à implantação (pública, privada, híbrida, comunitária) e quanto ao modelo de serviço (IaaS, PaaS e SaaS).

3 Estado da Arte

Este capítulo tem como objetivo descrever as principais abordagens para realizar testes no ambiente da Computação em Nuvem, principalmente em um ambiente de Software como um Serviço (SaaS).

Uma explanaçao sobre o que é teste em SaaS e os principais tipos de testes feitos nesta nuvem atualmente é realizada.

Por fim, uma análise sobre alguns artigos que trabalham com teste em SaaS é feita com o propósito de destacar os principais pontos de cada um, como, por exemplo, o tipo de teste realizado, a solução proposta, e a presença ou não de métricas.

3.1 Testes em *Software as a Service* (SAAS)

O ambiente SaaS está em constante expansão no mercado mundial devido suas características e vantagens [4] [43]. Portanto, as equipes de manutenção e testes de empresas como *Salesforce* [60] e *Wordpress* [70] realizam determinados tipos de testes com o objetivo de verificar o desempenho e funcionamento de suas respectivas nuvens SaaS, a fim de entregarem um ambiente satisfatório aos seus usuários.

Em [15], um levantamento acerca da definição de teste em SaaS foi realizado. Após esta atividade, foi conceitualizado que teste em SaaS:

> Refere-se a diferentes tipos de atividades de validação em um processo de teste para avaliar a qualidade do SaaS em entregar os serviços funcionais sob demanda em uma infraestrutura de nuvem. Como o teste de software convencional, o processo de teste em SaaS deve validar os componentes funcionais subjacentes, a integração funcional e serviços, assim como requisitos QoS (*Quality of Service*) não funcionais, como desempenho, confiabilidade, disponibilidade, escalabilidade, segurança, e tolerância a falhas. Contudo, teste em SaaS deve garantir a qualidade do *software* SaaS em seu multi-clientelismo, customização/configuração, compatibilidade

com interface usuário, conectividade, e melhorias contínuas dinâmicas [15].

3.1.1 Trabalhos relacionados com testes em SaaS

A seguir, alguns trabalhos relacionados com testes em SaaS serão demonstrados. Os principais tipos de teste são identificados em cada um destes trabalhos, além de mostrar a forma como cada um abordou este assunto.

Testando a Escalabilidade de Aplicações SaaS [67]

Neste trabalho, Tsai et al. [67] discutem o teste de escalabilidade de aplicações SaaS. A partir da constatação de que o paradigma da Computação em Nuvem e as aplicações SaaS receberam significativa atenção nos últimos anos, eles discutem que o teste em SaaS tem várias características únicas, as quais são:

- Efetivo quanto ao custo: a computação em nuvem reduz o custo de hardware e de software alavancando os recursos de nuvem usando recursos virtuais;
- Sob demanda: validação e verificação online em tempo real e em larga escala;
- Teste automático: teste online dinâmico sem custo manual;
- Teste escalável: suportando o multi-clientelismo;
- Contínuo: os serviços de teste trabalham a qualquer tempo;
- O teste pode ser feito até mesmo concorrentemente durante a operação das aplicações.

Para eles, a principal chave para testar a escalabilidade das aplicações SaaS são as métricas utilizadas para tal objetivo. Eles identificaram que as três principais métricas para este tipo de teste são velocidade, eficiência, e a escalabilidade da propriamente dita. Contudo, estas métricas não podem ser consideradas em um ambiente de nuvem devido a complexidade do ambiente de nuvem não ser contemplada por estas métricas; a possibilidade de que diferentes *workloads* que executam nas aplicações SaaS demonstrarem diferentes desempenhos uns dos outros, e as métricas não levarem isso em consideração.

Um conjunto de prováveis métricas de escalabilidade voltadas para aplicações em nuvem foi organizada pelos autores para realizar o teste de escalabilidade. A Tabela 1 exibe estas métricas e suas descrições.

Métrica	Descrição
Tempo de processamento	Mede o tempo de processamento da aplicação
Consumo de recurso	Mede a taxa de consumo de recursos da aplicação
Relação desempenho/recurso	Reflete o relacionamento entre desempenho e os recursos usados
Variância de métrica	Reflete a variância das métricas anteriores

Tabela 1. Métricas de escalabilidade propostas em [67]

A medição da relação desempenho/recursos (*Performance Resource Ratio* – PRR) da aplicação SaaS considera não apenas o tempo que ela realiza para fazer suas tarefas, mas também os recursos consumidos durante o processo. A fórmula para medir o PRR é mostrada a seguir:

$$\text{PRR} = \frac{1}{Tw} * \frac{1}{Cr} \quad [67]$$

Onde T_w representa o tempo de espera para acesso à aplicação SaaS e C_R representa a taxa de consumo do recurso da aplicação SaaS, sendo representada pela fórmula abaixo:

$$C_R = \sum Ri * Ti \quad [67]$$

Onde R_i significa a alocação do recurso *i*, que pode ser largura de banda de entrada e saída, uso de CPU e de memória, e T_i significa o tempo no qual o recurso *i* é usado.

A partir do valor do PRR, pode-se calcular a estabilidade da aplicação através da variável PC (*Performance Change*), que é definida pela seguinte fórmula:

$$PC = \frac{PRR(t) * W(t)}{PRR(t') * W(t')} \quad [67]$$

Onde o valor ideal para PC é a unidade.

Outra questão que foi considerada pelos autores é a complexidade da nuvem, e devido este fato, o valor de PC pode variar entre diferentes execuções de teste. Assim, outra variável que mede a escalabilidade da aplicação chama-se variância de desempenho (*Performance Variance* – PV). Segundo eles, esta métrica mede efetivamente a escalabilidade, uma vez que os *workloads* que executam na aplicação SaaS mudam de desempenho constantemente.

Os autores também sugerem que técnicas de mineração de dados podem ser aplicadas durante o processo de teste, pois problemas como a existência de gargalo no processo de teste de escalabilidade, a ausência na correlação de parâmetros e a escalabilidade e o grande número de casos de teste podem ser resolvidas, respectivamente, através de algoritmos de seleção de recursos, criação de regras de associação e mineração dos relacionamentos de entrada/saída.

Avaliação de Desempenho e de Escalabilidade de SaaS em Nuvens [18]

Neste artigo proposto por Gai et al., eles propõem um método de medição para medir a performance e a escalabilidade de nuvens SaaS. Este método consiste em novos modelos gráficos formais e métricas para avaliar o desempenho das nuvens SaaS, e analisar a escalabilidade do sistema na nuvem.

Os modelos gráficos formais propostos em [18] tem o objetivo de auxiliarem no desenvolvimento de métricas bem definidas que permitam a visualização e avaliação dinâmica e para o desempenho do sistema e a escalabilidade de aplicações SaaS em uma nuvem. Estes modelos são exibidos em gráficos de radar, isto é, um método gráfico de exibir dados multivariados na forma de um gráfico bidimensional de três ou mais variáveis quantitativas representadas nos eixos que começam a partir de um mesmo ponto. Através do uso deste tipo de gráfico, as métricas propostas pelos autores são:

- ***Computing Resource Allocation Meter (CRAM):*** indicador que apresenta a quantidade total de alocações de recursos (armazenamento, memória, tráfego de rede, CPU, e memória *cache*) para a aplicação SaaS *S* em uma nuvem no

tempo de avaliação *t*. A Figura 8 exibe o formato do gráfico de radar para esta métrica.

- ***Computer Resource Utilization Meter (CRUM)***: esta métrica mostra e mede a taxa de utilização dos recursos da nuvem (em porcentagem) pela aplicação SaaS durante a validação desta aplicação na nuvem. Esta métrica é útil para engenheiros e clientes na avaliação de e monitoramento do desempenho da aplicação. A Figura 9 exibe o formato do gráfico em radar para esta métrica.
- ***System Load Meter (SLM***): métrica usada para medir as cargas do sistema que hospeda a aplicação SaaS. Os autores consideram que as cargas de uso do sistema estão distribuídas em três classes: carga de acesso do usuário, carga do tráfego de comunicação e carga do acesso ao armazenamento de dados. Ao contrário dos gráficos que representam as métricas CRAM e CRUM, o gráfico desta métrica é representada através de um triângulo.
- ***System Capacity Meter (SCM)****:* métrica que mede a capacidade do sistema que hospeda a aplicação SaaS. Esta métrica é baseada a partir de três parâmetros: a carga de trabalho do sistema (S*ystem Load* – SL), os atuais recursos alocados (*Resource Allocated* – RA) e o atual desempenho do sistema (*System Performance* – SP). Assim como a métrica SLM, esta métrica é representada através de um triângulo.
- ***System Effective Capacity (SEC)***: esta métrica mede a capacidade efetiva do sistema. Ao contrário da métrica anterior, esta métrica se baseia somente nos atuais recursos utilizados pela aplicação SaaS. Esta métrica é calculada a partir de três parâmetros: a carga do sistema (SL), o desempenho do sistema (SP), e a utilização dos recursos (*Resource Utilization* – RU).
- ***Effective System Scalability (ESS)***: esta métrica, que verifica a escalabilidade do sistema que hospeda a aplicação SaaS, tem como base a métrica SEC e corresponde à relação entre o aumento de carga do sistema e o aumento da capacidade do sistema.
- ***Effective Scalable Range (ESR)***: métrica que corresponde à diferença entre o valor mínimo da métrica SEC e o valor máximo desta mesma métrica.

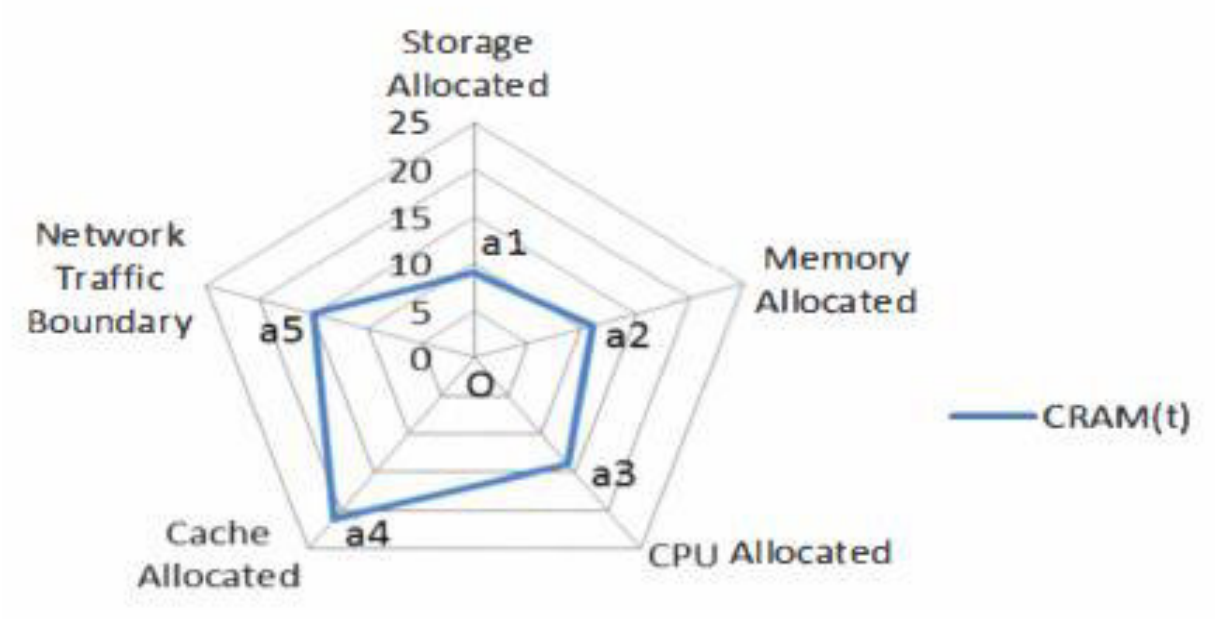

Figura 8. Gráfico em radar da métrica CRAM [18]

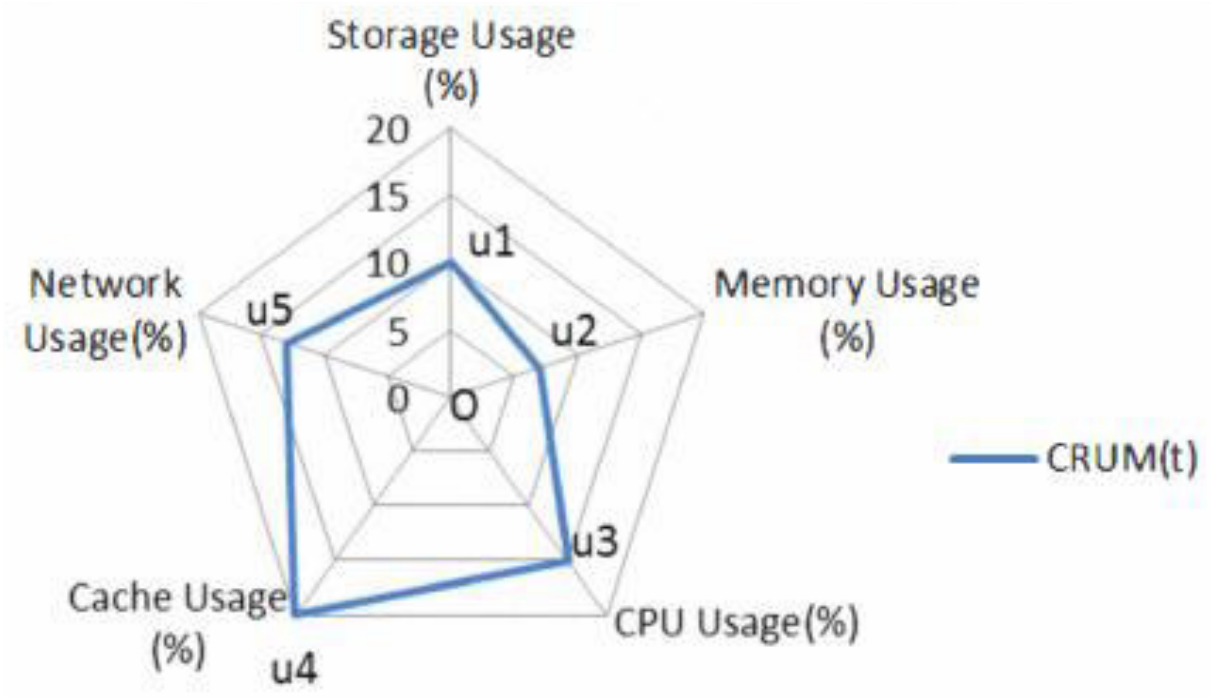

Figura 9. Gráfico em radar da métrica CRUM [18]

Testing as a Service (TaaS) em Nuvens [14]

Esta forma de oferecer testes como um serviço para nuvens SaaS, outros tipos de nuvens e aplicações baseadas em nuvens, oferece um modelo de serviço sob demanda para teste de software, no qual as solicitações de testes são processadas em um ambiente de teste escalável, localizado em nuvem e baseado

em acordos em níveis de serviço (*Service Level Agreement* – SLA) pré-definidos. Além disso, TaaS oferece um novo modelo de negócio para testes de software baseado em faturamento. Através disto, os usuários finais conseguem obter diferentes serviços de teste usando o método de pagamento por teste (*pay-as-you-test*), alcançado o compartilhamento de gastos e a redução de custos.

O TaaS é importante por permitir :

- Compartilhamento dos recursos e redução dos custos no processo de teste;
- Ambientes de teste escaláveis com virtualização [17];
- Serviço de teste automatizado sob demanda [58];
- Pagamento por teste a qualquer momento;
- Serviços de teste baseados em multi-clientelismo;
- Certificação de qualidade por terceiros.

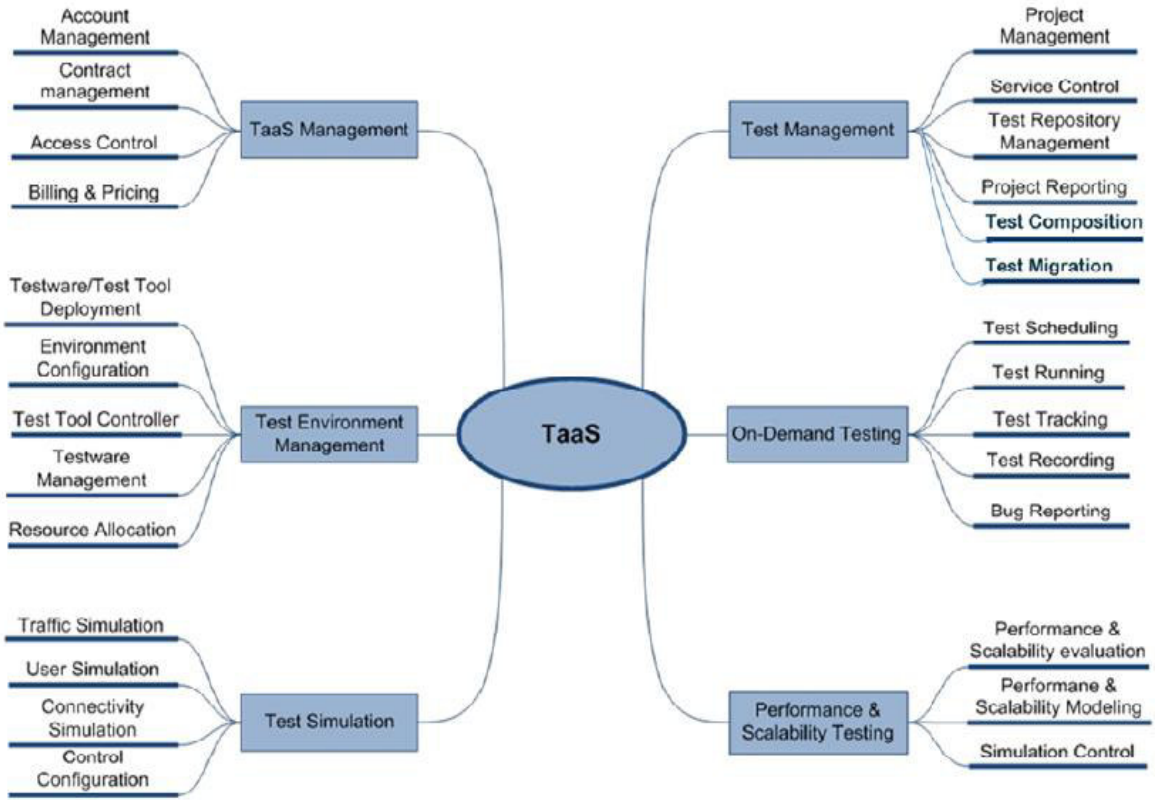

Figura 10. Classificação dos diferentes serviços TaaS [14]

O TaaS em uma infraestrutura de nuvem refere-se à entrega de diversas execuções de teste automatizadas sob demanda, serviços de controle e serviços de gerenciamento de projeto aos clientes baseados em SLAs bem definidos por

padrões QoS. A Figura 10 acima exibe uma classificação detalhada dos serviços TaaS.

Outras características podem ser identificadas para este serviços de teste [14]:

- Recursos computacionais e recursos de teste são agrupados organizadamente;
- O teste em aplicações SaaS é feito também com seus banco de dados;
- Teste multi-cliente de aplicações SaaS;
- Serviços inteligentes de automação de teste sob-demanda;
- Validação contínua e re-teste;
- Integração e composição das soluções de teste;

Os autores identificaram desafios técnicos em contruir um ambiente TaaS [14]. O primeiro desafio é como realizar a validação contínua do SaaS e o teste de regressão, uma vez que a característica do multi-clientelismo impõe esta dificuldade. O segundo desafio é a integração e composição da solução do teste. Os autores apontam que este desafio existe devido a [16]:

- Carência de padrões de qualidade bem definidos para os protocolos de conectividade de SaaS e interfaces de interação; e
- Carência de integração custo-efetiva, além de soluções de serviço de composição e *frameworks* para facilitar a integração de ferramentas de teste, montagem, e composição em nuvens, e até mesmo sobre as nuvens.

O terceiro desafio é o trancafiamento tecnológico que os usuários podem se submeter durante o uso do TaaS.

Teste de SaaS em Nuvens – Questões, Desafios, e Necessidades [15]

Os autores deste artigo indicam que, como a Computação em Nuvem está avançando rapidamente no mercado internacional, da mesma forma está avançado o desenvolvimento e teste de nuvens SaaS.

A Tabela 2 exibe algumas tarefas e seus objetivos no teste em SaaS.

Além destas tarefas e seus objetivos, o teste em SaaS pode ser realizado a partir de três ambientes de teste:

- Plataforma de teste e desenvolvimento de SaaS: suporta a construção, implantação, e validação com as ferramentas e facilidades fornecidas. Um exemplo deste ambiente é o Salesforce [58].
- Ambiente de teste baseado em nuvem pública/privada: permite aos engenheiros implantar e testar aplicações SaaS usando seus ambientes de teste configurados e recursos computacionais.
- Infraestrutura de teste TaaS baseada em nuvem: fornece vários serviços de teste sob demanda em uma infraestrutura de nuvem controlada por terceiros.

Um processo de teste em SaaS constitui-se em um conjunto de seis passos:

1) Teste unitário de componentes
2) Teste de integração de componentes
3) Teste de recursos funcionais baseado no cliente
4) Teste de sistema baseado no cliente
5) Teste dos recursos SaaS baseado no cliente
6) Teste contínuo baseado no cliente

Em cada etapa do processo de teste, existem sub-testes realizados para validar completamente o funcionamento do SaaS. A Figura 11 exibe o processo de teste em SaaS, os seis passos e os respectivos testes realizados em cada passo.

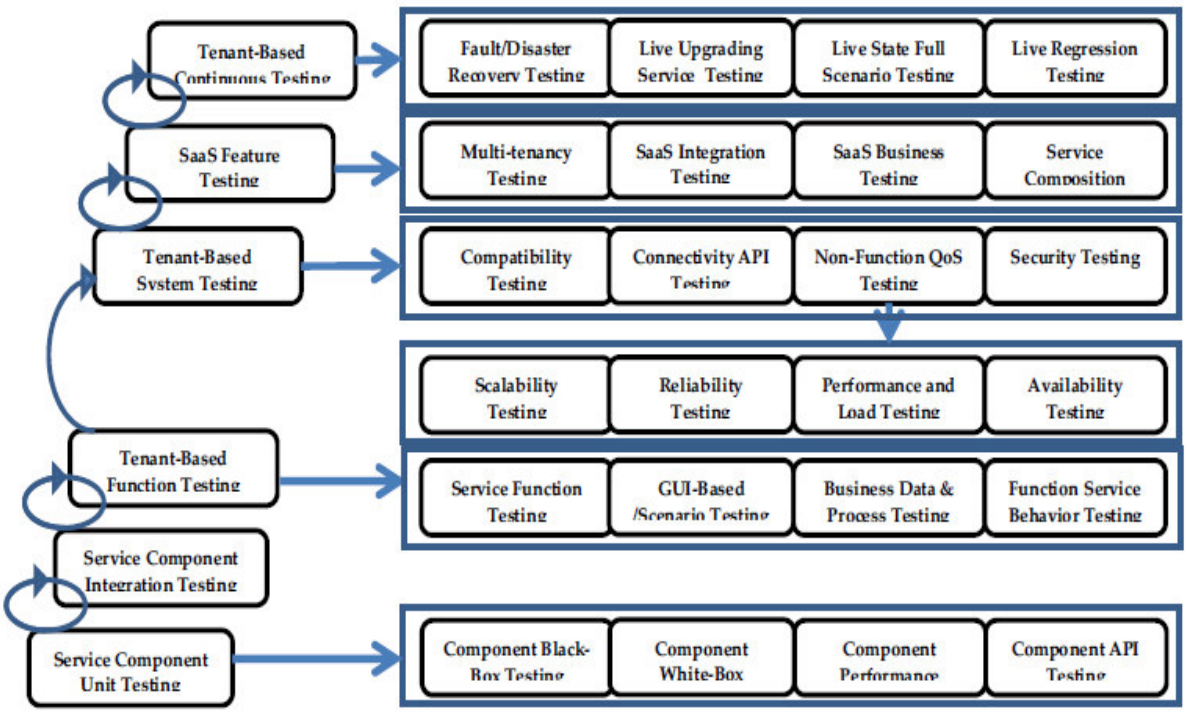

Figura 11. Um processo de teste em SaaS e as respectivas tarefas de teste [15]

Tarefas do teste em SaaS	Objetivo
Teste componente	Realizar teste em caixa preta e em caixa branca para componentes.
Teste funcional	Testar funções de serviço baseado em cliente, comportamentos, fluxos de trabalho e transações.
Teste de integração	Realizar integração entre sistemas SaaS e outros sistemas. Verificar a integração de serviços multi-clientes.
Teste *multi-tenancy*	Testar serviços e funções baseados em multi-clientelismo
Teste de conectividade	Avaliar a qualidade dos APIs de conectividade do SaaS
Teste de *upgrade* contínuo	Validar as atualizações contínuas do SaaS sempre que novos clientes são adicionados
Teste de segurança	Avaliar a segurança do SaaS baseado em clientes múltiplos ou clientes únicos em bancos de dados, fluxos de trabalho, transações, e funções. Avaliar a privacidade do usuário e segurança do sistema do SaaS.

Tabela 2. Tarefas e objetivos do teste em SaaS (alterada de [15])

3.2 Análise dos trabalhos relacionados

A partir do levantamento dos trabalhos relacionados a testes em SaaS, pode-se realizar as seguintes observações:

1) A aplicação de testes em SaaS tem como intenção verificar e validar o funcionamento deste tipo de nuvem. As tarefas de teste e as formas com que os testes podem ser realizados foram apresentadas para explicar melhor o escopo dos testes em SaaS;

2) O tópico de testes em SaaS é um assunto que tem atraído atenção da comunidade acadêmica, pois estes testes consideram as características

particulares de uma aplicação SaaS. Da mesma forma, vários desafios são levantados devidos estas mesmas características;

3) Nestes trabalhos relacionados [67] [18], verifica-se a utilização de métricas e gráficos com o objetivo de inferir quantitativamente o resutado dos testes propostos para SaaS. Destaca-se em especial os testes de escalabilidade e de desempenho que são propostos por [18] e [67], através do uso de métricas e gráficos. Além destes testes, em [15] observa-se a realização de outros tipos de teste, como o teste baseado em componentes, testes funcionais, testes de segurança, teste *multi-tenancy*, entre outros tipos, os quais são orientados para o bom funcionamento das nuvens SaaS;

4) Uma solução de teste em SaaS pode ser estabelecida a partir de um *framework* de realização de teste, passando por uma nuvem pública ou privada até um ambiente completo (no caso, o TaaS [14]) de realização e configuração de testes em SaaS;

5) Dentre os trabalhos relacionados, não verifica-se a existência de abordagens de Engenharia Dirigida por Modelo (MDE) para geração de casos de teste exclusivamente para aplicações SaaS e que contemplem outros tipos de teste além dos identificados;

A partir desta análise, conclui-se que é necessário estabelecer uma solução baseada em Engenharia Dirigida por Modelos que permita a geração de casos de teste exclusivamente para aplicações SaaS, pois as atividades de teste são reduzidas em seu custo, uma vez que o paradigma da MDE permite esta redução de custos por meio da geração de modelos, assim como reduz a complexidade da realização dessas tarefas.

Além da necessidade dessa solução, verifica-se que os testes de confiabilidade, disponibilidade, usabilidade e aceitação não são considerados nos trabalhos relacionados. Portanto, é necessário estabelecer estes tipos de teste, pois estes testes estão mais orientados ao usuário, e dessa forma a sua qualidade é melhor verificada.

A Tabela 3 exibe o quadro comparativo dos trabalhos realizados, levando em consideração o nível do teste, o(s) teste(s) realizado(s), a utilização ou não de métricas e a geração ou não de casos de teste.

3.3 Síntese

Neste capítulo, o Estado da Arte foi apresentado, com o objetivo de mostrar o atual *status* acerca de testes em SaaS.

Este tópico, que tem tido cada vez mais atenção da comunidade acadêmica internacional, conta com diversas abordagens de teste, com uso de métricas e gráficos, e que pode ser realizada através de vários ambientes de teste (nuvens públicas ou privadas, ambientes TaaS ou *frameworks* de execução de teste). O processo de testar uma nuvem SaaS conta com a realização de testes como teste de integridade, teste de componente, teste *multi-tenancy*, isto é, baseado nos múltiplos clientes que utilizam a mesma aplicação SaaS, teste de desempenho e de escalabilidade.

Finalizando este capítulo, uma análise dos trabalhos realizados no campo de teste em SaaS foi apresentada com objetivo de retratar os principais pontos presentes nesta área.

Itens avaliados	**Testando a Escalabilidade de Aplicações SaaS [67]**	**Avaliação de Desempenho e de Escalabilidade de SaaS em Nuvens [18]**	***Testing as a Service* (TaaS) em Nuvens [14]**	**Teste de SaaS em Nuvens – Questões, Desafios, e Necessidades [15]**
Nível de teste	SaaS	SaaS	SaaS, outros tipos de nuvens e aplicações baseadas em nuvens	SaaS
Testes realizados	Teste de desempenho e de escalabilidade	Teste de desempenho e de escalabilidade	Teste de desempenho e de escalabilidade	Testes funcionais, de integração, de componente, de segurança, teste *multi-tenancy*¸ de desempenho, etc.
Utilização de métricas	Sim	Sim	Não	Não
Geração de casos de teste	Não	Não	Não	Não

Tabela 3. Análise dos trabalhos relacionados

4 Uma Abordagem para Testes em SaaS de Código Aberto

Este capítulo tem como objetivo apresentar a solução proposta por este projeto, isto é, um *framework* para gerar casos de teste para nuvens SaaS de código aberto (normalmente chamada de *open-SaaS*).

Inicialmente, a estrutura do *framework* e sua forma de funcionamento são apresentados. Em seguida, as métricas adotadas pelo *framework* para medir SaaS de código aberto são exibidas.

Em seguida, os metamodelos e modelos que compõem o *framework* são explicados.

Finalizando o capítulo, a metodologia para geração dos casos de testes é apresentada.

4.1 *Framework* para Geração de Casos de Teste para SaaS de Código Aberto

A Figura 12 apresenta o *framework* proposto para geração de casos de teste para nuvens SaaS de código aberto. No centro da Figura 12, há uma linha tracejada separando o *framework* em duas partes. O lado esquerdo mostra a geração dos casos de teste para SaaS de código aberto e o lado direito representa a geração de código fonte para SaaS de código aberto.

O modelo de teste é conforme a um Metamodelo para Teste Independente de Plataforma baseado em Métricas (MPIT baseado em Métricas), definido na Seção 4.3.1. Nesta abordagem, o Modelo Independente de Plataforma (PIM) é conforme ao Metamodelo UML [55]. O modelo de teste refere-se ao PIM.

Um teste abstrato (PSM abstrato de teste) é gerado a partir de um modelo de teste graças a uma definição de transformação modelo-a-modelo (*Model 2 Model* – M2M). Um teste abstrato é conforme a um Metamodelo para Teste Abstrato de Nuvem para SaaS (MACT4SaaS) apresentado na Seção 4.3.2.

Um teste concreto (PSM concreto de teste) é gerado a partir de um teste abstrato graças a uma definição de transformação M2M. Um teste concreto refere-se a um PSM. Um teste concreto é conforme ao metamodelo do PHP e usa um modelo do API do PHPUnit para teste em PHP. A escolha do API do PHPUnit para teste em PHP foi realizada porque o *Wordpress* [70] é baseado em PHP.

Casos de teste (código fonte em PHPUnit) para os testes de disponibilidade, confiabilidade, usabilidade e aceitação são gerados a partir de um PSM de teste concreto graças a uma definição de transformação M2M. Casos de teste referem-se ao código fonte em PHP do SaaS em código aberto. Tanto o código de teste quanto o código fonte são conformes à gramática EBNF do PHP.

As métricas de disponibilidade, confiabilidade, usabilidade e aceitação foram definidas como atributos dos casos de teste. Elas serão explicadas na seção a seguir. A razão para usar métricas neste contexto é para avaliar o SaaS em código aberto [38] nos critérios de disponibilidade, confiabilidade, usabilidade e aceitação. Os testes respectivos a estas métricas foram escolhidos devido à carência de atenção por parte das equipes de teste e de manutenção das nuvens SaaS com relação a estes critérios.

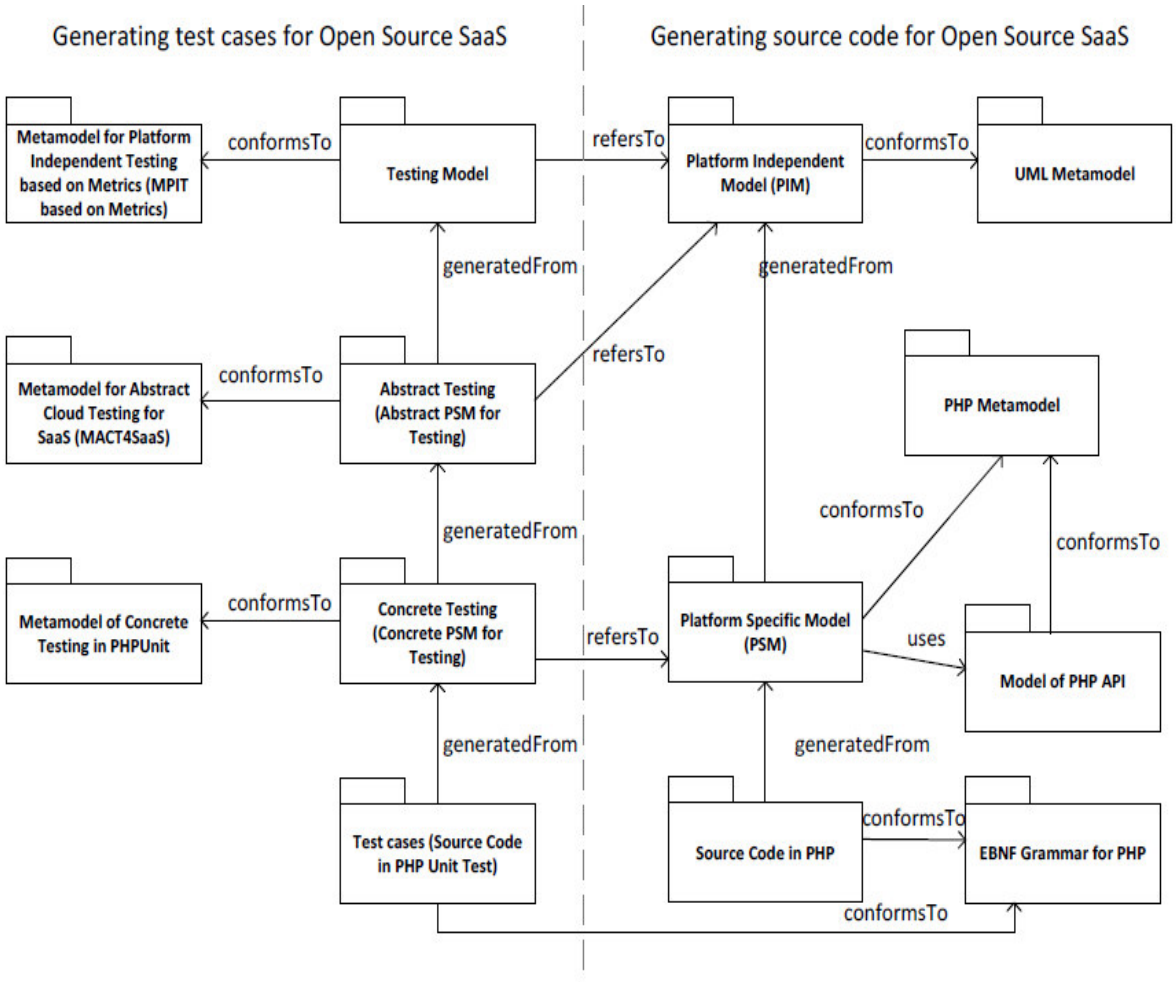

Figura 12. *Framework* para geração de casos de teste para SaaS de código aberto (FCTSaaS)

4.2 Métricas para Avaliação do SaaS em Código Aberto

Novas métricas para os testes de disponibilidade, confiabilidade, usabilidade e aceitação para SaaS de código aberto foram propostas. A métrica de disponibilidade, redefinida a partir da métrica de disponibilidade presente em [37], é proposta na próxima seção. A métrica de confiabilidade é baseada no trabalho apresentado em [38], sendo redefinida com o propósito de medir com maior rapidez e facilidade este critério. A métrica de usabilidade é criada usando como base a definição de usabilidade no ISO/DIS 9241-11 [27], e a métrica de aceitação é definida de acordo com a taxa de cenário de uso, o nível de segurança e as métricas de disponibilidade, confiabilidade e usabilidade.

4.2.1 Métrica de disponibilidade

A métrica de disponibilidade (*Robustness of Service* – ROS) [38] representa a razão entre o tempo disponível para invocar o *SaaS* e o tempo total para operá-lo. Nesta dissertação, a métrica de disponibilidade foi adaptada a partir da fórmula presente em [38], sendo medida a partir da expressão como segue:

$$ROS = \frac{TAD-(SFT+STM+TSI)}{TAD} \text{ (adaptada de [38])} \qquad (1)$$

Onde:

- ROS é a métrica de disponibilidade;
- TAD (*Time After Deployment*) é o tempo após a implantação do *SaaS*;
- SFT (*Service Failure* Time) é o tempo de falha do serviço;
- STM (*Stopped Time for Maintenance*) é o tempo de parada para manutenção programada do *SaaS*;
- TSI (*Time for Security Issues*) é o tempo em que o serviço teve as funcionalidades afetadas por problemas de segurança incluindo o tempo sem estar parado.

A introdução do STM ocorreu devido a crença de que um serviço deve estar executando todo o tempo após a sua implantação e o tempo de manutenção deve impactar diretamente na disponibilidade. TSI foi introduzido porque os problemas de

segurança devem penalizar a disponibilidade a partir do momento que eles acontecem e não apenas quando geram uma falha. Por exemplo, um incidente onde um usuário não-autorizado acessa o serviço não pode gerar uma falha no serviço (i.e. o serviço não é parado), mas compromete seu comportamento correto de estar seguro. A faixa da métrica de disponibilidade varia de 0 a 1. O valor 1 significa que uma aplicação open-SaaS é totalmente disponível, e o valor 0 significa que uma aplicação open-SaaS está totalmente indisponível. De acordo com Shari L. Pfleeger e Joanne M. Atlee, disponibilidade é "a probabilidade de um sistema estar operando com sucesso, de acordo com suas especificações, em um determinado momento" [59]. Então, é razoável considerar que o tempo reservado para manutenção (programada ou não) e o tempo de ocorrência dos problemas de segurança (parando o sistema ou não) tem impacto na disponibilidade. Ao contrário de Pfleeger e Atlee[1] [59], é importante o uso do tempo real (i.e., o tempo medido por um relógio) para calcular a disponibilidade, porque um serviço pode ser invocado em curtos intervalos de tempo ou em paralelo, portanto uma aproximação para tempo contínuo é razoável.

4.2.2 Métrica de confiabilidade

Uma métrica de confiabilidade mede a estabilidade de uma aplicação open-SaaS contra defeitos e falhas e a quão boas são as suas respostas ao usuário. Os conceitos de defeito e falha são importantes na definição da métrica de confiabilidade. Esta dissertação utiliza os conceitos presentes em [63], no qual defeito é "um estado incorreto do sistema, isto é, um estado do sistema que não é esperada pelos projetistas do sistema" [63], e falha é "um evento que ocorre em algum ponto no tempo quando o sistema não entrega um serviço conforme esperado por seus usuários" [63]. Um erro humano pode levar a um defeito, que por sua vez pode levar a uma falha.

Esta métrica é definida com três submétricas: a cobertura de tolerância contra defeitos (*Coverage of Fault Tolerance* – CFT), a cobertura de recuperação contra falhas (*Coverage of Failure Recovery* – CFR) e a precisão do serviço (*Service Accuracy* – SA) [38]. A submétrica CFT mede a tolerância do SaaS a defeitos. É

[1] Para Shari L. Pfleeger e Joanne M. Atlee [59], a disponibilidade é medida em momentos específicos de tempo e não durante o tempo de execução.

calculada como uma relação entre o número de defeitos que não se tornaram falhas e o númer total de defeitos:

$$CFT = \frac{\text{número de defeitos que não se tornam falhas}}{\text{número total de defeitos}} \text{ [38]} \quad (2)$$

A faixa de resultados está entre 0 e 1, e o maior valor indica que uma aplicação open-SaaS tem grande durabilidade contra vários defeitos.

A submétrica CFR mede a capacidade de uma aplicação open-SaaS recuperar-se de falhas em um período de tempo. Ele é medido como uma relação entre o número de falhas reparadas de uma aplicação open-SaaS e o número total de falhas:

$$CFR = \frac{\text{número de falhas reparadas}}{\text{número total de falhas}} \text{ [38]} \quad (3)$$

A faixa de resultados está entre 0 e 1, e o maior valor indica que a aplicação open-SaaS tem alta cobertura contra falhas.

A submétrica SA mede a relação entre o número de respostas corretas para os requisitos do consumidor e o número total de requisitos em um período específico de tempo [38]:

$$SA = \frac{\text{número de respostas corretas}}{\text{número total de solicitações}} \text{ [38]} \quad (4)$$

O resultado varia entre 0 e 1, e o maior valor indica que uma aplicação open-SaaS tem alta precisão. Essas três submétricas estão presentes nas fórmula da métrica de confiabilidade.

De acordo com Pfleeger e Atlee [59], confiabilidade é “a probabilidade de um sistema operar sem apresentar falhas, de acordo com condições específicas, em um determinado intervalor de tempo.” Um problema de segurança pode causar alterações nos resultados de um sistema, então a presença de problemas de segurança pode impactar na confiabilidade. Ao contrário de Pfleeger e Atlee [59], o uso do tempo real (i.e. o tempo medido pelo relógio) é proposto para calcular a

confiabilidade, porque considera-se que um serviço em uma nuvem está executando todo o tempo, executando uma funcionalidade ou esperando para executá-lo.

O Índice de Ausência de Problemas de Segurança (*Index of Absence of Security Issues* – IAS) é proposto a fim de impactar os problemas de segurança na métrica de confiabilidade e é definido como segue:

$$IAS = \frac{TAD\text{-}TSI}{TAD} \quad (5)$$

Onde:

- TAD é o tempo após a implantação do *SaaS*;
- TSI é o tempo enquanto o service teve as funcionalidades afetadas por problemas de segurança, incluindo o tempo sem estar parado.

A fórmula original para a métrica de confiabilidade proposta em [38] foi adaptada e é definida como segue:

$$RM = (W_{CFT} \times CFT + W_{CFR} \times CFR + W_{SA} \times SA + W_{IAS} \times IAS) \times MF \quad (6)$$

W_{CFT}, W_{CFR}, W_{SA} e W_{IAS} são pesos das submétricsa CFT, CFR, SA e IAS, respectivamente, onde $W_{CFT} + W_{CFR} + W_{SA} + W_{IAS} = 1$. Para definir esses pesos, foi usado uma combinação convexa [61], isto é, cada peso é uma divisão entre a respectiva submétrica e a soma de todas submétricas relacionadas. Por exemplo, na definição do peso W_{CFT}, a divisão é definida como segue:

$$W_{CFT} = \frac{CFT}{CFT+CFR+SA+IAS} \quad (7)$$

Similarmente, os pesos W_{CFR}, W_{SA} e W_{IAS} são definidas como segue:

$$W_{CFR} = \frac{CFR}{CFT+CFR+SA+IAS} \quad (8) \qquad W_{SA} = \frac{SA}{CFT+CFR+SA+IAS} \quad (9)$$

$$W_{IAS} = \frac{IAS}{CFT+CFR+SA+IAS} \quad (10)$$

MF (*Maturity Factor*) é fator de maturidade e é definido como segue:

$$MF = 1 - 0.5 \times e^{-\frac{5 \times TAD}{TBM}} \quad (11)$$

Onde:

- TAD é o tempo após a implantação do *SaaS*;
- TBM é o tempo para o serviço *SaaS* estar maduro.

Substituindo essas equações na Equação (6), a nova fórmula para a métrica de confiabilidade foi gerada como segue:

$$RM = \left(\frac{CFT^2 + CFR^2 + SA^2 + IAS^2}{CFT + CFR + SA + IAS}\right) \times \left(1 - 0.5 \times e^{-\frac{5 \times TAD}{TBM}}\right) \quad (12)$$

O resultado varia entre 0 e 1. O valor 1 significa que uma aplicação open-SaaS é totalmente confiável.

Sem o fator de maturidade, um serviço recentemente implantado teria confiabilidade igual ou próxima a 1, pois os defeitos, falhas e problemas de segurança não seriam conhecidos. Enquanto um serviço implantado depois de meses teria confiabiliade menor que 1 devido os defeitos, falhas e problemas de segurança serem conhecidos. MF permite definir um fator que regula a confiabilidade de acordo com o tempo de maturidade do serviço. Assim, um serviço que está recentemente implantado e os defeitos, falhas e problemas de segurança ainda são desconhecidos terão uma confiabilidade igual a 0,5. Quando este serviço está alcançando o tempo de maturidade, então o MF está mais próximo de 1, e a confiabilidade passa a ser determinada devido aos defeitos, falhas e problemas de segurança. TBM é configurado pelo usuário e este valor reflete por quanto tempo o usuário espera que um *SaaS* estará estável e capaz de ser lançado. Uma vez que o *SaaS* seja lançado, RM continua a ser calculada, mas o MF continuará próximo a 1.

4.2.3 Métrica de usabilidade

A métrica de usabilidade (*Usability Metric* – UM), baseada no ISO/DIS 9241-11 [27] e nas pesquisas de Kerzazi et al. em [32] e Mirnig et al. em [46], mede “até que ponto um sistema, produto ou serviço pode ser usado por usuários especificados para alcançar objetivos especificados com efetividade, eficiência e

satisfação em um contexto específico de uso" [27]. Ela é medida em relação a quatro critérios definidos em [32] e [27]: efetividade, eficiência, satisfação do usuário e contexto de uso. Efetividade está relacionada à "precisão e perfeição nos quais os usuários especificados podem alcançar objetivos especificados em ambientes particulares" [32]. Eficiência está relacionada aos "recursos gastos em relação à precisão e perfeição dos objetivos alcançados" [32]. Satisfação do usuário está relacionada "ao conforto e aceitabilidade do sistema em funcionamento aos seus usuários e outras pessoas afetadas pelo seu uso" [32]. O contexto de uso está relacionado aos "usuários, tarefas, equipamentos (*hardware*, software e materaisi), e os ambientes físico e social no qual um produto é usado" [32]. Neste caso, um sistema em funcionamento é a aplicação *open-SaaS* utilizado por um usuário.

A fim de obter a métrica de usabilidade, foram reusadas os critérios da efetividade, eficiência, satisfação do usuário e contexto de uso propostos por Kerzazi et al. em [32] e no ISO/DIS 9241-11 [27]. O perfil de usuário foi inserido como outro critério de obtenção da métrica de usabilidade. Este último critério permite medir o nível de confiança que se deve ter com a resposta de um usuário em um questionário oferecido a eles após o uso da aplicação *open-SaaS*. A Tabela 4 apresenta o questionário apresentado aos usuários e que permite calcular a métrica de usabilidade. Cada questão do formulário possui apenas uma resposta entre cinco respostas possíveis e cada uma tem um valor adequado conforme segue: A = 1.0; B = 0.8; C = 0.6; D = 0.4; E = 0.0.

A métrica de usabilidade é definida como segue:

$$UM = W_{ES} \times ES + W_{EY} \times EY + W_{US} \times US + W_{CU} \times CU \quad (13)$$

Onde:

ES é efetividade; EY é eficiência; US é satisfação do usuário; CU é contexto de uso; W_{ES}, W_{EY}, W_{US} e W_{CU} são pesos onde $W_{ES} + W_{EY} + W_{US} + W_{CU} = 1$.

ES, EY, US e CU são definidos como segue:

$$ES = \frac{\sum_{i=1}^{n}\left[up_i \times \left(\frac{\sum_{j=1}^{m} x_{i,j}}{m}\right)\right]}{n} \quad (14)$$

$$EY = \frac{\sum_{i=1}^{n}\left[up_i \times \left(\frac{\sum_{k=1}^{p} y_{i,k}}{p}\right)\right]}{n} \quad (15) \qquad US = \frac{\sum_{i=1}^{n}\left[up_i \times \left(\frac{\sum_{l=1}^{q} z_{i,l}}{q}\right)\right]}{n} \quad (16)$$

$$CU = \frac{\sum_{i=1}^{n}\left[up_i \times \left(\frac{\sum_{f=1}^{s} w_{i,f}}{s}\right)\right]}{n} \quad (17)$$

Onde:

- up_i é a métrica do perfil de usuário para o usuário *i*;
- $x_{i,j}$ é o valor correspondente à resposta do usuário *i* sobre a questão *j* relacionada a efetividade;
- $y_{i,k}$ é o valor correspondente à resposta do usuário *i* sobre a questão *k* relacionada a eficiência;
- $z_{i,l}$ é o valor correspondente à resposta do usuário *i* sobre a questão *l* relacionada à satisfação do usuário;
- $w_{i,f}$ é o valor correspondente à resposta do usuário *i* sobre a questão *f* relacionada ao contexto de uso;
- *n* é o número de usuários que responderam ao questionário. As variáveis *m, p,q* e *s* é o número total de questões sobre efetividade, eficiência, satisfação do usuário e contexto de uso, respectivamente.

up_i é definido como segue:

$$up_i = \frac{\sum_{a=1}^{r} u_a}{r} \quad (18)$$

Onde:

- u_a é o valor correspondente à resposta do usuário *i* sobre a questão *a* relacionada ao perfil do usuário;
- r é o número total de questões relacionadas ao perfil de usuário.

Aplicando combinação convexa, a fórmula da métrica de usabilidade é definida como segue:

$$UM = \frac{ES^2 + EY^2 + US^2 + CU^2}{ES+EY+US+CU} \quad (19)$$

PERFIL DE USUÁRIO	Qual é o seu conhecimento sobre a lógica de negócio da empresa?	A. Especialista / B. Bom / C. Regular / D. Fraco / E. Nenhum
	Como é o seu nível de conhecimento sobre a aplicação *SaaS*?	A. Especialista / B. Bom / C. Regular / D. Fraco / E. Nenhum
	Qual é a frequência que você usa a aplicação *SaaS*?	A. Diariamente B. Semanalmente C. Mensalmente D. Bimensalmente E. Nunca
EFETIVIDADE	A aplicação *SaaS* permitiu a você realizar suas tarefas?	A. Todo tempo / B. 80% do tempo / C. 50% do tempo / D. 20% do tempo / E. Nunca
	Você reconheceu que necessita de outros *softwares* para completar suas tarefas?	A. Nunca / B. 20% do tempo C. 50% do tempo / D. 80% do tempo / E. Nunca
	As funcionalidades fornecidas pela aplicação *SaaS* são suficientes para fazer as tarefas desejadas?	A. Todo o tempo / B. 80% do tempo / C. 50% do tempo / D. 20% do tempo / E. Nunca
EFICIÊNCIA	As tarefas realizadas por você usando a aplicação *SaaS* foram executadas rapidamente e economizaram seu tempo?	A. Todas as vezes / B. 80% das vezes / C. 50% das vezes / D. 20% das vezes / E. Nunca
	O tempo de resposta da aplicação *SaaS* foi como esperado?	A. Todas as vezes / B. 80% das vezes / C. 50% das vezes / D. 20% das vezes / E. Nunca
	A aplicação *SaaS* foi fácil de usar?	A. Todas as vezes / B. 80% das vezes / C. 50% das vezes / D. 20% das vezes / E. Nunca
SATISFAÇÃO DO USUÁRIO	Você está satisfeito com a aplicação *SaaS* que você usou?	A. Todas as vezes / B. 80% das vezes / C. 50% das vezes / D. 20% das vezes / E. Nunca
	A interface usuário foi intuitiva para uso?	A. Todas as vezes / B. 80% das vezes / C. 50% das vezes / D. 20% das vezes / E. Nunca
	Você realizou suas tarefas na aplicação *SaaS* com poucos passos (cliques, formulários, tabelas, etc.)?	A. Todas as vezes / B. 80% das vezes / C. 50% das vezes / D. 20% das vezes / E. Nunca

Tabela 4. Questionário elaborado para gerar a métrica de usabilidade (baseado em [32], [46] e [27])

O resultado da métrica de usabilidade varia entre 0 e 1. O valor 1 significa que uma aplicação open-SaaS é totalmente utilizável.

Por envolver a participação de seres humanos na utilização da aplicação *open-SaaS* do *Wordpress* e na resolução do questionário para o cálculo da métrica de usabilidade, esta pesquisa foi submetida para o Comitê de Ética em Pesquisa da Universidade Federal do Maranhão (CEP/UFMA). O comprovante de envio e o respectivo número de protocolo são exibidos no Anexo 8.4.

4.2.4 Métrica de aceitação

Teste de aceitação é um "teste conduzido para determinar se um sistema satisfaz seus critérios de aceitação e para permitir ao consumidor determinar se aceita o sistema" [66]. O teste de aceitação é conduzido pelo consumidor e planos de teste (ou cenários de teste) são fornecidos a fim de determinar se os requisitos implementados no produto (sistema de *software*) satisfaz as demandas ou expectativas do consumidor.

Uma métrica de aceitação é proposta para medir o teste de aceitação (*Acceptance Test* – AC). Os valores que a métrica possui e as suas condições estão presentes na Tabela 5:

Métrica	Valor	Condição
AC	1	se $\forall f \in F, \exists t_{ac}$ / *status* = *Success*
	0	se $\exists f \in F, \exists t_{ac}$ / *status* = *Failed* e *classification* = *HighRelevance*
	$(N_{HR} + N_{LR}) / S_{ac}$	se $\exists t_{ac}$ / *status* = *Failed* e *classification* ≠ *HighRelevance*

Tabela 5. Métrica AT e seus respectivos valores e condições

Onde:

- *f* é a funcionalidade de um sistema de *software* a ser testado;
- *F* é um conjunto que contem todas as funcionalidades de um sistema de *software*;
- t_{ac} é um teste de aceitação relacionado à funcionalidade *f*;
- *status* é o resultado do teste de aceitação e *status* ∈ {*Success, Failed*}.

Quando um teste de aceitação passa com sucesso, então o *status* é *Success*, do contrário o *status* é *Failed*.

- *classification* define a relevância da funcionalidade a ser testada, sendo que *classification* ∈ {*HighRelevance, LowRelevance*}. Uma funcionalidade classificada como *HighRelevance* deve ter teste de aceitação com *status* = *Success*, do contrário o sistema de *software* será parcialmente comprometido, mas sem consequências perigosas. Por exemplo, um *internet banking* tem as funcionalidades de transferir dinheiro e enviar mensagem SMS para cada transação e para um celular específico. A funcionlidade de transferir dinheiro é essencial e deve ter teste de aceitação com *status* = *Success*, mas se o teste de aceitação tiver *status* = *Failed*, então o sistema de *internet banking* pode continuar executando. Em outras palavras, um problema na funcionalidade de enviar mensagem SMS não parará o sistema de *internet banking*, porque a funcionalidade tem baixa relevância e não é essencial.
- N_{HR} é o número de testes de aceitação classificados com *HighRelevance* que passa com *status* = *Success*;
- N_{LR} é o número de testes de aceitação classificados com *LowRelevance* que passam com *status* = *Success*;
- S_{ac} é o número total de testes de aceitação.

4.3 Descrição dos metamodelos

A apresentação dos metamodelos de teste a seguir tem como objetivo mostrar suas classes, atributos e relacionamentos para geração dos casos de teste. Eles são estruturados com o objetivo de facilitar a geração dos casos de teste de disponibilidade, confiabilidade, usabilidade e aceitação para aplicações open-SaaS.

4.3.1 Metamodelo para Teste Independente de Plataforma baseado em Métricas

O Metamodelo para Teste Independente de Plataforma baseado em Métricas (MPIT baseado em Métricas) permite criar modelos de teste contendo métricas de disponibilidade, confiabilidade, usabilidade e aceitação. A Figura 13 apresenta MPIT baseado em Métricas. Ela reutiliza algumas classes presente no Metamodelo de Teste Independente de Plataforma definido em [50] e introduz novas classes e

novos acréscimos realizados em cima do Metamodelo de Teste Independente de Plataforma [50]. As novas classes e os acréscimos são mostrados a seguir:

- Classes *AvailabilityTest*, *ReliabilityTest*, *UsabilityTest* e *AcceptanceTest*, que representam os testes de disponibilidade, confiabilidade, usabilidade e aceitação, respectivamente. Eles herdam da classe *TestType*;
- Classes *AvailabilityMetric, ReliabilityMetric*, *UsabilityMetric* e *AcceptanceMetric*, que representam as métricas de disponibilidade, confiabilidade, usabilidade e aceitação, respectivamente. Seus atributos são o nome, a descrição e todas as variáveis que as formam e que foram mostradas na seção 4.2;
- Acréscimo de uma composição entre as classes *TestCase* e *TestData*;
- Acréscimo de uma associação entre as classes *TestType* e *TestClass*, mantendo a bidirecionalidade entre elas;
- Acréscimo das classes *TestPlan*, *ProcessTest*, *Functionality, DataSet, Input*, *ExpectedOutput* e *ObtainedOutput*, que fazem parte da realização do teste de aceitação na aplicação *open-SaaS*;
- Acréscimo da classe *TestingContainer*, que representa um repositório de testes feitos na aplicação *open-SaaS*;
- Acréscimo das enumerações *KindAcTest*, *FRelevance* e *StatusTest*.

O Metamodelo de Teste Indepente de Plataforma baseado em Métricas possui as seguintes classes que foram herdadas do Metamodelo de Teste Independente de Plataforma [50]:

- *Element*: superclasse do Metamodelo. Possui como atributo um nome e uma descrição;
- *TestType*: classe que identifica os tipos de teste realizados. Possui como atributos um campo para comentários (*comments*), um campo para agendamento (*schedule*) e o relatório de teste (*testReport*);
- *TestSuite*: classe que representa o conjunto de teste que contém casos de teste;
- *TestCase*; classe que representa o caso de teste;
- *Procedures*: classe descreve os procedimentos realizados pelo caso de teste;
- *TestData*: classe que representa os dados que compõem o *TestCase*;

- *TestClass*: classe que representa o nome da classe que implementa o caso de teste;
- *Method*: classe que representa o método que compõe a classe de teste. Possui como atributo uma variável booleana que representa se o método é *static* ou não;
- *Attribute*: classe que representa o atributo e que pode compor a classe de teste;
- *Parameter*: classe que representa o parâmetro e que compõe o *Method*. Possui como atributos o nome, o tipo de parâmetro, isto é, inteiro, *float*, *string*, etc., e uma variável booleana que representa se o parâmetro é de retorno ou não;
- *AttributeType*: classe que representa o tipo do atributo e que compõe a classe *Attribute*. Possui como atributo o nome do tipo do atributo, isto é, inteiro, *float*, *string*, etc.

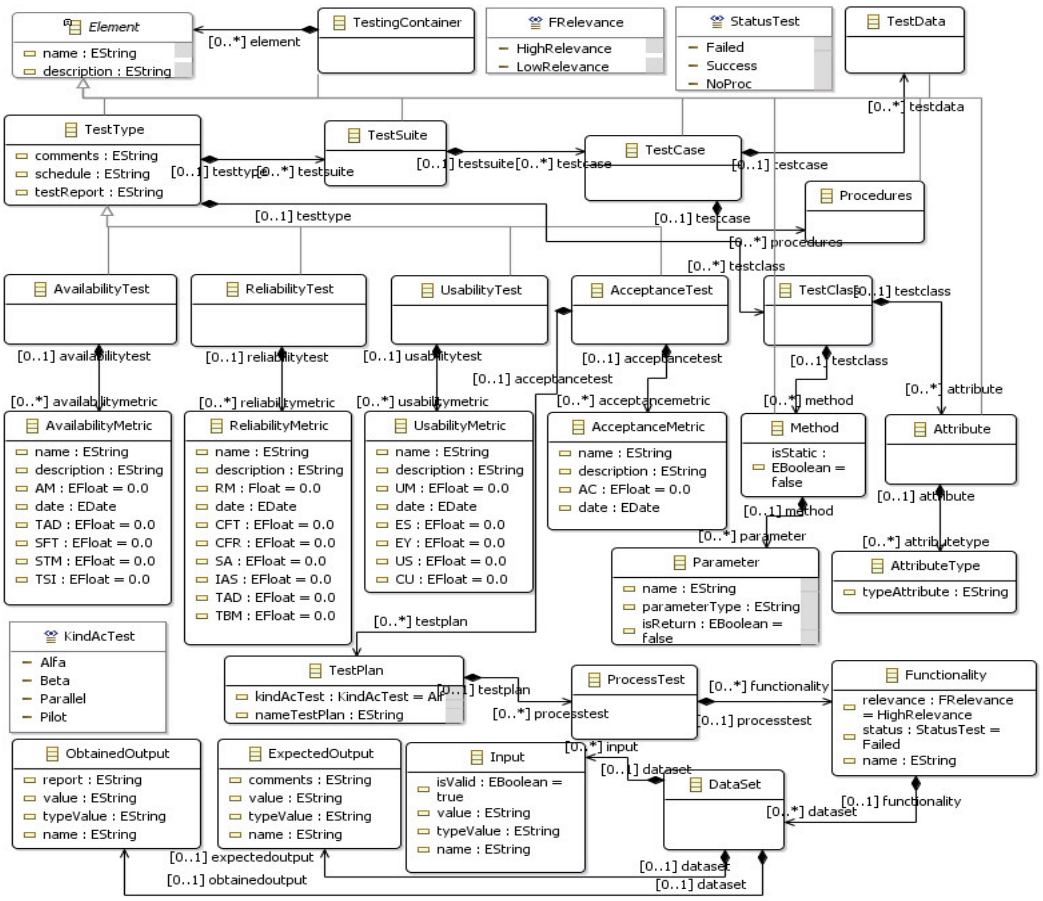

Figura 13. Metamodelo para Teste Independente de Plataforma baseado em Métricas (MPIT baseado em Métricas) (adapta e extende o trabalho apresentado em [50])

4.3.2 Metamodelo para Teste Abstrato de Nuvem para SaaS (MACT4SaaS)

Este metamodelo contem classes que representam alguns elementos relacionados a qualquer nuvem *open-SaaS*, assim como contem todas classes de MPIT baseado em Métricas. A Figura 14 exibe MACT4SaaS.

No lado superior da Figura 14, alguns elementos relacionados a nuvens open-SaaS são representados. Esses elementos são:

- *SaaSProvider*: contem informação sobre o provedor *open-SaaS*;
- *ContentManagementSystem*: contem informação específica relacionada ao sistema de gerenciamento de conteúdo (*Content Management System* – CMS). *Wordpress* é um exemplo de CMS que permite a criação de *blogs*;
- *Persistence*: contem informação e operações relacionadas a gravação de dados de forma persistente para CMS;
- *Action*: contem a descrição de ações realizadas pelo CMS, incluindo entradas e saídas esperadas.
- *Outras classes*: alguns *SaaS* são fornecidos como aplicação que é acessada via navegador *Web* ou chamada a APIs (*Application Programming Interfaces*). *Mail, Message, News, Office* (por exemplo, editor de texto, apresentação, planilhas de trabalho), *Schedule*, *Storage*, *Backup* e *Antivirus* são exemplos de aplicações que são fornecidas como *SaaS*. Neste trabalho, o foco fica em *ContentManagementSystem* no PSM abstrato e no *Wordpress* no PSM concreto.

4.3.3 Metamodelo e Modelos para construir um Teste Concreto

Aplicações *SaaS* podem ser desenvolvidas em muitas plataformas, por exemplo Java, dotNet e PHP. Nesta seção, algunas bases para construir PSM concreto para teste baseado em PHP e PHPUnit [3] foram fornecidas. Para este processo, um metamodelo para PHP, um modelo para a API do PHPUnit, e um padrão para modelo de teste concreto, incluindo teste não-funcional e teste de aceitação, são fornecidos.

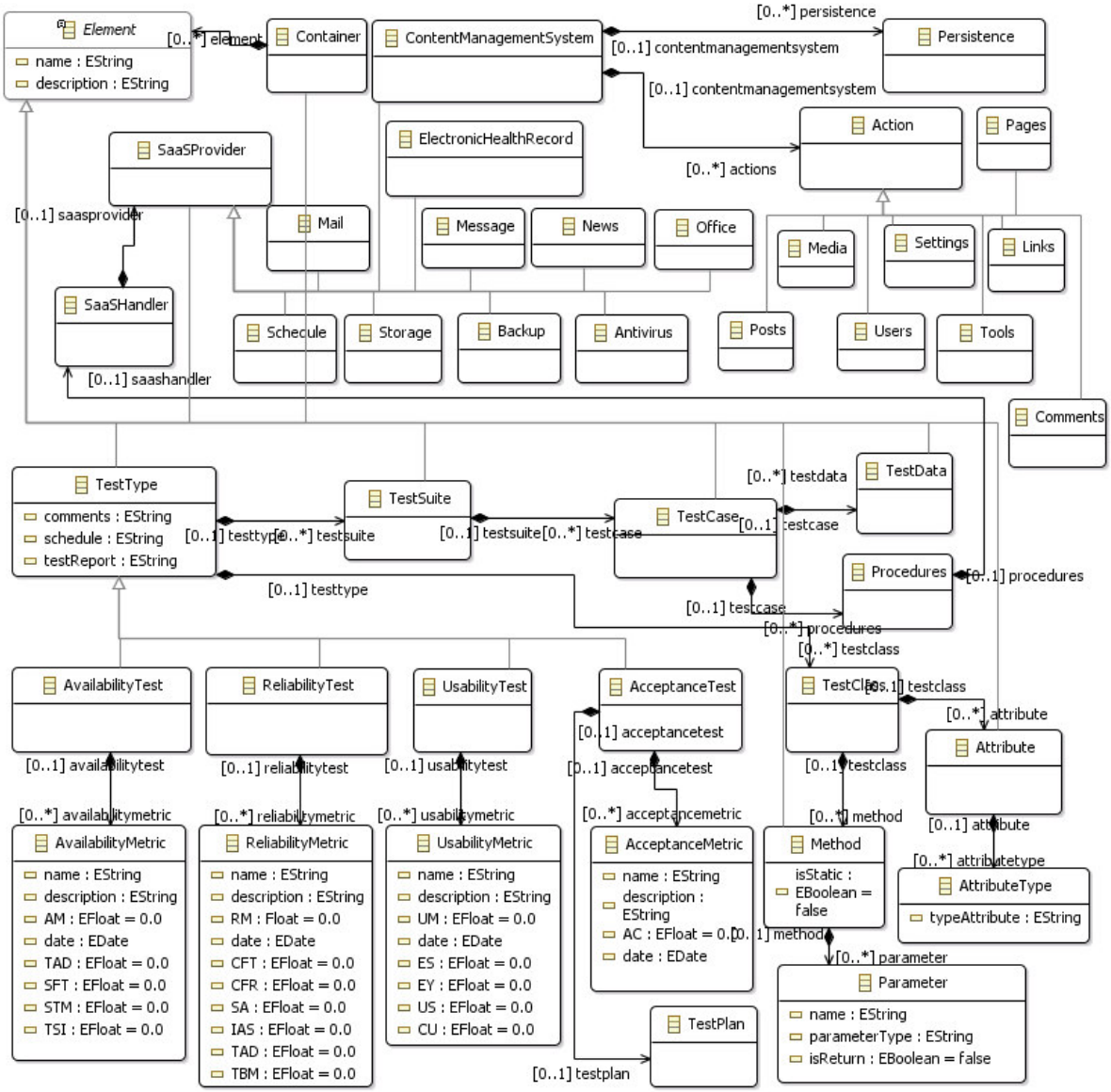

Figura 14. Metamodelo para Teste Abstrato de Nuvem para SaaS (MACT4SaaS)

A Figura 15 apresenta um metamodelo para PHP. A Figura 16 apresenta um modelo para a API do PHPUnit. A Figura 17 apresenta um padrão para modelo de teste concreto. Um modelo para API do PHPUnit é conforme ao metamodelo do PHP, e o padrão para modelo de teste concreto é conforme ao metamodelo do PHP e reusa alguns elementos do modelo para API do PHPUnit. A Figura 16 apresenta *PHPUnit_Framework_TestCase* como uma classe abstrata a ser extendida por outras classe que contem as funções de teste. Figura 17 apresenta *TemplateTest* que tem funções para implementar o teste de disponibilidade (*testAvailability*), o teste de confiabilidade (*testReliability*), o teste de usabilidade (*testUsability*) e o teste

de aceitação (*testAcceptance*). Algumas classes são fornecidas a fim de completar tais funções como *AvailabilityTest* e *AvailabilityMetric*.

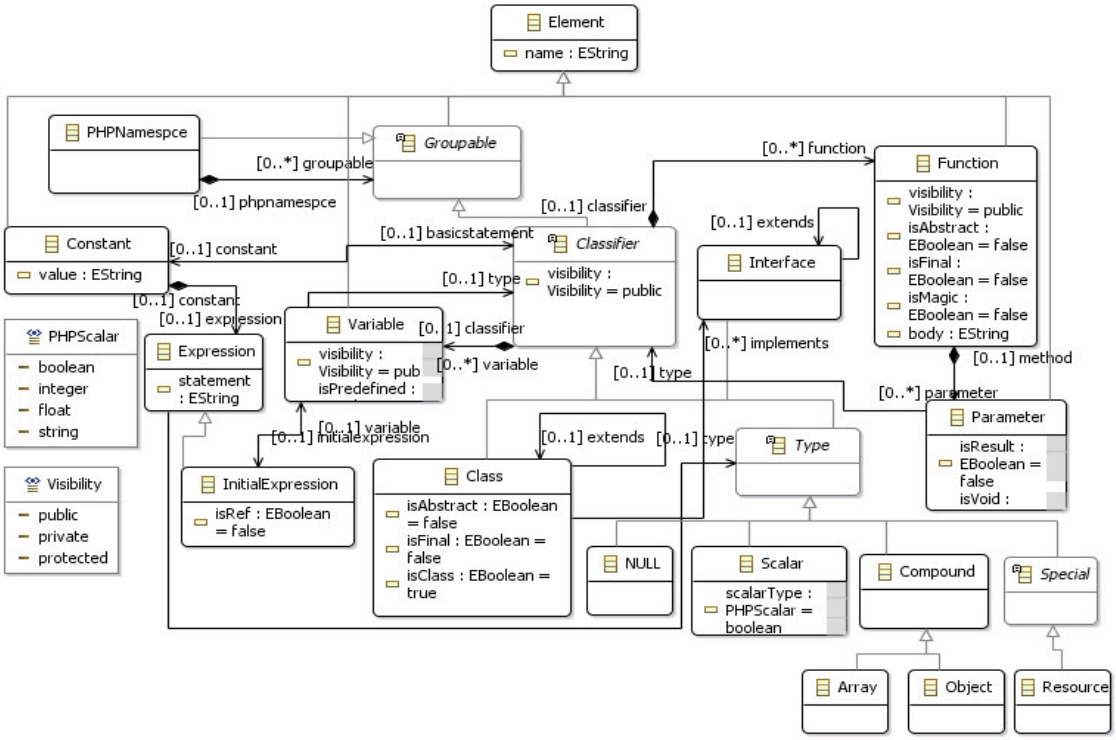

Figura 15. Metamodelo para PHP

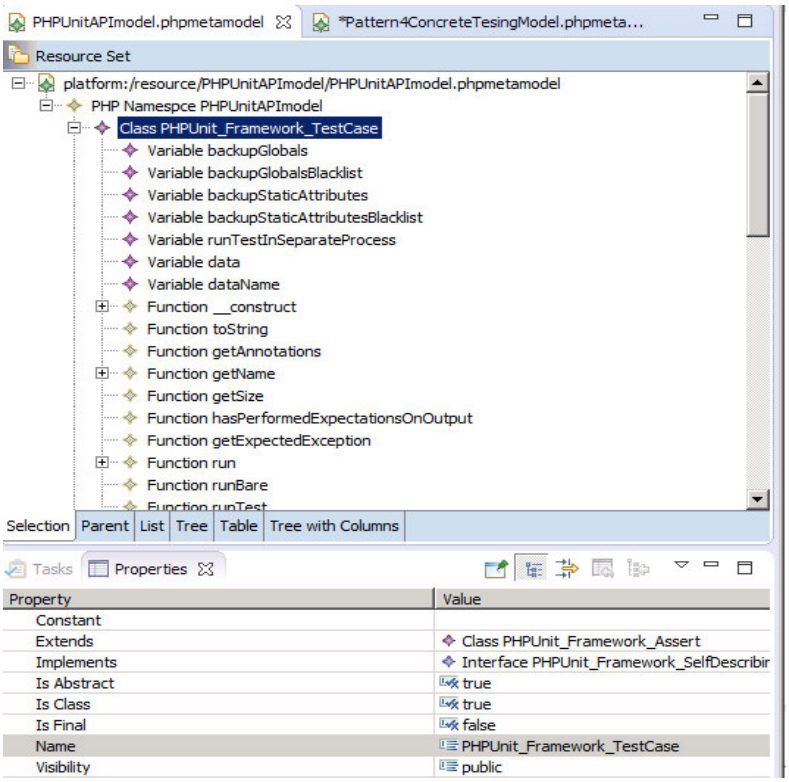

Figura 16. Um modelo para a API do PHPUnit

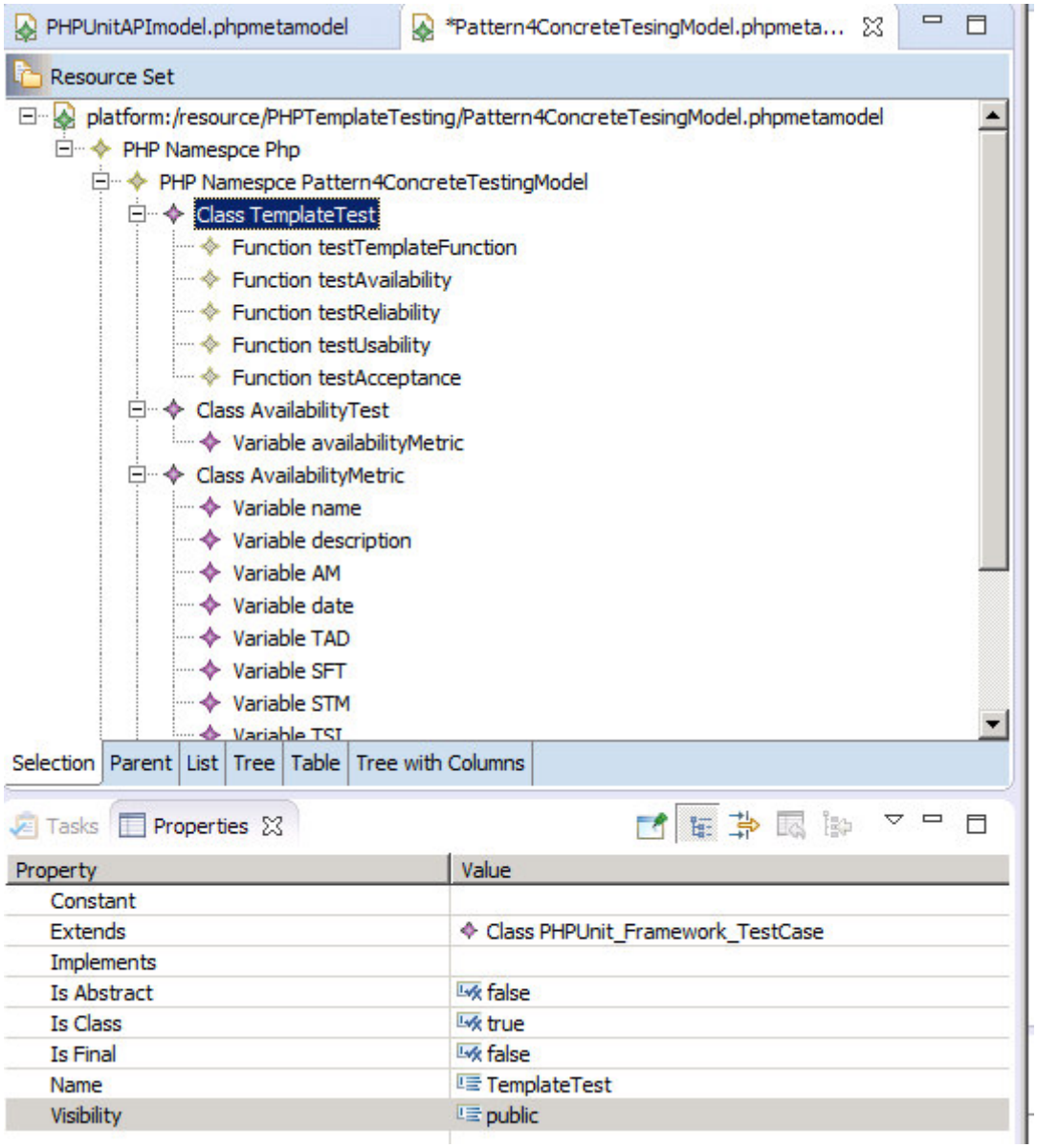

Figura 17. Padrão para modelo de teste concreto

4.4 Metodologia para Geração de Casos de Teste para Aplicações Open-SaaS

Uma metodologia para gerar os casos de teste de disponibilidade, confiabilidade, usabilidade e aceitação é proposta a seguir com o intuito de exibir a forma como é feita esta geração.

Como foi dito na Seção 1.2 e demonstrado na Tabela 3, existe uma carência dos testes citados acima em nuvens open-SaaS, devido as equipes de teste privilegiarem os testes de escalabilidade, desempenho, funcional, *multi-tenancy* e de componentes, que verificam tão-somente as funcionalidades da nuvem. Por isso, era

necessário criar uma solução que contemplasse estes tipos de teste, mais orientados ao usuário.

A metodologia para geração de casos de teste compõe-se das etapas exibidas na Figura 18.

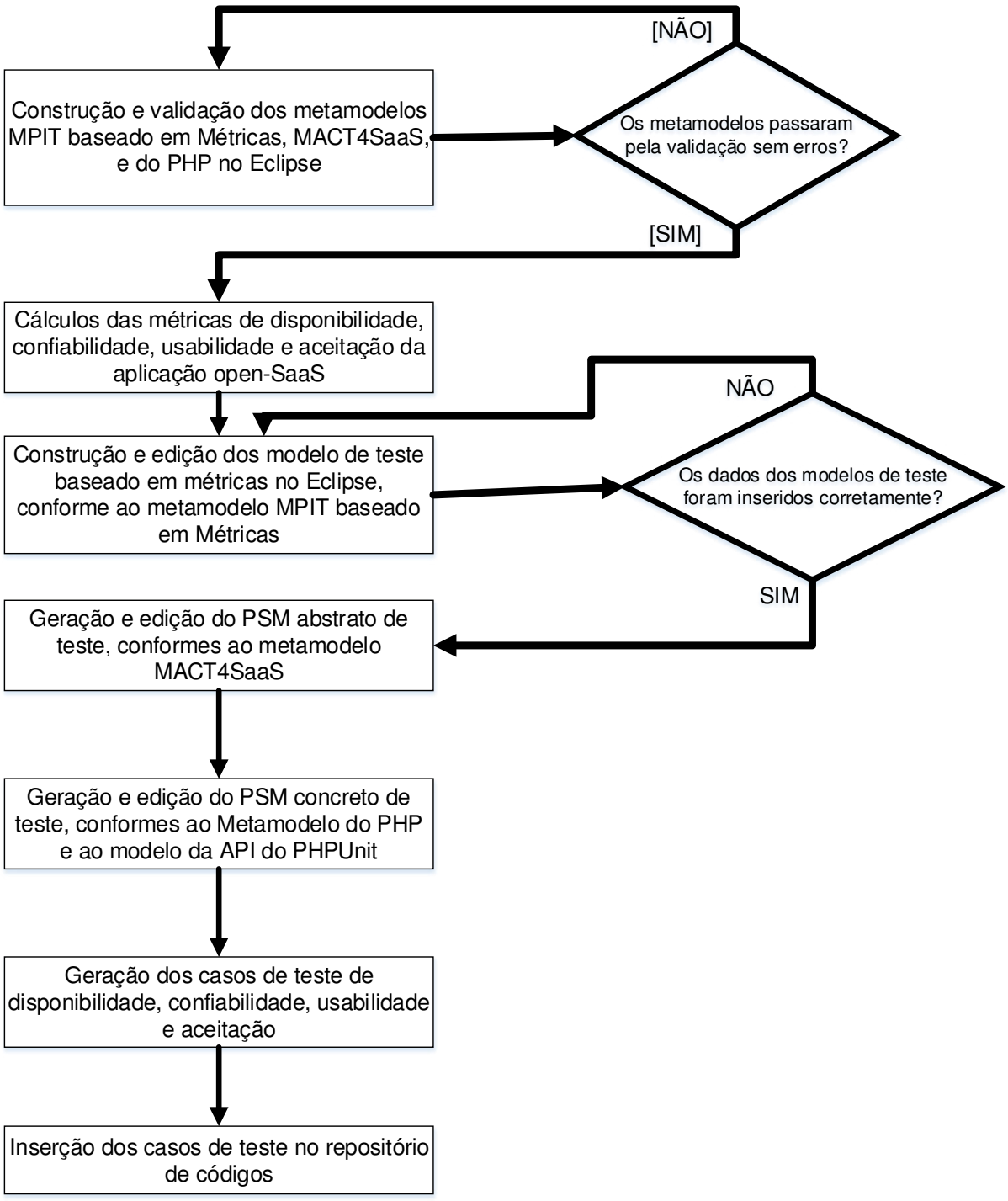

Figura 18. Metodologia para geração de casos de teste

4.5 Síntese

Neste capítulo uma abordagem para geração de casos de teste foi apresentada, na qual consiste do *framework* FCTSaaS que possui em sua estrutura metamodelos e modelos conformes a estes metamodelos. A geração destes modelos deve-se à criação de definições de transformação entre os metamodelos participantes.

A estrutura do *framework* foi apresentada e seu modo de funcionamento foi explicado. Este *framework* gera códigos de teste e códigos de aplicação que são referenciados pelos códigos de teste. Definições de transformação M2M e M2C levam à geração dos dois tipos de código, os quais referem-se à gramática EBNF do PHP.

Métricas de disponibilidade, confiabilidade, usabilidade e aceitação foram propostas e suas fórmulas foram definidas. Estas métricas foram definidas a fim de medir a qualidade das aplicações open-SaaS e são atributos dos metamodelos que foram propostas.

Em seguida, os metamodelos que fazem parte do *framework* foram exibidos, assim como seus componentes. O Metamdelo para Teste Independente de Plataforma baseado em Métricas (MPIT baseado em Métricas), o Metamodelo para Teste Abstrato de Nuvem para SaaS (MACT4SaaS), o Metamodelo do PHP, o modelo do API do PHPUnit [3] são os componentes que participam da geração dos modelos de teste, PSMs abstratos de teste e PSMs concretos de teste até a geração dos códigos de teste. O Metamodelo para Teste Independente de Plataforma baseado em Métricas é uma adaptação e alteração do trabalho exibido em [50] e que apresenta os testes e as métricas de disponibilidade, confiabilidade, usabilidade e aceitação. Este metamodelo participa da criação dos modelos de teste baseados em métricas, que são conformes a este Metamodelo. O segundo metamodelo participada da geração dos PSMs abstratos de teste, que são conformes a ele. O terceiro metamodelo participa da geração dos PSMs concretos de teste, os quais são conformes a ele, e dos códigos de teste.

Por fim, uma metodologia para geração dos casos de teste para aplicações *open-SaaS* foi proposto, como uma forma de exibir claramente o procedimento para a geração dos casos de teste e do código fonte da aplicação *open-SaaS*.

5 Implementação do Protótipo do *Framework* para Casos de Teste em SaaS de Código Aberto

Este capítulo apresenta a implementação do protótipo responsável pela geração dos casos de teste para aplicações open-SaaS. Este protótipo foi construído por meio do EMF (*Eclipse Modeling Framework*) [10], que auxilia na criação de metamodelos e facilita as atividades necessárias à geração de código e/ou de casos de teste.

O protótipo *SaaSTestTool* será apresentado, o qual implementa o *framework* FCTSaaS (Capítulo 4).

Em seguida, cada um de seus componentes e suas funções dentro do protótipo serão exibidos e explicados.

A implementação da geração dos casos de teste para aplicações open-SaaS é detalhada, com destaque para a construção dos metamodelos envolvidos, a criação do modelo de teste baseado em métricas, a geração do modelo de teste abstrato e concreto, e por fim, a geração do código de teste resultante.

5.1 Prototipagem do *framework* FCTSaaS

A Figura 19 apresenta o protótipo SaaSTestTool que extende o *framework* FCTSaaS.

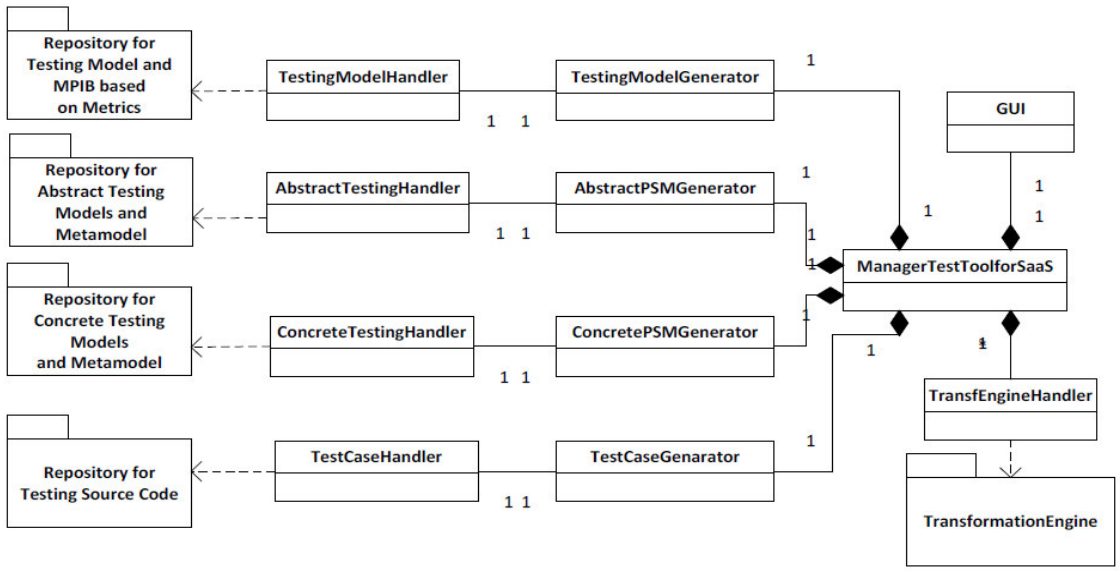

Figura 19. SaaSTestTool: protótipo do *framework* FCTSaaS

O protótipo é responsável por gerar casos de teste para aplicações open-SaaS. Ele gerencia os repositórios dos metamodelos de teste, dos modelos de teste abstratos, dos modelos de teste concreto e dos código de teste. Definições de transformação são escritos em ATL e executados no plugin ATL para Eclipse [8].

5.1.2 Componentes do *SaaSTestTool*

O protótipo possui os seguintes componentes (Figura 19):

- *ManagerTestTool4SaaS*: responsável por gerenciar o processo de gerar código de teste em PHPUnit a partir de modelos de alto nível como Modelos de Teste Abstrato, Modelos de Teste Concreto e Modelos de Teste. Ele também gerencia a máquina de transformação através do componente *TransfEngineHandler*;
- *GUI*: interface gráfica do protótipo;
- *Testing Model Generator*: tem um editor e permite ao usuário criar um modelo de teste que faz referência ao PIM;
- *Abstract PSM Generator*: tem como entrada um modelo de teste e gera como saída um modelo de teste abstrato. Para este propósito, ele chama uma máquina de transformação para executar uma definição de transformação de modelo escrito em uma linguagem de transformação como ATL;
- *Concrete PSM Generator*: leva como entrada um Modelo de Teste Abstrato e gera como saida um Modelo de Teste Concreto. Para este propósito, ele chama uma máquina de transformação para executar uma definição de transformação de modelo escrito em uma linguagem de transformação como ATL;
- *Test Case Generator*: tem como entrada um Modelo de Teste Concreto e dá como saída um código de teste baseado no PHPUnit;

Os componentes *Abstract PSM Generator*, *Concrete PSM Generator*, e *Test Case Generator* reutilizam os componentes presentes no plugin ATL para Eclipse [8] para a criação dos modelos de teste abstrato, dos modelos de teste concreto e dos códigos de teste.

Figura 20 apresenta um *screenshot* do protótipo *SaaSTestTool* desenvolvido em EMF e no ambiente Eclipse Luna.

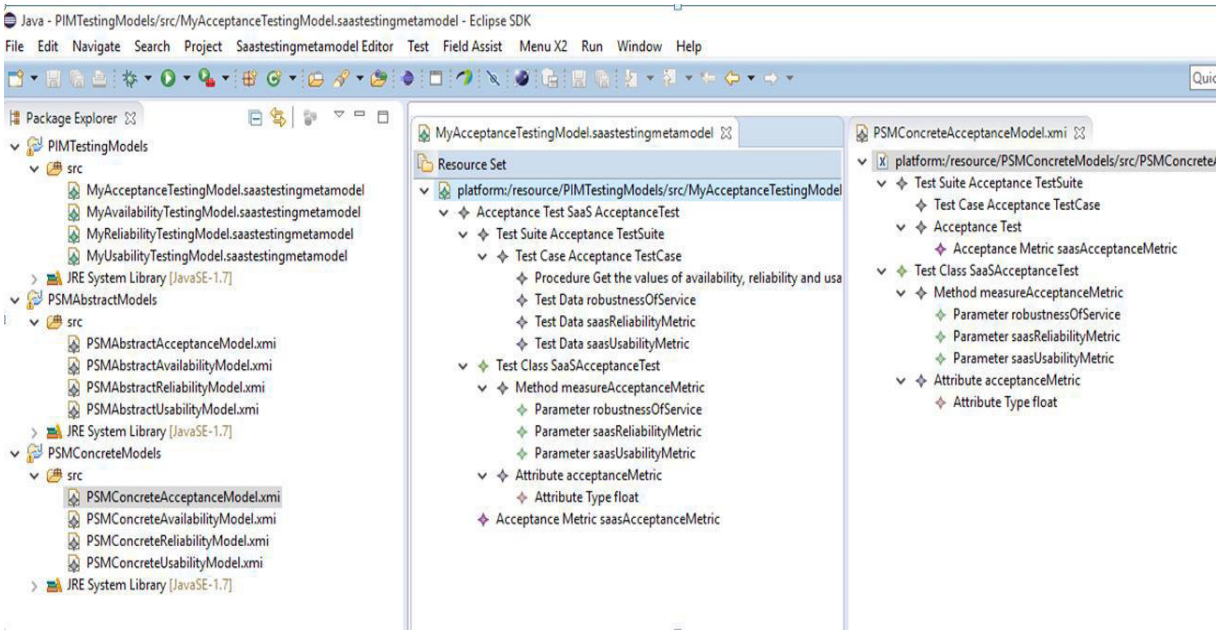

Figura 20. Protótipo *SaaSTestTool* para suportar o teste de SaaS

Para a geração dos modelos de teste abstrato, dos modelos de teste concreto e dos códigos de teste, definições de transformação foram escritas utilizando o plugin ATL para Eclipse [8]. Estas definições envolvem os metamodelos MPIT baseado em Métricas, MACT4SaaS, o Metamodelo do PHP e o modelo da API do PHPUnit. Os anexos 8.1, 8.2 e 8.3 contem as definições de transformações construídas.

Após a geração dos modelos de teste abstrato e dos modelos de teste concreto por meio das definições de transformação e dos componentes do protótipo, os códigos de teste de disponibilidade, confiabilidade, usabilidade e aceitação são gerados. O Código 1 exibe o resultado do framework FCTSaaS para a geração do caso de teste de confiabilidade.

Este código de teste tem os seguintes atributos:

- *RM*: representa o valor da métrica de confiabilidade;
- *CFT***:** representa a tolerância do SaaS aos defeitos [38];
- *CFR*: representa a capacidade do open-SaaS para recuperar-se de falhas em um período de tempo [38];
- *SA*: mede a relação entre o número de respostas corretas às requisições do usuário e o número total de requisições em um período específico de tempo [38];

- *IAS*: representa o impacto dos problemas de segurança na métrica de confiabilidade;
- *TAD*: representa o tempo após a implantação do *SaaS*;
- *TBM*: representa o tempo para o *SaaS* estar maduro.

```
class ReliabilityMetric {
 public $ var RM;
 public $ var CFT;
 public $ var CFR;
 public $ var SA;
 public $ var IAS;
 public $ var TAD;
 public $ var TBM;
}
```

Código 1. Código de teste para o caso de teste de confiabilidade

O *framework* FCTSaaS, além de gerar este código de teste para o caso de teste de confiabilidade, ele gera os códigos de teste dos casos de teste de disponibilidade, usabilidade e aceitação.

5.2 Implementação da geração dos casos de teste para open-SaaS

A geração dos casos de teste foi realizada inicialmente através da construção dos metamodelos de teste MPIT baseada em Métricas, do MACT4SaaS, do Metamodelo para PHP e do modelo do API do PHPUnit. Em seguida, procedeu-se à geração do modelo de teste baseado em métricas, dos PSMs abstratos de teste, dos PSMs concretos de teste e, por fim, à geração dos códigos de teste para os casos de teste de disponibilidade, confiabilidade, usabilidade e aceitação. Estas etapas serão descritas a seguir

5.2.1 Construção dos Metamodelos de Teste

Esta etapa da geração dos códigos de teste foi realizada dentro do ambiente Eclipse Luna, com a ferramenta EMF.

Em EMF, todo metamodelo criado é conforme ao *Ecore*. O Ecore é um metamodelo que compõe o *framework* central do EMF e é usado para descrever modelos e suporte em tempo de execução (*runtime*) para os modelos incluindo

notifição de mudanças, suporte persistente com serialização XMI (*XML Metadata Interchange*), e uma API eficiente para manipular objetos EMF (classes, atributos, relacionamentos) genericamente [9].

O primeiro metamodelo de teste construído foi o MPIT baseado em Métricas. O arquivo *SaaSTestingMetamodel.ecore*, que representa este metamodelo, é exibido no lado esquerdo da Figura 21, junto com os componentes deste metamodelo, do lado direito da figura. Em seguida, foi construído o metamodelo MACT4SaaS, representado pelo arquivo *AbstractCloudTestMetamodel4SaaS.ecore*, e este arquivo, é apresentado do lado esquerdo na Figura 22, com seus componentes mostrados do lado direito. Por fim, foi construído o Metamodelo do PHP, representado pelo arquivo *PHPMetamodel.ecore*. Este arquivo é localizado no lado esquerdo da Figura 23, e seus componentes são apresentados do lado direito da mesma figura.

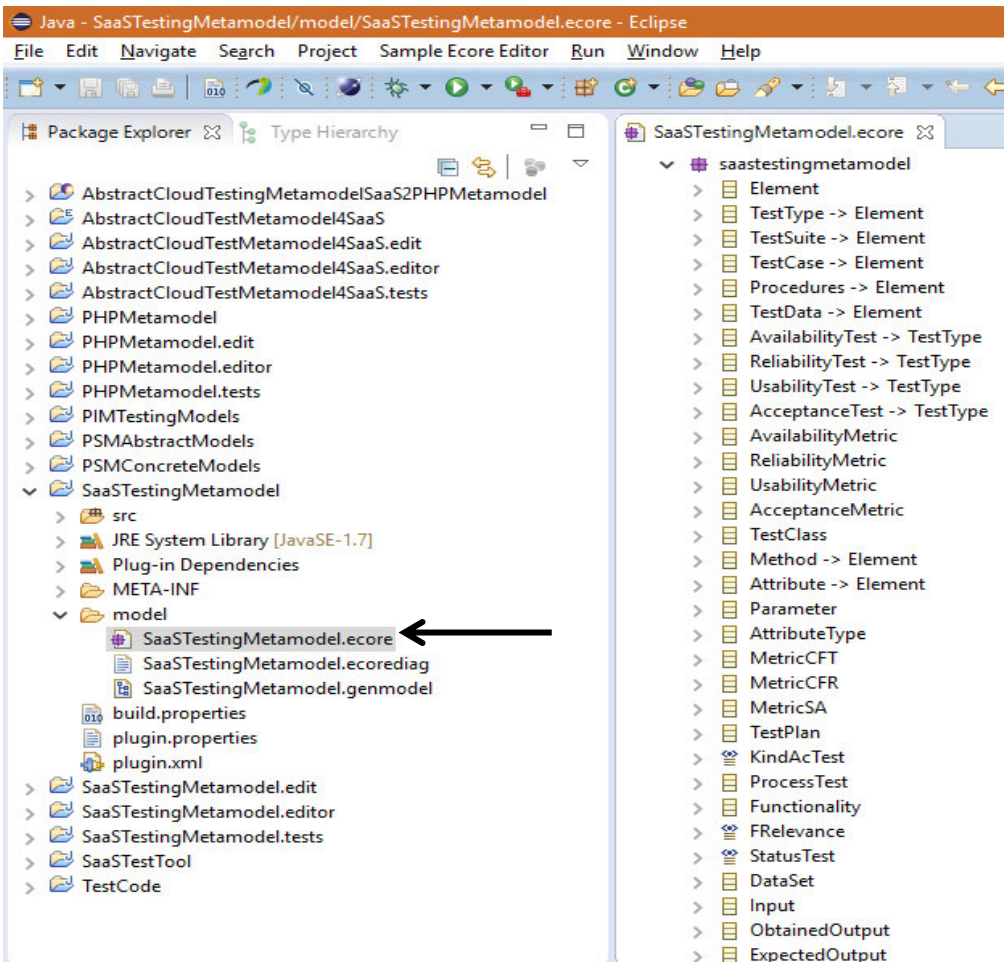

Figura 21. Arquivo *SaaSTestingMetamodel.ecore*, que representa o MPIT baseado em Métricas

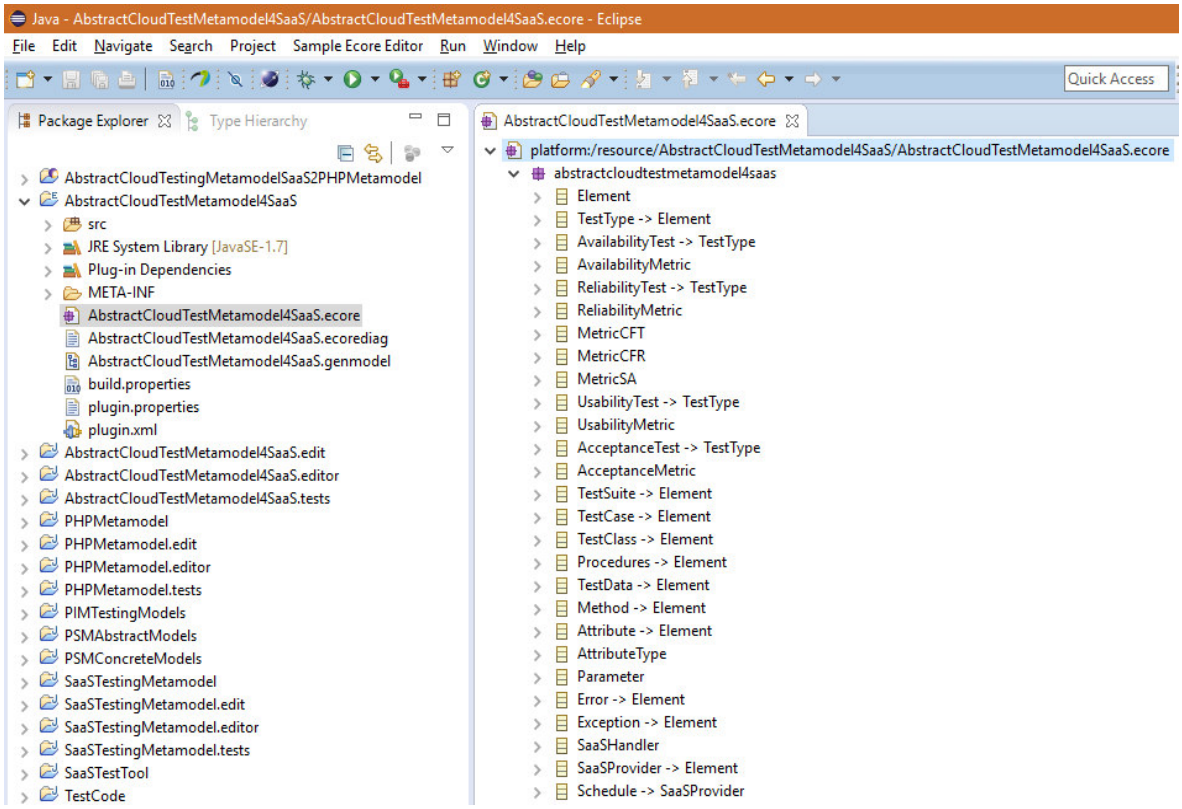

Figura 22. Arquivo *AbstractCloudTestMetamodel4SaaS.ecore*, que representa o MACT4SaaS

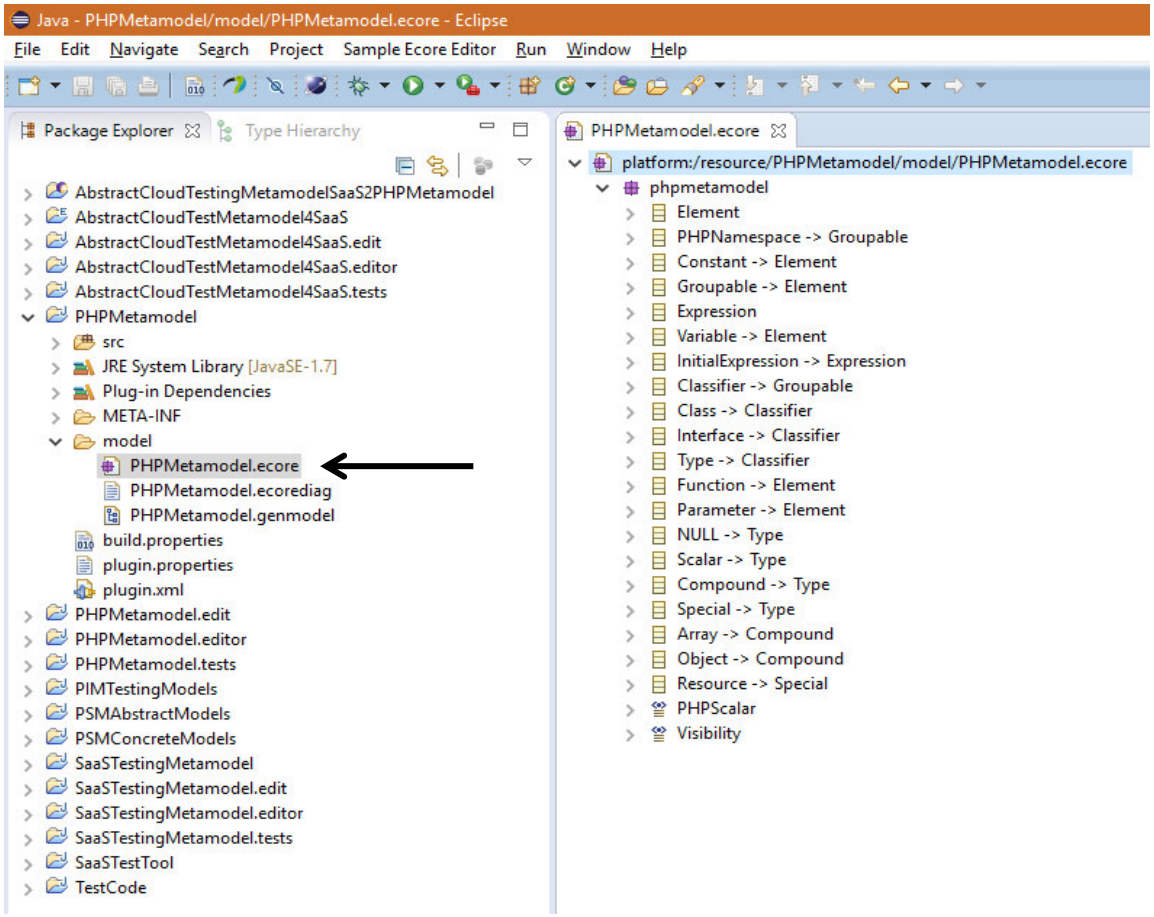

Figura 23. Arquivo *PHPMetamodel.ecore*, que representa o Metamodelo do PHP

5.2.2 Geração dos modelos de teste

Partindo destes arquivos, procedeu-se à geração dos modelos de teste. Esta etapa foi possível graças à ferramenta *genmodel* do EMF, que realizou a geração de código dos metamodelos e dos plugins para os metamodelo de teste. A Tabela 6 exibe o nome dos plugins para cada metamodelo criado.

Metamodelo de teste	**Plugins gerados pela ferramenta *genmodel* do EMF**
Metamodelo para Teste Independente de Plataforma baseado em Métricas	SaaSTestingMetamodel.edit
	SaaSTestingMetamodel.editor
	SaaSTestingMetamodel.tests
Metamodelo para Teste Abstrato de Nuvem para Saas	AbstractCloudTestMetamodel4SaaS.edit
	AbstractCloudTestMetamodel4SaaS.editor
	AbstractCloudTestMetamodelo4SaaS.tests
Metamodelo do PHP	PHPMetamodel.edit
	PHPMetamodel.editor
	PHPMetamodel.tests

Tabela 6. Plugins gerados pela ferramenta *genmodel* para os metamodelos de teste

A partir destes *plugins*, uma nova instância do ambiente Eclipse Luna foi gerada, o *TestingModelGenerator* (ver Figura 24), e dessa forma foi construído os modelos de teste baseado em métricas, conforme ao metamodelo MPIT baseado em Métricas. A Figura 24 exibe um fragmento do componente *TestingModelGenerator* juntamente com os quatro modelos de teste criados.

5.2.3 Geração dos modelos abstratos de teste

Esta etapa foi realizada com a participação dos metamodelos MPIT baseado em Métricas e MACT4SaaS. Uma definição de transformação, chamada *SaaSTestingMetamodel2AbstractCloudTestingMetamodel.atl*, foi criada através do plugin ATL para Eclipse [8], envolvendo estes metamodelos, para a geração do modelo abstrato de teste, que possui o formato *xmi*. A Figura 25 exibe o componente

*AbstractPSM*Generator do protótipo *SaaSTestTool* (ver Figura 19), que reusa o plugin da ATL [8] para a geração dos PSMs abstratos de teste. A definição de transformação é exibida no Anexo 8.1.

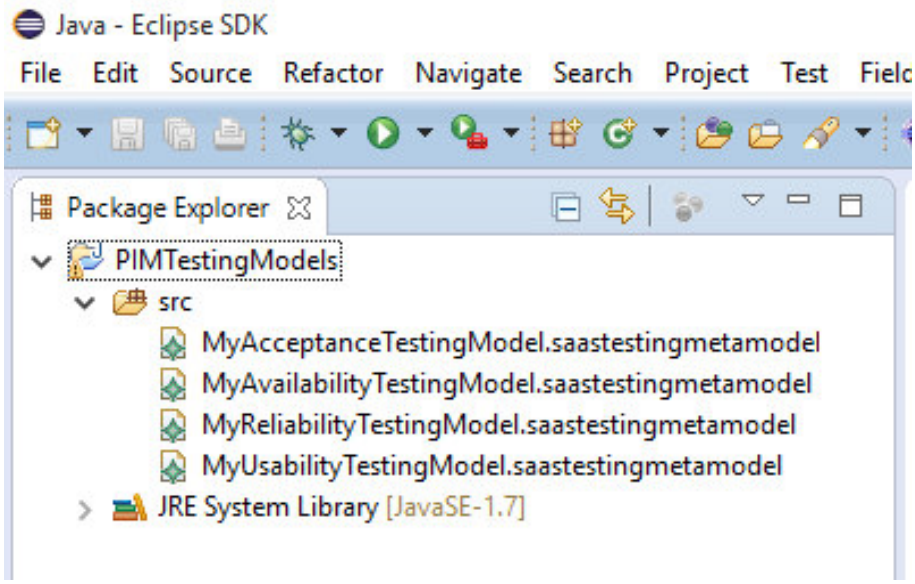

Figura 24. Modelos de teste criados no *TestingModelGenerator* (fragmento)

5.2.4 Geração do modelo concreto de teste

Esta etapa foi realizada através do componente *ConcretePSMGenerator* (ver Figura 19) do protótipo *SaaSTestTool*. Uma definição de transformação, chamada *AbstractCloudTestingMetamodelSaaS2PHPMetamodel.atl*, foi criada, envolvendo os metamodelos MACT4SaaS e o Metamodelo do PHP, para a geração do modelo concreto de teste, que possui o formato XMI. A Figura 26 apresenta o componente responsável por esta geração. Ela utiliza a definição de transformação criada para esta geração e reusa o plugin da ATL para Eclipse [8] para a geração dos PSM concreto de teste. As regras de transformação que compõem esta definição estão presentes no Anexo 8.2.

5.2.5 Geração dos códigos de teste

Esta etapa envolve o componente *TestCaseGenerator* do protótipo *SaaSTestTool* (ver Figura 19). Uma definição de transformação, com o nome *PHPUnit2Code*, foi criada para a geração dos códigos de teste, que por sua vez possuem formato *php*. A Figura 27 exibe este componente, que reusa o plugin da ATL para Eclipse [8] para a geração dos códigos de teste. As regras de transformação que compõem esta definição de transformação estão presentes no Anexo 8.3.

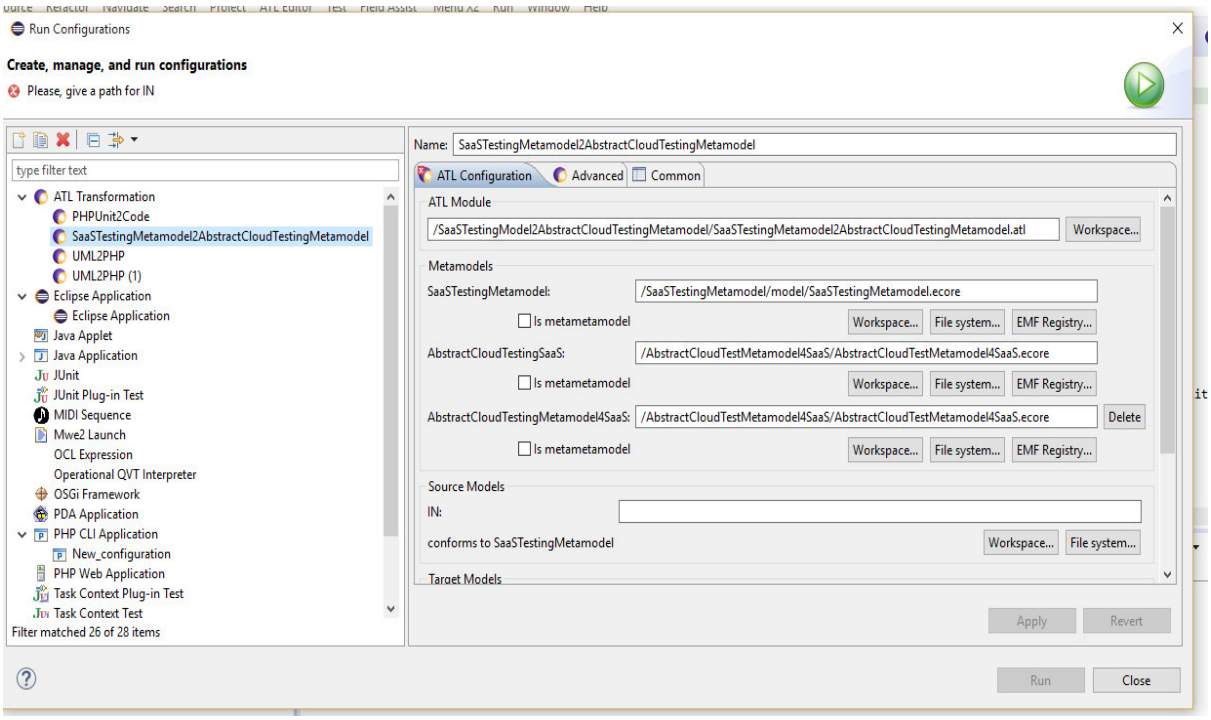

Figura 25. Componente *AbstractPSMGenerator* do protótipo *SaaSTestTool*, que reusa o plugin da ATL para Eclipse [8] para gerar os PSMs abstratos de teste

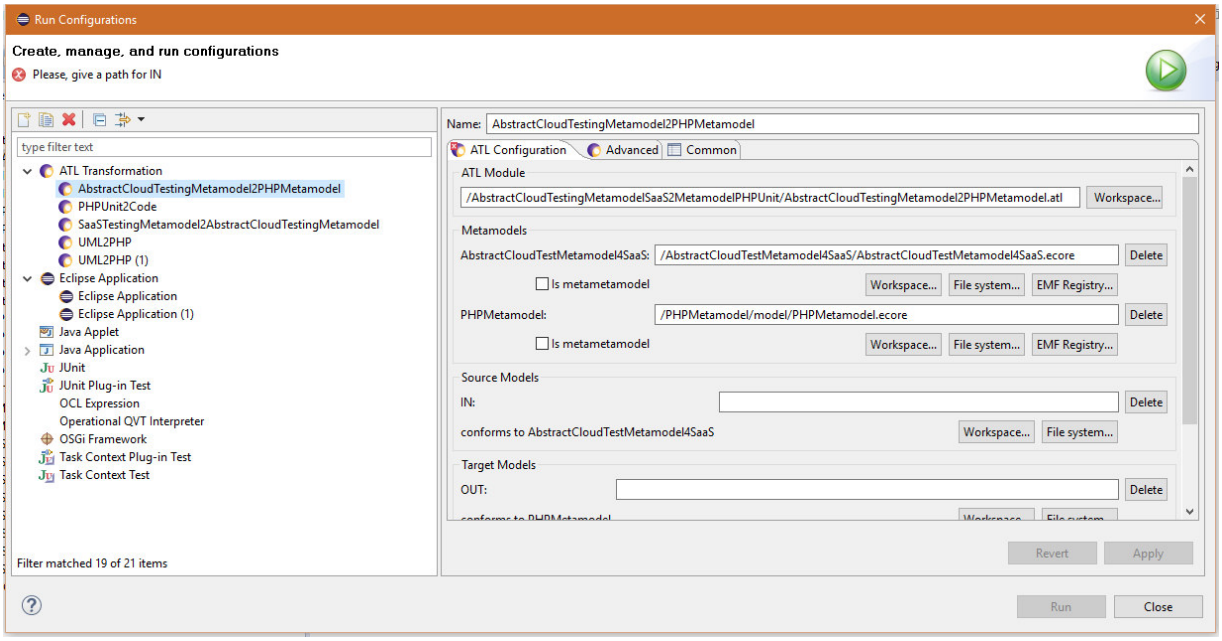

Figura 26. Componente *ConcretePSMGenerator* do protótipo *SaaSTestTool*, que reusa o plugin da ATL para Eclipse [8] para gerar o PSM concreto de teste

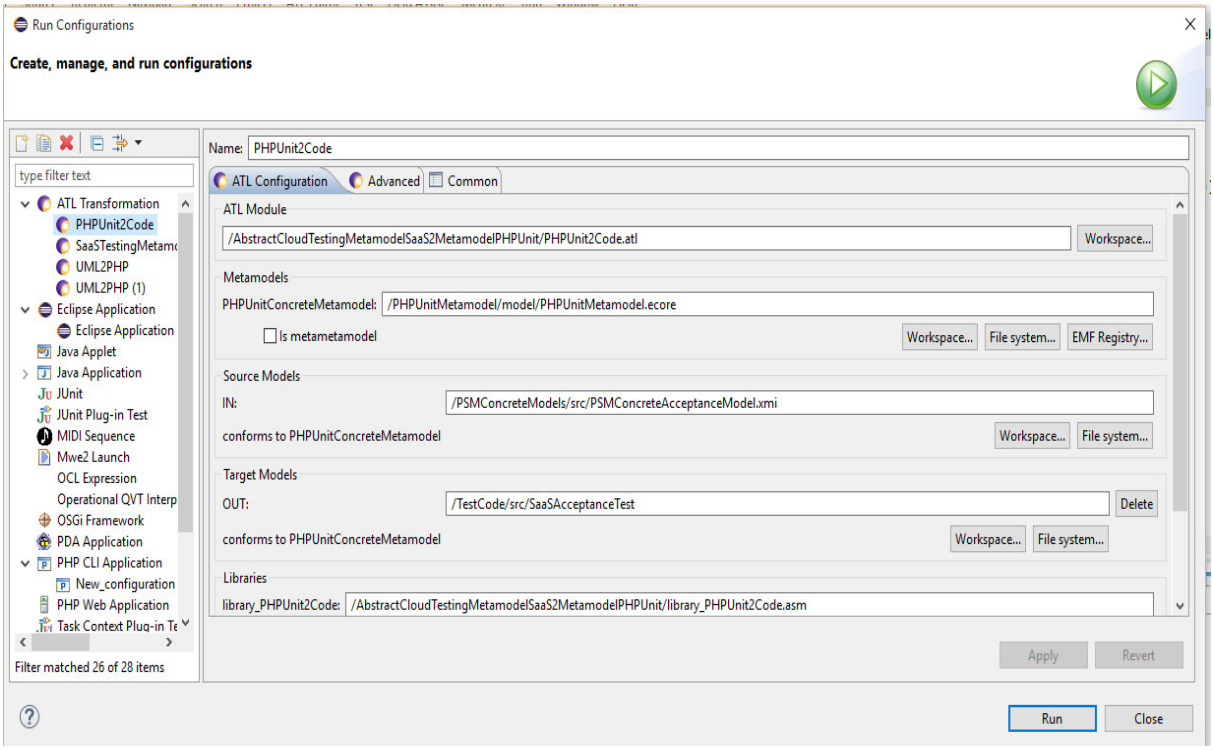

Figura 27. Componente *TestCaseGenerator* do protótipo *SaaSTestTool*, que reusa o plugin da ATL para Eclipse [8] para gerar os códigos de teste

5.3 Síntese

Neste capítulo, a implementação do protótipo do *framework* FCTSaaS foi exibida.

Os componentes do protótipo foram explicados e sua funções foram detalhadas. O protótipo *SaaSTestTool* foi construído a partir dos metamodelos de teste (ver Capítulo 4) e compõe-se de um gerador dos modelos de teste (*TestingModelGenerator*) e de um manipulador dos modelos de teste (*TestingModelHandler*); de um gerador de PSMs abstratos para teste (*AbstractPSMGenerator*) e de um manipulador destes PSMs (*AbstractPSMHandler*); de um gerador de PSMs concretos de teste (*ConcretePSMGenerator*) e de um manipulador destes PSMs (*ConcretePSMHandler*); e de um gerador de casos de teste (*TestCaseGenerator*) e de um manipulador destes casos de teste (*TestCaseHandler*). Todos estes componentes são gerenciados pelo *ManagerTestTool4SaaS*, que está integrada a uma GUI e a uma manipuladora da máquina de transformação, o *TransfEngineHandler*. A máquina de transformação utilizada foi a máquina da linguagem ATL. Os componentes *AbstractPSMGenerator*, *ConcretePSMGenerator* e *TestCaseGenerator* e seus manipuladores reusam o plugin da ATL para Eclipse [8] para gerar os PSMs abstratos de teste, PSMs concretos de teste e os códigos de teste, respectivamente..

A implementação da geração dos casos de teste para aplicações SaaS foi detalhada.

6 Exemplo Ilustrativo

A fim de ilustrar as funcionalidades do *framework* FCTSaaS proposto e seu protótipo, um exemplo ilustrativo que consiste da geração de códigos de teste para uma aplicação do *Wordpress* é exibido.

O projeto de teste consistiu na criação de um *blog* (aplicação *open-SaaS* padrão do *Wordpress*) e posterior hospedagem em um servidor de alto desempenho pertencente à empresa *Hostgator* [23]. Os testes de disponibilidade, confiabilidade, usabilidade e aceitação foram realizadas através do cálculo das respectivas métricas. A geração dos modelos de teste, modelos abstratos de teste, modelos concretos de teste e os esqueletos dos códigos de teste foram gerados dentro do ambiente Eclipse Luna, por meio de definições de transformação criadas com o *plugin* ATL do Eclipse [8].

A Figura 28 exibe o PIM para a aplicação do *Wordpress* criada. Ela apresenta alguns componentes presentes no ambiente open-SaaS do *Wordpress*.

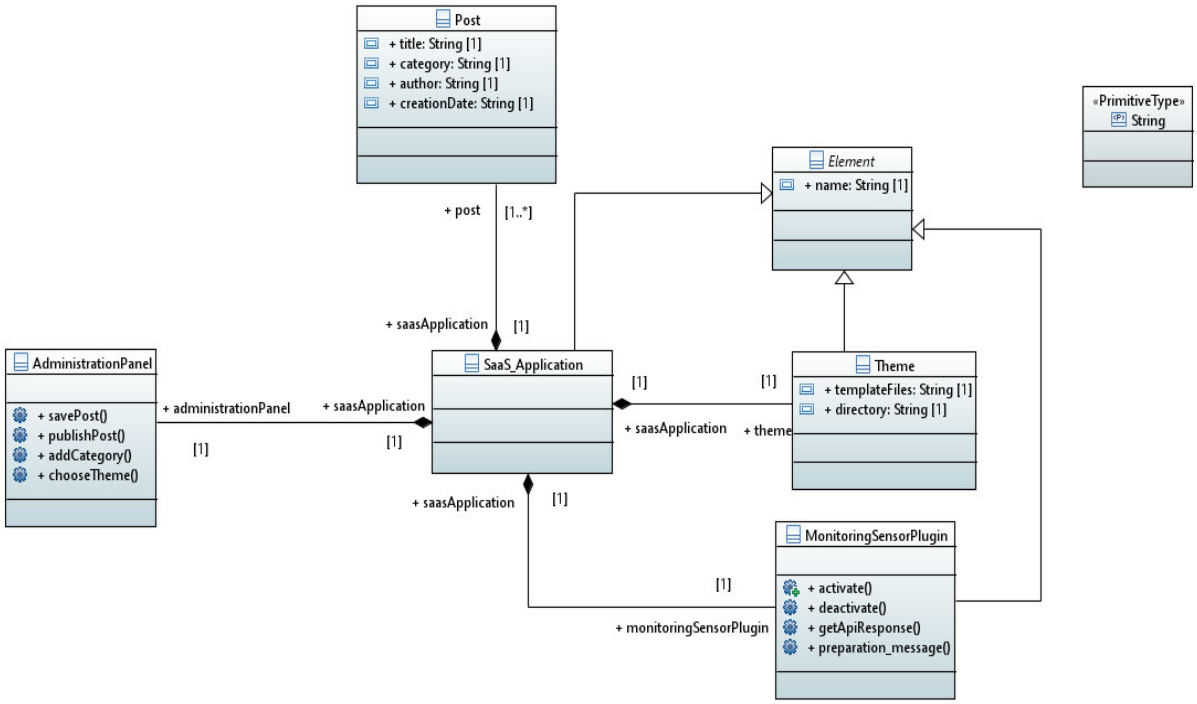

Figura 28. PIM de aplicação *open-SaaS* do Wordpress

Na Figura 12, o PIM é referenciado pelos modelos de teste criados. Isto significa que os modelos de teste de disponibilidade, confiabilidade, usabilidade e aceitação referenciam as classes deste PIM da aplicação SaaS.

O diagrama de classe contem os seguintes elementos:

- *SaaS_Application*: é a principal classe do diagrama. É a aplicação open-SaaS propriamente dita, criada no *Wordpress*. Ele tem um nome como atributo, herdado da classe abstrata *Element*. É composta das classes *AdministrationPanel*, *Theme*, *Post* e *MonitoringSensorPlugin*;
- *Element*: é a superclasse do diagrama. Ele tem um nome como atributo;
- *AdministrationPanel*: esta classe representa o painel de administração da aplicação *open-SaaS* do *Wordpress*. Através deste painel, atividades como salvar e publicar *posts*, acrescentar categorias de *posts* e escolher temas podem ser realizados. Os métodos que representam estas atividades e que estão presentes na classe são *savePost*(), *publishPost*(), *addCategory*() e *chooseTheme*();
- *Post*: esta classe representa os *posts* presentes na aplicação *open-SaaS* do *Wordpress*. Ele possui como atributos um título, uma categoria, o nome do autor e a data de criação;
- *Theme*: esta classe representa o tema da aplicação *open-SaaS* do *Wordpress*. Um tema no *Wordpress* é a tela de fundo da aplicação open-SaaS. Os atributos desta classe são *templateFiles* (o nome dos arquivos que compõem o tema), e *directory* (o diretório onde se encontra o tema);
- *MonitoringSensorPlugin*: esta classe representa o *plugin* específico para *Wordpress* [65] que monitora o *uptime* de uma aplicação open-SaaS. Ele compõe a classe *SaaS_Application*. Alguns métodos deste *plugin* são *activate*(), *deactivate*(), *getApiResponse*() e *preparation_message*().

6.1 Editando e gerando modelos de teste

A Figura 29 apresenta o *TestingModel* conforme ao metamodelo MPIT baseado em Métricas (Figura 13) e seu objetivo é testar um sistema de *blog* como o *Wordpress* apresentado na Figura 26.

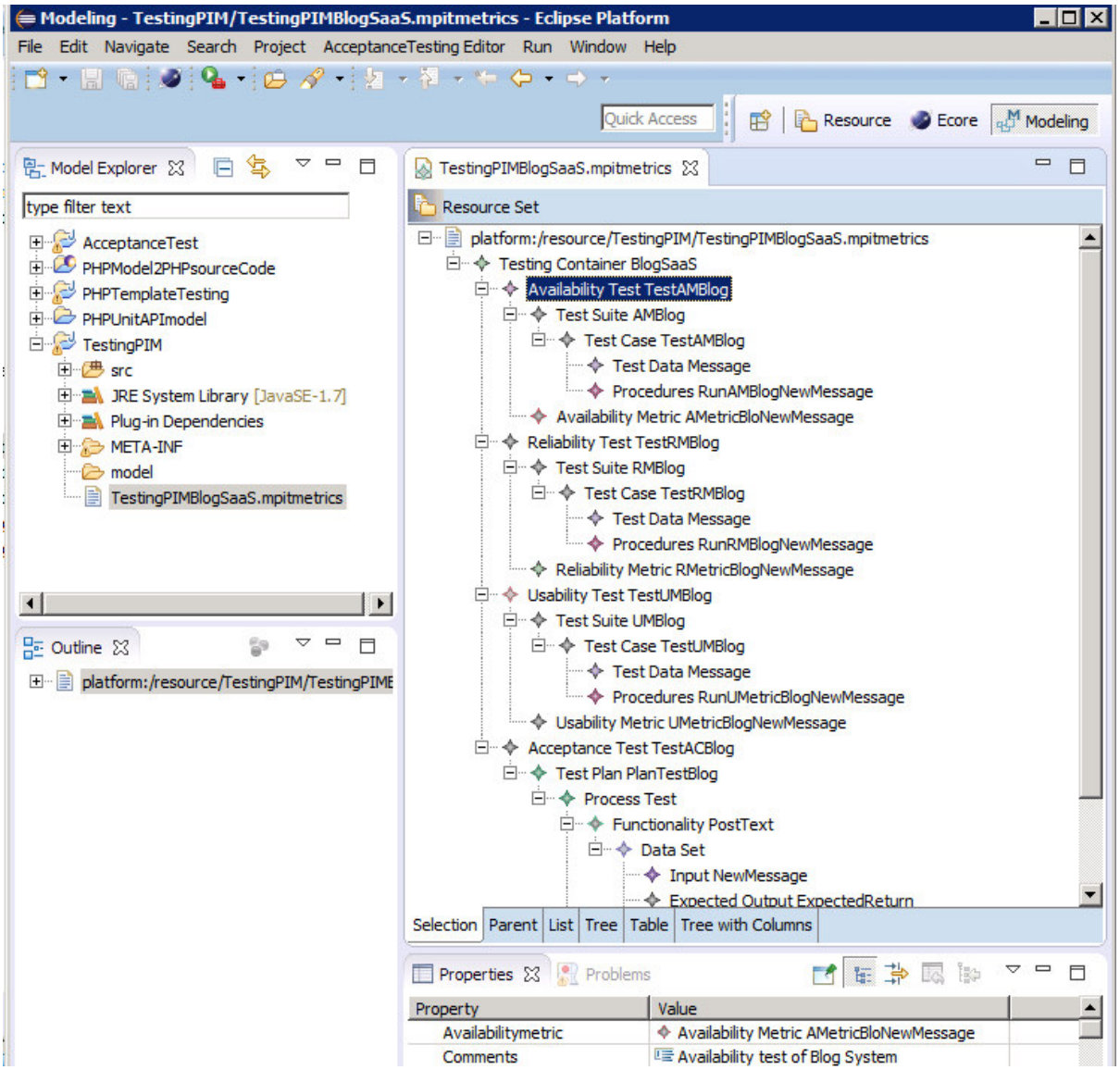

Figura 29. Modelo de teste baseado em métricas para o sistema de *blog*

Uma definição de transformação (apresentada no Anexo 8.1) tem como entrada *TestingModel based on Metrics* (o modelo PIM de teste) e gera como saída *Abstract Testing* (o modelo abstrato de teste). A Figura 30 apresenta o *Abstract Testing* para o sistema de *blog* proposto.

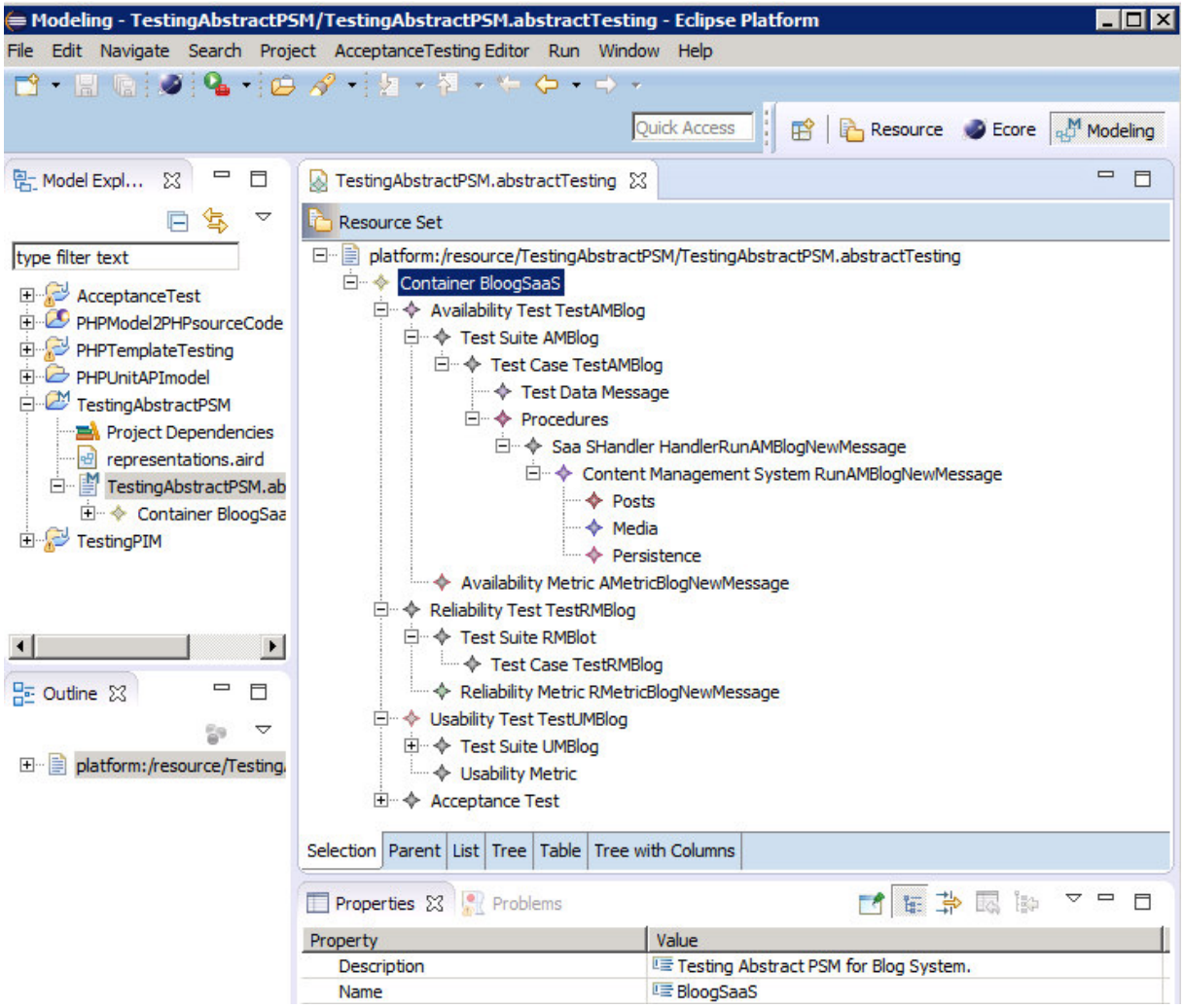

Figura 30. Modelo de teste abstrato

Outra definição de transformação (apresentada no Anexo 8.2) toma como entrada *Abstract Testing* e gera como saída *Concrete Testing* (o modelo concreto de teste). Outra definição de transformação (presente no Anexo 8.3) toma como entrada *Concrete Testing* e gera os casos de teste.

Uma vez que o código fonte PHP executável é baseado em PHPUnit, *SaaSTestTool* preenche as variáveis de métricas de acordo com os valores obtidos. Um cenário controlado foi criado para teste e é descrito como segue: um *shutdown* foi forçado no servidor e uma quebra de comunicação foi realizada (incluindo o *hardware* e o *software* do Wordpress), as respostas simuladas foram colocadas no formulário da métrica de usabilidade (ver Tabela 4), e defeitos, falhas e respostas erradas foram geradas.

A fim de obter o valor da métrica de disponibilidade (ROS), uma execução da aplicação SaaS (o sistema de *blog*) foi simulada durante três meses (TAD = 120960

minutos), onde foram obtidos os seguintes valores: SFT = 23 minutos, STM = 15 minutos e TSI = 18 minutos. Portanto, a métrica de disponibilidade ROS foi igual a 0,999.

A fim de obter a métrica de confiabilidade (RM), uma execução da aplicação SaaS (i.e. o sistema de *blog*) foi simulada, assim como defeitos (i. e. número total de defeitos = 4), falhas (i .e. número total de falhas = 3), falhas reparadas (i. e. número de falhas reparadas = 3) e solicitações (i. e. número total de solicitações = 1030 e número de respostas corretas = 1026) foram simulados. A partir destes valores, os valores para as submétricas CFT, CFR e SA foram de 0,25, 1 e 0,996, respectivamente. Tomando os valores de TAD e TSI, o valor de IAS resultante foi de 0,999. Colocando todos os valores na Equação 12 e considerando TAD = 120960 e TBM = 241920 (i. e. seis meses para ser considerado maduro), o valor da métrica RM foi de 0,902.

A fim de obter a métrica de usabilidade (UM), foi realizada uma simulação com seis usuários que preencheram o questionário presente na Tabela 4, com as suas respostas e respectivos valores presentes no Anexo 8.5. Considerando CU = 0, isto é, o contexto do usuário foi ignorado nesta simulação. Assim, colocando esses valores na Equação 19, o valor para a métrica UM obtido foi 0,584.

Durante a simulação do teste de aceitação, um plano de teste foi elaborado para determinar que o sistema de *blog* satisfaz os requisitos e é compatível com a expectativa de seus usuários. Considerando a Tabela 5, foram simulados três testes de aceitação como segue: $N_{HR} = 1$, $N_{LR} = 1$ e $S_{AC} = 3$. Assim, o valor da métrica de aceitação (AC) foi de 0,666.

O Código 2 apresenta o código fonte gerado de acordo com a sintaxe PHP e a partir do modelo concreto de teste (*Concrete Testing*).

O Código 3 apresenta o modelo de teste baseado em métricas preechido com os valores obtidos pela execução dos casos de teste.

```
class BlogSaaSTest extends PHPUnit_Framework_TestCase{
      public $var availabilityTest;
      public $var reliabilityTest;
      public $var usabilityTest;
      public $var acceptanceTest;

      public function testTemplateFunction() {body;}
      public function testAvailability() {body;}
      public function testReliability() {body;}
      public function testUsability() {body;}
      public function testAcceptance() {body;}
}
class AvailabilityTest extends TestType{
      public $var availabilityMetric;
}
class AvailabilityMetric{
      public $var name;
      public $var description;
      public $var AM;
      public $var date;
      public $var TAD;
      public $var SFT;
      public $var STM;
      public $var TSI;
}
class UsabilityTest{
      public $var usabilityMetric;
}
class UsabilityMetric{
      public $var UM;
      public $var ES;
      public $var EY;
      public $var US;
      public $var CU;
}
class ReliabiityTest{
      public $var reliabilityMetric;
}
class ReliabilityMetric{
      public $var RM;
      public $var CFT;
      public $var CFR;
      public $var SA;
      public $var IAS;
      public $var TAD;
      public $var TBM;
}
class AcceptanceTest{
      public $var acceptanceMetric;
      public $var testPlan;
}
class AcceptanceMetric{
      public $var AC;
}
```

Código 2. Caso de teste de disponibilidade, confiabilidade, usabilidade e aceitação

```
<? xml version ="1.0" encoding ="UTF -8"?>
< MPITMetric:TestingContainer xmi:version =" 2.0 " xmlns:xmi ="http://www.omg.org/XMI" xmlns:xsi ="
http: // www .w3.org /2001/ XMLSchema - instance" xmlns:MPITMetric ="
http://www.example.org/MPITMetric" name =" BlogSaaS " description =" ">

<element xsi:type =" MPITMetric:AvailabilityTest" name ="TestAMBlog" description =" Availability test of
Blog System" comments ="Availability test of Blog System ">
 <testsuite name = " AMBlog ">
 <testcase name = " TestAMBlog ">
 <testdata name = " Message "/>
 <procedures name = " RunAMBlogNewMessage "/>
 </ testcase >
 </ testsuite >
 < availabilitymetric name = " AMetricBloNewMessage " description =" Availability   metric is calculated from
11/01/2015 until 01/30/2016 " AM=" 0.999 " TAD =" 120960.0 " SFT =" 23.0 " STM =" 15.0 " TSI=" 18.0 "/>
</ element >

<element xsi:type =" MPITMetric:ReliabilityTest " name =" TestRMBlog " description ="Reliability test of
Blog system." schedule =""">
 <testsuite name =" RMBlog ">
 <testcase name =" TestRMBlog ">
 <testdata name =" Message "/>
 <procedures name =" RunRMBlogNewMessage "/>
 </ testcase >
 </ testsuite >
 < reliabilitymetric name =" RMetricBlogNewMessage " description ="" RM=" 0.902 "
 CFT =" 0.25 " CFR =" 1.0 " SA=" 0.996 " IAS =" 0.999 " TAD =" 3.0 " TBM =" 6.0 "/>
 </ element >

 <element xsi:type =" MPITMetric:UsabilityTest " name =" TestUMBlog " description ="Usability test of Blog
system.">
 <testsuite name =" UMBlog ">
 <testcase name =" TestUMBlog ">
 <testdata name =" Message "/>
 <procedures name =" RunUMetricBlogNewMessage "/>
 </ testcase >
 </ testsuite >
 < usabilitymetric name =" UMetricBlogNewMessage " description =" Usability Metric for BlogNewMessage "
UM=" 0.584 " ES=" 0.571 " EY="0.6 " US=" 0.581 "/>
 </ element >

 <element xsi:type =" MPITMetric:AcceptanceTest " name =" TestACBlog " description ="Acceptance test of
Blog system.">
 < acceptancemetric AC=" 0.666 "/>
 <testplan nameTestPlan =" PlanTestBlog ">
 < processtest >
 < functionality name =" PostText ">
 ***
 </ functionality >
 </ processtest >
 </ testplan >
 </ element >
 </ MPITMetric:TestingContainer >
```

Código 3. Modelo de teste (PIM de teste) preenchido com os valores obtidos pela execução dos casos de teste

6.2 Síntese

Neste capítulo, o processo para a geração dos casos de teste de disponibilidade, confiabilidade, usabilidade e aceitação foi demonstrado, a fim de ilustrar as funcionalidades do *framework* FCTSaaS proposto e seu protótipo *SaaSTestTool*.

O PIM da aplicação criada no *Wordpress* (um sistema de *blog*) foi exibido e os seus componentes foram detalhados. Os elementos do PIM são referenciados pelo modelo de teste demonstrado no Código 3.

O modelo de teste baseado em métrica, modelo de teste abstrato, modelo de teste concreto e casos de teste foram exibidos. Eles são resultados dos componentes presentes no *framework* FCTSaaS (Figura 17).

Os resultados das métricas inseridas no modelo de teste baseado em métricas demonstram os níveis de disponibilidade, confiabilidade, usabilidade e aceitação da aplicação *open-SaaS* construída no *Wordpress*. Por exemplo, o valor da métrica de usabilidade exibido neste capítulo significa que a aplicação do *Wordpress*, considerando efetividade, eficiência e satisfação do usuário, possui um valor de 0,584 nesse critério, isto é, 58,4% de usabilidade.

7 Conclusões e Trabalhos Futuros

Neste capítulo, a conclusão obtida durante a realização do trabalho, os trabalhos futuros, os objetivos alcançados, as limitações encontradas e as contribuições com a realização do trabalho serão exibidos.

7.1 Conclusão do trabalho

Nesta dissertação, uma abordagem baseada em Engenharia Dirigida por Modelos foi proposta para suportar a geração dos casos de teste para uma aplicação *open-SaaS*. Esta abordagem contem um *framework*, denominada FCTSaaS, projetada para gerar casos de teste de confiabilidade, disponibilidade, usabilidade e aceitação para uma aplicação *open-SaaS*.

Inicialmente, um levantamento bibliográfico [50] [18] [15] [16] [17] [71] [26] [67] foi realizado, e através deste levantamento, os principais testes realizados para nuvens SaaS e/ou *open-SaaS* foram identificados: teste de escalabilidade, teste funcional, teste de desempenho, teste baseado em componentes e teste *multi-tenancy*, ao passo que testes como teste de disponibilidade, confiabilidade, usabilidade e aceitação, orientados pelo uso dos usuários, são ignorados ou relegados pelas equipes de teste e manutenção. Por este motivo, novos tipos de teste para SaaS e *open-SaaS* relacionados aos testes de disponibilidade, confiabilidade, usabilidade e aceitação foram acrescentados.

A fim de medir quantitativamente a qualidade de aplicações open-SaaS, métricas de disponibilidade, confiabilidade, usabilidade e aceitação foram propostas. A métrica de disponibilidade, denominada *Robustness of Service* (ROS), foi proposta em [38] e redefinida a partir de [38] na abordagem proposta nesta dissertação. Uma nova versão da métrica de confiabilidade, também proposta em [38], é criada usando um exemplo de combinação convexa apresentada em [61]. A métrica de usabilidade é definida considerando os conceitos presentes na ISO 9241 [32] [27]. Um questionário e uma fórmula para a métrica são criados para medir a usabilidade. Uma fórmula para a métrica de aceitação foi criada a partir das métricas de disponibilidade, confiabilidade e usabilidade e dos conceitos presentes em [66].

Um protótipo, denominado *SaaSTestTool*, foi criado a fim de provar a efetividade da abordagem. Este protótipo gera códigos de teste para uma aplicação

SaaS de código aberto baseada em PHP, como, por exemplo, uma aplicação construída no *Wordpress* [70]. Este protótipo foi construído dentro do ambiente de programação Eclipse Luna com o auxílio da ferramenta EMF [9] e do plugin ATL para Eclipse [8].

Um exemplo ilustrativo baseado no *Wordpress* é apresentado a fim de mostrar como o protótipo *SaaSTestTool* pode ser usado passo a passo para obter código de teste baseado em PHPUnit [3].

7.2 Objetivos alcançados

Os seguintes objetivos foram alcançados durante o desenvolvimento da abordagem:

- Utilização da abordagem MDE para construção de um *framework* (FCTSaaS) gerador de casos de teste para aplicações SaaS de código aberto;
- Aplicação do *framework* para a geração do caso de teste de confiabilidade, usabilidade, disponibilidade e aceitação para a execução em aplicações SaaS de código aberto. Estes casos de teste são gerados a partir dos modelos de teste, os quais referenciam os elementos presentes no PIM da aplicação *open-SaaS*;
- Proposição de métricas de disponibilidade, confiabilidade, usabilidade e aceitação, que compõem os testes de disponibilidade, confiabilidade, usabilidade e aceitação para aplicações SaaS de código aberto;
- Construção de protótipo que implementa o *framework* a fim de executar os testes definidos na aplicação open-SaaS do *Wordpress* [70] e verificar quantitativamente a qualidade da aplicação através das métricas de disponibilidade, confiabilidade, usabilidade e aceitação. A partir dos valores destas métricas, observa-se que a aplicação *open-SaaS* construída no Wordpress atendeu positivamente aos usuários que os utilizaram durante o período em que esteve ativo (três meses), pois alcançou um nivel de disponibilidade de 99,9% do tempo, um nível de confiabilidade de 90,2%, um nivel de usabilidade de 58,4% (considerando efetividade, eficiência e satisfação do usuário) e, por fim, um nível de aceitação de 66,6%.

- Redução no tempo para geração de casos de teste;
- Aumento de qualidade na geração de casos de teste.

7.3 Contribuições

A principais contribuições realizadas com a abordagem proposta são apresentadas a seguir:

- Proposta de uma abordagem baseada em MDE, aplicada no processo de geração de casos de teste para modelos SaaS de código aberto (*open-SaaS*);
- Geração de casos de teste de disponibilidade, usabilidade, confiabilidade e aceitação para modelos SaaS de código aberto a partir de um *framework* construído para este objetivo;
- Proposta de métricas de disponibilidade, aceitação, usabilidade e confiabilidade como uma forma de análise da qualidade presentes no modelo SaaS de código aberto do Wordpress;
- Proposta de uma metodologia para geração de casos de teste para modelos SaaS de código aberto;
- Construção de protótipo para implementar a solução apresentada no *framework* FCTSaaS para gerar casos de teste para aplicações *open-SaaS;*
- Verificação das funcionalidades presentes no modelos SaaS de código aberto do *Wordpress;*
- Uma vez que são propostas métricas de confiabilidade, usabilidade, disponibilidade e aceitação, os usuários poderão ter um conhecimento maior do modelo SaaS do *Wordpress* concernentes aos testes propostos nesta dissertação.

7.4 Limitações

Algumas limitações foram encontradas durante a criação da abordagem de geração de casos de teste para aplicações SaaS. Elas serão detalhadas a seguir:

- O *framework* FCTSaaS e seu protótipo somente realizam o trabalho de criar definições de transformação com a linguagem ATL. Sendo assim, é necessário estudar a aplicação deste *framework* com outras linguagens de transformação de modelos, como Epsilon [36] e MOF2QVT [56];

- O *framework* FCTSaaS gera somente o esqueleto dos códigos de teste e dos códigos de aplicação, não sendo responsável pela execução dos mesmos. Sendo assim, é necessário implementar o corpo destes códigos e verificar a execução dos mesmos;

- O questionário criado para geração do modelo de teste de usabilidade foi respondido por somente 6 usuários. É necessário que uma quantidade maior de usuários seja envolvida durante o teste de aplicações open-SaaS;

7.5 Trabalhos futuros

Os trabalhos futuros que precisam realizados nesta abordagem são as seguintes:

- Geração de código de aplicação open-SaaS: devido à prioridade dada à geração de códigos de teste como forma de provar a efetividade do *framework* FCTSaaS e do protótipo *SaaSTestTool*, não se realizou a geração do código fonte da aplicação open-SaaS escolhida. Este trabalho é necessário a fim de mostrar na prática que o *framework* e o protótipo são capazes de realizar esta etapa da metodologia para geração de casos de teste;

- Integração com as ferramentas SAMT4MDE e MT4MDE: estas ferramentas definidas em [1] e [41] permitem a geração semi-automática de definição de transformação e de regras de transformção, como explicado no Capítulo 3. Este trabalho permitirá ao protótipo a geração semi-automática das definições de transformação exibidas no Capítulo 4 e de outras definições de transformação que vierem a ser criadas para outros tipos de teste;

- Aplicação da abordagem de geração casos de teste para nuvens IaaS e PaaS: esta abordagem realiza a geração de casos de teste exclusivamente para aplicações open-SaaS, uma vez que o código fonte de um open-SaaS é acessível ao usuário. Entretanto, é necessário extender o campo de aplicação desta abordagem para outros ambientes de Computação em Nuvem como IaaS e PaaS para descobrir até onde ele é efetivo em sua proposta.

Referências Bibliográficas

[1] ALOUINI, W. ; GUEDHAMI, O. ; HAMMOUDI, S. ; GAMMOUDI, M.; LOPES, D. “Semi-Automatic Generation of Transformation Rules in Model Driven Engineering: The Challenge and First Steps.” *International Journal of Software Engineering and Its Applications*, v. 5, p. 73-88, 2011.

[2] Amazon EC2. Amazon Elastic Compute Cloud (Amazon EC2). Disponível em <http://aws.amazon.com/pt/ec2/>. Acesso em 08-07-2015.

[3] BERGAN, S. *PHPUnit*. Link de acesso: <http://www.phpunit.de>. Data de acesso 23-09-2014.

[4] BORGES, H.; de SOUZA, J., SCHULZE, B.; MURY, A., april 2012. “Automatic generation of platforms in cloud computing”. In: *Network Operations and Management Symposium (NOMS)*, 2012 IEEE. pp. 1311 - 1318

[5] BOZGA, M.; FERNANDEZ, J. Cl; GHIRVU, L.; GRAF, S.; KRIMM, J. P. AND MOUNIER, L. “IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In WING, J.M., WOODCOCK, J., and DAVIES, J., editors, *Proceedings of FM’99 (Toulouse, France)*, volume 1708 of *LNCS*, pages 307–327. Springer, September 1999.

[6] CAVARRA, A.; DAVES, J.; THIERRY, J.; MOUNIER, L.; HARTMAN, A.; AND OLVOVSKY, S. “Using UML for Automatic Test Generation”, *In Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[7] CUADRADO, J. S.; GUERRA, E.; AND LARA, J. de. “A component model for model transformations”. *IEEE TSE*, 40(11):1042–1062, 2014.

[8] Eclipse. *Atlas Transformation Language (ATL)*. Disponível em <https://eclipse.org/atl/>. Data de acesso: 13-08-2015.

[9] Eclipse. *Eclipse Modeling Framework (EMF)*. Disponível em http://eclipse.org/modeling/emf/. Acesso em 19-01-2015.

[10] Eclipse. *GMT Project*. Disponível em: <https://eclipse.org/gmt/>. Data de acesso: 13-08-2015.

[11] Eclipse. *Graphical Modelling Framework* (GMF). Acessível em: <http://www.eclipse.org/modeling/gmp/>. Data de acesso: 11-08-2015.

[12] FÉHER, P.; AND LENYGEL, L. “The challenges of a model transformation language”. In *IEEE 19th International Conference and Workshops on Engineering of Computer Based Systems (ECBS) (2012)*, pp. 324–329.

[13] FUENTES, L. AND VALLECILLO, A. 2004. “An Introduction to UML Profiles”. UPGRADE, The European Journal for the Informatics Professional, 5(2) pp. 5–13.

[14] GAO, J., BAI, X., TSAI, W., and UEHARA, T. (2013) “Testing as a Service (TaaS) on Clouds.” *Proceedings of the 7th IEEE International Symposium on Service Oriented System Engineering (SOSE 2013)* (pp. 212-222). IEEE.

[15] GAO, J.; BAI, X.; TSAI, W. T.; and UEHARA, T. “SaaS Testing on Clouds. Issues, Challenges, and Needs.” *SOSE*, 2013.

[16] GAO, J.; BAI, X.; AND TSAI, W. T. “Cloud-Testing – Issues, Challenges, Needs and Practice”. Published by *Software Engineering: An International Journal (SEIJ), Vol. 1,* September, 2011.

[17] GAO, J., MANJULA, K., ROOPA, P., SUMALATHA, E., BAI, X., TSAI, W., UEHARA, T. “A Cloud-Based TaaS Infrastructure with Tools for SaaS Validation, Performance, and Scalability Evaluation.” (2012) *Proceedings of the 4th International Conference on Cloud Computing Technology and Science (ICCCTS 2012)* (pp. 464-471).

[18] GAO, J., PATTABHIRAMAN, P., BAI, X., AND TSAI, W. (2011). “SaaS Performance and Scalability Evaluation in Clouds”. *Proceedings of the 6th IEEE International Symposium on Service Oriented System Engineering (SOSE 2011)* (pp. 61-71). IEEE.

[19] GARAVEL, H., LANG, F., MATEESCU, R., AND SERWE, W. “CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes”, *In Proc. of the 19th International Conference on Computer Aided Verification*, pp. 158-163, 2007.

[20] GEELAN, J. “Twenty one experts define cloud computing”. Disponível em http://virtualization.sys-con.com/node/612375”, 2008. Acesso em 25-05-2015.

[21] GONG, C.; LIU, J.; ZHANG, Q.; CHEN, H.; and GONG, Z. “The Characteristics of Cloud Computing,” in *39th International Conference on Parallel Processing Workshops (ICPPW)*, 2010, pp. 275–279.

[22] GOOGLE. Google App Engine. Disponível em <https://cloud.google.com/appengine/>. Acesso em 08-007-2015.

[23] HOSTGATOR. Disponível em <www.hostgator.com.br>. Acesso em 08-11-2015.

[24] HP Helion Eucalyptus. Disponível em <https://www.eucalyptus.com/eucalyptus-cloud/iaas>. Acesso em 08-07-2015.

[25] IBM HAIFA RESEARCH LABORATORY. *Model Driven Testing Tools – Overview*, September 2003.

[26] INÇKI, K., ARI, I., AND SOZER, H. “A survey of software testing in the cloud,” in *SERE (Companion)*, pp. 18–23, 2012.

[27] ISO, ISO/DIS 9241-11(en): Ergonomics of human-system interaction - Part 11: Usability: Definitions and concepts, Available at: https://www.iso.org/obp/ui/#iso:std:iso:9241:-11:dis:ed-2:v1:en, access date: 05/01/2016.

[28] JAVED, A. Z.; STROOPER, P. A.; AND WATSON, G. N. “Automated generation of test cases using model-driven architecture”, *In Proc. of the ICSE 2nd International Workshop on Automation of Software Test (AST)*, 2007.

[29] JEDEJA, Y.; MODI, K., “Cloud Computing – Concepts, Architecture and Challenges”, Dept. of Computer Engineering & IT, U.V. Patel College of Engineering, Ganpat University, 2012.

[30] JOUAULT., F AND KURTEV, I. “Transforming Models with ATL”.In *MoDELS 2005 Workshops*, LNCS 3844, pages 128-138. Springer, 2006.

[31] KENT, S. Model Driven Engineering. IFM 2002, LNCS 2335, pp. 286 –298, 2002.

[32] KERZAZI, N. and LAVALLEE, M., “Inquiry on usability of two software process modeling systems using ISO/IEC 9241”, *24th Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2011.

[33] KHERAJANI, M.; and SHRIVASTAVA, A. “Impact of Cloud Computing Platform Based on Several Software Engineering Paradigm”. In *International Journal of Advanced Computer Research* (IJACR), pp. 67-71, 2011.

[34] KING, T. M.; GANTI, A.; AND FROSLIE, D. “Enabling Automated Integration Testing of Cloud Application Services in Virtualized Environments.” In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON ’11, pages 120–132, Riverton, NJ, USA, 2011. IBM Corp.

[35] KLEPPE, A., WARMER, J., AND BAST, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*, 1st. ed. Addison-Wesley, August 2003.

[36] KOLOVOS, D.; PAIGE, R.; AND POLACK, F. “The Epsilon Transformation Language.” In *ICMT 2008*, LNCS 5063, pages 46-60. Springer, 2008.

[37] LAWLEY, M. J. AND STEEL, J. “Practical Declarative Model Transformation With Tefkat,” *In Satellite Events at the MoDELS 2005 Conference*, *LNCS Vol. 3844*, 2005.

[38] LEE, J.Y., LEE, J.W., CHEUN, D.W., AND KIM, S.D.: “A Quality Model for Evaluating Software-as-a-Service in Cloud Computing”. In: *7th ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2009),Washington, DC, USA, IEEE Computer Society (2009)*, 261–266.

[39] LIMA, B., SOUSA J., LOPES, D., “Using MDA to Support Hypermedia Document Sharing”. *IEEE International Conference on Software Engineering Advances (ICSEA 2007)*, French Riviera, France, 2007.

[40] LIU, J.; HE, K.; LI, B.; HE, C.; AND LIANG, P. “A Transformation Definition Metamodel for Model Transformation”. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC’05)*, *pp. v-xiv, Las Vegas, Nevada, USA*, 2005.

[41] LOPES, D.; HAMMOUDI, S.; BÉZIVIN, J.; AND JOUAULT, F. “Generating Transformation Definition from Mapping Specification: Application to Web Service Platform”. *The 17th Conference on Advanced Information System Engineering (CaiSE’05)*, LNCS 3520 (June 2005), 309-325.

[42] LOPES, D., HAMMOUDI, S., BÉZIVIN, J. and JOUAULT, F. “Mapping Specification in MDA: from Theory to Practice, In: *Proceedings of the 1th International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA'2005)*. Springer-Verlag, 2005, s. 253-264.

[43] MATHUR, P; and NISHCHAL, N. "Cloud Computing: New challenge to the entire computer industry", 2010 *1st International Conference on Parallel, Distributed and Grid Computing (PDGC - 2010)*.

[44] MELL, P.; and GRANCE, J. “The NIST definition of Cloud Computing.” National Institute of Standard and Technology – NIST, Tech. Rep., 2011.

[45] MELO, Simone A. B de; LOPES, D.; ABDELOUAHAB, Z.: “Developing Medical Information System with MDA and Web Services”. *Software Engineering Research and Practice 2006*: pp. 562-568.

[46] MIRNIG, A. G.; MESCHTSCHERJAKOV, A. et al. “A Formal Analysis of the ISO 9241-210: Definition of User Experience”. *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, 2015.

[47] MOHAMAD, R. P.; KOLOVOS, D. AND PAIGE, R. “Modeling Workloads, SLAs and their Violations in Cloud Computing” in *Proceedings of the Fourth York Doctoral Symposium on Computer Science*, *Department of Computer Science, The University of York, UK*, October 2011, pp. 71-76.

[48] MUSSA, M.; OUCHANI, S.; AL SAMMANE, W.; AND HAMOU-LHADJ, A. “A survey of model-driven testing techniques,” in *9th Internacional Conference on Quality Software*, 2009, pp. 167-172.

[49] OLDEVIK, J. *MOFScript User Guide*. *Version 0.6 (MOFScript v 1.1.11)*, 2nd ed. Eclipse Project, http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf, November 2006.

[50] OLIVEIRA, J. B. “Uma Abordagem baseada em Engenharia Dirigida por Modelos para suportar o Teste de Sistemas de Software na Plataforma de Computação em Nuvem”. 2012. 192 f. Dissertação (Mestrado em Engenharia de Eletricidade) – Universidade Federal do Maranhão, São Luís. 2012.

[51] OMG. *Meta Object Facility (MOF) specification – version 1.4, formal/01-11-02.*

[52] OMG (2000). *Unified Process Model* (UPM) – document number ad/2000-05-05.

[53] OMG (2003). *Common Warehouse Metamodel* (CWM) Specification – document number formal/03-03-02.

[54] OMG (2014).OMG Meta Object Facility (MOF) Core Specification version 2.4.2 – document number formal/2014-04-03.

[55] OMG (2015). *Unified Modeling Language* (UML) v2.5 – document number formal/15-03-01.

[56] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification – Version 1.2,* February 2015. Available at: http://www.omg.org/spec/QVT/1.2/formal-15-02-01.pdf.

[57] OMG. XML Metadata Interchange (XMI). September 2015. Disponível em < http://www.omg.org/spec/XMI/#sts=XML%20Metadata%20Interchange%20%28XMI%29%C2%A0>.

[58] PARVEEN, T.; AND TILLEY, S. “When to Migrate Software Testing to the Cloud?”. *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW),* pp. 424-427, 2010.

[59] PFLEEGER, S. L.; AND ATLEE, J. M. “Software Engineering: Theory and Practice”, 4th edition, Prentice Hall, 2010.

[60] SALESFORCE. Salesforce. Disponível em <https://www.salesforce.com/saas/>. Acesso em 08-07-2015.

[61] SHARMA, A. K. “Text Book of Elementary Statistics”, 1st edition, Discovery Publishing House, 2005.

[62] SILVA, M.; LOPES, D.; AND ABDELOUAHAB, Z. (2006). “A Remote IDS based on Multi-Agent Systems, Web Services and MDA”. *Proceedings of the IEEE International Conference on Software Engineering Advances (ICSEA)*, pp. 64-69, 2006.

[63] SOMMERVILLE, I. Software engineering (6th ed.). Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001.

[64] SOUSA, H.; LOPES, D.; ABDELOUAHAB, Z.; CLARO, D. AND HAMMOUDI, S. “An Approach for Model Driven Testing: framework, metamodels and tools”. Computer Systems Science and Engineering, v. 26, p. 41-55, 2011.

[65] Super Monitoring. *Super Monitoring* (Online). Disponível em <https://wordpress.org/plugins/website-monitoring>. Data de acesso: 11/04/2014.

[66] SYSTEMS AND SOFTWARE ENGINEERING - VOCABULARY, in ISO/IEC/IEEE 24765:2010E , pp.1-418, 2010.

[67] TSAI, W; HUANG, Y; SHAO, Q; AND BARRETT, M. “Testing the scalability of SaaS applications,” in *IEEE RTSOAA ’11*, 2011.

[68] W3C: Web Services Description Language (WSDL) 1.1. (2001)

[69] WIMMER, M.; STROMMER, M.; KARGL, H.; AND KRAMLER, G.. “Towards model transformation generation by-example”. In: *Proceedings of 40th HICSS 2007, CD-ROM / Abstracts Proceedings, 3–6 Jan. 2007*, Waikoloa, Big Island, HI, USA.

[70] Wordpress. Disponível em http://wordpress.org/. Acesso em 14-01- 2014.

[71] WU, CS and LEE, YT. “Automatic SaaS Test Cases Generation based on SOA in the Cloud Service”. (2012) *Proceedings of the 4th International Conference on Cloud Computing Technology and Science (ICCCTS 2012)* (pp. 349-354). IEEE.

[72] ZHENLONG, P.; ZHONGHUI, O. Y.; YOULAN, H. “The Application and Development of Software Testing in Cloud Computing Environment”. In *International Conference on Computer Science and Service System*, 2012, pp.450-454.

8 Anexos

8.1 Anexo A - Regras de transformação de *Metamodel for Platform Independent Testing* baseado em Métricas (MPIT baseado em Métricas) para *Metamodel for Abstract Cloud Testing for SaaS* (MACT4SaaS)

```
-- @path
SaaSTestingMetamodel=/SaaSTestingMetamodel(1)/model/SaaSTestingMetamodel.ec
ore
-- @path
AbstractCloudTestingSaaS=/AbstractCloudTestMetamodel4SaaS/AbstractCloudTest
Metamodel4SaaS.ecore

module SaasTestingMetamodel2AbstractCloudTestingSaaS;
create OUT : AbstractCloudTestingSaaS from IN : SaaSTestingMetamodel;

rule TestType2TestType{
      from testtype: SaaSTestingMetamodel!TestType(testtype.oclIsTypeOf
                  (SaaSTestingMetamodel!TestType))
      to testtype1: AbstractCloudTestingSaaS!TestType
      (
            name <- testtype.name,
            description <- testtype.description,
            comments <- testtype.comments,
            schedule <- testtype.schedule,
            testReport <- testtype.testReport,
            testSuite <- testtype.testSuite,
            testClass <- testtype.testClass
      )
}

rule AvailabilityTest2AvailabilityTest{
      from availabilitytest:
            SaaSTestingMetamodel!AvailabilityTest(availabilitytest.
            oclIsTypeOf(SaaSTestingMetamodel!AvailabilityTest))
      to availabilitytest1: AbstractCloudTestingSaaS!AvailabilityTest
      (
            availabilityMetric <- availabilitytest.availabilityMetric
      )
}

rule ReliabilityTest2ReliabilityTest{
      from reliabilitytest:
            SaaSTestingMetamodel!ReliabilityTest(reliabilitytest.
            oclIsTypeof(SaaSTestingMetamodel!ReliabilityTest))
      to reliabilitytest1: AbstractCloudTestingSaaS!ReliabilityTest
      (
            reliabilityMetric <- reliabilitytest.reliabilityMetric
      )
}

rule UsabilityTest2UsabilityTest{
      from usabilitytest:
            SaaSTestingMetamodel!UsabilityTest(usabilitytest.
            oclIsTypeOf(SaaSTestingMetamodel!UsabilityTest))
      to usabilitytest1: AbstractCloudTestingSaaS!UsabilityTest
```

```
	(
		usabilityMetric <- usabilitytest.usabilityMetric
	)
}

rule AcceptanceTest2AcceptanceTest{
	from acceptancetest:
		SaaSTestingMetamodel!AcceptanceTest(acceptancetest.
		oclIsTypeOf(SaaSTestingMetamodel!AcceptanceTest))
	to acceptancetest1: AbstractCloudTestingSaaS!AcceptanceTest
	(
		acceptanceMetric <- acceptancetest.acceptanceMetric,
		testPlan <- acceptancetest.testPlan
	)
}

rule AvailabilityMetric2AvailabilityMetric{
	from availabilitymetric: SaaSTestingMetamodel!AvailabilityMetric
	to availabilitymetric1: AbstractCloudTestingSaaS!AvailabilityMetric
	(
		name <- availabilitymetric.name,
		description <- availabilitymetric.description,
		ROS <- availabilitymetric.ROS,
		date <- availabilitymetric.date,
		TAD <- availabilitymetric.TAD,
		SFT <- availabilitymetric.SFT,
		STM <- availabilitymetric.STM,
		TSI <- availabilitymetric.TSI,
		availabilityTest <- availabilitymetric.availabilityTest
	)
}

rule ReliabilityMetric2ReliabilityMetric{
	from reliabilitymetric: SaaSTestingMetamodel!ReliabilityMetric
	to reliabilitymetric1: AbstractCloudTestingSaaS!ReliabilityMetric
	(
		name <- reliabilitymetric.name,
		description <- reliabilitymetric.description,
		RM <- reliabilitymetric.RM,
		date <- reliabilitymetric.date,
		CFT <- reliabilitymetric.CFT,
		CFR <- reliabilitymetric.CFR,
		SA <- reliabilitymetric.SA,
		IAS <- reliabilitymetric.IAS,
		TAD <- reliabilitymetric.TAD,
		TBM <- reliabilitymetric.TBM
		reliabilityTest <- reliabilitymetric.reliabilityTest
	)
}

rule UsabilityMetric2UsabilityMetric{
	from usabilitymetric: SaaSTestingMetamodel!UsabilityMetric
	to usabilitymetric1: AbstractCloudTestingSaaS!UsabilityMetric
	(
		name <- usabilitymetric.name,
		description <- usabilitymetric.description,
		UM <- usabilitymetric.value,
		date <- usabilitymetric.date,
		ES <- usabilitymetric.ES,
		EY <- usabilitymetric.EY,
		US <- usabilitymetric.US,
```

```
            CU <- usabilitymetric.CU
            usabilityTest <- usabilitymetric.usabilityTest
      )
}

rule AcceptanceMetric2AcceptanceMetric{
      from acceptancemetric: SaaSTestingMetamodel!AcceptanceMetric
      to acceptancemetric1: AbstractCloudTestingSaaS!AcceptanceMetric
      (
            name <- acceptancemetric.name,
            description <- acceptancemetric.description,
            AC <- acceptancemetric.AC,
            date <- acceptancemetric.date,
            acceptanceTest <- acceptancemetric.acceptanceTest
      )
}

rule TestSuite2TestSuite{
      from testsuite: SaaSTestingMetamodel!TestSuite
      to testsuite1: AbstractCloudTestingSaaS!TestSuite
      (
            name <- testsuite.name,
            description <- testsuite.description,
            testType <- testsuite.testType,
            testCase <- testsuite.testCase
      )
}

rule TestCase2TestCase{
      from testcase: SaaSTestingMetamodel!TestCase
      to testcase1: AbstractCloudTestingSaaS!TestCase
      (
            name <- testcase.name,
            description <- testcase.description,
            testSuite <- testcase.testSuite,
            procedures <- testcase.procedures,
            testData <- testcase.testData
      )
}

rule Procedures2Procedures{
      from procedures: SaaSTestingMetamodel!Procedure
      to procedures1: AbstractCloudTestingSaaS!Procedure
      (
            name <- procedures.name,
            description <- procedures.description,
            testCase <- procedures.testCase
      )
}

rule TestData2TestData{
      from testdata: SaaSTestingMetamodel!TestData
      to testdata1: AbstractCloudTestingSaaS!TestData
      (
            name <- testdata.name,
            description <- testdata.description,
            testCase <- testdata.testCase
      )
}
```

```
rule TestClass2TestClass{
      from testclass: SaaSTestingMetamodel!TestClass
      to testclass1: AbstractCloudTestingSaaS!TestClass
      (
            testType <- testclass.testType,
            method <- testclass.method,
            attribute <- testclass.attribute
      )
}

rule Method2Method{
      from method: SaaSTestingMetamodel!Method
      to method1: AbstractCloudTestingSaaS!Method
      (
            name <- method.name,
            description <- method.description,
            isStatic <- method.isStatic,
            testClass <- method.testClass,
            parameter <- method.parameter
      )
}

rule Attribute2Attribute{
      from attribute: SaaSTestingMetamodel!Attribute
      to attribute1: AbstractCloudTestingSaaS!Attribute
      (
            name <- attribute.name,
            description <- attribute.description,
            testClass <- attribute.testClass,
            attributeType <- attribute.attributeType
      )
}

rule Parameter2Parameter{
      from parameter: SaaSTestingMetamodel!Parameter
      to parameter1: AbstractCloudTestingSaaS!Parameter
      (
            name <- parameter.name,
            parameterType <- parameter.parameterType,
            isReturn <- parameter.isReturn,
            method <- parameter.method
      )
}

rule AttributeType2AttributeType{
      from attributetype: SaaSTestingMetamodel!AttributeType
      to attributetype1: AbstractCloudTestingSaaS!AttributeType
      (
            typeAttribute <- attributetype.typeAttribute,
      )
}

rule TestPlan2TestPlan{
    from testplan: SaaSTestingMetamodel!TestPlan
    to testplan1: AbstractCloudTestingSaaS!TestPlan
    (
      acceptanceTest <- testplan.acceptanceTest
    )
}
```

```
rule TestingContainer2Container{
    from testingcontainer: SaaSTestingMetamodel!TestingContainer
    to container: AbstractCloudTestingSaaS!Container
    (
     name <- testingcontainer.name,
     description <- testingcontainer.description,
     element <- testingcontainer.element
    )
}
```

8.2 Anexo B - Regras de transformação de *Metamodel for Abstract Cloud Testing for SaaS* (MACT4SaaS) para Metamodelo do PHP

```
-- @path
AbstractCloudTestMetamodel4SaaS=/AbstractCloudTestMetamodel4SaaS/AbstractCl
oudTestMetamodel4SaaS.ecore
-- @path PHPMetamodel=/PHPMetamodel/model/PHPMetamodel.ecore

module AbstractCloudTestingMetamodel2PHPMetamodel;
create OUT : PHPMetamodel from IN : AbstractCloudTestMetamodel4SaaS;

rule TestClass2Class{
      from testclass:
            AbstractCloudTestMetamodel4SaaS!TestClass(testclass.oclIsTypeOf
            (AbstractCloudTestMetamodel4SaaS!TestClass))
      to testclass1: PHPMetamodel!Class
      (
            name <- testclass.name,
            visibility <- 'public',
            isAbstract <- 'false',
            isFinal <- 'false',
            isClass <- 'true',
            function <- testclass.method,
            variable <- testclass.attribute
      )
}

rule Method2Function{
      from method:
            AbstractCloudTestMetamodel4SaaS!Method(method.oclIsTypeOf
            (AbstractCloudTestMetamodel4SaaS!Method))
      to funtion: PHPMetamodel!Function
      (
            name <- method.name,
            visibility <- 'public',
            isAbstract <- 'false',
            isFinal <- 'false',
            isMagic <- 'false',
            body <- '{body;}',
            classifier <- method.testClass,
            parameter <- method.parameter
      )
}

rule Attribute2Variable{
      from attribute:
            AbstractCloudTestMetamodel4SaaS!Attribute(attribute.oclIsTypeOf
            (AbstractCloudTestMetamodel4SaaS!Attribute))
      to variable: PHPMetamodel!Variable
      (
            name <- attribute.name,
            visibility <- 'public',
            isPredefined <- 'false',
            classifier <- attribute.testClass,
            type <- attribute.attributeType
      )
}
```

```
rule AttributeType2Expression{
      from attributetype:
            AbstractCloudTestMetamodel4SaaS!AttributeType(attributetype.ocl
            IsTypeOf(AbstractCloudTestMetamodel4SaaS!AttributeType))
      to expression: PHPMetamodel!Expression
      (
            statement <- attributetype.typeAttribute
      )
}

rule Parameter2Parameter{
      from parameter:
            AbstractCloudTestMetamodel4SaaS!Parameter(parameter.oclIsTypeOf
            (AbstractCloudTestMetamodel4SaaS!Parameter))
      to parameter1: PHPMetamodel!Parameter
      (
            name <- parameter.name,
            isResult <- 'false',
            isVoid <- 'false',
            method <- parameter.method,
            type <- parameter.parameterType
      )
}
```

8.3 Anexo C – Regras de transformação de Metamodelo do PHP para Código

```
 -- PHP2SourceCode_query . atl
 query PHPUnit2Code = pHPmetamodel ! Element . allInstances () ->
          select (e | e. oclIsTypeOf (pHPmetamodel!Class) or
                      e. oclIsTypeOf (pHPmetamodel!Interface)) ->
          collect (x | x. toString (). writeTo ('C:/temp/'+ x.
name + '.php '));
uses library_PHPUnit2Code;
-- libraryPHPUnit2Code.atl
library libraryPHPUnit2Code;

helper context pHPmetamodel!Interface def: toString () : String =
       'interface ' + self . name + '{\n ' + self . function -> iterate (i;
       acc : String = ' ' | acc + i. toString ()) + '\n' + '}';

helper context pHPmetamodel!Class def: toString () : String =
      'class ' + self.name + self.getExtends () +
      self.getImplements()+ '{\n ' +
      self.variable->iterate(i; acc : String = ' ' | acc + i. toString ())+
          '\n' + self.function->iterate(i; acc : String = ' ' | acc +
          i.toString ()) + '\n' + '}';

helper context pHPmetamode!Class def: getImplements():String =
        self.implements->iterate(i; acc : String = '' | acc +
        if acc = '' then ' implements ' else ', ' endif + i. name );

helper context pHPmetamodel!Class def:getExtends():String =
        if (self.extends.oclIsTypeOf(pHPmetamodel!Class )) then
        'extends' + self.extends.name else '' endif ;

helper context pHPmetamodel!Variable def: toString(): String =
       '\t' + if(self.visibility = #public) then 'public ' else 'private'
       endif + ' $ var ' + self.name + ';\n';

helper context pHPmetamodel!Function def: toString():String =
       '\t' + if(self.visibility = #public) then 'public' else 'private'
       endif + ' function ' + self . name + '(' +
       self.parameter->iterate(i; acc : String = '' | acc +
       if acc = '' then '' else ', ' endif + i.toString()) + ') {' +
       'body ;'+ '}'+ '\n';

helper context pHPmetamodel!Parameter def: toString() : String =
       ' $ var ' + self . name ;
```

8.4 Anexo D – Comprovante de envio da pesquisa para o Comitê de Ética em Pesquisa da Universidade Federal do Maranhão (CEP/UFMA)

Como esta pesquisa envolveu a participação de seres humanos durante o uso da aplicação *open-SaaS* do *Wordpress* e durante a resolução do questionário na obtenção do valor da métrica de usabilidade (Seção 4.2.3), ela foi enviada para o Comitê de Ética em Pesquisa da universidade Federal do Maranhão (CEP/UFMA) para apreciação ética. Através da Plataforma Brasil, o documento a seguir foi gerado com o respectivo número de protocolo CAAE nº 52520515.4.0000.5087.

UNIVERSIDADE FEDERAL DO MARANHÃO UFMA

COMPROVANTE DE ENVIO DO PROJETO

DADOS DO PROJETO DE PESQUISA

Título da Pesquisa: UMA ABORDAGEM BASEADA EM ENGENHARIA DIRIGIDA POR MODELOS E COMPUTAÇÃO EM NUVEM PARA SUPORTAR O TESTE DE MODELOS SAAS DE CÓDIGO ABERTO

Pesquisador: GERSON LOBATO RABELO FILHO

Versão: 1

CAAE: 52520515.4.0000.5087

Instituição Proponente: Universidade Federal do Maranhão

DADOS DO COMPROVANTE

Número do Comprovante: 002467/2016

Patrocionador Principal: Financiamento Próprio

Endereço: Avenida dos Portugueses, 1966 CEB Velho
Bairro: Bloco C,Sala 7, Comitè de Ética **CEP:** 65.080-040
UF: MA **Município:** SAO LUIS
Telefone: (98)3272-8708 **Fax:** (98)3272-8708 **E-mail:** cepufma@ufma.br

8.5 Anexo E – Formulário de usabilidade preenchido por seis usuário durante teste da aplicação SaaS (simulação)

	Questão	Usuário, resposta e valores correspondentes às respostas											
		Usuário 1		Usuário 2		Usuário 3		Usuário 4		Usuário 5		Usuário 6	
		Resp.	Valor	Resp.	Valor	Resp.	Valor	Resp.	Valor	Resp.	Valor	Resp.	Valor
Perfil do usuário	1	B	0,8	D	0,4	B	0,8	C	0,6	A	1	C	0,6
	2	C	0,6	C	0,6	C	0,6	B	0,8	B	0,8	B	0,8
	3	B	0,8	B	0,8	C	0,6	B	0,8	B	0,8	C	0,6
Efetividade	1	B	0,8	B	0,8	B	0,8	B	0,8	A	1	B	0,8
	2	A	1	C	0,6	B	0,8	B	0,8	B	0,8	A	1
	3	A	1	C	0,6	C	0,6	C	0,6	C	0,6	A	1
Eficiência	1	A	1	B	0,8	B	0,8	B	0,8	B	0,8	A	1
	2	B	0,8	B	0,8	A	1	A	1	B	0,8	B	0,8
	3	B	0,8	B	0,8	B	0,8	B	0,8	B	0,8	B	0,8
Satisfação do usuário	1	A	1	B	0,8	A	1	B	0,8	B	0,8	B	0,8
	2	B	0,8	B	0,8	A	1	B	0,8	C	0,6	A	1
	3	A	1	C	0,6	A	1	C	0,6	C	0,6	B	0,8

upi	0,73		0,6		0,67		0,73		0,87		0,67

ESi	0,68		0,4		0,49		0,54		0,69		0,62
EYi	0,64		0,48		0,58		0,64		0,69		0,58
USi	0,68		0,44		0,67		0,54		0,58		0,58

ES	0,57
EY	0,6
US	0,58