PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CONJUNTO DE CARACTERÍSTICAS PARA TESTE DE DESEMPENHO: UMA VISÃO A PARTIR DE MODELOS

MAICON BERNARDINO DA SILVEIRA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Avelino Francisco Zorzo

Porto Alegre
2012

S463p Silveira, Maicon Bernardino da
Conjunto de características para teste de desempenho : uma visão a partir de modelos / Maicon Bernardino da Silveira. – Porto Alegre, 2012.
105 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Avelino Francisco Zorzo.

1. Informática. 2. Engenharia de Software. 3. Software – Avaliação. I. Zorzo, Avelino Francisco. II. Título.

CDD 005.1

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Conjunto de Características para Teste de Desempenho: Uma Visão a Partir de Modelos***", apresentada por Maicon Bernardino da Silveira como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 08/03/2012 pela Comissão Examinadora:

Prof. Dr. Avelino Francisco Zorzo - Orientador — PPGCC/PUCRS

Prof. Dr. Felipe Rech Meneguzzi - — PPGCC/PUCRS

Prof. Dr. Adenilso da Silva Simão - — USP

Homologada em 10/07/2012, conforme Ata No. 014/2012 pela Comissão Coordenadora.

Prof. Dr. Paulo Henrique Lemelle Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

aos meus pais Gerci e Ceroni (*in memoriam*)
aos meus irmãos Romaldo, Euzébio, Jeferson e Cledissom
às minhas irmãs Roseli, Joceli e Sheila
e em especial, à minha amada Rosangela

AGRADECIMENTOS

À Deus. A força da fé me manteve firme e perseverante.

Agradeço o apoio financeiro obtido durante o período, oportunizado pela DELL, em forma de bolsa taxas. Além da oportunidade de desenvolver a minha pesquisa como bolsista vinculado ao Projeto de Desenvolvimento em Tecnologia da Informação (PDTI) no convênio DELL/PUCRS, como integrante do projeto "Center of Competence in Performance Testing".

Aos colegas do CePES e aos membros do grupo de pesquisa "CoC Perf" por terem compartilhado momentos descontraídos e também estressantes durante o período do mestrado. Ao colega Elder de Macedo Rodrigues um agradecimento especial por toda a ajuda e colaboração prestada.

Ao meu orientador, professor Avelino Francisco Zorzo, por todos os ensinamentos, críticas e conselhos e, principalmente, pela confiança. Saiba que os desafios lançados colaboraram muito com o meu crescimento, tanto pessoal quanto profissional. Espero que possamos continuar trabalhando juntos por muito tempo.

Agradeço a ajuda do professor Fernando Luís Dotti pelas considerações efetuadas no plano de estudo e pesquisa e no seminário de andamento.

Gostaria de agradecer o apoio do professor Flávio Moreira de Oliveira pela oportunidade profissional e contribuições na publicação de artigos.

Agradecimento ao projeto PROCAD, financiado pela CAPES, oportunizando a missão de estudos realizada na Universidade de São Paulo - USP em São Carlos/SP. Um agradecimento também a todos os integrantes do LabES, em especial ao professor Adenilso da Silva Simão pela acolhida durante a visita.

MUITO OBRIGADO ao "grande" e mais que amigo: João Batista Mossmann pelo incentivo, motivação, conselhos e ajudas. Além das cervejadas e churrascadas para manter a vida social em dia. E aos demais amigos que contribuíram de alguma forma para a realização deste trabalho.

À minha família, pelo apoio, carinho e confiança dado em todos os momentos de minha vida e, por compreenderem minha ausência nos encontros familiares durante este período.

Um agradecimento especial a minha querida mãe, a "Véia". Obrigado, do fundo do coração por sempre acreditar e confiar no meu potencial. Eu, o "Maiquinho", dedico este trabalho a você, pelo carinho e por ter me ensinado os valores e caráter que tenho hoje.

Agradecimento incondicional a minha futura esposa, Rosangela dos Santos Cabrera pela paciência e por tudo que passamos juntos neste período de crescimento profissional. Sem a tua ajuda e apoio, com certeza não teria conseguido.

CONJUNTO DE CARACTERÍSTICAS PARA TESTE DE DESEMPENHO: UMA VISÃO A PARTIR DE MODELOS

RESUMO

O processo de teste de *software* possui um custo elevado se comparado com as demais etapas de desenvolvimento de *software*. A automação do teste de *software* por meio do reuso de artefatos de *software*, *e.g.*, modelos, tem sido uma boa alternativa para mitigar estes custos, reduzindo o tempo de geração e execução dos casos de teste, tornando mais eficiente e eficaz este processo. Nesse sentido, a abordagem de Teste Baseado em Modelos (*Model Based Testing - MBT*) está crescendo na Engenharia de *Software*. MBT é uma técnica que consiste na geração automática dos artefatos de teste com base em informações extraídas dos modelos de *software*, que inclui também a especificação dos aspectos que serão testados. O presente trabalho tem por objetivo estudar técnicas e metodologias para MBT e avaliar as características dos diferentes modelos aplicados em MBT. A principal contribuição deste estudo é a análise das características dos modelos que são utilizados no teste de desempenho em aplicações *web*. Em outra perspectiva, a pesquisa norteia a investigação de modelos e métodos para geração de sequências de teste. Assim, apresenta a abordagem de geração de casos de teste baseado em MEFs (Máquinas de Estados Finitos), conceituando MEF e ainda, descreve o processo de geração das sequências de teste através do método HSI (*Harmonized State Identification*). Por outro lado, a implementação de um *plug-in* para a ferramenta PLeTs, que implementa uma linha de produtos de *software*, baseado no modelo UML SPT, que interpreta o conjunto de características para teste de desempenho desenvolvido, é apresentado em um estudo de caso real. Com esta análise, definem-se quais características devem estar presentes no modelo para a modelagem das interações do usuário com o SUT (*System Under Test*), buscando o maior reuso deste modelo ao longo do ciclo de vida de desenvolvimento do *software*.

Palavras-chave: Teste Baseado em Modelos; Teste de Desempenho; Modelos Formais; Máquina de Estados Finitos; UML; Teste de *Software*; Linha de Produto de *Software*.

SET OF FEATURES FOR PERFORMANCE TESTING: A PERSPECTIVE FROM MODELS

ABSTRACT

Software testing process has a high cost when compared to the other stages of software development. Automation of software testing through reuse of software artifacts (e.g. models) is a good alternative for mitigating these costs and making the process much more efficient and effective. In this sense, the Model-Based Testing (MBT) approach has increased in Software Engineering. MBT is a technique of automatic generation of testing artifacts based on software models, which also includes the specification of the aspects to be tested. This work aims to study techniques and methodologies for MBT and to evaluate the features of the different models applied in MBT. The main contribution of this study is to analyze the features of the models that are used in performance testing of web applications. In another perspective, it presents an approach for generating test cases based on FSM (Finite State Machines). Furthermore, it describes the process of generating the test cases using the HSI method (Harmonized State Identification). Besides, the implementation of a plug-in in the $\mathrm{PL}e\mathrm{T}s$ tool based on UML SPT profile, which interprets the set of features designed for performance testing, is presented in a real case study. Our work, defines which features must be present in a model when modeling user interactions with a SUT (System Under Test). This will allow the increasing in automation and in reusing testing artifacts throughout the systems development life cycle. Furthermore, our solution is generated automatically by a Software Product Line (SPL).

Keywords: Model-Based Testing; Performance Testing; Formal Models; Finite State Machines; UML; Software Testing; Software Product Line.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

CTMC	*Continuous Time Markov Chains*
DTMC	*Discrete Time Markov Chains*
FSM	*Finite Machine State*
HOL	*Higher Order Logic*
HSI	*Harmonized State Identification*
IEEE	*Institute of Electrical and Electronics Engineers*
MARTE	*Modeling and Analysis of Real-Time and Embedded Systems*
MBT	*Model Based Testing*
MC	*Markov Chain*
MDD	*Model-Driven Development*
MEF	Máquina de Estados Finitos
MTTF	*Mean Time To Failure*
MTTR	*Mean Time To Recovery*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
PN	*Petri Nets*
PL*e*T*s*	*Product Line Testing Tool*
SAN	*Stochastic Automata Network*
Skills	*Workforce Planning: Skill Management Prototype Tool*
SPE	*Software Performance Engineering*
SPL	*Software Product Line*
SPT	*Schedulability, Performance and Time*
SUT	*System Under Test*
TDD	*Test Driven Development*
TPN	*Timed Petri Nets*
TPS	*Transações por Segundo*
TTCN-3	*Testing and Test Control Notation version 3*
UCML	*User Community Modeling Language*
UML	*Unified Modeling Language*
UTP	*UML Testing Profile*
VV&T	Validação, Verificação e Teste

SUMÁRIO

1. INTRODUÇÃO

Hoje em dia, cada vez mais pessoas e empresas usam programas de computador para automatizar suas atividades, delegando aos sistemas a realização de tarefas complexas. Esse uso generalizado de sistemas de computadores também tem aumentado o número de falhas residuais de *software* que geram defeitos para os usuários [97] [7]. Portanto, é importante que durante o desenvolvimento de um sistema sejam aplicadas diferentes técnicas para garantir que seja fornecido um serviço confiável.

A capacidade de oferecer um serviço que possa ser, justificadamente, confiável é conhecido como dependabilidade [97]. Os principais atributos que integram a dependabilidade são a confiabilidade, disponibilidade, segurança, confidencialidade, integridade e manutenabilidade. De acordo com a taxonomia apresentada em [97], a confiabilidade do sistema pode ser alcançada por quatro técnicas: 1) Prevenção de Falhas (*Fault Prevention*) - prevenir a ocorrência ou introdução de falhas; 2) Tolerância a Falhas (*Fault Tolerance*) - evitar defeitos em serviços na presença de falhas; 3) Remoção de Falhas (*Fault Removal*) - reduzir a quantidade e a gravidade das falhas, e; 4) Previsão de Falhas (*Fault Forecasting*) - para estimar o número de falhas, sua incidência futura, e as prováveis consequências destas falhas [97].

Vários trabalhos que proveem a dependabilidade do sistema através da tolerância a falhas, a prevenção de falhas e a previsão de falhas estão presentes na literatura, *e.g.*, [8] [9] [10] [11] [12].

Embora as quatro técnicas sejam usadas para atingir dependabilidade de *software*, a técnica mais utilizada em todas as áreas de desenvolvimento de *software* na indústria é a remoção de falhas, por meio do teste de *software*. Teste de *software* é um processo que visa, intencionalmente, encontrar o maior número de defeitos de um programa durante sua execução [13], ou que tem atividades para validar os requisitos de um programa, determinando se os resultados esperados são atingidos [14]. Entretanto, devido à evolução dos sistemas e ao aumento de suas funcionalidades, os mesmos estão se tornando tão complexos que testá-los é uma tarefa difícil [15]. Portanto, é necessário implementar um processo de teste para mitigar a execução de testes ao produto de *software*. Este processo visa reduzir o impacto do custo financeiro e otimizar a eficácia dos testes, além de melhorar a qualidade do produto de *software* [7].

Uma das técnicas que auxiliam o processo de teste de *software* é o Teste Baseado em Modelos (*Model-Based Testing - MBT*) [1]. Esta técnica consiste na geração dos casos de teste e/ou *scripts* de teste baseado nos modelos do *software*. Devido a isso, os modelos incluem em sua especificação as características do *software* que serão testadas [1]. Além disso, o uso de MBT apresenta várias vantagens, como por exemplo, a probabilidade de redução da má interpretação dos requisitos do sistema por um engenheiro de teste e/ou a redução do tempo do teste [7].

Por esses motivos, diversos trabalhos [16] [17] [18] [19] [20] têm proposto ferramentas que utilizam essa técnica para automatizar o processo de teste. No entanto, em nenhum deles é proposto uma maneira de utilizar o conhecimento e artefatos gerados no desenvolvimento de uma ferramenta, a fim de gerar novas ferramentas a partir de variações dos artefatos já desenvolvidos. Assim, uma das

abordagens para obter o reuso de artefatos é a implementação de uma Linha de Produto de *Software* (*Software Product Line - SPL*), que é uma família de sistemas de *software* que compartilham um conjunto de características comuns. Uma linha de produto atende às necessidades específicas de um mercado em particular, e são desenvolvidos a partir de um conjunto comum de características básicas [21].

O modelo tradicional de *software* é o desenvolvimento de um sistema único, enquanto a abordagem de linha de produto de *software* estende este modelo para uma família de sistemas de *software*. Uma SPL envolve a análise de diversos requisitos funcionais. Estes requisitos são agrupados em conjuntos de características similares a uma família de *software*. Para projetar-se a arquitetura de uma SPL baseada nas características identificadas, essa arquitetura é composta pelas seguintes *features* [22]:

- Comuns - (obrigatórias) que devem ser requeridos por todos os sistemas membros da família;
- Opcionais - sendo referenciados por apenas alguns membros da família, e;
- Variantes - (alternativos) que determinam sistemas com versões diferentes dos demais sistemas da família.

Conforme [23], SPL é um paradigma que apresenta diversas vantagens e benefícios ao desenvolvimento de *software*, pois possibilita melhorias no desenvolvimento e gerenciamento de projetos, tais como: menor tempo de mercado, menor custo, melhor qualidade, maior produtividade e maior velocidade de entrada em novos mercados. Estes benefícios se dão em função da flexibilidade que os componentes de *software* possuem na arquitetura da família do *software*, permitindo a reusabilidade destes componentes.

Nesse contexto, a proposta deste trabalho faz parte de um projeto maior, o qual busca contribuir com uma pesquisa de doutorado, cuja proposta de tese é aplicar os conceitos de uma arquitetura de referência de teste no desenvolvimento de uma linha de produto de *software* para MBT. Assim, a proposta apresentada neste trabalho propõe investigar as características que são necessárias para representar o comportamento do usuário sob o SUT (*System Under Test*) em diferentes modelos, *e.g.*, Máquinas de Estados Finitos - MEF (*Finite State Machines - FSM*), Redes de Filas (*Queueing Network - QN*), Redes de Petri (*Petri Nets - PN*), Cadeias de Markov (*Markov Chain - MC*), Redes de Autômatos Estocásticos (*Stochastic Automata Network - SAN*); além de modelos semi-formais, como por exemplo, os perfis de diagramas UML, tais como SPT [24] [25] e MARTE [26] [27]. Esses modelos serão utilizados pelos produtos derivados da linha, a fim de gerar os casos e/ou *scripts* de teste para serem aplicados sobre o SUT.

É importante ressaltar que os modelos inferidos a partir das características definidas neste trabalho foram utilizados por outro trabalho de mestrado [5] cujo autor é integrante do mesmo grupo de pesquisa. Esse trabalho utilizará como entrada os casos de teste abstratos gerados e as informações extraídas dos modelos estudados relacionadas ao teste de desempenho, com o objetivo de instrumentalizar os casos de teste em *scripts* de teste para determinadas ferramentas de teste de

desempenho. Esse trabalho compreende ainda, a análise das características que serão necessárias para a geração dos *scripts*, baseadas nas diferentes ferramentas para teste de desempenho.

Inicialmente, com base na comparação das características a serem levantadas pelos dois trabalhos, tem-se como hipótese ideal que um conjunto de características presentes nas ferramentas estudadas seja um conjunto menor e que esteja contido dentro do conjunto de características que os modelos proveem (Figura 1.1-c). No entanto, não é desejável que estes conjuntos sejam disjuntos (Figura 1.1-a), neste caso todas as informações que os modelos fornecerem não serão aproveitadas para a geração dos *scripts* de teste das ferramentas. Todavia, a hipótese mais provável é que o conjunto de características que as ferramentas necessitam seja uma intersecção com o conjunto de características dos modelos analisadas na proposta deste projeto de pesquisa (Figura 1.1-b).

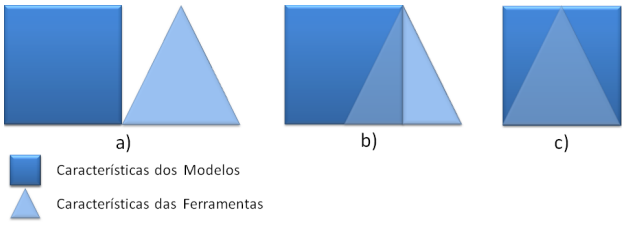

Figura 1.1: Relação entre as características dos modelos e ferramentas

O objetivo geral desta dissertação é investigar técnicas que busquem identificar e extrair automaticamente dos modelos, características que facilitem a geração de casos de teste abstratos para teste de desempenho, aplicando a abordagem MBT em uma ferramenta derivada de uma SPL.

Para resolvê-lo faz-se necessário atingir os seguintes objetivos específicos:

a) Realizar experimentos com modelos e notações para teste de *software*;

b) Utilizar um estudo de caso que implementem os modelos e notações estudadas em MBT;

c) Desenvolver um *plug-in* para a ferramenta $\mathrm{PL}e\mathrm{T}s$ [28] baseado no modelo proposto;

d) Implementar um algoritmo existente para realizar a extração de informações relevantes ao teste de *software* baseado em modelos;

e) Gerar uma forma comum de representar as características extraídas dos modelos estudados;

f) Prover um conjunto comum de características para que sejam gerados os casos de teste abstratos.

Ao alcançar o objetivo proposto, almejam-se resultados relevantes no contexto de teste de desempenho baseados em modelos para a área de teste de *software*.

Esta dissertação está organizada como segue. O Capítulo 2 apresenta uma breve contextualização sobre alguns conceitos abordados ao longo da pesquisa, tais como: teste de *software*, teste de desempenho, processo e taxonomia de MBT, além do mapeamento sistemático em MBT. O Capítulo 3 discute os modelos aplicados em MBT, avaliando as características necessárias aos modelos,

especificamente, para teste de desempenho; e apresenta a proposta de um conjunto de características para teste de desempenho em aplicações *web*. O Capítulo 4 descreve o processo de geração de casos de teste baseados em MEFs, detalhando sua implementação através do método HSI. O Capítulo 5 apresenta um estudo de caso utilizado para demonstrar a implementação do conjunto de características para teste de desempenho. Este conjunto de características proposto é implementado por meio de uma ferramenta MBT, a qual é um produto derivado de uma SPL. Finalmente, o Capítulo 6 apresenta as conclusões, perspectivas para trabalhos futuros a partir dos resultados alcançados com a pesquisa.

2. TESTE BASEADO EM MODELO

Este capítulo tem por objetivo descrever o embasamento teórico que facilite a compreensão de alguns conceitos posteriormente abordados no desenvolvimento deste trabalho.

2.1 Teste de *Software*

A palavra teste vem do Latim *Testum*, que significa panela de barro. Historicamente, esta panela era usada para medir o peso de vários elementos, ou seja, pôr à prova. As pessoas crescem realizando testes na escola, fazendo com que a maioria delas saiba, mesmo que intuitivamente, o significado da palavra. Atualmente, a palavra é aplicada em vários contextos, que visam medir ou avaliar os conhecimentos ou habilidades de pessoas ou processos [29].

Teste de *software* (ou simplesmente teste) é um processo que visa, intencionalmente, encontrar defeitos de um programa ou sistema durante sua execução [13]. Ou ainda, um processo que envolva toda a atividade que tem por objetivo avaliar um requisito de um programa ou sistema e determinar que ele atenda aos resultados esperados [14]. Uma definição formal é dada pela IEEE (*Institute of Electrical and Electronics Engineers*) como sendo um processo de execução do sistema ou de algum componente do sistema, aplicado em um ambiente controlado, observando ou registrando o comportamento do mesmo, a fim de analisar determinados aspectos do sistema ou componente [30].

O processo de teste de *software* é efetivamente visto como destrutivo, por tentar encontrar os defeitos em um programa. Um caso de teste bem sucedido é aquele que na sua execução, consegue fazer com que ocorram defeitos no programa. Eventualmente, deseja-se usar testes de *software* para avaliar o grau de confiança que um programa faz o que é suposto fazer, e não faça o que não se propõe a fazer, porém os resultados podem ser mais satisfatórios através da exploração de defeitos do sistema [13]. Ou seja, determinar a confiabilidade de que o programa ou sistema faz o que supostamente ele propôs fazer [29].

Outra definição de teste está relacionada à mensuração da qualidade do *software* [29]. Teste de *software* é o primeiro processo de garantia da qualidade de *software* aplicada ao controle de qualidade do produto de *software* [14]. Qualidade se traduz no cumprimento de requisitos [29], em outras palavras é o nível em que um sistema, componente ou processo satisfaz os requisitos especificados pelas necessidades ou expectativas de clientes, usuários ou *stakeholders* [30] [31].

Nesta seção, são apresentados alguns princípios básicos que permeiam esta área de conhecimento, a fim de esclarecer alguns conceitos. Primeiramente, a diferença dos termos Falha (*Fault*), Erro (*Error*) e Defeito (*Failure*) podem ser definidos conforme [97]:

- Falha está relacionada à aplicação propriamente dita, e são geradas por pessoas ao tentar solucionar um problema aplicando métodos, técnicas e ferramentas. Eles podem resultar na manifestação de erros no produto de *software*, *i.e.*, o fato da implementação da solução

estar diferente da especificação do *software*, *i.e.*, o sistema não está em conformidade com a especificação;

- Erro é a consequência causada por uma falha de algum artefato de *software*. A transição do estado correto do sistema ou serviço para o estado incorreto, posteriormente, resulta em um defeito no sistema ou serviço, o erro é especificamente esta transição;

- Defeito, por sua vez, é gerado através de erros do sistema, ou seja, a resposta do *software* não é a esperada pelo usuário. Em outras palavras, o defeito é caracterizado por um estado inconsistente ou inesperado do serviço ou sistema conforme a especificação definida pelo usuário. Ocasionalmente, alguns erros podem nunca serem gerados, todavia um defeito sempre é causado por um ou diversos erros.

Um exemplo, para elucidar, a relação entre os conceitos de falha, erro e defeito, pressupõe-se que um determinado computador possui algum problema em sua fonte, e que isto altere a tensão que alimenta os seus componentes eletrônicos. Esta etapa se caracteriza como uma falha. Porém, se esta alteração de tensão resultar na troca de valores de alguns bits de 0 para 1 ou vice-versa, a falha se transformará em erro. Neste momento, se o erro gerado não for tratado, então o sistema poderá travar ou ainda alterar alguma informação, *e.g.*, banco de dados. Assim, este erro causará um defeito ao usuário final.

Outros conceitos que devem ser esclarecidos estão relacionados aos artefatos de teste. Por isso, a próxima seção define os conceitos relacionados aos artefatos utilizados pelo processo de teste, os quais serão mencionados ao longo do trabalho.

2.1.1 Artefatos de Teste

A atividade de teste requer uma série de artefatos que formalizam a realização do processo de teste de *software*. A seguir serão listadas as definições que conceituam estes artefatos [7] [29] [30]:

- Plano de Teste - é um documento que descreve o planejamento do teste, desde a declaração do objetivo, a definição do escopo, abordagens e técnicas implementadas, cronograma das atividades e recursos necessários para realizar cada uma delas, as responsabilidades de cada um dos envolvidos e até os riscos que demandam um plano de contingência;

- Cenário de Teste - é um documento que compõe um conjunto de casos e/ou *scripts* de teste e a sequência em que devem ser executados. Além disso, todo cenário deve conter um objetivo específico a ser alcançado pelo teste, *e.g.*, avaliar um determinado requisito não-funcional. Para cenários de teste de desempenho, eles ainda possuem a distribuição da carga aos diferentes casos e/ou *scripts* de teste;

- Caso de Teste - um caso de teste é um conjunto de entradas que combinadas com condições e procedimentos possam ser executadas pelo testador, a fim de que possibilite determinar

um critério de sucesso ou defeito de acordo com um objetivo específico, além de avaliar o cumprimento de um requisito específico do *software*;

- Especificação de Caso de Teste - é um documento que contém a especificação de um requisito ou caso de uso que deverá ser atendido por um ou vários casos de teste;
- Projeto de Teste - é um documento que define a especificação dos testes a serem realizados, contém a descrição do conjunto de testes, além dos detalhes das características ou combinação de aspectos identificados para cada teste associado.

Estes artefatos são desenvolvidos, em diferentes fases do produto de *software*, na realização das diversas atividades que compõem o processo de teste de *software*: iniciação, planejamento, controle e monitoração, execução e encerramento. Na próxima seção, será detalhado o processo de teste de *software* apresentando as diferentes técnicas de teste utilizadas.

2.1.2 Técnicas de Teste

Atualmente existem muitas maneiras de se testar um *software*. Todavia, algumas técnicas que foram desenvolvidas para encontrar defeitos de sistemas baseados no paradigma de desenvolvimento estrutural, persistem até hoje para diversos sistemas, *e.g.*, sistemas orientados a objeto. Embora os paradigmas de desenvolvimento tenham evoluído, as principais técnicas continuam sendo o teste funcional e o teste estrutural.

Teste Funcional

O teste funcional, também comumente conhecido como teste caixa preta (*Black Box*), ou ainda *Data-Driven*, ou *Input/Output Driven*, ou teste baseado em especificação, é o processo de teste que visa identificar as inconformidades do *software*. Dado um conjunto de entradas e as condições de execução do teste, sendo ignorados os mecanismos internos do sistema, são avaliadas as saídas produzidas pelo sistema, a fim de que os objetivos tenham sido alcançados. Ou seja, os resultados estejam em conformidade com os requisitos funcionais estabelecidos pelo usuário [7] [13] [30] [31] [32].

Teste Estrutural

O teste estrutural, ainda citado como teste caixa branca (*White Box*), ou caixa de vidro (*Glass-box*) visa testar a estrutura interna de partes ou de componentes do sistema baseado nos requisitos de teste de uma dada implementação. A técnica está intrinsecamente vinculada aos detalhes do desenvolvimento do código fonte, com o intuito de testar os caminhos lógicos do programa, colocando à prova elementos de controle de fluxo como: condições, repetições, comandos, desvios, uso de variáveis e caminhos. Os resultados obtidos com o teste estrutural são complementares às demais técnicas de teste [13] [14] [15].

Todavia, a maioria dos testes são baseados nas especificações do *software*, o que não é o caso deste tipo de teste, em que são avaliadas os diferentes caminhos que o programa pode executar, a partir de sua implementação. Muitas vezes, apesar de testados os possíveis caminhos mapeados de execuções de um programa, não é garantido que o programa não possa vir a falhar, pois algum caminho pode não ter sido identificado. E r5ainda, a execução de um comando ou programa com falha, pode não resultar na geração de defeitos percebidos pelo usuário final [7]. Portanto, o teste estrutural é relevante para fatores de qualidade do *software* como manutenabilidade, estrutura e confiança, pois inclui casos de teste que não são avaliados por meio de testes funcionais.

2.1.3 Fases de Teste

Teste de *software* pode ser classificado de diversas formas, depende do ponto de vista da equipe do projeto e sua experiência com o processo de desenvolvimento de *software*. Nesta subseção, são explicadas as fases de testes de *software* de acordo com os processos de Validação, Verificação e Teste (VV&T), além de como e quando cada um pode ser aplicado no processo de desenvolvimento de *software* [13]. O processo de teste de *software* pode ser definido em cinco fases [13]: teste unitário, teste de integração, teste de sistema, teste de validação, teste de implantação e teste de regressão.

Teste Unitário

Teste unitário, ou teste de módulo, é o processo que tem em vista testar pequenas partes ou componentes isolados do sistema, como por exemplo: funções, procedimentos, métodos, classes, etc. Este tipo de teste é um dos primeiros a serem implementados no processo de desenvolvimento de *software*, muitas vezes são automatizados pela equipe de teste por meio da própria ferramenta de desenvolvimento e executadas pelo desenvolvedor em tempo de desenvolvimento. Desta forma, os objetivos dos testes unitários são de resolver pequenas falhas dos módulos do sistema, evitando que estas falhas se propaguem para outros níveis ou tipos de testes aplicados em etapas subsequentes, a facilidade de depuração da falha pelo fato dela estar isolada em um determinado módulo, e a alternativa de aplicar o processo de teste do *software* simultaneamente para vários módulos do sistema [7] [13] [15].

Teste de Integração

O Teste de integração pode ser implementado aplicando duas abordagens: não incremental ou '*Big-Bang*' e incremental ou iterativa. Na primeira, deve-se testar cada módulo do sistema, independentemente, e depois testar suas combinações. Já na segunda, obriga-se em testar o novo módulo do sistema somente quando combinado com os demais módulos existentes já previamente testados. A abordagem de teste incremental é a melhor proposta, pois possibilita que defeitos entre as *interfaces* sejam detectadas mais rapidamente, além de permitir avaliar de forma mais simples a

complexidade das interações entre os módulos, uma vez que a complexidade cresce à medida que novos módulos sejam incorporados ao sistema [7] [13].

No teste de integração, após serem testados isoladamente cada unidade, o foco é dado no desenvolvimento da estrutura do sistema. O objetivo desta fase de teste é avaliar os defeitos entre os componentes, quando combinadas as diversas partes do sistema. Assim, o propósito do teste de integração é analisar as interações dos componentes do sistema e comprovar a consistência e a compatibilidade entre eles [13] [14] [15]. Como esta etapa depende de conhecimento prévio da estrutura do sistema, muitas vezes ela é executada pela própria equipe de desenvolvimento [15].

Teste de Sistema

Comparado com os demais tipos de teste, o teste de sistema é o processo mais difícil e o mais propenso a mal-entendimentos. Comumente este tipo de teste é confundido com o teste funcional, avaliando ser o teste 'completo' do sistema ou programa [7] [13].

A proposta do teste de sistema é uma atividade de verificação, que visa assegurar que o *software* esteja consistente com os objetivos especificados pelo usuário. Logo, a compreensão é a fundamental característica deste tipo de teste, pois a comparação das discrepâncias entre comportamento do sistema com a especificação dele resulta em falhas de tradução, assim torna este processo vital, uma vez que a severidade dos defeitos e a propensão de suas ocorrências serem avaliadas somente nesta etapa [7] [13] [15].

Para avaliar todas as implicações da especificação de requisitos do sistema com o resultado do comportamento deste sistema, existe uma vasta lista de categorias de testes de sistemas, tais como [13] [14]: 1) Teste de habilidade - assegura a existência de que cada função descrita nos objetivos do sistema tenha sido implementada, *i.e.*, identifica quais são as reais habilidades do sistema e se elas estão de acordo com a especificação; 2) Teste de usabilidade - avalia os problemas e desconforto da facilidade de uso das *interfaces* do sistema em relação aos critérios de interação humano-computador especificadas [33]; 3) Teste de segurança - teste que tenta corromper os mecanismos de segurança do programa; 4) Teste de desempenho - visa avaliar se o programa satisfaz os objetivos de desempenho especificados, como por exemplo - tempo de resposta das operações e taxas de transferências; 5) Teste de armazenamento - procura analisar a quantidade de memória principal e secundária utilizada pelo sistema, assim como identificar estouro de memória (*overflow*); 6) Teste de configuração - testa as diferentes configurações de *hardware*, sistemas operacionais, e/ou bancos de dados, avaliando os requisitos mínimos e máximos necessários para instalar e executar o *software*; 7) Teste de compatibilidade/conversão - verifica se o sistema atende aos objetivos de compatibilidade e com os procedimentos de conversão de uma versão do sistema para outra; 8) Teste de instalabilidade - ou teste de implantação é o teste que visa assegurar que o procedimento de instalação do sistema está de acordo com os objetivos do sistema, garantindo uma boa impressão e confiabilidade do sistema ao usuário. É um processo de validação que permite instalar e configurar um sistema com sucesso. O processo pode ser completo, parcial ou atualizações de instalação, ou ainda, a desinstalação de componentes do sistema. Objetivo é garantir que todos os

recursos e componentes sejam, de fato, instalados, as opções selecionadas funcionem corretamente, que os arquivos tenham sido criados e contenham os dados necessários, além da configuração apropriada do *software* seja efetuada [13].; 9) Teste de confiabilidade - avalia se o sistema é capaz de completar suas operações com sucesso em um determinado intervalo de tempo estabelecido nos objetivos do sistema, como por exemplo MTTF (*Mean Time To Failure*); 10) Teste de recuperação - teste que visa avaliar se a recuperação do sistema funciona corretamente, um objetivo de sistema é minimizar o tempo médio para recuperação (*Mean Time To Recovery - MTTR*). Um exemplo deste tipo de teste é simular defeitos em *hardware* a fim de avaliar a reação do sistema; 11) Teste de manutenabilidade - teste que considera os reparos e modificações do serviço de um sistema ser concluído em um determinado intervalo de tempo; 12) Teste de documentação - avalia a precisão e clareza da documentação do usuário para determinar a prévia representação do sistema; 13) Teste de procedimento - teste que verifica se os procedimentos documentados do sistema possam ser realizados por uma pessoa leiga.

Teste de Validação

Teste validação é uma atividade, como o próprio nome especifica, tem em vista validar a aceitabilidade do produto de *software*, ou seja, se o produto atende as expectativas do usuário em relação a sua qualidade [7] [13] [34].

Este teste possui três estratégias que podem ser implementadas, tais como [7]: 1) Teste de aceitação formal - aplicado em um ambiente altamente controlado, é um teste planejado e controlado, no qual se testa um subconjunto de casos de teste aplicados na etapa de teste do sistema, normalmente executado pelo usuário final; 2) Teste *Alfa* - são testes executados pelos usuários no ambiente de desenvolvimento, entretanto não seguem nenhum roteiro do que deve ser testado, apenas se identifica e se documenta as funções e regras de negócios exploradas; 3) Teste *Beta* - efetuado no próprio ambiente do usuário, é o menos rigoroso e ao mesmo tempo o mais subjetivo, pois o usuário tem liberdade em definir os critérios para determinar a aceitação ou rejeição do *software*.

Teste de Regressão

Teste de regressão é um teste aplicado após a manutenção de reparo de falhas ou melhorias de novas ou antigas funcionalidades do programa, pois o sistema passa a estar vulnerável à ocorrência de riscos relacionados aos novos defeitos que podem surgir. O objetivo é identificar se as funcionalidades da versão anterior do sistema permanecem funcionais e continuam válidas, caso contrário, afirma-se que o sistema "regrediu". Uma abordagem consiste em retestar todos os casos de testes, todavia ela nem sempre é efetiva, pois alguns casos de testes podem se tornar obsoletos já que as funcionalidades a serem testadas foram alteradas, excluídas ou trocadas [7] [13] [15] [34].

2.2 Teste de Desempenho

Desempenho é uma qualidade fundamental de sistemas de *software*, a qual afeta todas as camadas subjacentes de sistemas, tais como: sistemas operacionais, *middleware*, *hardware*, redes de comunicação, entre outros. Desta forma, com o intuito de melhorar a qualidade do produto de *software*, a Engenharia de Desempenho de *Software* - EDS (*Software Performance Engineering - SPE*) [35] aplica seus esforços para descrever e fornecer os meios a fim de melhorar o desempenho através de duas abordagens distintas: no ciclo inicial baseada em modelos preditivos ou no ciclo final baseadas em medições.

Nesse sentido, SPE apresenta um conjunto de atividades e heurísticas a fim de garantir que os esforços da análise de desempenho sejam usados durante todo o ciclo de desenvolvimento de *software*. O ponto de partida de uma análise de desempenho começa desde a análise de requisitos não-funcionais [36] até a avaliação das métricas e os resultados esperados pelos testes de desempenho [37]. O objetivo deste tipo de teste é identificar possíveis gargalos ou insuficiências de desempenho, determinando os limites de processamento do sistema, além de verificar se o desempenho do *software* satisfaz os requisitos especificados [35].

Uma maneira de guiar a implantação da SPE é através da realização do teste de desempenho baseado em medições do *software*. Teste de desempenho é uma das atividades da SPE, responsável por realizar testes de parte ou de todo o sistema sob carga normal e/ou carga de estresse. Teste de desempenho pode ser dividido basicamente em duas categorias [38]:

- Teste de carga (*Load Testing*) - visa determinar ou validar o comportamento de um sistema sob condições de normais de carga. Principalmente, a fim de verificar se o sistema atende aos requisitos de desempenho especificados;

- Teste de estresse (*Stress Testing*) - foco em determinar o comportamento de um sistema quando ele é utilizado em condições onde a carga é superior àquela esperada. Além de revelar os defeitos e determinar os pontos de falhas quando o sistema está sob enormes cargas.

Para isso, diferentes conjuntos de ferramentas de automação e verificação foram criados para aumentar a eficiência na execução de cenários de teste de desempenho. Existem várias ferramentas para a geração automática e execução de cenários de teste, para citar algumas: Apache JMeter [39], HP LoadRunner [40], IBM Rational Performance Tester [41], Borland SilkPerformer [42] e Microsoft Visual Studio [43].

Nesse contexto, diferentes tipos de modelos são utilizados para modelar o teste de desempenho. Esses modelos, quando utilizados neste âmbito de teste, têm como objetivo formalizar os requisitos não-funcionais que o sistema deve atender. Além disso, também podem ser utilizados para proporcionar a automação por meio da abordagem MBT. Máquinas de Estados Finitos (MEF) [44], Cadeias de Markov [45], Redes de Autômatos Estocásticos [46], Redes de Filas [47] e Redes de Petri [48] são exemplos de modelos formais suscetíveis à integração com a abordagem MBT.

Em outro panorama, existem modelos que são usados extensivamente em ambientes industriais, *e.g.*, UCML (*User Community Modeling Language*) [49], o perfil UML SPT (*Schedulability, Performance and Time*) [25] e o perfil UML MARTE (*Modeling and Analysis of Real-Time and Embedded Systems*) [27]. Aos modelos dos perfis UML são adicionados estereótipos e rótulos a fim de anotar informações relacionadas ao teste de desempenho, *e.g.*, número de usuários, caminhos de rede dos aplicativos, comportamento do usuário ou informações de carga.

2.3 Processo MBT

Teste baseado em modelos (*Model-Based Testing - MBT*) é uma técnica para a geração automática de artefatos de teste com base em modelos extraídos de artefatos de *software* [50]. O principal objetivo do MBT é a criação de artefatos de teste que descrevam os requisitos e comportamento do próprio sistema, visando automação do processo de teste de *software* [51]. Assim, o teste baseado em modelos requer a elaboração de modelos para atenuar os esforços das atividades de teste. Engenheiros de teste quando elaboram os casos de testes do sistema, implicitamente, constroem estes modelos mentalmente.

Portanto, encapsular o comportamento e a estrutura do sistema através de modelos possibilita que a equipe compartilhe e reutilize estes artefatos. Desta forma, a partir destes modelos, é possível extrair informações contidas neles para a geração de novos artefatos de teste, tais como casos de teste e *scripts* [1] [52]. Uma vez desenvolvido o modelo, ele pode ser utilizado de várias formas e por várias etapas ao longo do ciclo de vida do desenvolvimento do produto de *software*, tais como: melhoria da qualidade de especificações dos requisitos, a geração de código, análise de confiabilidade do sistema e geração de teste [51].

O custo do teste de *software* está relacionado ao número de iterações e casos de teste que são executados durante o processo de desenvolvimento. O teste tem como pressuposto ser uma das etapas mais onerosas e caras do desenvolvimento de *software* [13] [53]. Desta forma, MBT é uma ótima abordagem para mitigar esta premissa, automatizando o processo de geração de casos e/ou *scripts* de teste com o objetivo de reduzir tempo e custo do processo de teste [17].

A técnica de elaboração de modelos para sua aplicação no processo de teste de *software* e, análise da confiabilidade de sistemas vem sendo estudada e desenvolvida há mais de duas décadas. Entretanto, poucas empresas adotaram MBT em suas atividades, permanecendo a maioria delas no processo manual do teste [51].

A modelagem de *software* é uma técnica importante que deve ser implementada no desenvolvimento de *software*, pois permite a captura e compartilhamento do conhecimento acerca do sistema, aumentando a qualidade da especificação e o reuso dos modelos desenvolvidos à medida que o sistema evolui [51]. A criação de modelos formais baseados nos requisitos exige dos engenheiros e analistas de testes a detecção de informações que na maioria das vezes estão implícitas em documentos tradicionais de especificação, incluindo comentários e estereótipos ao modelo que enriquecem a qualidade da especificação. Desta forma, estas informações incrementadas ao modelo servirão para

a criação de novos artefatos, ou ainda, permitir a automação de outros processos para melhorar a comunicação da equipe e a qualidade dos artefatos desenvolvidos. Esta modelagem pode ser implementada em vários contextos, pois a abordagem de MBT se aplica a diversas técnicas e/ou fases de teste de *software*. Nos primeiros estudos, a modelagem de *software* se limitava à técnica de caixa preta. Atualmente, os modelos já são capazes de abstrair inúmeras outras informações, podendo ser aplicada a técnica MBT para realização de outros tipos de testes, como o teste de desempenho [54].

O processo da abordagem MBT exige a realização de atividades específicas, que fogem da habitual atividade de teste de *software*, pois ela requer da equipe uma adaptação do seu processo de teste, com investimentos em ferramentas e treinamentos. As principais atividades de MBT que definem seu processo podem ser ilustradas conforme a Figura 2.1 e são descritas a seguir [1]:

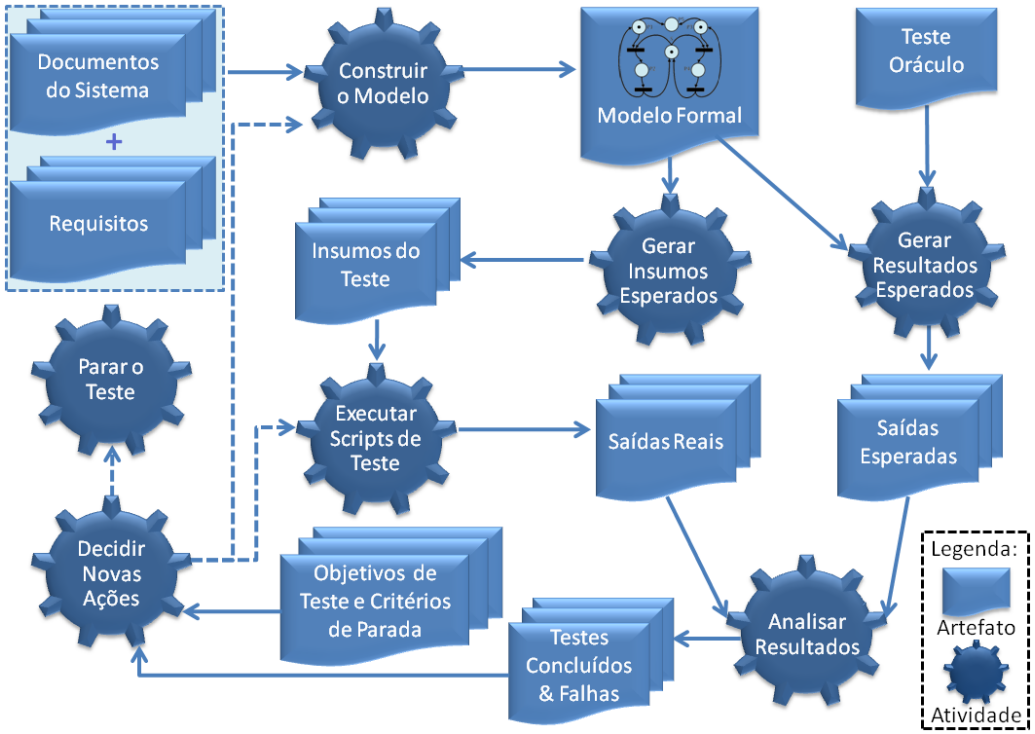

Figura 2.1: Atividades de teste baseado em modelos (adaptado de [1])

- Construir o modelo - construção de um modelo baseado na especificação dos requisitos do sistema, que definem a estrutura e o comportamento do próprio sistema sob teste. Nesta etapa, define-se a escolha do modelo de acordo com a aplicação a ser desenvolvida;

- Gerar insumos esperados - utilização do modelo formal desenvolvido para geração dos casos de teste ou *scripts* de teste, a fim de obter as entradas e saídas esperadas na execução dos testes, em outras palavras, a solução proposta em MBT deve prover uma ferramenta para geração dos casos de teste e/ou produzirem os *scripts* de teste;

- Gerar resultados esperados - geração de um mecanismo que determine se os resultados de uma execução do teste realizado estão corretos ou não. Este mecanismo é o Oráculo de Teste (*Test*

Oracle), que atua como o critério que determina se o resultado obtido é igual ao resultado esperado [15]. Desta forma, baseado no modelo formal, extraem-se os resultados esperados para a etapa de análise;

- Executar os *scripts* de teste - execução dos *scripts* de testes desenvolvidos anteriormente, sendo submetidos aos sistemas sob teste e armazenando os resultados do processamento de cada caso de teste;
- Analisar os resultados - uma vez executados os testes, é realizada a comparação dos resultados obtidos com os resultados esperados, gerando relatórios e gráficos analíticos dos resultados obtidos;
- Decidir novas ações - a partir da análise dos resultados, deve-se decidir qual caminho percorrer: modificar o modelo a fim de corrigir defeitos encontrados, tais como: gerar e executar mais testes, decidir quando parar de testar o sistema e implantar o produto no cliente e, estimar a qualidade do *software*;
- Parar o teste - final do processo, quando terminar o teste do sistema e publicar o sistema de *software*.

A execução destas atividades do processo pode gerar uma série de vantagens à equipe de teste, tais como [1]:

- Cronogramas mais curtos, com menor custo e melhor qualidade;
- Identificação de ambiguidades nas especificações dos modelos nas fases iniciais do processo;
- Melhoria na comunicação entre os desenvolvedores e testadores, em razão do desenvolvimento do modelo;
- Capacidade de geração automática dos *scripts* de teste em diversos casos de testes não-repetitivos;
- Mecanismos de teste para executar automaticamente os *scripts* gerados;
- Facilidade de atualização dos cenários de teste quando houver mudanças de requisitos;
- Capacidade de avaliar os testes de regressão, identificando qual o nível de cobertura o teste obtêm;
- Permite avaliar a qualidade e a confiabilidade do *software*.

Contudo, estas vantagens requerem um investimento em ferramentas e treinamento da equipe. Ou seja, o perfil do profissional necessário para trabalhar com teste baseado em modelos requer habilidades e conhecimentos sobre modelos formais, tais como: teoria dos autômatos, teoria dos

grafos, linguagens formais, além de saber criar e interpretar estes modelos [1]. Desta forma, a abordagem MBT se tornará eficiente somente se a equipe for qualificada, ou se a gerência estiver disposta a investir na formação e qualificação da equipe de teste. Por isso, a técnica requer um alto investimento na sua implantação, obtendo retornos a médio e longo prazo [1].

2.4 Taxonomia MBT

O trabalho de pesquisa desenvolvido em [55] [56] teve como resultado uma proposta de taxonomia para MBT. Os autores definem sete diferentes dimensões que agrupam um conjunto de características aplicadas em MBT. Para isto, classificam diversas abordagens incorporadas em ferramentas de MBT, com o objetivo de compreender as limitações da abordagem e entender as questões envolvidas na integração de MBT em um processo de desenvolvimento de *software*.

A Figura 2.2 apresenta a taxonomia MBT agrupada em três grandes atividades: *Modelo*, *Geração de Teste* e *Execução de Teste*. A partir destas atividades sete dimensões de MBT foram identificaram [55] [56].

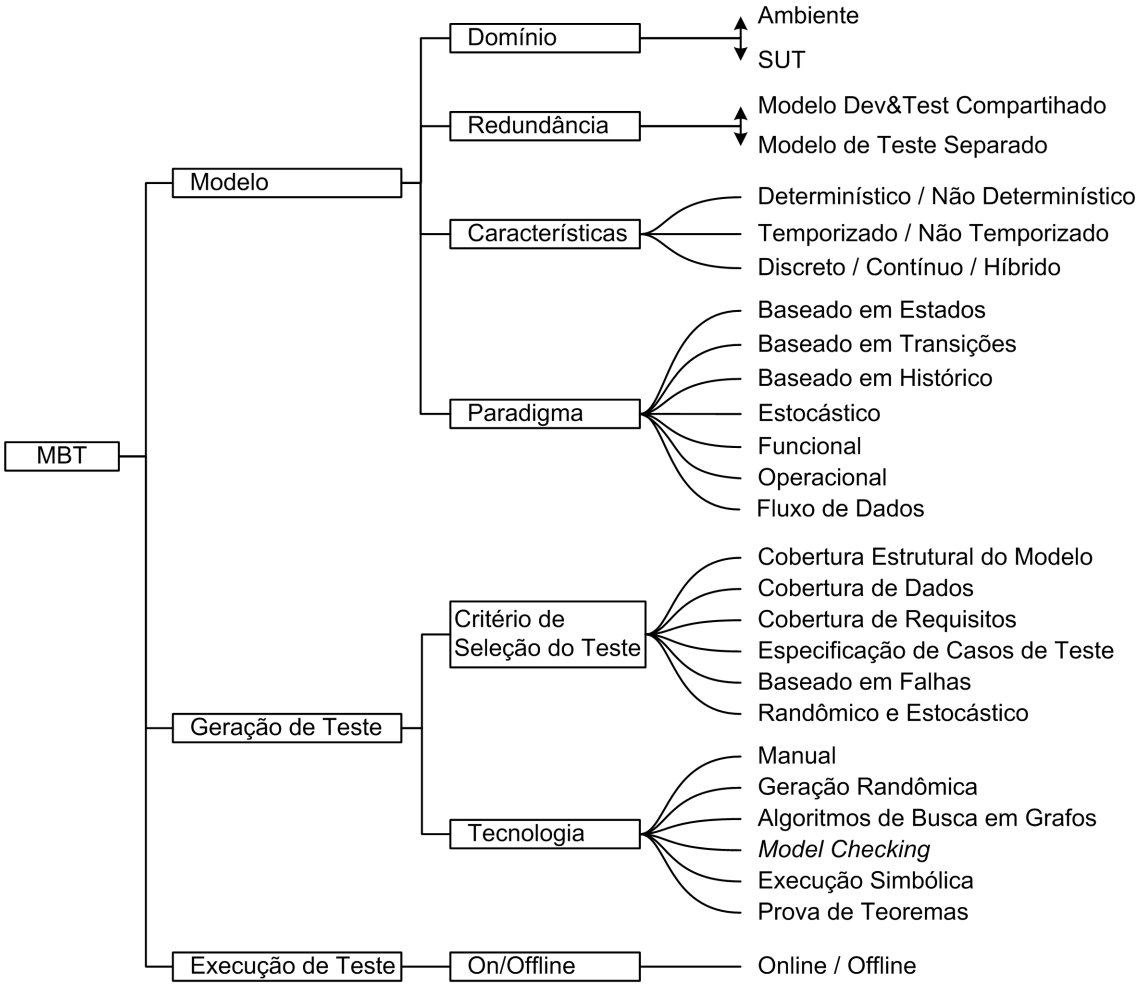

Figura 2.2: Taxonomia MBT (adaptado de [2])

2.4.1 Características da Taxonomia MBT

As dimensões definidas na Figura 2.2 foram concebidas com conceitos ortogonais, pois elas se influenciam mutuamente. Por exemplo, caso um projeto use um modelo contínuo em vez de um discreto, este é suscetível em limitar a escolha do paradigma de modelagem, de critérios de seleção, teste e da tecnologia de geração dos casos de teste. As setas verticais indicam uma faixa contínua de alternativas para as folhas indicarem alternativas mutuamente exclusivas, enquanto que as linhas curvas indicam alternativas que não são necessariamente mutuamente exclusivas, (*e.g.*, algumas ferramentas podem usar mais de uma geração de tecnologia, e é comum e desejável para apoiar diversos tipos de critérios de seleção de teste).

Domínio (*Subject*)

O domínio é a primeira dimensão, e diz respeito ao escopo do modelo. Este pode ser aplicado para avaliar o comportamento do sistema (*System Under Test - SUT*), ou então, com a intenção de analisar o comportamento do ambiente do SUT. Na maioria dos casos, ambos os modelos são utilizados.

Redundância (*Redundancy*)

Os modelos desenvolvidos para MBT podem ser aplicados em diferentes abordagens. Basicamente, eles diferem do nível de redundância entre o modelo de teste e o modelo de desenvolvimento. Enquanto o primeiro permite a geração tanto de código fonte quanto casos de teste, a segunda abordagem implementa um modelo específico para teste, baseado na especificação de documentação do SUT.

Características (*Characteristics*)

As características dos modelos estão relacionadas ao não-determinismo, incorporam o comportamento temporal e com a natureza contínua ou discreta dos modelos. A distinção em termos de dinâmica e comportamento entre as diferentes características dos modelos são fundamentais, pois impactam na escolha das demais dimensões da taxonomia, como o paradigma de modelagem e os critérios para seleção de teste.

Paradigma (*Paradigm*)

Os paradigmas estão relacionados às diferentes notações de modelagem de comportamento dos sistemas, tais como [55] [56]:

- Notação baseada em estados (*State-Based (or Pre/Post) Notations*) - representa um conjunto de variáveis que avaliam o estado interno do sistema em um instante de tempo. As operações são definidas por uma pré e pós-condição. Um exemplo desta notação é a OCL (*Object Constraint Language*);

- Notação baseada em transições (*Transition-Based Notations*) - descreve o comportamento do sistema através das transições entre seus estados. Um dos representantes desta notação são as MEFs onde os nós da MEF representam os estados mais importantes do sistema e os arcos representam as ações ou operações do sistema. Outro exemplo de notação muito utilizada são os diagramas de estados da UML;

- Notação baseada em histórico (*History-based Notations*) - implementa a lógica temporal para avaliar o comportamento do sistema por meio das sequências de iterações entre os componentes do sistema;

- Notação funcional (*Functional Notations*) - descreve um sistema como uma coleção de funções matemáticas. Estas funções podem ser de primeira ordem, como especificações algébricas, ou ordem superior, *e.g.*, a notação HOL (*Higher Order Logic*). Esta abordagem não é muito aplicada em MBT quando comparada com outras notações, isto em razão do seu elevado grau de abstração e na dificuldade em escrevê-la;

- Notação operacional (*Operational Notations*) - representa uma coleção de processos executáveis, executando em paralelo. Comumente aplicados em sistemas distribuídos e de protocolos de comunicação. Um exemplo desta notação são as redes de Petri (*Petri Nets - PN*);

- Notação estocástica (*Stochastic Notations*) - representa um sistema por um modelo probabilístico de eventos. Alguns formalismos como cadeias de Markov (*Markov Chain - MC*) ou redes de autômatos estocásticos (*Stochastic Automata Network - SAN*) podem ser implementados;

- Notação de fluxo de dados (*Data-Flow Notations*) - concentrar-se no fluxo de dados em vez do fluxo de controle. Como exemplo os diagramas de blocos utilizados no Matlab Simulink.

Critério de Seleção de Teste (*Test Selection Criteria*)

Nesta dimensão, o objetivo é definir os recursos que são necessários para controlar a geração de testes. A taxonomia apresenta um conjunto de seis critérios. Em geral, nem sempre existe o melhor critério a ser aplicado, mas é responsabilidade do engenheiro de teste configurar o ambiente para a geração de teste, a escolha mais adequada do critério de seleção dos testes e especificações de casos de teste. O estudo detalha cada um destes critérios que são: cobertura estrutural, cobertura de dados, cobertura de requisitos, especificação dos casos de teste, baseado em falhas e, randômico e estocástico.

Tecnologia (*Technology*)

A sexta dimensão é a tecnologia que deve ser usada durante a geração do teste. Uma das principais vantagens de MBT é o seu potencial para a automação. Esta tarefa é baseada em um modelo do SUT e a especificação do caso de teste, possivelmente dado como um modelo de ambiente

com restrições adicionais. Os casos de teste podem ser derivados estocasticamente ou por meio de algoritmos de busca em grafos, *model checking*, execução simbólica ou prova de teoremas dedutivos.

On-line/Off-line

A última dimensão está relacionada ao tempo de geração e execução dos casos de teste. Eles podem ser geridos de duas formas: testes *on-line* ou testes *off-line*. Testes *on-line* significam que o algoritmo de geração dos casos de teste reage com as saídas reais do SUT. Esta abordagem às vezes é necessária quando o modelo possui características não-determinísticas, fazendo com que o gerador de casos de teste analise qual o caminho que o SUT percorreu, e siga o mesmo caminho no modelo. Para testes *off-line* os casos de teste são gerados, estritamente, antes de serem executados. Geração de casos de teste *off-line* para modelos não-determinísticos é mais onerosa, exigindo a geração de casos de teste em formatos de árvores ou grafos, ao invés de sequências. Os testes *off-line*, na sua maioria são pragmáticos, e permitem o gerenciamento e execução por outras ferramentas de gerenciamento de teste. Uma das vantagens da abordagem *off-line* é a geração dos casos de teste separado da execução, permitindo que eles sejam executados diversas vezes em diferentes máquinas e/ou ambientes.

2.5 Mapeamento Sistemático em MBT

O objetivo principal desta seção é fornecer uma visão ampla sobre as abordagens MBT propostas nos últimos cinco anos. Além disso, este mapeamento sistemático em MBT deve ajudar pesquisadores e/ou organizações privadas a compreender os últimos temas propostos que envolvem modelos e ferramentas para MBT.

A questão principal de pesquisa deste mapeamento sistemático em MBT é "Quais trabalhos de MBT têm sido propostos nos últimos cinco anos e quais são suas principais características?" Para responder esta questão de pesquisa é importante estruturar sua resposta conforme alguns critérios.

Na última década, Engenharia de *Software* baseada em evidências (*Evidence-Based Software Engineering - EBSE*) tem atraído interesse da área de Engenharia de *Software* [57]. Recentemente, outra metodologia tem sido adotada, Estudo de Mapeamento Sistemático (*Systematic Mapping Studies*) [58]. Esta metodologia é indicada para proporcionar uma visão geral de algum tópico da área em que se esteja trabalhando, identificando áreas adequadas para a realização de revisões sistemáticas da literatura (*Systematic Literature Reviews*) e lacunas na qualidade em estudos primários [59].

Esta seção descreve um estudo de mapeamento sistemático [58], aplicado a fim de mapear o campo MBT, através da síntese de evidências que sugerem importantes implicações para a prática, bem como identificar tendências de pesquisa, questões em aberto, e áreas de melhoria.

Os resultados de um estudo de mapeamento sistemático são obtidos a partir de um processo definido de: pesquisa, triagem, avaliação e análise de estudos relevantes. Por isso, o mapeamento sistemático apresenta uma síntese e um resumo objetivo das evidências relevantes. Com base na

análise dos resultados dos estudos primários, orientados por um conjunto de questões de pesquisa, o mapeamento sistemático destaca as tendências e identifica lacunas para futuras pesquisas em MBT.

2.5.1 Processo de Mapeamento Sistemático

Um processo de mapeamento sistemático é apresentado em [58], o qual descreve o processo de mapeamento sistemático na Engenharia de *Software* e o compara com revisões sistemáticas. Com base nisso, os autores definem as diretrizes para a realização de um mapeamento sistemático.

O processo adaptado de [58] definido neste mapeamento sistemático foi dividido em fases, atividades e artefatos. Cada fase possui duas atividades, e cada atividade por sua vez resulta em um artefato conforme mostrado na Figura 2.3.

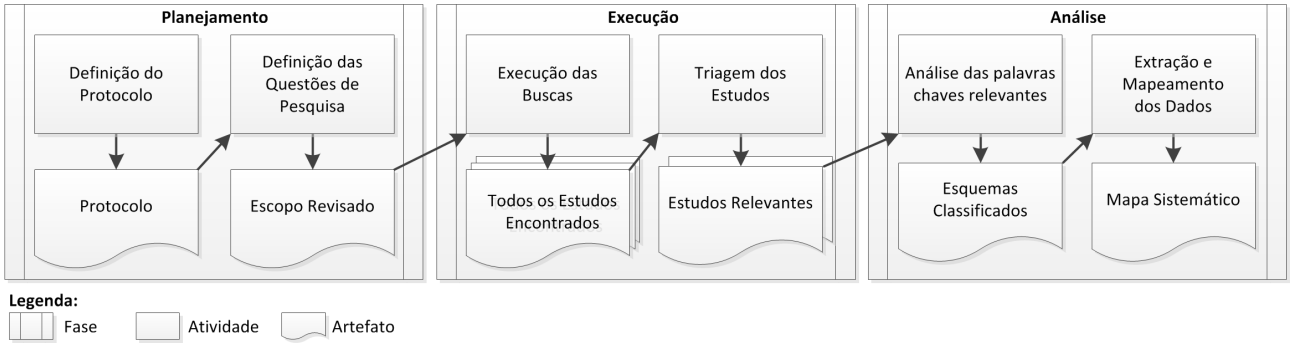

Figura 2.3: Processo de mapeamento sistemático

2.5.2 Planejamento

Inversamente às revisões de literatura tradicionais, um mapeamento sistemático é conduzido de acordo com um planejamento. Para isto, é necessário definir o escopo da pesquisa e ter como resultado um protocolo previamente definido a ser seguido durante sua execução. Este protocolo foi desenvolvido baseado nos protocolos apresentados em [58] [60]. O protocolo do mapeamento é o ponto de partida, cujas principais características são a definição das questões de pesquisa e os métodos necessários para aplicar a condução da revisão, tais como: estratégia de busca, critérios de inclusão e exclusão, processo de seleção dos estudos, critérios de avaliação da qualidade e processo de extração de dados. Este protocolo foi executado por dois revisores e sugestões de melhoria foram discutidas e implementadas ao protocolo final.

Desta forma, os principais requisitos identificados foram os seguintes: a) *População* - pesquisas em teste de *software*; b) *Intervenção* - abordagens de MBT; c) *Resultado* - abordagens, métodos, metodologias, técnicas, modelos, especificações e ferramentas que são usadas atualmente no processo de MBT, e quais são suas características; d) *Problema* - identificar quais as abordagens de MBT, métodos, metodologias, técnicas, modelos, especificações e ferramentas têm sido propostas nos últimos cinco anos e quais são suas principais características; e) *Aplicação* - a comunidade de pesquisa em MBT.

As questões de pesquisa secundárias do estudo são definidas a seguir:

RQ1. Quantos trabalhos em MBT foram publicados desde 2006?

RQ2. Quais são as ferramentas MBT comerciais e acadêmicas?

RQ3. MBT é utilizada por quais domínios de aplicação?

RQ4. Que modelos ou especificações são usados?

RQ5. Quais são os principais grupos de pesquisa que trabalham neste tópico?

RQ6. Quais são os estudos mais citados?

A estratégia de busca e seleção dos estudos primários foi definida de acordo com as fontes de busca, língua dos trabalhos, termos e sinônimos e de acordo com a *string* de busca para a revisão:

a) *Fontes* - as bases de dados eletrônicas indexadas usadas na busca foram: ACM Digital Library, IEEExplore, SpringerLink, Scopus e Compendex; b) *Língua* - Inglês por ser a língua internacionalmente aceita nos principais eventos e conferências de trabalhos científicos; c) *Termos e Sinônimos* - Uma estrutura de perguntas baseado em [59] foi usado para a construção das *strings* de busca a ser aplicado nas fontes selecionadas (ver Tabela 2.1); d) *String* - A *string* de busca foi desenvolvida usando o operador `OR` para alternar as palavras dos sinônimos de cada termo e o operador `AND` para se agregar os principais termos de intervenção, população e resultado, conforme Tabela 2.1. A *string* de busca genérica está representada no quadro abaixo:

```
 ((MBT OR "model-based testing" OR "model based
testing" OR "model-based test" OR "model based
test" OR "model-based software testing" OR "model
based software testing") AND (approach OR method OR
methodology OR technique) AND (software))
```

Tabela 2.1: Definição da *string* de busca

Estrutura	**Termos**	**Sinônimos**
População	*Software*	
Intervenção	*Model-based Testing*	MBT *Model-based Test* *Model-based Software Testing*
Resultado	*Approach*	*Method* *Methodology* *Technique*

2.5.3 Execução

O mapeamento sistemático foi conduzido por dois revisores durante cinco meses (de Fevereiro 2011 a Junho 2011). As buscas foram executadas no intervalo de datas entre 16 de Fevereiro a 4 de

março de 2011, baseada na *string* de busca definida na estratégia de busca. Esta *string* foi aplicada e ajustada de acordo com as funcionalidades de cada motor de busca.

Concluídas as buscas das *strings* nas bases de dados eletrônicas, um total de 803 estudos foram encontrados. Um dos revisores, autor desta dissertação, iniciou o processo de seleção dos estudos primários. Durante esta etapa 448 estudos foram excluídos por serem duplicados. Na próxima etapa, os dois revisores, baseando-se no título, palavras-chaves e leitura do resumo, o escopo de 355 estudos foi analisado, identificando aqueles que apresentavam alguma contribuição relevante[1] para MBT. Destes foram excluídos mais 305 trabalhos. Na etapa seguinte, os 50 estudos pré-selecionados foram lidos integralmente pelos revisores, e mais 4 trabalhos foram excluídos por não atenderem aos critérios de seleção. Na última etapa, os 46 trabalhados pré-selecionados foram submetidos ao formulário de avaliação da qualidade, resultando na desclassificação de mais 13 estudos e a inclusão dos 33 estudos primários para o processo de mapeamento sistemático. A Tabela 2.2 apresenta um resumo dos estudos retornados, estudos incluídos, e a porcentagem de estudos incluídos para cada fonte de busca. É importante ressaltar que há algumas sobreposições entre as diferentes bases de dados. Por exemplo, os onze estudos encontrados e incluídos em Compendex também foram retornados por Scopus.

Tabela 2.2: Estudos retornados e incluídos para cada fonte

Fonte	**Retornados**	**Incluídos**	**%**
ACM	104	7	21,21%
Compendex	219	11	33,33%
IEEE	160	13	39,39%
Scopus	289	0	-
SpringerLink	31	2	6,07%

A avaliação de qualidade dos estudos foi aplicada através de critérios de avaliação de qualidade como passo final no processo de seleção dos estudos primários. Os estudos classificados como: *bom*, *muito bom* ou *excelente* foram incluídos como estudos primários. Para isso, cada revisor realizou, novamente, a leitura completa dos estudos previamente selecionados, a fim de avaliar de forma independente cada um destes estudos ao responder as perguntas do formulário de avaliação de qualidade. Desta forma, foram extraídos dados de todos os 33 estudos primários selecionados.

2.5.4 Análise

Nesta subseção é descrita a avaliação qualitativa da literatura em relação às questões de pesquisa do estudo. Os 33 estudos primários foram relacionados com o domínio da pesquisa para avaliar sua aplicação nos diferentes níveis de teste: *Unitário*, *Integração*, *Sistema*, *Validação* e *Regressão*, uma sexta categoria foi atribuída para os trabalhos que não puderam ser classificados em nenhuma das mencionadas anteriormente, denominada *Não se aplica*.

[1] Trabalhos com as seguintes propostas: técnicas, metodologias, abordagens, modelos, linguagens, notações ou ferramentas; para serem aplicados na modelagem de teste e/ou geração de artefatos de teste.

O número de estudos primários descrevendo cada nível de teste foi: 1 (*Validação*), 6 (*Integração*), 20 (*Sistema*), 5 (*Regressão*) e 1 (*Não se aplica*). Observa-se na análise qualitativa do estudo que nenhum aborda o teste unitário. Todavia, pode-se destacar que mais de 60% dos estudos tratam sobre teste de sistema, vale ressaltar que muitos estudos classificados neste nível abordam o teste de *interface* (*GUI testing*). Em relação ao ano de publicação dos estudos primários selecionadas, a distribuição da quantidade de estudos por ano foi de: 4 (2006), 4 (2007), 7 (2008), 6 (2009) e 12 (2010).

A Figura 2.4 (adaptado de [61]) mostra o gráfico bolha com a distribuição das aplicações (eixo x central) dos estudos primários em relação ao ano de publicação (eixo y esquerdo) e fase de teste (eixo y direito). A bolha na intersecção dos eixos contém a quantidade dos estudos primários.

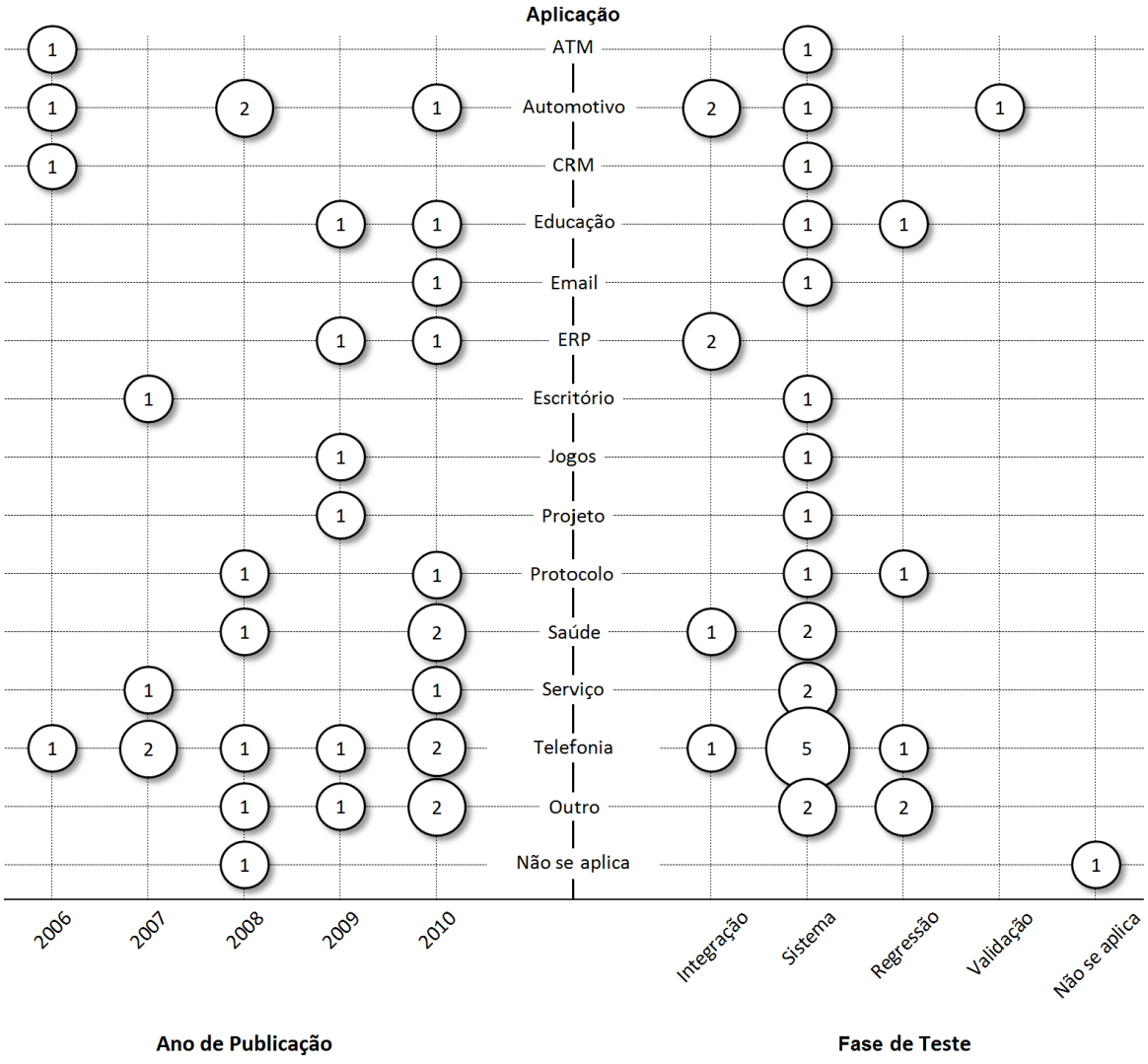

Figura 2.4: Distribuição do domínio dos estudos em relação ao ano de publicação e fase de teste

A análise deste gráfico visa responder as questões de pesquisa **RQ1** (*Quantos trabalhos em MBT foram publicados desde 2006?*) e **RQ3** (*MBT é utilizada por quais domínios de aplicação?*). Desta forma, respondendo a questão **RQ1**, foram publicados 33 estudos baseados nos critérios de busca utilizados pelo mapeamento sistemático. Enquanto que a questão **RQ3** destaca estudos que

utilizam MBT nos seguintes domínios aplicação: ATM, automotivo, CRM, educação, e-mail, ERP, escritório, jogos, projeto, protocolo, saúde, serviço e telefonia. Evidentemente, que MBT pode ser aplicado em vários domínios de aplicação, mas alguns deles se destacam em relação aos outros, tais como: automotivo, saúde e telefonia.

Mutuamente, a partir do processo de extração e mapeamento dos dados podem-se destacar ainda os resultados do mapeamento em relação ao processo MBT, com os seguintes resultados: 5 (*Modelo de Teste*), 2 (*Transformação de Modelo*), 18 (*Geração do Caso de Teste*), 1 (*Instanciação do Caso de Teste*), 1 (*Seleção do Caso de Teste*), 3 (*Validação*) e 3 (*Outros*). A maioria dos estudos descrevem técnicas e métodos para geração dos casos de teste e poucos descrevem detalhes de como criar os modelos de teste.

2.6 Considerações

A modelagem de sistemas é parte essencial do processo de desenvolvimento de *software*. Com o crescente avanço da complexidade dos atuais sistemas, os modelos tornam-se fundamentais para permitir a abstração de problemas. Atualmente, o uso destes modelos tende a ampliar, agregando ou refinando estes modelos para sua aplicação em outras atividades, ou ainda, o seu reuso em outras fases do desenvolvimento do *software*, como por exemplo, teste de *software*. Desta forma, a abordagem MBT tem se tornado uma boa alternativa por ser baseada em modelos, sendo uma aliada na tentativa de reduzir os esforços e custos do processo de teste, além de prover a melhoria da qualidade do *software*.

3. CONJUNTO DE CARACTERÍSTICAS PARA TESTE DE DESEMPENHO

Este capítulo sintetiza o estudo de modelos e notações aplicadas ao teste de desempenho em MBT, com o intuito de analisar as diferentes características contidas em cada modelo para a geração automática de artefatos de teste. Ainda, apresenta a proposta do conjunto de características para teste de desempenho. Além disso, compara e correlaciona o resultado deste conjunto de características com a proposta de outro trabalho de dissertação, o qual compreende a análise das características que serão necessárias para a geração dos *scripts* de teste, baseadas nas diferentes ferramentas para teste de desempenho.

3.1 Análise de Modelos para Teste de Desempenho

A área de Engenharia de Desempenho de *Software* trabalha com modelos a fim de melhorar a qualidade do produto de *software*. Para isto, ela faz uso de duas abordagens, uma baseada em medições e outra baseada em modelos. Assim, é desta forma que MBT se interliga com a área de teste de desempenho.

Neste contexto, baseado na taxonomia MBT apresentada na Seção 2.4, foram definidas algumas características para serem analisadas nos diferentes modelos e notações estudados. O estudo desenvolvido limitou-se em analisar os formalismos, modelos e notações aplicados para teste de desempenho.

O resultado gerado por este estudo é a Tabela 3.1 que apresenta a relação entre as características da taxonomia MBT e os modelos estudados. Algumas das características propostas possuem mais de uma alternativa, mutuamente exclusivas, como por exemplo, domínio, o qual determina se um modelo especifica o ambiente ou o SUT, nunca ambos.

Os trabalhos analisados que aplicam modelos formais em teste de *software*, tais como: *e.g.*, Máquinas de Estados Finitos - MEF (*Finite State Machines - FSM*), Redes de Filas (*Queueing Network - QN*), Redes de Petri (*Petri Nets - PN*), Cadeias de Markov (*Markov Chain - MC*), Redes de Autômatos Estocásticos (*Stochastic Automata Network - SAN*), possuem as mesmas características. Quanto ao domínio, todos estes modelos formais descrevem o ambiente do sistema, e também o comportamento entre os componentes do sistema. Nos trabalhos avaliados, os formalismos foram classificados como modelo de teste separado, uma vez que se tratam exclusivamente do modelo do ambiente. Para MC e SAN ambos são não-determinísticos, temporizados, híbridos (dinâmica) e estocásticos (paradigma). Modelos com dinâmica de comportamento híbrido são modelos que permitem ser modelados como discretos ou contínuos, mas não possuem a mesma característica em uma mesma cadeia. MC foi caracterizado nesta classe por apresentar as Cadeias de Markov de Tempo Discreto (DTMC) e de Tempo Contínuo (CTMC).

Os exemplos de PN estudados apresentam redes de dinâmica discreta (comportamento temporal,

Tabela 3.1: Síntese dos modelos e seu mapeamento com a taxonomia MBT

Características	**Subcaracterísticas**	**Modelos**										
		UTP	**TTCN-3**	**UCML**	**SPT**	**MARTE**	**PerfTTCN**	**MEF**	**QN**	**MC**	**PN**	**SAN**
Domínio	Ambiente							X	X	X	X	X
	SUT	X	X	X	X	X	X					
Redundância	Modelo Desenv&Teste Compartilhado	X			X	X						
	Modelo de Teste Separado		X	X			X	X	X	X	X	X
Não-determinístico									X	X	X	X
Não-temporizado			X	X			X	X	X		X	
Dinâmica	Discreto	X	X	X	X	X	X	X	X		X	
	Contínuo											
	Híbrido									X		X
Paradigma (Notação)	Baseado em Estados	X	X	X	X	X	X					
	Baseado em Transições							X				
	Baseado em Histórias											
	Funcional *											
	Operational *								X		X	
	Estocástico									X		X
	Fluxo de Dados											

* Nos estudos avaliados não foram encontrados trabalhos que aplicam os paradigmas funcional e operacional.

no qual o fenômeno acontece em determinados instantes de tempo dentro de um período), se enquadram no paradigma operacional, não-determinístico e não-temporizado. Apesar de existir o conceito de tempo através das Redes de Petri Temporizadas (*Timed Petri Nets - TPN*) associando um tempo fixo no disparo de transições, até onde se pôde identificar, esta variação do formalismo não foi encontrada em nenhum estudo relacionado ao teste de desempenho de *software*. MEF também foi classificada como dinâmica discreta, mas com determinismo, não-temporizado e paradigma baseado em notação de transição. QN, por outro lado, pertence ao paradigma operacional, com dinâmica discreta, não-determinístico e não-temporizado.

Os demais modelos semi-formais ou informais (UTP (*UML Testing Profile*), TTCN-3 (*Testing and Test Control Notation version 3*), UCML (*User Community Modeling Language*), SPT (*Schedulability, Performance and Time*), MARTE (*Modeling and Analysis of Real-Time and Embedded Systems*) e PerfTTCN) estudados possuem domínio no modelo de comportamento do SUT e não do ambiente. Todos são modelos determinísticos, pois todos os parâmetros do sistema são previamente determinados, ou seja, os parâmetros do sistema não fazem uso de variáveis aleatórias e probabilísticas. Neste sentido, estes modelos são classificados como paradigma baseado em estados, por avaliarem o comportamento do SUT em um determinado instante de tempo, e por este mesmo motivo, o comportamento dinâmico destes modelos é avaliado como discreto. Por se tratarem de modelos que caracterizam o SUT, as métricas que estes modelos permitem medir são essencialmente

transações por segundo, requisições por segundo e tempo de resposta das transações do SUT.

Uma das características que mais difere estes modelos semi-formais ou informais é a redundância entre os modelos de desenvolvimento e modelos de teste. As linguagens UTP, SPT e MARTE foram classificados como temporizadas por implementarem em suas notações aspectos de tempo na modelagem. Para estas linguagens os modelos de desenvolvimento podem agregar estereótipos UML, a fim de adicionar informações relevantes ao teste de *software*. Por isso, estes modelos foram classificados como modelos compartilhados de desenvolvimento e teste. Por outro lado, as notações TTCN-3, UCML e PerfTTCN requerem o desenvolvimento de um modelo exclusivo para o teste de *software*, utilizando os modelos de desenvolvimento apenas como especificação do SUT.

O mais comum é o desenvolvimento de dois modelos: o modelo do SUT utilizado com modelos semi-formais ou informais e o modelo do ambiente do SUT implementado, normalmente, com modelos formais [2]. Novas notações e versões estão surgindo visando estreitar esta relação, uma delas é o perfil UML MARTE para sistemas em tempo-real e embarcados, o qual permite a modelagem do sistema em diversas camadas, integrando modelos de *software*, *hardware* e plataformas.

Uma das contribuições deste estudo foi a análise destas características dos modelos que são utilizados no teste de desempenho aplicando MBT. Com esta análise, definem-se quais características devem estar presentes nos modelos para a modelagem do SUT, buscando o maior reuso deste modelo ao longo do ciclo de vida de desenvolvimento do *software*. Esta abordagem é interessante uma vez que o modelo limita a escolha de critérios de seleção, tecnologias de teste e de geração dos casos de teste.

3.2 Conjunto de Características dos Modelos para Teste de Desempenho

Baseado na análise dos modelos aplicados em MBT para teste de desempenho (Seção 3.1), juntamente com a análise dos estudos selecionados no mapeamento sistemático apresentado na Seção 2.5 e as experiências *ad hoc*, observações e práticas desenvolvidas no projeto de pesquisa, foi proposto um conjunto de características para o teste de desempenho em aplicações *web*.

Analisando estes diferentes modelos, linguagens e notações estudadas, um conjunto de características necessárias para a geração de casos de teste abstratos para teste de desempenho é apresentado na Tabela 3.2.

As características foram classificadas como obrigatórias ou opcionais, pois determinam a obrigatoriedade das características mínimas necessárias para compor um modelo de teste de desempenho. Desta forma, permitindo que a técnica MBT possa ser aplicada neste modelo para geração de cenários e casos de testes abstratos. Como exemplo, pode-se citar a característica "Tempo de Espera" (*Think Time*), por ser uma característica que pode ser omitida durante a execução do teste, ou substituída por um valor padrão entre cada uma das interações do usuário com o SUT.

O conjunto de características ainda foi classificado de acordo com sua relação com os artefatos de teste, dividindo-se em: "Cenário de Teste", "Casos de Teste" e, "Plano de Teste". Esta classificação pode auxiliar o engenheiro de teste de desempenho, familiarizado com os artefatos de teste, a

Tabela 3.2: Classificação das características para teste de desempenho

Características	Obrigatória	Opcional	Artefatos de Teste: Cenário de Teste	Artefatos de Teste: Caso de Teste	Artefatos de Teste: Plano de Teste
Dados		X		X	
Parâmetro		X		X	
Probabilidade		X	X		
Requisição	X			X	
Requisições por Segundo		X			X
SUT	X		X		
Tempo de Espera		X		X	
Tempo de Execução	X		X		
Tempo de Finalização		X	X		
Tempo de Inicialização		X	X		
Tempo de Resposta	X				X
Transação		X		X	
Transações por Segundo	X				X
Usuários de Finalização		X	X		
Usuários de Inicialização		X	X		
Usuários Virtuais	X		X		
Utilização de Recursos		X			X
Vazão		X			X

compreender melhor a distribuição e hierarquia das características, caso venha fazer uso do conjunto de características para teste de desempenho a fim de aplicá-lo em algum outro modelo ou formalismo para modelagem do teste.

O artefato de teste "Cenário de Teste" compreendem tanto informações relacionadas ao próprio cenário quanto características da carga de trabalho (*Workload*). O "Caso de Teste" define as características que determinam o comportamento do SUT, enquanto que o "Plano de Teste" destaca as características relacionadas às métricas mensuradas para atender aos objetivos do teste.

Conforme mostra a Tabela 3.2, seis características são classificadas como obrigatórias, enquanto que doze características são opcionais. Já o artefato de teste "Cenário de Teste" possui oito características, enquanto que "Caso de Teste" e "Plano de Teste" possuem cinco características, respectivamente.

As características que compõem o conjunto de características para teste de desempenho são definidas como:

- Dados - refere-se à entrada de dados que serão fornecidos para a aplicação ao executar os casos de teste;

- Parâmetro - representa os campos do formulário ou parâmetros correlacionados aos *Dados*;
- Probabilidade - probabilidade de execução de uma determinada sequência de atividades por um usuário virtual;
- Requisição - define as requisições (atividade, operação, ação, chamada, métodos, evento) que o usuário virtual executará na aplicação *web*;
- Requisições por Segundo (RPS) - é uma solicitação de qualquer espécie feita a partir do usuário virtual para o aplicativo sendo testado. No contexto de aplicações *web*, refere-se às requisições HTTP (*Hypertext Transfer Protocol*). Quanto maior for o resultado desta métrica, mais requisições da aplicação são processadas por segundo;
- *System Under Test (SUT)* - representa o endereço do servidor de aplicação (*host*) onde o sistema é executado;
- Tempo de Espera - denota o tempo entre o momento em que a atividade se torna disponível para o usuário e o momento em que o usuário decide executá-la. Por exemplo, o tempo para preencher um formulário antes de sua submissão;
- Tempo de Execução - denota o tempo de execução ou duração de um cenário de teste;
- Tempo de Finalização - refere-se à fração de tempo que cada "Usuários de Finalização" é finalizado no teste;
- Tempo de Inicialização - determina a fração de tempo que cada "Usuários de Inicialização" é inicializado no teste;
- Tempo de Resposta - intervalo de tempo entre o pedido e o início/conclusão do serviço;
- Transação - define as transações existentes dentro de um caso de teste;
- Transações por Segundo (TPS) - é um tipo de vazão aplicado em sistemas de processamento de transações ou aplicações *web*;
- Usuários de Finalização - define a quantidade de usuários que finalizarão o teste em cada fração de tempo definido em "Tempo de Finalização";
- Usuários de Inicialização - define a quantidade de usuários que iniciarão o teste em cada fração de tempo definido em "Tempo de Inicialização";
- Usuários Virtuais - define o número de usuários simultâneos que estarão executando o sistema;
- Utilização de Recursos - fatia de tempo em que o sistema permanece ocupado, atendendo a requisições. Esta característica está vinculada aos recursos consumidos pelo sistema durante seu processamento, tais como: memória, processador, CPU, disco, etc;

- Vazão (*Throughput*) - taxa de atendimento de pedidos pelo sistema. Mas no domínio de aplicações *web*, esta característica está relacionada à rede, ou seja, bits por segundo (bps).

Este estudo limitou-se em avaliar as características, presentes na literatura, dos modelos aplicados em MBT para teste de desempenho. No entanto, no que diz respeito à análise das características das ferramentas de teste de desempenho, considera-se uma limitação deste estudo. Entretanto, este tópico foi abordado na pesquisa de outro aluno de mestrado [5] integrante do mesmo projeto de pesquisa, o qual será descrito na seção a seguir.

3.3 Modelo de Características para Ferramentas de Teste de Desempenho

Esta seção apresenta uma síntese do modelo de características para ferramentas de teste de desempenho, desenvolvido por outra pesquisa de mestrado [5]. Com o objetivo de correlacionar os resultados da pesquisa com os resultados do conjunto de características para teste de desempenho proposto (Seção 3.2). Assim, permitindo a análise dos resultados obtidos de acordo com as hipóteses (Figura 1.1) avaliadas no início da pesquisa.

O estudo limitou-se em analisar as características implementadas para cada gerador de carga, *i.e.*, ferramentas para teste de desempenho, que descrevem o processo de geração dos cenários e/ou *scripts* de teste, por meio do estudo e investigação de diversos trabalhos publicados na área, incluindo relatórios técnicos da indústria.

Um dos critérios para a seleção das ferramentas a serem pesquisadas baseou-se em uma pesquisa de mercado publicada por [62]. Para cada ferramenta selecionada o autor estabeleceu um conjunto de características necessárias para a criação de cenários e/ou *scripts* de teste, totalizando a análise de trinta trabalhos utilizados como referência para a elaboração do modelo.

Em uma segunda etapa o autor correlaciona as diferentes características a fim de classificar aquelas que são similares, pois são tratadas com diferentes aspectos pelos autores ou fabricantes. Outra premissa adotada foi selecionar as características pertencentes a mais de uma ferramenta, *i.e*, excluindo as características identificadas como exclusivas de determinada ferramenta. O autor assevera que um dos objetivos para o modelo é torná-lo independente de tecnologia, privilegiando e generalizando as características comuns encontradas.

A Tabela 3.3 apresenta a lista das características elegidas, agrupadas pelas ferramentas de teste de desempenho estudadas. Vale destacar que algumas características estão presentes em todas as ferramentas, enquanto que outras são específicas de determinadas ferramentas.

As características apresentadas na Tabela 3.3 que compõem o modelo de características para ferramentas de teste de desempenho são descritas a seguir:

- Contadores - refere-se ao conjunto de contadores que serão utilizados para medir o desempenho do ambiente. Cada uma das ferramentas analisadas possui um conjunto de contadores que pode ser configurado para o teste. Durante a execução do teste para cada ferramenta as informações referentes aos contadores são mostradas em tabelas e gráficos;

Tabela 3.3: Características das ferramentas para teste de desempenho [5]

Características	Apache JMeter	HP LoadRunner	Rational Performance Tester	Borland SilkPerformer	Microsoft Visual Studio
	Ferramentas para Teste de Desempenho				
Contadores	X	X	X	X	X
*Dados	X	X	X	X	X
Host da Aplicação	X	X	X	X	X
*Parâmetros	X	X	X	X	X
Perfil da Carga de Trabalho		X		X	X
Quantidade de Usuários	X	X	X	X	X
Quantidade de Usuários da Rampa de Descida		X	X	X	
Quantidade de Usuários da Rampa de Subida		X	X	X	X
Quantidade de Usuários por *Script*	X	X	X	X	X
Tempo de Aquecimento			X	X	X
Tempo de Duração do Teste	X	X	X	X	X
Tempo de Pensamento	X	X	X	X	X
Tempo de Rampa de Descida		X	X	X	
Tempo de Rampa de Subida	X	X	X	X	X
*Transações	X	X	X	X	X
*URLs da Aplicação	X	X	X	X	X

* Características incorporadas ao modelo baseado no processo de geração de casos de teste de desempenho para aplicações *web* [38]

- Dados - refere-se à entrada de dados externos que serão fornecidos ao gerador de carga durante a execução dos *scripts* de teste;
- *Host* da Aplicação - refere-se às informações de configuração dos endereços IP da aplicação para onde será gerada a carga;
- Parâmetros - determina os parâmetros enviados ao servidor para processar a requisição HTTP;
- Perfil da Carga de Trabalho - refere-se ao perfil de teste que será executado. Normalmente, esta característica é previamente definida por meio de estereótipos para os perfis de usuários, diferenciando basicamente na forma como os usuários iniciam o teste, se todos ao mesmo tempo ou de forma gradativa de acordo com as configurações da rampa de subida;
- Quantidade de Usuários - refere-se ao número de usuários que farão requisições ao(s) ende-

reço(s) configurado(s);

- Quantidade de Usuários da Rampa de Descida - define a quantidade de usuários que deixarão de fazer carga no sistema em um determinado tempo;
- Quantidade de Usuários da Rampa de Subida - define a quantidade de usuários que iniciarão o teste em um determinado tempo;
- Quantidade de Usuários por *Script* - refere-se à distribuição da "Quantidade de Usuários" para cada *script* de teste;
- Tempo de Aquecimento - define um período de tempo em que a ferramenta de teste irá coletar informações referentes aos contadores configurados para o teste. Durante esse período não é realizado nenhum tipo de carga no sistema. O objetivo é verificar se não há influência de nenhum fator externo à aplicação que possa, por exemplo, estar concorrendo por recursos e com isso, afetar os resultados do teste;
- Tempo de Duração do Teste - refere-se ao tempo total de execução do teste;
- Tempo de Pensamento - refere-se ao tempo que cada usuário leva para realizar determinada atividade entre duas requisições consecutivas, *e.g.*, preenchendo algum formulário *web*;
- Tempo de Rampa de Descida - define o tempo que levará para um conjunto de usuários deixarem o teste (processo de finalização do teste);
- Tempo de Rampa de Subida - define o tempo que levará para um determinado conjunto de usuários iniciarem o teste (processo de inicialização do teste);
- Transações - especifica o conjunto de transações contidas em um *script* de teste;
- URLs da Aplicação - especifica o endereço URL que será requisitado ao servidor processar após a submissão de alguma requisição HTTP da aplicação.

3.4 Considerações

As características apresentadas no conjunto de características para teste de desempenho (Tabela 3.2) também podem ser utilizadas em um contexto diferente da SPL. Apesar do forte acoplamento no que diz respeito a sua implementação, propõe-se que o modelo seja abstrato, *i.e.*, genérico o suficiente para ser estendido e aplicado em outros contextos, *e.g.*, *Model-Driven Development (MDD)* e *Test Driven Development (TDD)*.

A Tabela 3.4 apresenta a correlação entre as características dos modelos (Seção 3.2) e ferramentas (Seção 3.3). É importante ressaltar que o conjunto de características presentes nas ferramentas para teste de desempenho foram superiores aos encontrados nos modelos para teste de desempenho.

Tabela 3.4: Mapeamento da correlação entre as características dos modelos e ferramentas

Características dos Modelos	**Características das Ferramentas**	**Correlação**
SUT	*Host* da Aplicação	X
Usuários Virtuais	Quantidade de Usuários	X
Probabilidade	Quantidade de Usuários por *Script*	X
Tempo de Execução	Tempo de Duração do Teste	X
Tempo de Inicialização	Tempo de Rampa de Subida	X
Usuários de Inicialização	Quantidade de Usuários da Rampa de Subida	X
Tempo de Finalização	Tempo de Rampa de Descida	X
Usuários de Finalização	Quantidade de Usuários da Rampa de Descida	X
Tempo de Espera	Tempo de Pensamento	X
Requisição	URLs da Aplicação	X
Parâmetro	Parâmetros	X
Dados	Dados	X
Transação	Transações	X
Transações por Segundo	Contadores	X
Tempo de Resposta		
Requisições por Segundo		
Vazão		
Utilização de Recursos		
	*Tempo de Aquecimento	
	*Perfil da Carga de Trabalho	

* Características incorporadas ao conjunto de característica para teste de desempenho

Se bem que a maioria das características, apesar de serem tratadas com nomenclaturas distintas, mostraram-se compatíveis umas com as outras.

Inicialmente, com base na comparação das características a serem analisadas pelas duas pesquisas, foram definidas três hipóteses para a correlação entre as características dos modelos e ferramentas. A suposição otimista tinha como hipótese ideal que um conjunto de características presentes nas ferramentas estudadas fosse um conjunto menor e que estivesse contido dentro do conjunto de características que o modelo provê (Figura 1.1-c). Todavia, a hipótese pessimista definia não ser desejável que estes conjuntos fossem disjuntos (Figura 1.1-a), neste caso todas as informações que os modelos fornecerem não seriam aproveitadas para a geração dos *scripts* de teste das ferramentas. Entretanto, a hipótese mais provável, e que fora evidenciado a partir da correlação entre as características apresentada na Tabela 3.4, foi que o conjunto de características que as ferramentas necessitam seja uma intersecção com o conjunto de características analisadas na proposta deste projeto de pesquisa (Figura 1.1-b).

Complementando esta análise, destacam-se as características rotuladas com "*", as quais estão presentes nas características das ferramentas, mas não possuem correlação direta com as características dos modelos. Entretanto, os modelos são capazes de absorver estas informações, *e.g.*, a característica "Perfil de Carga de Trabalho" é uma informação que está distribuída em outras carac-

terísticas dos modelos, tais como: “Usuários Virtuais”, “Probabilidade” e “Requisição”, dependendo exclusivamente do nível de abstração aplicado para modelar as interações dos usuários com o SUT. Já a característica “Tempo de Aquecimento” poderia ser adicionada ao modelo, *e.g.*, através de um diagrama UML de casos de uso.

4. GERAÇÃO DE CASOS DE TESTE BASEADO EM MEF

No Capítulo 3 foram analisados diversos modelos aplicados ao teste de desempenho. Um destes modelos são as máquinas de estados finitos, as quais são implementadas através da técnica de "Teste Baseado em Máquinas de Estados Finitos", doravante denominado "Teste Baseado em MEFs". Esta é uma das diversas técnicas que integram a abordagem de teste baseado em modelos (ver Capítulo 2). Neste sentido, a técnica de teste baseado em MEFs torna-se uma importante abordagem para detectarmos os possíveis aspectos de comportamento e controle de fluxo do sistema, a fim de definir as sequências de entrada para execução dos testes [63].

Nas últimas décadas vem crescendo a pesquisa acadêmica e o interesse da indústria na área de teste baseado em MEFs. Esta técnica faz uso de uma MEF para modelar o sistema sob teste (*System Under Test - SUT*) [64]. O teste realizado a partir de uma MEF se caracteriza por um conjunto de entradas que produz um conjunto de saídas. Estas saídas quando comparadas com um oráculo de teste (*Test Oracle*), processo capaz de definir se o caso de teste foi realizado com sucesso ou não, permite automatizar a execução dos testes, delegando a tarefa de decisão de quais casos de testes produziram ou não a saída esperada.

Teste baseado em MEFs permite encontrar o maior número de defeitos em uma implementação, a qual se assume heuristicamente que pode ser modelada como uma MEF pertencente a um domínio de defeitos. Essa heurística é conhecida como hipótese de teste, sendo necessária para que um conjunto finito de testes possa ser gerado, uma vez que o número potencial de implementações para uma especificação de determinado sistema é infinito [65]. Dessa forma, a implementação I de uma MEF permite verificar se ela está de acordo com sua especificação M.

Definição 1. *A notação de uma MEF é representada por uma quíntupla* $M = (I, O, S, \int_T, \int_O)$*, sendo que [66]:*

- I *(Input) é um conjunto finito não vazio de elementos de entradas;*
- O *(Output) é um conjunto finito de elementos de saída;*
- S *(State) é um conjunto finito não vazio de estados* S_i*, incluindo o estado especial* S_0*, chamado estado inicial;*
- $\int_O : (S \times I) \rightarrow O$ *é a função de saída;*
- $\int_T : (S \times I) \rightarrow S$ *é a função da transição de um estado para outro.*

Chow [44] assevera que os defeitos de uma MEF implementada podem ser classificados como:

- Defeito de Transferência - a implementação I é dita ter defeitos de transferências se I não é equivalente a M e I possa ser alterado para que seja equivalente a M, ajustando apenas a função transição de I (sem adicionar ou excluir estados em I), *i.e.*, quando a transição alcança um estado incorreto;

- Defeito de Saída - a implementação I é dita ter defeitos de saída, se I não é equivalente à especificação M e I possa ser modificado para se tornar equivalente a M, mudando apenas a função de saída de I (sem adição ou exclusão de estados I), *i.e.*, quando a transição produz uma saída incorreta;

- Estados Extras/Faltantes - a implementação I é dita ter estados extras/faltantes se a fim de que a implementação I esteja de acordo com sua especificação M, o número de estados em I deve ser reduzido/aumentado.

Essas classes de defeitos quando modeladas por MEFs através da estimativa do número de estados da implementação, permite a geração de um conjunto finito de sequências de teste. A combinação dos conjuntos de entradas e saídas produzidas relacionados com as transições entre os estados do sistema definem o conjunto de sequências de teste da MEF. Existem diversos métodos de geração de sequências de teste, cada um deles atende a determinadas características da MEF. Dentre os métodos mais conhecidos podem-se destacar os métodos: *Transition Tour* (TT) [67], *Distinguishing Sequence* (DS) [68], W [44], *Unique Input/Output* (UIO) [69], Wp [66], *Switch-Cover* [70], *Harmonized State Identification* (HSI) [71] [72], H [73], *State Counting* [74] e SPY [75].

4.1 Máquina de Estados Finitos

Uma Máquina de Estados Finitos - MEF (*Finite State Machine - FSM*) [64], popularmente conhecida como máquina de estados, é uma máquina hipotética composta por um conjunto de estados finitos e transições entre os estados, em que ações são executadas nestas transições [76], ou ainda pode ser definido como um "Autômato Finito" [77].

A literatura considera que existem dois tipos de MEFs: máquina de Mealy [78] e máquina de Moore [79]. O comportamento de ambas é idêntico, mas suas implementações diferem em um aspecto, a definição da saída. Na máquina de Mealy as saídas são representadas na transição, enquanto que na máquina de Moore são representadas no próprio estado.

Máquina de estados finitos é uma alternativa viável para projetar e testar os componentes de *software* [44]. Uma das tarefas mais onerosas do processo de teste é a elaboração das diversas sequências de entradas a fim de que sirvam como dados para o teste. Máquinas de estados são os modelos mais adequados para este fim, uma vez que MEFs são aplicáveis a qualquer modelo de especificação que possa ser descrito com um número finito (normalmente muito pequeno) de estados específicos. Além do que, um conjunto de entradas (dados) varia de acordo com o "estado" exato do *software*, essa característica faz de modelos baseados em estados forneçam uma adaptação lógica para o teste de *software*. Em contrapartida, sistemas complexos geram MEFs com um grande números de estados, tornando o modelo difícil de ser construído e mantido [1].

Uma MEF pode ser representada por um diagrama de estados finitos (conforme retrata a Figura 4.1, adaptado de [44]) ou por uma tabela de transições de estados (segundo mostra a Tabela 4.1, adaptado de [76]). No primeiro, os estados são representados por nodos, enquanto que as transições

são arestas direcionadas que ligam um estado ao outro. Cada aresta, usualmente, é rotulada para indicar uma operação, condição ou evento associado à transição, assim como a entrada que gera a transição e a saída que ela produz. No segundo, a tabela de transição, as linhas indicam os estados, enquanto que as colunas representam as entradas [7] [76].

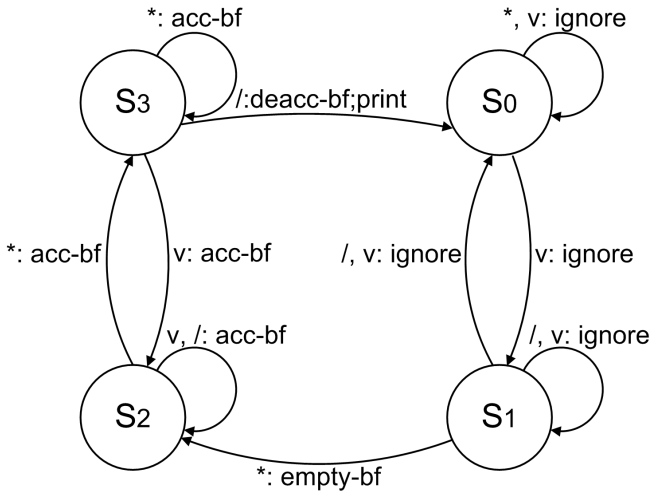

Figura 4.1: Exemplo de diagrama de transição de estados

Tabela 4.1: Exemplo de tabela de transição de estados

Estado \ Entrada	*	/	v	*	/	v
S_0	ignore	ignore	ignore	S_0	S_1	S_0
S_1	empty	ignore	ignore	S_2	S_1	S_0
S_2	acc-bf	acc-bf	acc-bf	S_3	S_2	S_2
S_3	acc-bf	deacc-bf; print	acc-bf	S_3	S_0	S_2

A Figura 4.1 ilustra um exemplo, adaptado de [76], de uma máquina de estados finitos de um impressor de comentários (*Comment Printer*). A MEF possui quatro estados (S_0, S_1, S_2 e S_3) e dez transições. Em cada aresta da transição, o elemento a esquerda do símbolo dois pontos (:) identifica a entrada consumida pela transição, e os caracteres a direita do símbolo representam a saída processada pela transição.

O conjunto de entradas aceitas pela MEF é formado pelos símbolos: "*", "/" e "v", onde "v" significa qualquer caractere diferente de "*" e "/". Estes caracteres especiais quando combinados no formato "/*" e "*/", o conjunto de caracteres entre eles formam um comentário de texto. Estas entradas são processadas a partir do conjunto de saída formado pela definição das seguintes operações [76]: 1) *ignore* - ação nula; 2) *empty-bf* - *buffer* := <>, *i.e.*, vazio; 3) *acc-bf* - *buffer* := *buffer* concatenado com o caractere atual; 4) *deacc-bf* - *buffer* := *buffer* com o caractere mais à direita truncado; 5) *print* - imprime o conjunto de caracteres do *buffer*.

Outro exemplo de MEF é a Figura 4.2 (adaptado de [80]) que apresenta uma determinística, completamente especificada, fortemente conexa e minimal. A notação desta MEF é representada

conforme Definição 1, sendo:

- $I = \{x, y\}$;
- $O = \{0, 1\}$;
- $S = \{S_0, S_1, S_2, S_3\}$;
- $\int_O = \{(S_0 \times x \to 1), (S_0 \times y \to 0), ..., (S_3 \times x \to 1), (S_3 \times y \to 0)\}$;
- $\int_T = \{(S_0 \times x \to S_2), (S_0 \times y \to S_1), ..., (S_3 \times x \to S_2), (S_3 \times y \to S_3)\}$.

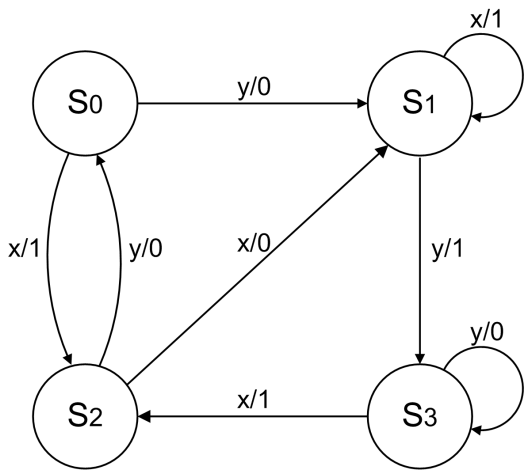

Figura 4.2: Exemplo de MEF M

As funções de saída e transição são complementarmente representadas na Tabela 4.2. Além disso, a MEF contém os seguintes parâmetros: número de estados $(n) = 4$, de entradas $(k) = 2$, de saídas $(l) = 2$, e de transições $(t) = 8$.

Tabela 4.2: Função de saída e transição da MEF M

$\int_O \to \int_T$	x	y
S_0	$1 \to S_2$	$0 \to S_1$
S_1	$1 \to S_1$	$1 \to S_3$
S_2	$0 \to S_1$	$0 \to S_0$
S_3	$1 \to S_2$	$0 \to S_3$

Com o propósito de esclarecer a aplicabilidade de determinados métodos de geração de sequência, faz-se necessário a definição de algumas propriedades acerca de uma MEF M:

Definição 2. *M é "não determinística" caso ela possua ao menos um estado com duas ou mais transições para um mesmo símbolo de entrada definido no alfabeto de entrada. Caso contrário, ela é "determinística".*

Definição 3. *M é completamente especificada, ou simplesmente "completa", se para todo estado de M existe uma transição definida para cada símbolo do alfabeto de entrada. Do contrário, é dito que M é parcialmente especificada ou "parcial".*

Definição 4. *M é "inicialmente conexa" se para cada estado S_i existe uma sequência de entrada a qual leva M de S_0 para S_i. M é "fortemente conexa" se para cada par de estados (S_i, S_j) existe uma sequência de entrada que leva M de S_i para S_j.*

Definição 5. *M é chamada "minimal"para uma MEF completa (ou "reduzida" para uma MEF parcial) quando ela não tem quaisquer dois estados equivalentes, i.e., para todo par de estados (S_i, S_j), existe ao menos uma sequência de entradas cuja saída produzida seja distinguível;*

A geração de sequências de teste requer a combinação de outras sequências básicas utilizadas como resultado parcial para a obtenção das sequências finais. Algumas sequências básicas citadas na academia e que são utilizadas neste trabalho são definidas a seguir:

- "Conjunto *state cover*" (Q) de uma MEF é um conjunto formado pelas sequências de transferências geradas a partir do estado inicial para cada um dos seus estados, incluindo a sequência ϵ;
- "Conjunto *transition cover*" (P) de uma MEF é um conjunto formado pelas sequências de entrada o qual avalia todas as transições pelo menos uma vez, partindo do estado inicial. Obrigatoriamente, o conjunto $Q \subset P$;
- "Conjunto de identificadores harmonizados" (*harmonized identifiers*) (H_i) ou família de separação (*separating family*) é um conjunto formado pela união das sequências de separação de S_i para cada S_j de M, sendo $i \neq j$. O conjunto final dos conjuntos H_i é denominado conjunto HSI.

Diferentes métodos requerem que as MEFs possuam determinados conjuntos de propriedades mencionadas acima. Desta forma, a escolha do melhor método a ser aplicado em um sistema depende das propriedades acerca da especificação do sistema (aplicabilidade); à classe de defeitos que pretende revelar (eficácia) e; ao tamanho dos conjuntos e sequências de teste geradas (eficiência).

Um método pode ser eficiente em sua geração de sequências de teste, mas não sendo eficaz ao encontrar os defeitos residuais. A relação destas características influencia também no custo do teste, uma vez que se os conjuntos ou sequências de teste gerados forem muito grandes podem resultar na inviabilidade de execução do teste.

4.2 Método HSI

O método HSI (*Harmonized State Identification*) [71] [72] é um método de geração de sequências de teste baseado em MEFs. Ele possui como diferencial a geração de sequências de teste tanto para MEFs parciais quanto para MEFs completas, *i.e.*, qualquer especificação reduzida, sendo o primeiro a implementar tal funcionalidade. Inicialmente, o método foi proposto como uma alternativa para

a caracterização do conjunto W. Por isso, ele é dito uma variação do método W [44], além de ser muito semelhante ao método Wp [66], ambos são métodos de geração de sequências de teste conhecidos pela comunidade acadêmica. Desta forma, o método HSI diferencia-se no critério de seleção do conjunto de caracterização, selecionando um subconjunto do método W, chamado de conjunto HSI. Por ser menor que o conjunto W, o conjunto de sequências de teste final também será menor, o que otimiza o processo de teste final. O método permite garantir a cobertura completa dos defeitos existentes. Característica essencial para o teste de *software* podendo atender um conjunto maior de MEFs. Esse método aplica o conceito de famílias de separação, o qual é um conjunto de identificadores do estado.

O exemplo de implementação do método HSI usado nesta seção é baseado na Figura 4.2. Assim, para exemplificar a aplicação do método HSI, a Tabela 4.3 apresenta os seguintes conjuntos *state cover* e *transition cover* agrupados pelo conjunto de estados da MEF M.

Tabela 4.3: *State cover* e *transition cover* da MEF M

Estado	S_0	S_1	S_2	S_3
State cover Q	ϵ	y	x	yy
Transition cover P	ϵ, x, y	xx, xy	yx, yy	yyx, yyy

Obtidos estes conjuntos, a próxima etapa para a implementação do método HSI requer a criação de uma lista de pares de estados (Figura 4.3) a partir do conjunto finito de estados S da MEF M.

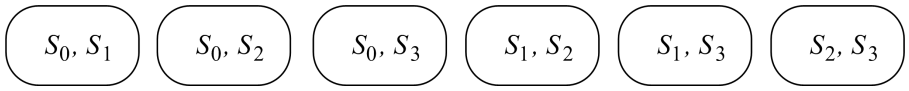

Figura 4.3: Lista dos pares de estados da MEF M

A etapa seguinte é baseada nos resultados apresentados na Tabela 4.4. Cada par de estados definido na Figura 4.3 é representado na coluna "par de estados origem". E em que cada par de estados aplicam-se as entradas aceitas pela MEF M baseada no conjunto de entradas I. Assim, os dois estados realizarão suas respectivas transições, alcançando um novo par de estados. Por exemplo, para a MEF M da Figura 4.2 o primeiro par de estados (S_0, S_1), aplicando a entrada x realiza a transição para o par de estados destino (S_2, S_1) produzindo a mesma saída 1 para ambos os estados e gerando um resultado válido. Entretanto, se o mesmo par de estados recebe como entrada y, a transição para o par de estados destino (S_1, S_3) é executada produzindo uma saída distinta $0/1$. Neste caso o resultado é considerado falho.

Para cada par de estados, o processo visa encontrar a menor sequência de entradas, conhecida como identificador harmonizado, que leva um par de estados origem a um novo par de estados destino produzindo saídas distintas. A ordem do par de estados é irrelevante, *e.g.*, $(S_1, S_2) = (S_2, S1)$. No caso do par de estados (S_0, S_1) o identificador harmonizado do par de estados é y.

Por outro lado, o par de estados (S_0, S_3) quando processa a entrada x realiza a transição para um par de estados ambíguo (S_2, S_2) com resultado válido, consequentemente por ambos produ-

zirem a mesma saída. Esta característica de ambiguidade resulta em descartar este identificador harmonizado, pelo fato de que todos os próximos pares sempre produzirão a mesma saída.

Tabela 4.4: Relação de transitividade dos pares de estados da MEF M

Par de Estados Origem	Entrada	Par de Estados Destino	Saída	Resultado
(S_0, S_1)	x	(S_2, S_1)	$1/1$	válido
(S_0, S_1)	y	(S_1, S_3)	$0/1$	falho
(S_0, S_2)	x	(S_2, S_1)	$1/0$	falho
(S_0, S_2)	y	(S_1, S_0)	$0/0$	válido
(S_0, S_3)	x	(S_2, S_2)	$1/1$	válido
(S_0, S_3)	y	(S_1, S_3)	$0/0$	válido
(S_1, S_2)	x	(S_1, S_1)	$1/0$	falho
(S_1, S_2)	y	(S_3, S_0)	$1/0$	falho
(S_1, S_3)	x	(S_1, S_2)	$1/1$	válido
(S_1, S_3)	y	(S_3, S_3)	$1/0$	falho
(S_2, S_3)	x	(S_1, S_2)	$0/1$	falho
(S_2, S_3)	y	(S_0, S_3)	$0/0$	válido

Todavia, aplicando a entrada y gera uma transição para o par de estados (S_1, S_3) que por sua vez aplicando a entrada x produz um resultado válido ao atingir o par de estados (S_1, S_2). Por outro lado, quando aplicada a entrada y gera um resultado falho para o par de estados ambíguo (S_3, S_3), porém com saídas distintas. Neste outro caso, o identificador harmonizado do par de estados (S_1, S_3) é yy. A Tabela 4.5 apresenta o resultado do conjunto de identificadores harmonizados da MEF M.

Tabela 4.5: Identificadores harmonizados dos pares de estados da MEF M

Par de Estados	(S_0, S_1)	(S_0, S_2)	(S_0, S_3)	(S_1, S_2)	(S_1, S_3)	(S_2, S_3)
Identificadores Harmonizados	y	x	yy	x	y	x

A partir dos identificadores harmonizados dos pares de estados, identificam-se quais as entradas relacionadas a cada estado da MEF contido na lista de pares de estados (Figura 4.3). Tem-se como exemplo o estado S_0 o qual possui como entradas relacionadas a este estado os seguintes identificadores harmonizados y, x, yy. Excluindo as entradas duplicadas e prefixos de outras, o identificador harmonizado final é formado por x, yy.

O resultado do processamento desta etapa é o conjunto de identificadores harmonizados, doravante denominado conjunto HSI, da MEF M seria:

$$HSI = \left\{ \begin{array}{lcl} HSI_0 & = & \{x, yy\} \\ HSI_1 & = & \{x, y\} \\ HSI_2 & = & \{x\} \\ HSI_3 & = & \{x, yy\} \end{array} \right\}$$

A etapa final consiste em executar o conjunto *transition cover* P, e para cada sequência P concatena-se a ela o identificador harmonizado do estado atual no término da execução da sequência

P. Como exemplo tem-se a sequência $P\{yx\}$ (Tabela 4.3). Executando essa sequência de entradas na MEF M da Figura 4.2, o estado final obtido é S_1. Desta forma, o conjunto de sequências de teste obtidas é na forma $yx.H_1$, resultando nas sequências de teste $\{yxx, yxy\}$. A Tabela 4.6 apresenta a concatenação do conjunto P com o conjunto HSI resultando no conjunto TS_{HSI}, também chamada de suíte de teste HSI.

Tabela 4.6: Conjunto TS_{HSI} da MEF M

Estado	S_0			S_1		S_2		S_3	
Conjunto P	ϵ	x	y	xx	xy	yx	yy	yyx	yyy
Conjunto HSI	$\epsilon.H_0$	$y.H_1$	$x.H_2$	$yy.H_3$	$yx.H_1$	$xx.H_1$	$xy.H_0$	$yyx.H_2$	$yyy.H_3$
Conjunto TS_{HSI}	x, yy	yx, yy	xx	$yyx, yyyy$	yxx, yxy	xxx, xxy	$xyx, xyyy$	$yyxx$	$yyyx, yyyyy$

Gerado o conjunto de testes HSI, depois de retiradas as sequências prefixos de outras e adicionando a função r (*reset*) se obtém o seguinte conjunto de testes com tamanho 41:

$$TS_{HSI} = \{ryxx, ryxy, rxxx, rxxy, rxyx, rxyyy, ryyxx, ryyyx, ryyyyy\}$$

4.3 Considerações

Esse capítulo apresentou os conceitos e aspectos gerais relacionados ao modelo de teste usando MEF, utilizada para representar o comportamento e aspectos do sistema sob teste. Além disso, de acordo com as características e propriedades das MEFs implementadas neste trabalho, descreveu-se o método HSI como abordagem para a geração das sequências de teste que compõem o conjunto de testes. Este método HSI estudado foi escolhido por possuir as propriedades desejadas, sendo uma das fundamentais o fato do método interpretar MEFs parcialmente especificadas.

Consequentemente, gerado o conjunto de testes, cada uma das sequências de testes podem ser instanciadas nos chamados casos de teste abstratos, e por sua vez concretizados em *scripts* de teste para uma determinada ferramenta de teste de desempenho.

No Capítulo 5 será detalhada a implementação dos conceitos supracitados através de um estudo de caso. Além da implementação do método HSI, será descrito o ambiente de teste da aplicação, bem como seus modelos de teste e o processo de geração dos casos de teste abstratos. Além disso, serão demonstrados os *scripts* de teste concretizados a partir dos casos de teste abstratos gerados.

5. ESTUDO DE CASO

A fim de avaliar o conjunto de características para teste de desempenho, na perspectiva dos modelos, descrito no Capítulo 3, este capítulo detalha a implementação de um estudo de caso, que apresenta os modelos de teste UML, o processo de transformação destes modelos em modelos formais, a geração dos casos de teste abstratos baseados nos modelos e, sua instrumentalização em *scripts* de testes para determinadas tecnologias de teste de desempenho, *i.e.*, geradores de carga de desempenho, *e.g.*, HP LoadRunner [40] e MS Visual Studio [43]. Além disso, como contribuição deste trabalho este capítulo apresenta uma linha de produto de *software* para teste baseado em modelos. Para apoiar as atividades MBT que serão desenvolvidas para o estudo de caso, um produto para teste de desempenho será derivado desta linha de produto de *software* para MBT.

5.1 Linha de Produto de *Software* para MBT

Existem diversos modelos que podem ser utilizados na implementação da técnica MBT. Cada um destes modelos possui um conjunto de características pertinentes ao seu formalismo, sendo que algumas características diferem de modelo para modelo. Estes modelos podem ser utilizados por ferramentas de teste de desempenho, que por sua vez podem ser utilizadas pelas equipes de teste a fim de atender as particularidades de cada SUT. Este processo exige um profundo conhecimento do formalismo destes modelos, além das variadas ferramentas, aumentando o custo e diminuindo a qualidade do processo de teste de *software*.

Uma boa alternativa para reduzir o custo e o tempo de qualificação de uma equipe de teste seria ter uma única ferramenta, que abranja todas as fases do processo de MBT. Idealizando esta ferramenta, ela seria capaz de descrever o modelo do sistema, gerar os casos e *scripts* de teste, executar os *scripts* de teste e analisar os resultados. Melhor ainda se esta ferramenta pudesse gerar os casos ou *scripts* de teste para diferentes ferramentas de teste (*e.g.* Jmeter [39], LoadRunner [40] e Visual Studio [43], etc.) e/ou diferentes tipos de testes (funcional, desempenho, segurança, etc.) sobre um mesmo SUT. Além disso, é desejável que a equipe de teste faça reuso dos artefatos implementados, (*e.g.*, modelos, componentes de *software*, *scripts* de teste). Assim, neste contexto, torna-se interessante projetar um conjunto de ferramentas (produtos) para MBT derivado de uma Linha de Produto de *Software* (*Software Product Line - SPL*) [23].

Uma SPL busca explorar os pontos comuns entre os sistemas a partir de um determinado domínio, e, ao mesmo tempo gerenciar a variabilidade entre eles [21]. O *Software Engineering Institute* (SEI) [23] assevera que uma abordagem SPL tem três conceitos principais: Desenvolvimento do Núcleo de Artefatos (Engenharia de Domínio), Desenvolvimento do Produto (Engenharia da Aplicação) e o Gerenciamento da Linha de Produto de *Software*. O núcleo de artefatos é a parte principal de uma SPL, e seus componentes pretendem representar, de forma clara, os aspectos comuns e variáveis dos futuros produtos. Assim, seguindo os conceitos da abordagem SPL, novos produtos variantes

podem ser rapidamente criados com base em uma arquitetura comum, modelos, componentes de *software*, etc. Por isso, SPL permite a rápida entrada de um produto no mercado, bem como torna mais fácil à produção em massa dos produtos de uma empresa.

As empresas estão descobrindo que a prática de desenvolvimento de um conjunto de sistemas que possuam características comuns pode, de fato, melhorar quantitativamente em termos de qualidade do produto e satisfação do cliente, atendendo eficientemente a demanda atual de personalização em massa dos produtos de *software*. No entanto, dado o grande número de produtos que podem estar presentes em uma SPL, é necessário controlar a variabilidade e semelhanças entre eles. O gerenciamento da variabilidade é usado para controlar os aspectos variáveis presentes nos produtos da SPL. O modelo de características (*Feature Models*) é um conceito importante para a modelagem da variabilidade. Quando os modelos de características são aplicados para representar a variabilidade nos diferentes domínios, as características (*feature*) da linha de produtos são analisadas e classificadas como: comuns (*commons*), opcionais (*optionals*) ou alternativas (*alternatives*) [22]. Características comuns representam as características que devem estar presentes em todos os produtos da SPL. Também é referenciada na literatura como características obrigatórias (*mandatory*), necessárias (*necessary*), ou características essenciais (*kernel*). As características opcionais representam as características que são suportadas por alguns produtos na SPL, e as características alternativas, também conhecidas na literatura como variantes, representam as características que são mutuamente exclusivas na SPL. Neste tipo de características, um conjunto de características são alternativas quando apenas uma delas pode estar presente em cada um dos produtos da SPL.

Uma das abordagens que implementa uma SPL é a $\mathrm{PL}e\mathrm{T}s$ PL [28]. A $\mathrm{PL}e\mathrm{T}s$ é uma linha de produto de *software* que utiliza uma ferramenta para modelagem, gerenciamento e derivação automatizada de produtos (ferramentas MBT). Os produtos derivados visam automatizar a geração, execução e coleta dos resultados do processo de MBT. A $\mathrm{PL}e\mathrm{T}s$ PL permite combinar estas atividades do processo MBT com diferentes tipos de testes de *software*, gerando ferramentas para MBT derivadas da linha de produto. O objetivo da $\mathrm{PL}e\mathrm{T}s$ PL não é apenas a reutilização de artefatos para tornar mais fácil e rápido o desenvolvimento de um novo produto da família, mas também para melhorar a criação, execução e a coleta dos resultados de testes [3]. Ela foi desenvolvida com a intenção de ser usada por engenheiros de *software*, programadores e engenheiros de teste, auxiliando o processo de definição e execução dos casos de teste e *scripts* de teste.

A Figura 5.1 (adaptado de [3]) apresenta o modelo de características da $\mathrm{PL}e\mathrm{T}s$ que representa a definição e visualização do escopo da linha de produto. Este modelo representa algumas das características que poderiam estar presentes em uma variante do *software*. No primeiro nível do modelo existem quatro características principais:

- *Parser* - representa uma das principais atividades de MBT, a construção e interpretação do modelo. É uma característica obrigatória e tem duas características dependentes: *UML - FSM* e *UML - PN*. Cada um dos *parsers* é usado para extrair as informações a partir dos modelos UML a fim de gerar um modelo formal (FSM ou PN);

- *Test Case Generation* - representa a etapa de geração dos casos de teste. É uma característica obrigatória e tem três características dependentes: *Functional Testing*, *Performance Testing* e *Security Testing*, um deles deverá ser selecionado em cada variante do *software*. O teste de segurança possui uma característica obrigatória dependente *UIO Method* [69]. Enquanto que o teste de desempenho possui dependência com *HSI Method* [71] [72]. Ambos são métodos de geração de sequências de teste com o objetivo de obter um conjunto de casos de teste a ser testado;

- *Script Generation* - representa a etapa de geração dos cenários e *scripts* de teste. É uma característica opcional, pois algumas ferramentas são capazes apenas de gerar casos de teste, *e.g.*, teste de segurança não tem uma ferramenta (até onde conseguimos identificar) que utiliza *scripts* para executar teste. Essa característica possui duas características dependentes: *Jmeter Script Generation*, *LR Script Generation* e *VS Script Generation*. Estas características representam, respectivamente, Jmeter [39], LoadRunner [40] e Visual Studio [43] (ferramentas de desempenho que usam *scripts* para realizar a execução do teste);

- *Execution* - representa a etapa de execução dos *scripts* de testes sobre o SUT, e também a coleta dos resultados produzidos durante a execução do teste a fim de compará-los com o oráculo de teste, *i.e.*, etapa da análise dos resultados. Esta característica possui características dependentes semelhantes à etapa anterior: *Jmeter Script Execution*, *LR Script Execution* e *VS Script Execution*. Ela é uma característica opcional, pois nem todas as ferramentas de teste possuem as funcionalidades de execução e/ou análise dos resultados.

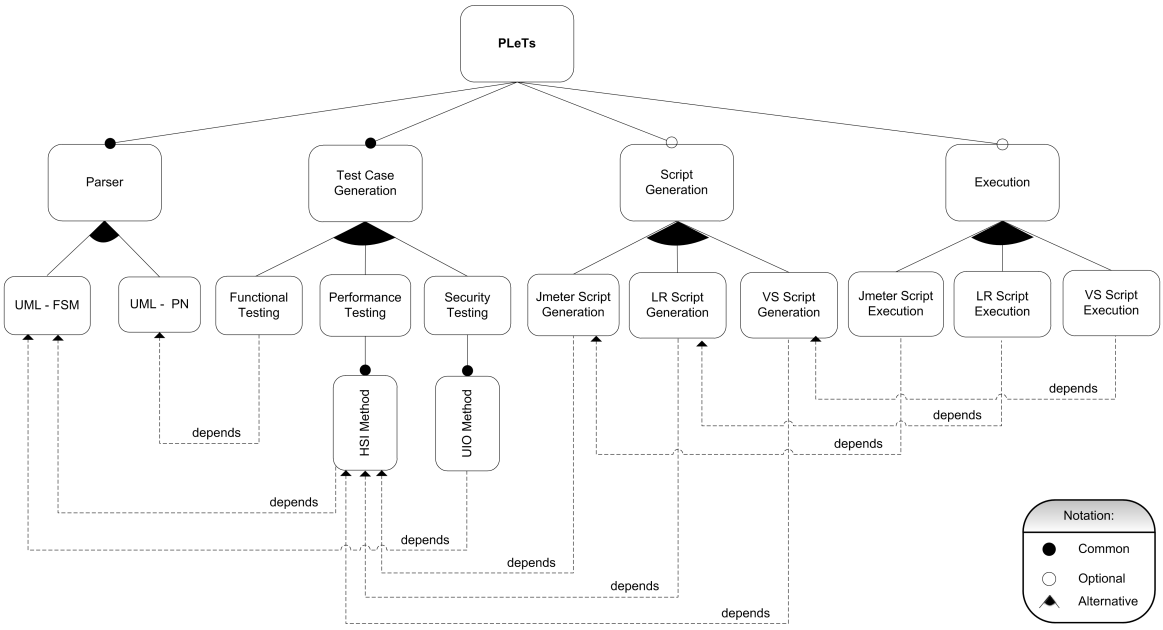

Figura 5.1: Modelo de características da $\mathrm{PL}e\mathrm{T}s$ PL (adaptado de [3])

É importante destacar que existem dependências (*depends*) entre algumas características (ver a linha tracejada na Figura 5.1). Por exemplo, se algum produto derivado da $\mathrm{PL}e\mathrm{T}s$ PL seleciona

a característica *Execution* e a característica variante *LoadRunner*, ele deve também selecionar a característica *Script Generation* e a característica variante *LoadRunner*, pois a ferramenta não é capaz de executar os testes sem um *script* de teste compatível.

Outro ponto importante é que o modelo de características pode ser estendido para suportar novas técnicas de teste ou de ferramentas, adicionando novas características dependentes das quatro principais características. Por exemplo, se alguém quiser adicionar novas características para trabalhar com a ferramenta SilkPerformer [81], poderia incluir novas características dependentes para as características principais *Script Generation* e *Execution*.

Para desenvolver a ferramenta com o propósito de representar a flexibilidade do modelo de características da $\mathrm{PL}e\mathrm{T}s$ PL, a arquitetura da ferramenta $\mathrm{PL}e\mathrm{T}s$ é baseada no conceito de *plug-ins* [82], que permite extensibilidade e flexibilidade para incorporar novas características. Com base nisto, a ferramenta permite selecionar, em tempo de execução, cada *plug-in* (representado por uma característica) que é necessário para executar uma atividade MBT e assim gerar um novo produto derivado da $\mathrm{PL}e\mathrm{T}s$ PL.

A Figura 5.2 apresenta o modelo de classes da ferramenta $\mathrm{PL}e\mathrm{T}s$ que reflete as características da linha de produto para MBT apresentado na Figura 5.1 da $\mathrm{PL}e\mathrm{T}s$ PL. Embora o modelo possua várias classes para derivar produtos $\mathrm{PL}e\mathrm{T}s$, as principais classes são: *BasePlugIn*, *ParserPlugIn*, *TestCaseGenerator*, *ScriptGeneration* e *ExecutionPlugIn*. Vale ressaltar que as funcionalidades de cada *plug-in* correspondem às características descritas anteriormente do modelo de características da Figura 5.1, exceto o *plug-in BasePlugIn* que é uma classe abstrata em que os demais *plug-ins* herdam dela.

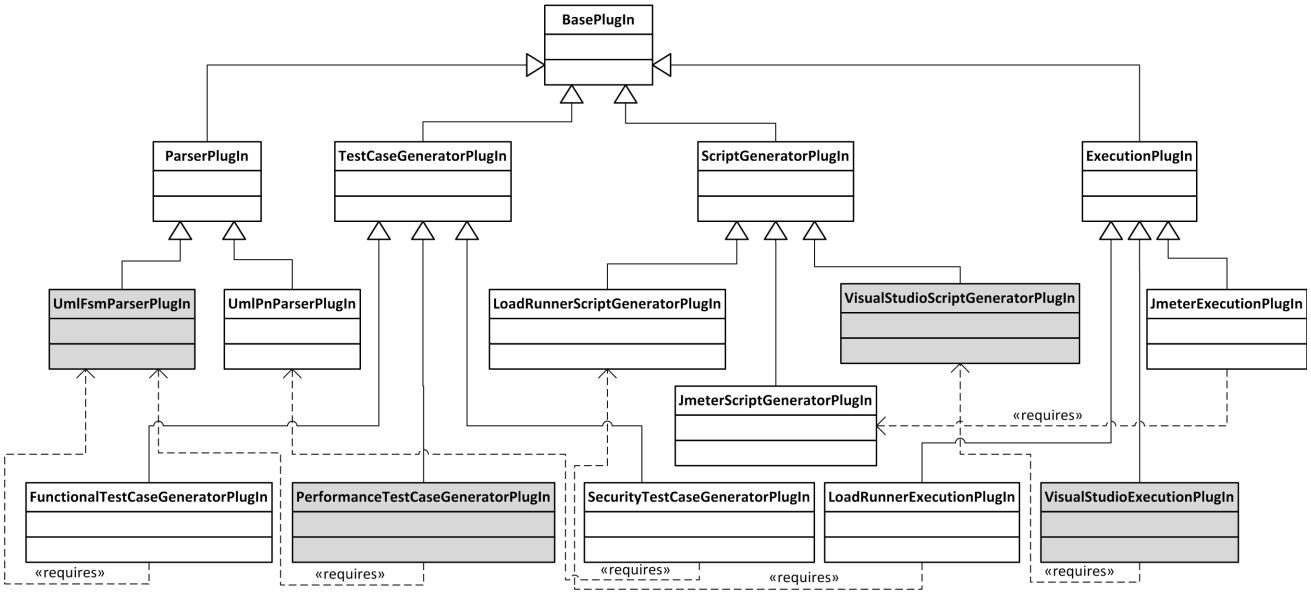

Figura 5.2: Modelo de classes UML da ferramenta $\mathrm{PL}e\mathrm{T}s$ (adaptado de [4])

5.2 Ferramenta MBT para Teste de Desempenho em Aplicações *Web*

Esta seção descreve como derivar um novo produto da $\mathrm{PL}e\mathrm{T}s$ PL. Este produto será desenvolvido com o intuito de executar teste de desempenho para aplicações *web* baseado em MBT. Para isto, inicialmente alguns requisitos foram analisados para compor as características necessária para a geração da ferramenta de teste de desempenho (produto a ser derivado da $\mathrm{PL}e\mathrm{T}s$). Entre os requisitos fundamentais que esta ferramenta deveria atender estão listados na Tabela 5.1.

Tabela 5.1: Requisitos funcionais da ferramenta de teste de desempenho (produto $\mathrm{PL}e\mathrm{T}s$)

ID	Requisitos
RF01	Deve usar os próprios modelos UML do SUT, previamente desenvolvidos, como modelo de teste de entrada para a ferramenta.
RF02	O conjunto de casos de teste gerados deve garantir a cobertura completa dos defeitos da aplicação.
RF03	Deve ser capaz de gerar *scripts* de teste de desempenho para a ferramenta Visual Studio.
RF04	Deve permitir a execução automática dos *scripts* de teste de desempenho para a ferramenta Visual Studio.

A partir destes requisitos, uma análise foi realizada de forma a mapeá-los com as respectivas características já existentes na $\mathrm{PL}e\mathrm{T}s$ PL. Esta análise permite identificar quais *plug-ins* podem ser reutilizados e quais novos *plug-ins* devem ser desenvolvidos. A análise identificou o reuso do *plug-in* `UmlFsmPlugIn`, mas também foi necessário o desenvolvimento de dois novos *plug-ins*: `VisualStudioScriptGeneratorPlugIn` e `VisualStudioExecutionPlugIn` a fim de atender aos requisitos `RF03` e `RF04` da ferramenta (Tabela 5.1). Além disto, foi necessário refatorar o *plug-in* `PerformanceTestCaseGeneratorPlugIn` com o objetivo de agregar uma nova subcaracterística, o método HSI, em razão da sua dependência com o *plug-in* `UmlFsmPlugIn` e a fim de atender ao requisito `RF02` (Tabela 5.1) da ferramenta.

A Figura 5.3 apresenta um novo modelo de classes UML do produto para teste de desempenho para aplicações *web* derivado da $\mathrm{PL}e\mathrm{T}s$ PL, doravante denominado $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$, o acrônimo atribuído ao nome do produto derivado corresponde aos artefatos (*plug-ins*) selecionados para configurá-lo: **U**ml**F**sm, ***P**erformance*, **V**isual **S**tudio. Este modelo de classes é uma instância do modelo de classes UML da $\mathrm{PL}e\mathrm{T}s$ (Figura 5.2), *i.e.*, é um novo produto derivado apenas das características destacadas (*plug-ins* na cor cinza) na $\mathrm{PL}e\mathrm{T}s$ PL. Como se pode observar na Figura 5.3, a ferramenta gerada é formada pelos *plug-ins*:
`UmlFsmParserPlugIn`, `PerformanceTestCaseGeneratorPlugIn`, `VisualStudioExecutionPlugIn` e `VisualStudioScriptGeneratorPlugIn`.

Nas seções a seguir serão descritos os detalhes de cada uma das funcionalidades que compõe este novo produto $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$. Além de demonstrar o uso desta ferramenta em um estudo de caso de uma aplicação *web*. Na Seção 5.4 tem por objetivo descrever o processo de modelagem UML implementando o conjunto de características para teste de desempenho proposto. Na Seção 5.5 detalha a etapa do *plug-in* `UmlFsmParserPlugIn`, enquanto que na Seção 5.6 destaca a etapa do *plug-in* `PerformanceTestCaseGeneratorPlugIn`. Estas duas etapas correspondem com a contribuição desta pesquisa com o modelo de características da $\mathrm{PL}e\mathrm{T}s$ PL. Já a Seção 5.7

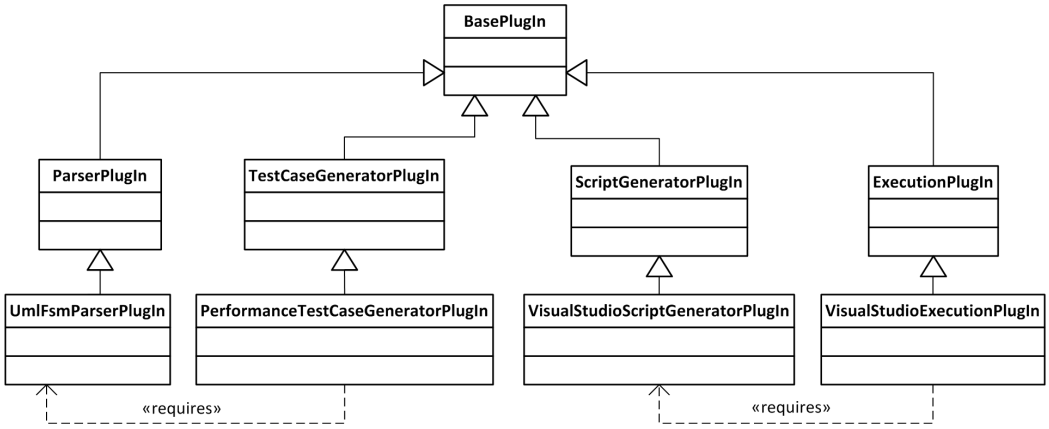

Figura 5.3: Modelo de classes UML do produto $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$ (adaptado de [4])

visa sintetizar a implementação do modelo de características para ferramentas de teste de desempenho (Seção 3.3) de autoria de outra dissertação de mestrado, a qual tem foco nas outras duas principais características da $\mathrm{PL}e\mathrm{T}s$ PL, os *plug-ins* `VisualStudioScriptGeneratorPlugIn` e `VisualStudioExecutionPlugIn`.

5.3 Aplicação *Skills*

Para demonstrar a implementação do conjunto de características para o teste de desempenho, além de exemplificar a utilização da ferramenta $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$, será utilizada uma aplicação *web* como estudo de caso chamada *Workforce Planning: Skill Management Prototype Tool*, doravante denominada *Skills*. A aplicação *Skills* tem por objetivo gerenciar os perfis profissionais de funcionários de uma empresa. Algumas das funcionalidades apresentadas pela aplicação é o gerenciamento do cadastro de habilidades, certificações e experiências de funcionários. Este *software* foi desenvolvido em linguagem de programação *Java*, utilizando o SGBD MySQL para persistência de dados e o TomCat como servidor *web*.

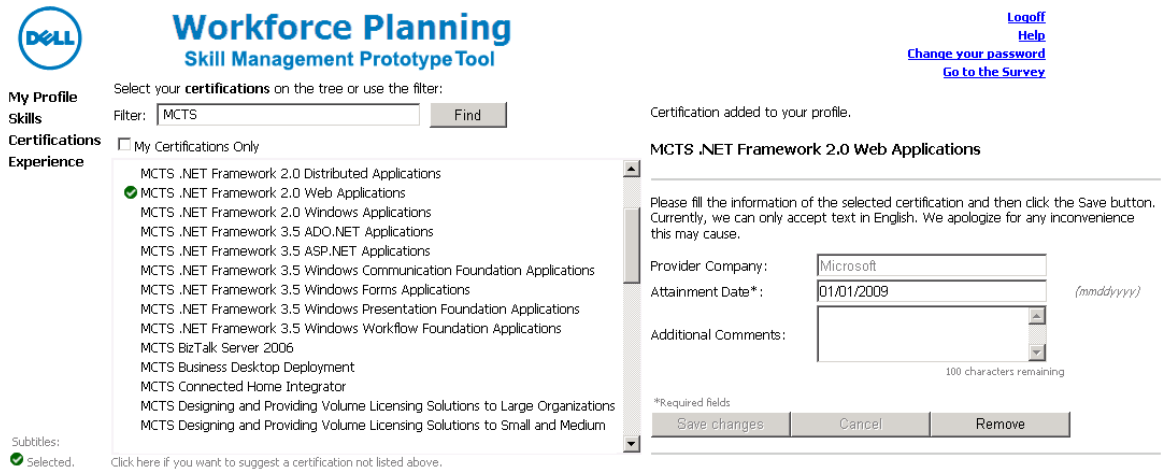

Figura 5.4: Aplicação *Skills*

A Figura 5.4 demonstra o usuário realizando o cadastro de uma certificação. Para isto, o usuário

clicou na opção "*Certifications*" no menu de navegação à esquerda da figura, após realizou um filtro ("*Filter*") informando o termo "MCTS" e submetendo este filtro ao clicar no botão "*Find*". Por sua vez, o sistema processou e renderizou uma árvore com a lista de certificações encontradas com o termo pesquisado. Nesta árvore, o usuário selecionou a opção "*MCTS .NET Framework 2.0 Web Applications*", para então o sistema habilitar o formulário localizado a direita da figura, permitindo preencher os campos Empresa Certificadora ("*Provider Company*"), Data de Realização ("*Attainment Date*") e Comentários Adicionais ("*Additional Comments*") a fim de persistir a certificação informada ao clicar no botão "*Save Changes*".

A Tabela 5.2 aborda os principais requisitos funcionais que a aplicação *Skills* implementa para atender seus objetivos.

Tabela 5.2: Requisitos funcionais da aplicação *Skills*

ID	Funcionalidade	Requisitos
RF01	Gerenciar Habilidades	O sistema deve permitir ao usuário adicionar e/ou editar suas habilidades a cerca de técnicas, metodologias, tecnologias, *software*, *hardware* ou línguas, informando o nível de proficiência, ano em que adquiriu a habilidade e o último ano que a implementou.
RF02	Gerenciar Certificações	O sistema deve possibilitar ao usuário adicionar e/ou editar suas certificações, sejam elas de formação acadêmica ou profissional realizadas.
RF03	Gerenciar Experiências	O sistema deve dar autonomia ao usuário para registrar suas experiências nas diversas áreas da indústria.
RF04	Alterar Senha	O sistema deve permitir ao usuário alterar sua senha.
RF05	Visualizar Perfil	O sistema deve possibilitar ao usuário consultar o seu perfil baseado nos dados cadastrados na aplicação (habilidades, certificações e experiências).

5.4 Modelagem UML

Trabalhos anteriores [28] [83] descrevem os componentes implementados na $\mathrm{PL}e\mathrm{T}s$ para o escopo em teste de segurança e teste funcional. Nesta abordagem, o escopo da aplicação é estendida para um novo domínio, teste de desempenho, reutilizando e/ou refatorando componentes previamente desenvolvidos na $\mathrm{PL}e\mathrm{T}s$ PL. Desta forma, um novo produto é gerado combinando características relacionadas a este determinado tipo de teste.

Neste sentido, o início do processo de geração dos casos de teste é a criação de um modelo de teste. Para isto, baseado no modelo UML SPT (*Schedulability, Performance and Time*) [24] [25], é adicionado ao diagrama de casos de uso (*Use Case Diagram - UC*), informações relacionadas aos cenários de teste, perfis do usuários e carga de trabalho (*Workload*). Em seguida, cada caso de uso é decomposto em um diagrama de atividades (*Activity Diagram - AD*), o qual descreve as ações do usuário para a realização de uma determinada tarefa do sistema. Em ambos os diagramas são inseridos informações e características relacionadas ao teste de desempenho, representadas por meio de estereótipos e rótulos (*tags*).

Como mencionado acima, ao usar modelos UML, estereótipos são a base para incluir as infor-

mações necessárias para geração dos casos de teste. Ao modelo da aplicação devem-se adicionar estereótipos de desempenho ao modelo, com o intuito de avaliar a escalabilidade da aplicação em cenários com maior demanda e concorrência. Assim, foram definidos seis estereótipos de desempenho conforme [28] [84]:

- «PApopulation» - Definida no ator do diagrama de casos de uso. Possui seis *tags*:
  - TDpopulation - representa o número de usuários virtuais que estarão executando a aplicação;
  - TDhost - define o endereço de rede ou caminho a ser executada a aplicação;
  - TDrampUpUser - determina o número de usuários virtuais que acessam o sistema a cada intervalo de tempo (TDrampUpTime) durante a inicialização do teste;
  - TDrampUpTime - representa o intervalo de tempo de cada ciclo de acesso a novos usuários ao sistema durante a inicialização do teste;
  - TDrampDownUser - determina o número de usuários virtuais que finalizam seu acesso ao sistema a cada intervalo de tempo (TDrampDownTime) durante a finalização do teste;
  - TDrampDownTime - representa o intervalo de tempo de cada ciclo de finalizações de acessos dos usuários ao sistema durante a finalização do teste.
- «PAprob» - define a probabilidade de distribuição do usuário na execução de determinadas atividades da aplicação. Esta informação é aplicada na associação entre os usuários e seus casos de uso no diagrama de casos de uso;
- «PAtime» - tempo de execução de determinado fluxo de atividades. Representado nos casos de uso no diagrama de casos de uso;
- «PAthinkTime» - denota o tempo ocioso entre a disponibilidade de execução da atividade e o início real da atividade pelo usuário da aplicação, como por exemplo o tempo de preenchimento de um formulário até sua submissão. Definido em cada transição da atividade no diagrama de atividades;
- «PAparameters» - define os parâmetros necessários para a execução dinâmica de determinadas atividades que fazem uso de informações de um domínio específico ou pré-definido. Este estereótipo é formado por três *tags:*
  - TDaction - determina o endereço em que os dados serão submetidos ou do *link* acessado pelo ator;
  - TDmethod - define o método de requisição HTTP utilizado pelo servidor;
  - TDparameters - denota os dados (parâmetros) enviados ao servidor para processar a requisição HTTP.

- «PAcounter» - representa os contadores de desempenho que se deseja avaliar durante a execução do teste.

O objetivo principal destes estereótipos é auxiliar a equipe de teste em duas atividades: melhorar o desempenho da aplicação e; prover informações ao modelo que possibilite a automação da geração dos *scripts* para o teste de desempenho. Para a geração de testes de desempenho, dois tipos de diagramas UML foram necessários para representar este tipo de teste, o diagrama de casos de uso e o diagrama de atividades.

Baseado nos requisitos funcionais da aplicação *Skills* definido na Tabela 5.2 o diagrama de casos de uso da Figura 5.5 representa estas funcionalidades dividida em quatro casos de uso, são eles: gerenciar habilidades, gerenciar experiências, gerenciar certificações e alterar senha. O modelo demonstra o comportamento de iteração de dois perfis de usuários com o sistema: gerente RH e empregado. Em termos de funcionalidades os perfis diferem especificamente na execução da atividade alterar senha, atribuída exclusivamente ao ator gerente RH.

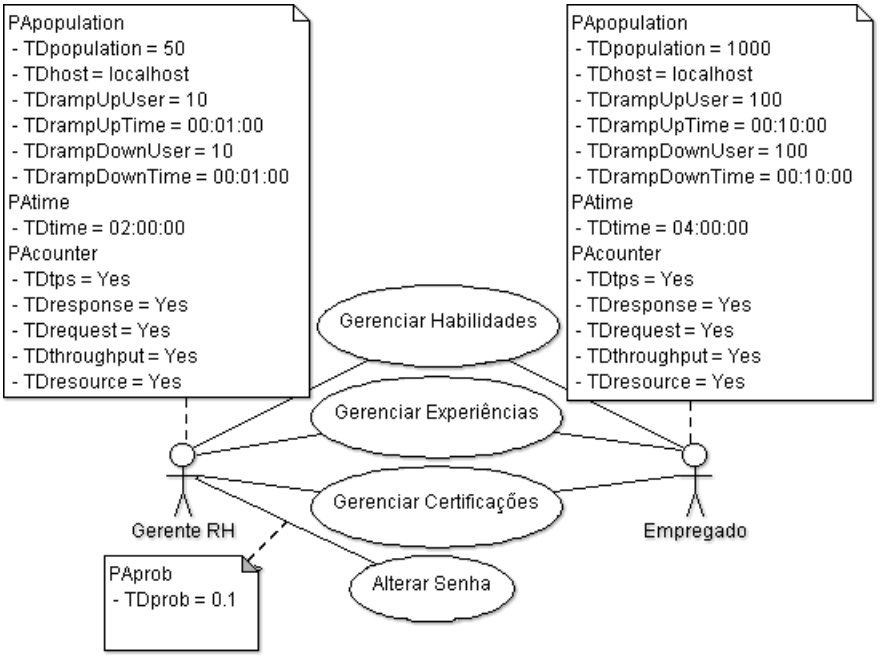

Figura 5.5: Diagrama de casos de uso da aplicação *Skills*

Aos atores definidos no modelo foram atribuídos três estereótipos: `PApopulation`, `PAtime` e `PAcounter`, sendo que basicamente eles diferem nos seguintes aspectos:

1. "Gerente RH" modela um perfil de 50 usuários virtuais, sendo que durante a inicialização do teste são acrescentados 10 usuários (`TDrampUpUser`) a cada minuto (`TDrampUpTime`), totalizando 5 minutos para a inicialização total do teste, para a finalização do teste (`TDrampDownUser` e `TDrampDownTime`) foram configurados os mesmos valores da inicialização. Assim, subtraindo o total de 10 minutos entre a inicialização e finalização do teste, resultando em 1h50min de execução do teste com 50 usuários virtuais concorrentes para uma duração total da execução do teste de 2h (`TDtime`);

2. "Empregado" modela um perfil de 1000 usuários virtuais, sendo que durante a inicialização do teste são acrescentados 100 usuários (`TDrampUpUser`) a cada 10 minutos (`TDrampUpTime`), totalizando 1h40min para a inicialização total do teste, assim como a finalização do teste (`TDrampDownUser` e `TDrampDownTime`) também foram configurados com os mesmos parâmetros. Totalizando 3h20min entre a inicialização e finalização do teste, resultando em 40min de execução do teste com os 1000 usuários virtuais concorrentes para uma duração total de execução do teste de 4h (`TDtime`).

Outra característica importante neste modelo é o parâmetro `TDprob` definido em cada associação entre ator e os casos de uso. Esta informação define a probabilidade de distribuição dos usuários virtuais em cada caso de uso relacionado a ele. A Tabela 5.3 apresenta a distribuição da probabilidade entre os atores e seus casos de uso, conforme demonstrado na Figura 5.5 que destaca a probabilidade do ator "Gerente RH" em executar o caso de uso "Alterar Senha" 10% (`TDprob`), o qual representa na Tabela 5.3 em 5 usuários virtuais dos 50 concorrentes. Vale ressaltar que esta probabilidade pode ser propagada aos casos de testes gerados para cada um dos diagramas de atividades.

É importante relatar que por limitação de espaço e tamanho da imagem (Figura 5.5), optou-se em não comentar todas as associações entre os atores e seus casos de uso. Desta forma, os demais valores de probabilidade apresentados na Tabela 5.3 estão rotulados no diagrama de casos de uso através do rótulo `TDprob`.

Tabela 5.3: Distribuição da probabilidade entre os atores e seus casos de uso

Ator	**Caso de Uso**	**Probabilidade**	**Usuários Virtuais**
Gerente RH	Gerenciar Habilidades	40%	20
	Gerenciar Certificações	30%	15
	Gerenciar Experiências	20%	10
	Alterar Senha	10%	5
			50
Empregado	Gerenciar Habilidades	40%	400
	Gerenciar Certificações	35%	350
	Gerenciar Experiências	25%	250
			1000

A próxima etapa da elaboração do modelo de teste é decompor cada caso de uso em um diagrama de atividades, o qual descreve as ações do usuário para a realização de uma determinada tarefa do sistema. Diagrama de atividades (*Activity Diagram - AD*) ilustra a sequência de atividades ou etapas que determinam um processo (*workflow*) complexo do sistema, com apoio a escolha, iteração e concorrência, como por exemplo um algoritmo ou fluxo de trabalho. Um AD demonstra o fluxo de controle de um componente do sistema, ele é uma variação de uma máquina de estados, na qual os estados são as atividades que representam a execução de operações e as transições são disparadas pela conclusão destas operações [85]. Graficamente, em um AD as elipses alongadas representam os estados de atividades e/ou ações, enquanto que as setas determinam as transições entre elas.

É possível aplicar desvios no fluxo que são representados por um losango. Desta forma, foram desenvolvidos quatro diagramas de atividades correspondentes aos quatro casos de uso apresentados na Figura 5.5.

Tipicamente, um diagrama de atividades representa as etapas que compõem um processo por meio das interações do usuário com o sistema. A Figura 5.6 apresenta o exemplo do cadastro de habilidades que descreve o processo do caso de uso "Gerenciar Habilidade" apresentado na Figura 5.5. Este diagrama representa o comportamento de interação do usuário com a aplicação *Skills*. A primeira interação é "Menu" que habilita as opções de escolha do sistema após seu acesso. Em seguida, determina a seleção do usuário com a opção "Habilidades" no menu do sistema. Então a atividade "Pesquisar Habilidade", permite realizar o filtro de determinada habilidade pela opção de pesquisa. Outra alternativa de realizar este filtro é executar a iteração "Árvore de Habilidades" que possibilita a navegação do usuário pelas diferentes habilidades carregadas na lista da árvore. Independente do caminho escolhido, depois de selecionada uma habilidade é permitido "Adicionar Habilidade" ainda não existente ou "Editar Habilidade" para uma habilidade já existente no cadastro do usuário. Por fim, a iteração "Deslogar" para o usuário sair do sistema. Este diagrama ainda possui quatro elementos de desvios no seu fluxo, os quais permitem a variabilidade de diferentes caminhos que o usuário pode executar ao realizar suas atividades. Maiores detalhes sobre a geração das sequências de teste serão descritos na Seção 5.6.

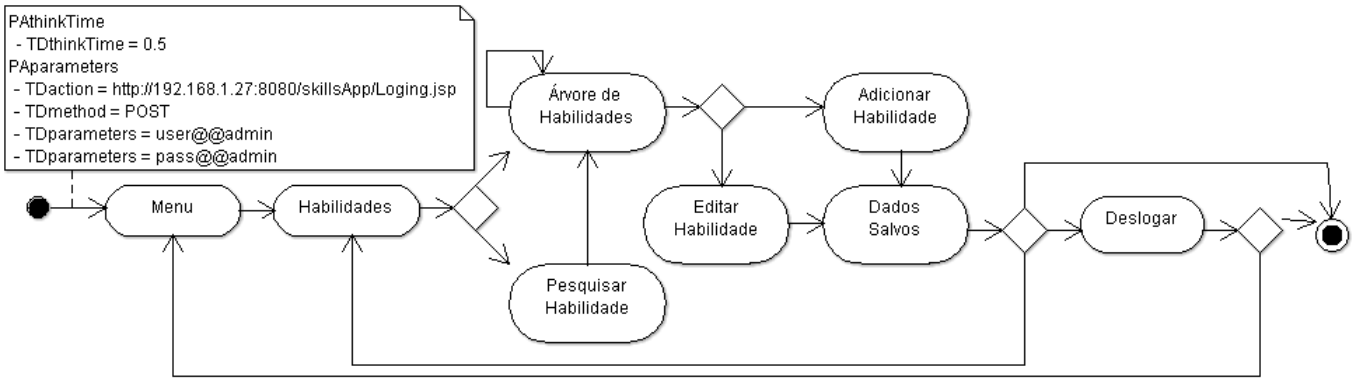

Figura 5.6: Diagrama de atividades do caso de uso "Gerenciar Habilidade"

Com relação às anotações das informações do teste de desempenho, os diagramas de atividades são rotulados basicamente com dois estereótipos: `PAparameters` e `PAthinkTime`. Na Figura 5.6, o exemplo apresenta uma transição anotada com estes estereótipos. O `TDthinkTime` define o tempo de 0.5 segundos para o preenchimento dos dados pelo usuário para a submissão da requisição HTTP, neste caso os dados correspondem aos itens "user" e "pass" da *tag* `TDparameters`. Estes dados serão submetidos para o endereço "http://192.168.1.27:8080/skillsApp/Loging.jsp" anotado na tag `TDaction` usando o método "POST" definido na *tag* `TDmethod`.

Observa-se que a *tag* `TDparameters` é a concatenação de duas informações: nome e valor, separados pelo delimitador "@@". Esta informação poderia ser gerada automaticamente ou ser importada de um arquivo de usuários e senhas, para diferentes cenários que o engenheiro de teste

desejasse testar. É importante destacar que todas as demais transições do modelo possuem suas transições anotadas, por limitação de espaço, optou-se em não comentar todas as transições.

Na Seção 5.5 serão apresentadas as tabelas com a lista de ações e parâmetros utilizados pelas demais transições apresentados nos diagramas de atividades. Assim como, os demais diagramas de atividades decompostos a partir dos casos de uso da Figura 5.5 podem ser consultados no Apêndice A.

Na Tabela 5.4, o conjunto de característica para o teste de desempenho é mapeado de acordo com os estereótipos e rótulos apresentados no estudo de caso. O coluna "Característica" apresenta o conjunto de característica para o teste de desempenho apresentado na Seção 3.2; a coluna "Estereótipo" lista os estereótipos implementados no modelo UML SPT para representar o modelo de teste; a coluna "Rótulo" descreve o nome de cada rótulo (*Tag*) vinculado aos estereótipos, são os rótulos que contém os valores das informações contidas no modelo; e a coluna "Diagrama" faz o mapeamento de qual dos diagramas UML (*UC - Use Case Diagram* ou *AD - Activity Diagram*) foi adicionada a respectiva informação.

Tabela 5.4: Mapeamento das características com os estereótipos

Característica	**Estereótipo**	**Rótulo**	**Diagrama**
Dados	PAparameters	TDdata	AD
Parâmetro	PAparameters	TDparameters	AD
Probabilidade	PAprob	TDprob	UC
Requisição	PAparameters	TDaction	AD
		TDmethod	
Requisições por Segundo	PAcounter	TDrequest	UC
SUT	PApopulation	TDhost	UC
Tempo de Espera	PAthinkTime	TDthinkTime	AD
Tempo de Execução	PAtime	TDtime	UC
Tempo de Finalização	PApopulation	TDrampDownTime	UC
Tempo de Inicialização	PApopulation	TDrampUpTime	UC
Tempo de Resposta	PAcounter	TDresponse	UC
Transação	PAparameters	TDtransaction	AD
Transações por Segundo	PAcounter	TDtps	UC
Usuários de Finalização	PApopulation	TDrampDownUser	UC
Usuários de Inicialização	PApopulation	TDrampUpUser	UC
Usuários Virtuais	PApopulation	TDpopulation	UC
Utilização de Recursos	PAcounter	TDresource	UC
Vazão	PAcounter	TDthroughput	UC

A característica "Dados", por ser opcional no conjunto de característica para o teste de desempenho, não foi adicionada ao modelo UML do estudo de caso. A característica "Transação" possui uma particularidade, pois não possui nenhum estereótipo vinculado a ela. Entretanto, a característica está implicitamente relacionada a cada atividade do diagrama de atividades.

Os diagramas UML têm a capacidade de fornecer uma visão sobre os aspectos mais importantes, a estrutura e o comportamento dos sistemas. Quando utilizados em combinação com anotações, podem ser usados para derivar representações abstratas dos modelos na forma de modelos intermediários. É importante salientar que a qualidade dos modelos está diretamente relacionada ao tempo investido durante a rotulagem dos elementos dos modelos UML.

5.5 Transformação de UML para MEF

Utilizando a ferramenta $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$, os diagramas UML modelados na Seção 5.4 são representados por um arquivo no formato XMI [86], que por sua vez é submetido ao *plug-in* (`UmlFsmParserPlugIn`) a fim de convertê-lo em um modelo formal, *e.g.*, FSM.

Primeiramente, o modelo é analisado e convertido em uma estrutura de dados correspondente aos próprios modelos UML. A próxima etapa é converter este modelo UML analisado em uma nova estrutura genérica, chamada de estrutura intermediária (Figura 5.7). Esta estrutura é responsável pela comunicação entre os *plug-ins* `UmlFsmParserPlugIn` e `PerformanceTestCaseGeneratorPlugIn` através da serialização da estrutura intermediária em formato XML.

Esta nova estrutura foi projetada a fim de resolver o problema de acoplamento existente na conversão entre os modelos UML e os modelos formais, os quais compartilhavam as mesmas classes, impedindo o isolamento dos componentes. Os modelos formais, *e.g.*, FSM, sempre irão ter como entrada a mesma estrutura de dados, *i.e.*, a estrutura intermediária. Assim, o uso desta estrutura facilitaria o desenvolvimento de novos *plug-ins*, pois "todos" passariam a receber o mesmo "formato" de dados. Desta forma, possibilitaria que novos produtos derivados da $\mathrm{PL}e\mathrm{T}s$ PL para MBT possam ter como modelo de entrada diferentes modelos, não se resumindo apenas aos modelos UML.

Outra característica deste processo, é a implementação de um documento XML que contém a hierarquia de equivalências entre os modelos UML e a estrutura intermediária. Assim, novos perfis UML que possam ser analisados pelo *plug-in* `UmlFsmParserPlugIn` ou até mesmo a adição de novos estereótipos ou rótulos aos modelos, permitirão a conversão transparente entre as estruturas.

Vale ressaltar que esta estrutura intermediária (conforme Figura 5.7) foi modelada para o escopo de teste de desempenho.Uma vez que se estendam estes conceitos a outros tipos de testes suportados pela $\mathrm{PL}e\mathrm{T}s$ PL, possivelmente esta estrutura deverá ser refatorada ou estendida, ou até mesmo desenvolvida uma nova estrutura que seja compatível com os novos *plug-ins* implementados.

O modelo de classes da estrutura intermediária apresentado na Figura 5.7 é composto por dez classes. Esta estrutura visa implementar o conjunto de características para teste de desempenho apresentado na Tabela 3.2. No mais alto nível da hierarquia, está a classe "`TestSuit`" que está associado a uma coleção de objetos da classe "`Scenario`". O cenário de teste de desempenho é o artefato que configura uma série de parâmetros do teste, *e.g.*, o número de usuários virtuais representado pela propriedade "`Population`". Além disso, a classe "`Scenario`" possui uma coleção de objetos da classe "`TestCase`". Esta classe, por sua vez, possui a lista de requisições (classe "`Request`") e transações (classe "`Transaction`") executadas pelo caso de teste. Cada "`Request`"

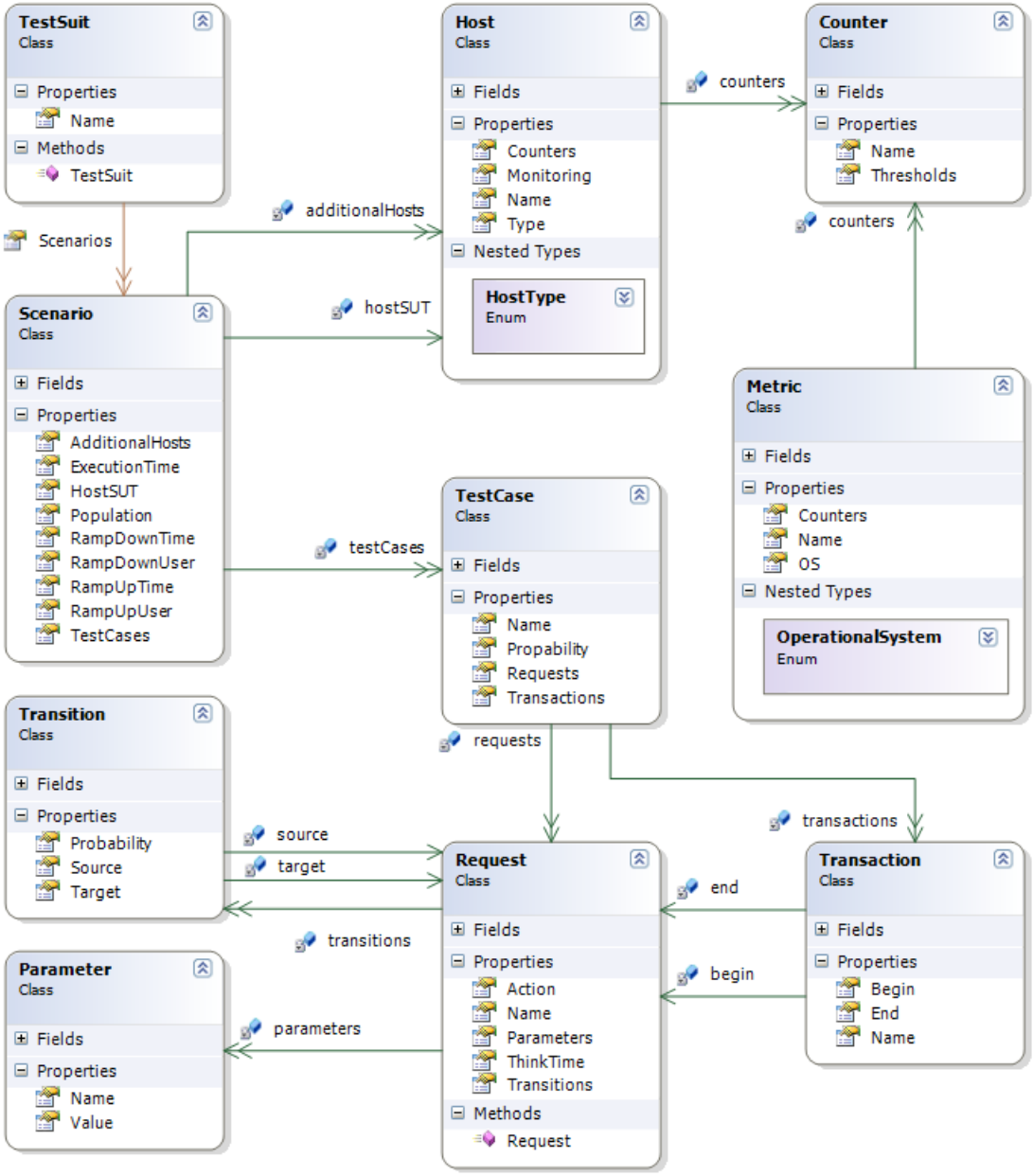

Figura 5.7: Modelo de classes da estrutura intermediária

pode ter associado a ele um ou mais parâmetros (classe "`Parameter`"), e ainda pode ter uma ou mais transações "`Transaction`" vinculadas a cada requisição. As transações são definidas no teste de desempenho com o objetivo de isolar o processamento de diferentes pontos de requisições, a fim de facilitar a identificação dos gargalos do sistema, além de possibilitar mensurar a métrica TPS para estas transações e validar se satisfaz seu critério de aceitação (*e.g.*, SLA (*Service Level Agreement*)). As demais classes são: "`Host`" que está associado ao cenário de teste, determinando o servidor (*host*) em que está hospedado o SUT, além dos demais servidores que possam estar associados ao SUT (*e.g.*, banco de dados); finalizando, as classes "`Counter`" e "`Metric`" que determinam os contadores de desempenho e suas métricas associadas a cada contador que estarão relacionados com os objetos da classe "`Host`".

Prosseguindo com o processo de transformação do modelo UML em MEFs, basicamente limitou-se aos modelos de atividades, por possuírem, essencialmente, as informações relacionadas as interações dos usuários com o SUT. A estratégia adotada é manter as demais informações relacionadas ao teste de desempenho anotadas no modelo de entrada, armazenadas na estrutura intermediária (Figura 5.7). Assim, a proposta é que a MEF seja responsável somente pela geração das sequências de teste que devem ser executadas para cada caso de teste.

O resultado previamente gerado na transformação do diagrama de atividades do caso de uso "Gerenciar Habilidades" mostrado na Figura 5.6 é a MEF correspondente retratada na Figura 5.8.

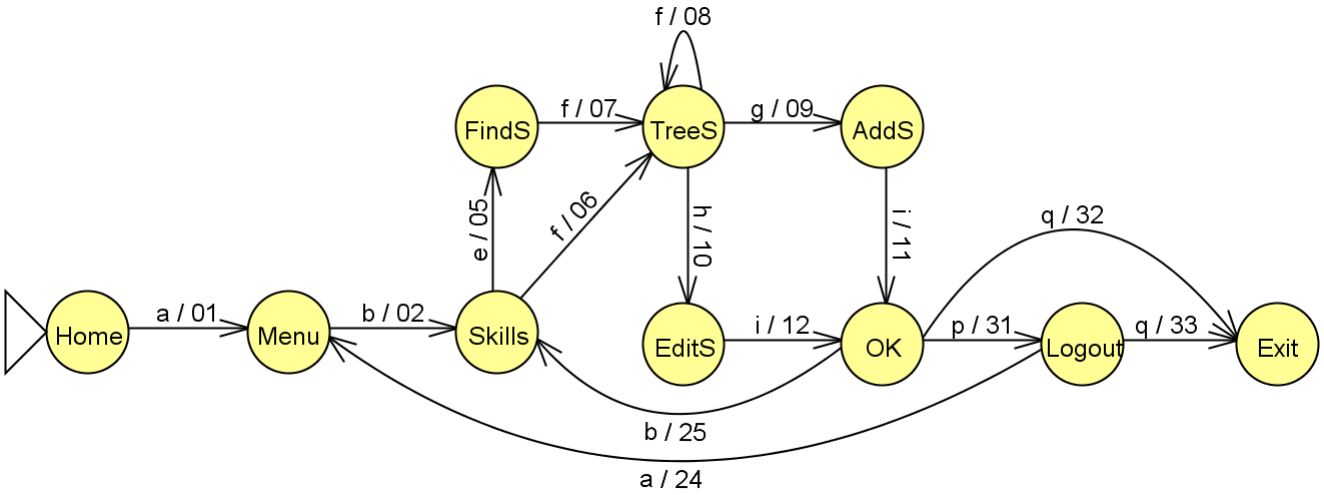

Figura 5.8: MEF gerada a partir do diagrama de atividades da Figura 5.6

O processo de transformação do diagrama de atividades em MEF seguiu os seguintes critérios:

- Os elementos de início e fim do diagrama de atividades foram convertidos em estados extras na MEF, sendo o início denominado "Home" e o fim "Exit";
- Cada atividade do diagrama de atividades foi convertido em um estado correspondente na MEF;
- Cada transição do diagrama de atividades também foi modelado em uma transição correspondente na MEF;
- O elemento de decisão não foi modelado na MEF, entretanto as transições oriundas da decisão foram alteradas na MEF, tendo como estado origem da transição o estado (atividade) que originou o desvio;
- O entrada consumida para cada transição na MEF foi concebida a partir da dependência funcional entre $S_j \rightarrow I$, *i.e.*, o estado destino implica em um novo alfabeto de entrada;
- A entrada gerada para cada transição na MEF foi modelada baseada na dependência entre $(S_i \times S_j \times A \times M \times P) \rightarrow O$, *i.e.*, a combinação do estado origem, estado destino, requisição, método e parâmetros implicam em um novo alfabeto de saída.

A partir da especificação dos critérios de conversão e da adaptação da abordagem de modelagem com MEFs para aplicações *web* proposta por [87], tem-se como resultado para a geração das MEFs da aplicação *Skills* a Tabela 5.5 (a) que define as entradas processadas pelas MEFs. As demais MEFs geradas a partir dos diagramas de atividades modelados para a aplicação *Skills* podem ser consultados no Apêndice B.

Tabela 5.5: Informações extraídas dos diagramas de atividades utilizadas pelas MEFs

(a) Entradas das MEFs

ID	Entrada
a	Menu
b	Habilidade
c	Certificação
d	Experiência
e	Pesquisar Habilidade
f	Árvore Habilidade
g	Adicionar Habilidade
h	Editar Habilidade
i	Dados Salvos
j	Pesquisar Certificação
k	Árvore Certificação
l	Adicionar Certificação
m	Editar Certificação
n	Adicionar Experiência
o	Visualizar Experiência
p	Deslogar
q	Exit
r	Senha
s	Senha Atual
t	Nova Senha

(b) Requisições de saídas das MEFs

ID	TDAction
A01	http://192.168.1.27:8080/skillsApp/Loging.jsp
A02	http://192.168.1.27:8080/skillsApp/ServletArvore
A03	http://192.168.1.27:8080/skillsApp/skillFields.jsp?type=new
A04	http://192.168.1.27:8080/skillsApp/ServletArvore?nomeArvore=skills&funcao=getFilhosADoPai&pai=Business Related&modo=showAll
A05	http://192.168.1.27:8080/skillsApp/skillFields.jsp?skill=Governance&type=new
A06	http://192.168.1.27:8080/skillsApp/skillsIndex.jsp
A07	http://192.168.1.27:8080/skillsApp/exit.jsp
A08	"exit"
A09	http://192.168.1.27:8080/skillsApp/certificationsIndex.jsp
A10	http://192.168.1.27:8080/skillsApp/certificationsFields.jsp?type=new
A11	http://192.168.1.27:8080/skillsApp/certificationsFields.jsp?certification=_3D_Studio_MAX&type=new
A12	http://192.168.1.27:8080/skillsApp/certificationsIndex.jsp
A13	http://192.168.1.27:8080/skillsApp/experiencesIndex.jsp
A14	http://192.168.1.27:8080/skillsApp/experiencesFields.jsp?numExperience=-1&experience=New_experience&type=new
A15	http://192.168.1.27:8080/skillsApp/ServletLista
A16	http://192.168.1.27:8080/skillsApp/changePW.jsp
A17	http://192.168.1.27:8080/skillsApp/ServletPassword

Da mesma forma, a Tabela 5.5 (b) apresenta as diferentes requisições, rotuladas através da *tag* `TDAction` aos diagramas de atividades. Conforme exemplo apresentado na Figura 5.6 em que destaca a requisição "http://192.168.1.27:8080/skillsApp/Loging.jsp" que é apresentada na tabela com o ID "A01", o qual é representado na Figura 5.8 como uma saída processada na transição entre os estados "Home" e "Menu". É importante destacar que o estereótipo `PAparameters`, além da *tag* `TDaction`, especifica a *tag* `TDmethod`, *e.g.* ver Figura 5.6, para determinar o método de submissão dos dados, assumindo somente dois valores: "POST" ou "GET".

Seguindo na mesma linha, a Tabela 5.6 (a) apresenta a lista dos parâmetros (`TDparameters`) rotulados em cada transição dos diagramas de atividades, que servirão para instrumentalizar os casos de teste abstratos após a geração das sequências de teste a partir da MEF.

Finalizando o processo de conversão dos diagramas de atividades em MEFs, baseado nos dados das tabelas preliminares apresentadas é possível correlacioná-las para obter a lista das saídas processadas pelas MEFs, conforme Tabela 5.6 (a). A combinação entre os estados de origem, estado de destino, parâmetros, métodos e requisições formam um novo elemento do alfabeto de saída.

Tabela 5.6: Parâmetros e saídas utilizadas pelas MEFs

(a) Parâmetros de saída das MEFs

ID	TDparameters
P01	user@@admin; pass@@admin
P02	link@@skills
P03	skills@@Governance
P04	null
P05	nomeArvore@@skills; funcao@@getNodosPai; modo@@showAll
P06	Proficiency_level_field@@Expert; Acquired_Date_field@@2000; Last_time_used_field@@2008; postType@@add; edit_fields@@Proficiency_level_field, Acquired_Date_field, Last_time_used_field
P07	Proficiency_level_field@@Expert; Acquired_Date_field@@2000; Last_time_used_field@@2009; postType@@edit;edit_fields@@Last_time_used_field; editbutton@@Edit; removebutton@@Remove
P08	link@@certification
P09	certification@@_3D_Studio_MAX
P10	nomeArvore@@certifications; funcao@@getNodosPai; modo@@showAll
P11	Provider_Company_field@@BrainBench; Attainment_Date_field@@02/02/2002; Additional_Comments_field@@; postType@@add; edit_fields@@Attainment_Date_field; savebutton@@Save changes
P12	Provider_Company_field@@BrainBench; Attainment_Date_field@@02/02/2003; Additional_Comments_field@@; postType@@edit; edit_fields@@Attainment_Date_field; savebutton@@Save changes; removebutton@@Remove
P13	companyNameField@@IBM; industryField@@Academic; areaField@@Engineering; roleLevelField@@Intermediate; dateBegin@@2006; dateEnd@@2009; postType@@add; edit_fields@@; relatedSkills@@; experienceId@@; playedRole@@
P14	id@@253947000995517
P15	senhaAtual@@admin; senhaNova@@admin2; user@@admin
P16	pwAtual@@admin; pwNovo@@admin2; pwNovoC@@admin2 resposta@@yes; postType@@save
P17	user@@admin; pass@@admin2

(b) Saídas das MEFs

ID	Estado Origem	Estado Destino	Método	Requisição	Parâmetros
01	Home	Menu	P	A01	P01
02	Menu	Skills	P	A02	P02
03	Menu	Certif	P	A02	P08
04	Menu	Exper	G	A13	P04
05	Skills	FindS	G	A03	P03
06	Skills	TreeS	P	A02	P05
07	FindS	TreeS	P	A04	P04
08	TreeS	TreeS	P	A05	P04
09	TreeS	AddS	P	A05	P06
10	TreeS	EditS	P	A05	P07
11	AddS	OK	G	A06	P04
12	EditS	OK	G	A06	P04
13	Certif	FindC	G	A10	P09
14	Certif	TreeC	P	A04	P04
15	FindC	TreeC	P	A02	P10
16	TreeC	TreeC	P	A11	P04
17	TreeC	AddC	P	A11	P11
18	TreeC	EditC	P	A11	P12
19	AddC	OK	G	A12	P04
20	EditC	OK	G	A12	P04
21	Exper	AddE	P	A14	P13
22	Exper	ViewE	P	A15	P14
23	AddE	OK	P	A13	P04
24	Logout	Menu	P	A01	P01
25	OK	Skills	P	A02	P02
26	OK	Certif	P	A02	P08
27	OK	Exper	G	A13	P04
28	ViewE	ViewE	G	A13	P04
29	ViewE	Logout	G	A07	P04
30	ViewE	Exit	P	A08	P04
31	OK	Logout	G	A07	P04
32	OK	Exit	P	A08	P04
33	Logout	Exit	P	A08	P04
34	Menu	Senha	G	A16	P04
35	Senha	Atual	P	A17	P15
36	Atual	Nova	P	A16	P16
37	Nova	Senha	G	A16	P04
38	Nova	Logout	P	A07	P04
39	Nova	Exit	P	A08	P04
40	Logout	Menu	P	A01	P17

5.6 Geração dos Cenários e Casos de Teste Abstratos

Uma vez gerada a estrutura intermediária, a ferramenta $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$ utiliza esta estrutura de dados como parâmetro de entrada para o *plug-in* `PerformanceTestCaseGeneratorPlugIn` conforme apresentado no modelo de classes na Figura 5.3. A primeira atividade deste *plug-in* é o processo de geração da MEF, o qual foi descrito em detalhes na transformação de modelo UML em MEF na Seção 5.5.

Estruturalmente, esta MEF é armazenada em um modelo de classes do próprio *plug-in* `PerformanceTestCaseGeneratorPlugIn`. Todavia, nem todas as informações contidas na es-

trutura intermediária (Figura 5.7) são convertidas, resumindo-se às informações necessárias para a geração das sequências de testes. O processo transformação dos modelos UML em MEFs foi descrito na Seção 5.5.

Geradas as MEFs baseadas nos modelos UML da aplicação *Skills*, a próxima etapa é a geração das sequências de teste. Para realizar esta tarefa podem ser usadas diversas tecnologias de geração de teste conforme apresentadas na taxonomia da Figura 2.2, tais como: geração randômica, *model checking* [88], entre outros. A abordagem apresentada no Capítulo 4 faz uso de métodos de geração de sequências de teste através de MEFs, *e.g.*, HSI *Method* [71].

O método HSI, descrito na Seção 4.2, foi escolhido por possuir as propriedades desejadas, *e.g.* o fato do método interpretar MEFs parcialmente especificadas. Este método foi implementado visando contribuir com uma nova funcionalidade para a linha de produto da $\mathrm{PL}e\mathrm{T}s$ PL , por meio do *plug-in* `PerformanceTestCaseGeneratorPlugIn`.

Após as MEFs serem previamente geradas pela ferramenta $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$, foi necessário aplicar um algoritmo para redução das MEFs [98], pois além de serem MEFs parcialmente especificadas, tratam-se de MEFs não reduzidas. Transformadas em MEFs reduzidas foi possível obter as sequências de teste de cada MEF executando o método HSI implementado e resultando no conjunto de testes HSI. Assim, a partir desse conjunto de testes, a ferramenta permite que possam ser concretizados em cenários e casos de testes abstratos.

A Tabela 5.7 apresenta as sequências de teste geradas a partir das entradas processadas para cada MEF. A tabela apresenta a lista dos diagramas de atividades, e a referência para a MEF gerada para cada diagrama, e por sua vez, a última coluna apresenta os conjuntos TS_{HSI}.

Tabela 5.7: Conjuntos TS_{HSI} das MEFs geradas da aplicação *Skills*

Diagrama de Atividades	**MEF**	**Conjunto HSI das Sequências de Teste**
Gerenciar Habilidades	Figura 5.8	$\{abeff, abfff, abfgiq, abfhib, abfhiq, abfgibf, abfgipq, abfgipab\}$
Gerenciar Certificações	Figura B.1	$\{acjkk, ackkk, ackmic, ackmiq, acklick, ackliqa, acklipac, acklipqa\}$
Gerenciar Experiências	Figura B.2	$\{adod, adoq, adnid, adniq, adopq, adnipq, adnipa, adopad\}$
Alterar Senha	Figura B.3	$\{arstr, arstq, arstpq, arstpar\}$

Estas sequências de teste geradas podem ser transformadas em uma descrição equivalente a casos de teste em linguagem natural, ou seja, os chamados casos de teste abstratos. O resultado esperado na transformação dos modelos UML SPT da aplicação *Skills* são os cenários e casos de teste abstratos, doravante denominado simplesmente de casos de teste abstratos. O caso de teste abstrato contém as informações necessárias das tarefas a serem realizadas pelo usuário. Enquanto que o cenário de teste abstrato possui as informações relacionadas ao contexto do teste e a distribuição da carga de trabalho entre os casos de teste que compõem o cenário.

A Figura 5.9 apresenta um caso de teste abstrato da funcionalidade "Gerenciar Habilidades" baseado nas sequências de teste geradas pelo conjunto TS_{HSI}, conforme Tabela 5.7. A sequência de teste adotada para exemplificar a instrumentalização do caso de teste é formada pela seguinte sequência de entradas aceitas pela da MEF Figura 5.8: $\{abfgipq\}$.

O caso de teste abstrato apresentado na Figura 5.9 implementa parte das características apresentadas no conjunto de características para teste de desempenho apresentados na Seção 3.2. Essencialmente, aquelas características provenientes dos diagramas de atividades, conforme apresentado na Tabela 5.4.

```
#Caso de Teste: Gerenciar Habilidades 7 - abfgipq
1. Menu
   <<TDmethod:POST>> <<TDaction:http://192.168.1.27:8080/skillsApp/Loging.jsp>>
   <<TDparameters:[user@@admin|pass@@admin]>>
   <<TDthinkTime : 5>>
   <<TDtransaction:[Home:Menu|Home:Exit]>>
2. Habilidades
   <<TDmethod:POST>> <<TDaction:http://192.168.1.27:8080/skillsApp/ServletArvore>>
   <<TDparameters:link@@skills>>
   <<TDthinkTime:5>>
   <<TDtransaction : [Habilidades:Adicionar Habilidade]>>
3. Árvore de Habilidades
   <<TDmethod:POST>> <<TDaction:http://192.168.1.27:8080/skillsApp/ServletArvore>>
   <<TDparameters : [nomeArvore@@skills|funcao@@getNodosPai|modo@@showAll]>>
   <<TDthinkTime:5>>
   <<TDtransaction : [Habilidades:Adicionar Habilidade]>>
4. Adicionar Habilidade
   <<TDmethod:POST>>
   <<TDaction:http://192.168.1.27:8080/skillsApp/skillFields.jsp?skill=Governance\&type=new>>
   <<TDparameters:[Proficiency\_level\_field@@Expert|Acquired\_Date\_field@@2000|
     Last\_time\_used\_field@@2008|postType@@add|edit\_fields@@Proficiency\_level\_field,
     Acquired\_Date\_field, Last\_time\_used\_field]>>
   <<TDthinkTime:15>>
   <<TDtransaction : [Adicionar Habilidade:Dados Salvos]>>
5. Dados Salvos
   <<TDmethod:GET>>
   <<TDaction:http://192.168.1.27:8080/skillsApp/skillsIndex.jsp>>
   <<TDthinkTime:5>>
   <<TDtransaction : [Dados Salvos:Deslogar]>>
6. Deslogar
   <<TDmethod:GET>>
   <<TDaction:http://192.168.1.27:8080/skillsApp/exit.jsp>>
   <<TDthinkTime:4>>
   <<TDtransaction : [Deslogar:Exit]>>
7. ‘‘Exit’’
   <<TDaction:exit>>
```

Figura 5.9: Caso de teste abstrato da funcionalidade “Gerenciar Habilidades”

Os casos de teste abstratos fazem uso de uma abordagem hierárquica, onde as atividades são enumeradas e estruturadas de acordo com a dependência entre as atividades do AD. Outro detalhe que é possível observar na descrição dos casos de teste abstratos é a variação dos valores adicionados aos parâmetros de cada atividade, demonstrando a flexibilidade de configuração dos modelos.

Esta mesma abordagem poderia ser estendida para transmitir o paralelismo e a sincronização modelados no diagrama de atividades por meio dos elementos UML *Fork* e *Join*. Desta forma, se uma atividade pertence a um determinado nível, ele deve cumprir todos os requisitos antes de proceder para a próxima atividade.

Por sua vez, a Figura 5.10 apresenta um cenário de teste abstrato do ator "Gerente RH". A quantidade de cenários de testes gerados a partir de um modelo UML está relacionado diretamente a quantidade de atores adicionados ao modelo, *i.e.*, para o estudo de caso da aplicação *Skills* foram gerados dois cenários de teste abstratos.

```
Nome do Cenário de Teste : Gerente RH
## Configuração do Teste
Usuários Virtuais : <<TDpopulation:50>>
Host do SUT : <<TDhost:localhost>>
Tempo de Execução : <<TDtime:7200>>
Tempo de Inicialização : <<TDrampUpTime:60>>
Usuários de Inicialização : <<TDrampUpUser:10>>
Tempo de Finalização : <<TDrampDownTime:60>>
Usuários de Finalização : <<TDrampDownUser:10>>
## Distribuição dos Casos de Teste:
<<TDprob:0.4>> <<TDpopulation:20>>
1. Gerenciar Habilidades
1.1. Gerenciar Habilidades 1 - abeff
1.2. Gerenciar Habilidades 2 - abfff
1.3. Gerenciar Habilidades 3 - abfgiq
1.4. Gerenciar Habilidades 4 - abfhib
1.5. Gerenciar Habilidades 5 - abfhiq
1.6. Gerenciar Habilidades 6 - abfgibf
1.7. Gerenciar Habilidades 7 - abfgipq
1.8. Gerenciar Habilidades 8 - abfgipab
<<TDprob:0.3>> <<TDpopulation:15>>
2. Gerenciar Certificações
2.1. Gerenciar Certificações 1 - acjkk
2.2. Gerenciar Certificações 2 - ackkk
2.3. Gerenciar Certificações 3 - ackmic
2.4. Gerenciar Certificações 4 - ackmiq
2.5. Gerenciar Certificações 5 - acklick
2.6. Gerenciar Certificações 6 - ackliqa
2.7. Gerenciar Certificações 7 - acklipac
2.8. Gerenciar Certificações 8 - acklipqa
<<TDprob:0.2>> <<TDpopulation:10>>
3. Gerenciar Experiências
3.1. Gerenciar Certificações 1 - adod
3.2. Gerenciar Certificações 2 - adoq
3.3. Gerenciar Certificações 3 - adnid
3.4. Gerenciar Certificações 4 - adniq
3.5. Gerenciar Certificações 5 - adopq
3.6. Gerenciar Certificações 6 - adnipq
3.7. Gerenciar Certificações 7 - adnipa
3.8. Gerenciar Certificações 8 - adopad
<<TDprob:0.1>> <<TDpopulation:5>>
4. Alterar Senha
4.1. Alterar Senha 1 - arstr
4.2. Alterar Senha 2 - arstq
4.3. Alterar Senha 3 - arstpq
4.4. Alterar Senha 4 - arstpar
## Contadores de Desempenho:
Transações por Segundo : <<TDtps:Yes>>
Tempo de Resposta : <<TDresponse:Yes>>
Requisições por Segundo : <<TDrequest:Yes>>
Vazão: <<TDthroughput:Yes>>
Utilização de Recursos : <<TDresource:Yes>>
```

Figura 5.10: Cenário de teste abstrato do ator "Gerente RH"

Um cenário de teste de desempenho agrega informações relacionadas ao contexto do teste e o conjunto dos casos de testes que devem ser testados, incluindo a distribuição do número de usuários virtuais para cada caso de teste. Desta forma, a figura está dividida em três blocos: 1) Configuração - carrega as características genéricas que são aplicadas a todo contexto do teste, basicamente, informações oriundas do modelo UML de diagrama de casos de uso. 2) Distribuição - são vinculados as diferentes sequências de testes geradas pelas MEFs. Observa-se que no cabeçalho de cada caso de teste abstrato constam as informações de probabilidade e sua respectiva quantidades de usuários virtuais propagadas. 3) Contadores - constam os dados de quais contadores de desempenho devem ser mensurados para o cenário de teste abstrato.

5.7 Instrumentalização dos Cenários e *Scripts* de Teste

Esta seção visa resumir a implementação dos *plug-ins* `VisualStudioScriptGeneratorPlugIn` e `VisualStudioExecutionPlugIn`, desenvolvidos como parte integrante dos objetivos específicos de outro projeto de pesquisa [5]. Além disso, visa demonstrar a geração dos cenários e *scripts* de teste de desempenho para a ferramenta MS Visual Studio, por meio da mesma ferramenta $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$ (conforme Figura 5.3).

Baseados nos cenários e *scripts* de teste abstratos apresentados na Seção 5.6, a próxima etapa tem por finalidade gerar as instâncias destes cenários e casos de teste abstratos. Estas instâncias são chamadas de casos de teste concretos ou executáveis, pois dependem da escolha da tecnologia utilizada para a geração dos cenários e *scripts* de teste, tais como: HP LoadRunner [40] e MS Visual Studio [43], entre outros. Para o estudo de caso da aplicação *Skills* foi utilizada a tecnologia MS Visual Studio baseada nas características da ferramenta $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$ gerada.

A abordagem proposta é a geração de casos de teste abstratos com o objetivo de que possam ser convertidos em *scripts* de teste para um conjunto de ferramentas de teste de desempenho, ao invés de gerar diretamente os casos de teste executáveis para uma determinada ferramenta, *e.g.*, HP LoadRunner. Adicionando esta etapa inicial ao processo de automação do teste de desempenho, possibilitando flexibilidade na escolha da ferramenta ou tecnologia na etapa de execução dos casos de teste de acordo com a demanda do projeto de *software* em desenvolvimento ou ainda o conhecimento técnico dos engenheiros de desempenho sobre determinadas tecnologias.

Uma das vantagens na geração de casos de teste abstratos é a possibilidade de converter estas informações em *scripts* que serão interpretados por geradores de carga, tais como: HP LoadRunner, MS Visual Studio ou qualquer outra ferramenta que utilize *scripts* para automação do teste. Em outras palavras, permite criar *scripts* de teste, independentemente da ferramenta ou tecnologia, sendo necessário apenas implementar um novo *plug-in* para a ferramenta $\mathrm{PL}e\mathrm{T}s$ a fim de converter os casos de teste abstratos em instâncias de teste executáveis para uma ferramenta de teste de desempenho específica.

5.7.1 *Scripts* Visual Studio

O módulo de teste de desempenho da ferramenta MS Visual Studio, estrutura seus cenários e *scripts* de teste em formato XML, dividindo-os em dois arquivos: 1) `LoadTest` - responsável por armazenar as informações de configuração do teste, distribuição dos perfis de carga de trabalho entre os *scripts* de teste, além dos contadores de desempenho que serão monitorados pela ferramenta; 2) `WebTest` - possui as informações referentes à interação do usuário com a aplicação, incluindo dados das requisições HTTP geradas, bem como seus parâmetros e as transações definidas entre as requisições.

A Figura 5.11 apresenta um trecho do código XML do cenário de teste gerado pela ferramenta $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$. Este cenário de teste concretizado foi instrumentalizado com as informações oriundas do cenário de teste abstrato gerado na etapa anterior, conforme Figura 5.10.

```
<LoadTest xmlns="http://microsoft.com/schemas/VisualStudio/TeamTest/2010" Description=""
    Name="LoadTestGerenteRH" Owner="" storage="ProjetoSkills.loadtest" Priority="2147483647"
    CssProjectStructure="" CssIteration="" DeploymentItemsEditable="" TraceLevel="None"
    Enabled="true" CurrentRunConfig="Run Settings" Id="1ee66e05-f282-483f-8d98-eb246791de49">
  <Scenarios>
    <Scenario Name="ScenarioGerenteRH" DelayBetweenIterations="0" PercentNewUsers="0"
        IPSwitching="true" TestMixType="PercentageOfTestsStarted" MaxTestIterations="0"
        ApplyDistributionToPacingDelay="true" DelayStartTime="0" AllowedAgents=""
        DisableDuringWarmup="false" >
      <ThinkProfile Value="0" Pattern="On"/>
      <LoadProfile Pattern="Step" InitialUsers="0" MaxUsers="50" StepUsers="10"
          StepDuration="0" StepRampTime="60"/>
      <BrowserMix>...</BrowserMix>
      <TestMix>...</TestMix>
      <NetworkMix>...</NetworkMix>
    </Scenario>
  </Scenarios>
  <CounterSets>
    <CounterSet Name="LoadTest" CounterSetType="LoadTest" LocId="">...</CounterSet>
    <CounterSet Name="Controller" CounterSetType="Controller" LocId="CS_Controller">...
        </CounterSet>
    <CounterSet Name="Agent" CounterSetType="Agent" LocId="CS_Agent">...</CounterSet>
  </CounterSets>
  <RunConfigurations>
    <RunConfiguration Name="Run Settings1" Description="" ResultsStoreType="Database"
        TimingDetailsStorage="AllIndividualDetails" SaveTestLogsOnError="true"
        SaveTestLogsFrequency="0" MaxErrorDetails="200" MaxErrorsPerType="1000"
        MaxThresholdViolations="1000" MaxRequestUrlsReported="1000" UseTestIterations="false"
        RunDuration="7200" WarmupTime="300" CoolDownTime="0" TestIterations="100"
        WebTestConnectionModel="ConnectionPerUser" WebTestConnectionPoolSize="50"
        SampleRate="5" ValidationLevel="High" SqlTracingConnectString=""
        SqlTracingConnectStringDisplayValue="" SqlTracingDirectory=""
        SqlTracingEnabled="false" SqlTracingMinimumDuration="500"
        RunUnitTestsInAppDomain="true">...</RunConfiguration>
  </RunConfigurations>
</LoadTest>
```

Figura 5.11: XML do cenário de teste gerado para o Visual Studio (*.LoadTest)

Entre as diversas características instrumentalizadas entre os artefatos de teste, destacam-se as informações registradas na *tag* `LoadProfile`, responsável por configurar o perfil de carga do teste, onde: a propriedade `MaxUser` corresponde à *tag* `TDpopulation`; a atributo `StepUsers` equivale à *tag* `TDrampUpUser`; e a atributo `StepRampTime` diz respeito à *tag* `TDrampUpTime`. Outra *tag* que sofreu alterações foi a `RunConfiguration` com a atributo `RunDuration` correlacionada com a *tag* `TDtime`. O processo de instrumentalização do cenário de teste é baseado em um *template*. Desta forma, as demais informações contidas no cenário, que não possuem procedência a partir do cenário abstrato, são informações padrões para qualquer cenário gerado pela ferramenta MBT para teste de desempenho.

A Figura 5.12 apresenta um trecho do código XML do *script* de teste gerada pela ferramenta $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$. Esse *script* de teste foi instanciado a partir do caso de teste abstrato apresentado na Figura 5.9.

Basicamente, o *script* de teste é composto pelas diversas requisições (*Requests HTTP*) que formam a sequência de teste utilizadas para compor o caso de teste abstrato. Entre as características correlacionadas entre os artefatos, destaca-se para o exemplo apresentado: 1) *Tag* `Request` - atributo `Method` equivale à *tag* `TDmethod`; atributo `Url` corresponde à *tag* `TDaction`; atributo

ThinkTime está correlacionado à *tag* TDthinkTime; 2) *Tag* FormPostParameter - atributos Name e Value dizem respeito à *tag* TDparameters.

```
1  <WebTest xmlns="http://microsoft.com/schemas/VisualStudio/TeamTest/2010" Priority="2147483647"
       Name="Gerenciar Habilidades 7" Id="b946df25-9d1b-4d37-ae55-e5f9ceb8a6ef" Enabled="True"
       Owner="" CssProjectStructure="" CssIteration="" Timeout="0" WorkItemIds="" Description=""
       CredentialUserName="" CredentialPassword="" PreAuthenticate="True" Proxy=""
       StopOnError="False" RecordedResultFile="">
2    <Items>
3      <TransactionTimer Name="Menu">
4        <Items>
5          <Request Method="POST" Url="http://192.168.1.27:8080/skillsApp/Loging.jsp"
               Version="1.1" ThinkTime="5" Timeout="300" ParseDependentRequests="True"
               FollowRedirects="True" RecordResult="True" ResponseTimeGoal="0"  Encoding="utf-8"
               Cache="False" ExpectedHttpStatusCode="0" ExpectedResponseUrl="" ReportingName="">
6            <FormPostHttpBody>
7              <FormPostParameter Name="user" Value="admin" RecordedValue="admin"
                   CorrelationBinding="" UrlEncode="True"/>
8              <FormPostParameter Name="pass" Value="admin" RecordedValue="admin"
                   CorrelationBinding="" UrlEncode="True"/>
9            </FormPostHttpBody>
10         </Request>
11       </Items>
12     </TransactionTimer>
13     <TransactionTimer>...</TransactionTimer>
14   </Items>
15   <ValidationRules>...</ValidationRules>
16 </WebTest>
```

Figura 5.12: XML do *script* de teste gerado para o Visual Studio (*.WebTest)

Assim como no Visual Studio (VisualStudioScriptGeneratorPlugIn), o processo de geração dos cenários e *scripts* de teste para a ferramenta LoadRunner (LoadRunnerScriptGeneratorPlugIn) é bem similar. Ambos compartilham a mesma estrutura de dados de entrada, a qual o *plug-in* PerformanceTestCaseGeneratorPlugIn provê durante a geração dos cenários e casos de teste abstratos.

Por fim, gerados os cenários e *scripts* de testes para a ferramenta Visual Studio, a ferramenta $\mathrm{PL}e\mathrm{T}s-\mathrm{UFPVS}$, por meio do *plug-in* VisualStudioExecutionPlugIn, executa sua última funcionalidade, o qual recebe como entrada os cenários e *scripts* gerados para então automatizar sua execução.

5.8 Resultados

Uma vez gerados os cenários e *scripts* de teste, Figuras 5.11 e 5.12, respectivamente, para a ferramenta Visual Studio, por meio do *plug-in* VisualStudioScriptGeneratorPlugIn e por intermédio do produto $\mathrm{PL}e\mathrm{T}s-\mathrm{UFPVS}$, é executado o *plug-in* VisualStudioExecutionPlugIn, recebendo como parâmetros de entrada os *scripts* de testes produzidos.

A execução do *plug-in* VisualStudioExecutionPlugIn é responsável por inicializar a tecnologia de teste de desempenho escolhida pelo produto gerado, nesse caso o Visual Studio. Ao carregar os *scripts* a execução do mesmo também é inicializada, com base na configuração do cenário de teste gerado.

Na Figura 5.13 apresenta a execução do cenário de teste apresentado na Figura 5.11. O Visual Studio durante sua execução utiliza o programa `Perfmon` (*Performance Monitor*) que é o monitor de atividades do computador para sistemas operacionais Windows ©, para monitorar os contadores de desempenho definidos na configuração do cenário de teste. Uma vez que o `Perfmon` permite criar contatores personalizados, esses podem ser extendidos ou personalizados de acordo com as necessidades do teste. Dessa forma, permitindo flexibilidade no processo de monitoramento do teste de desempenho.

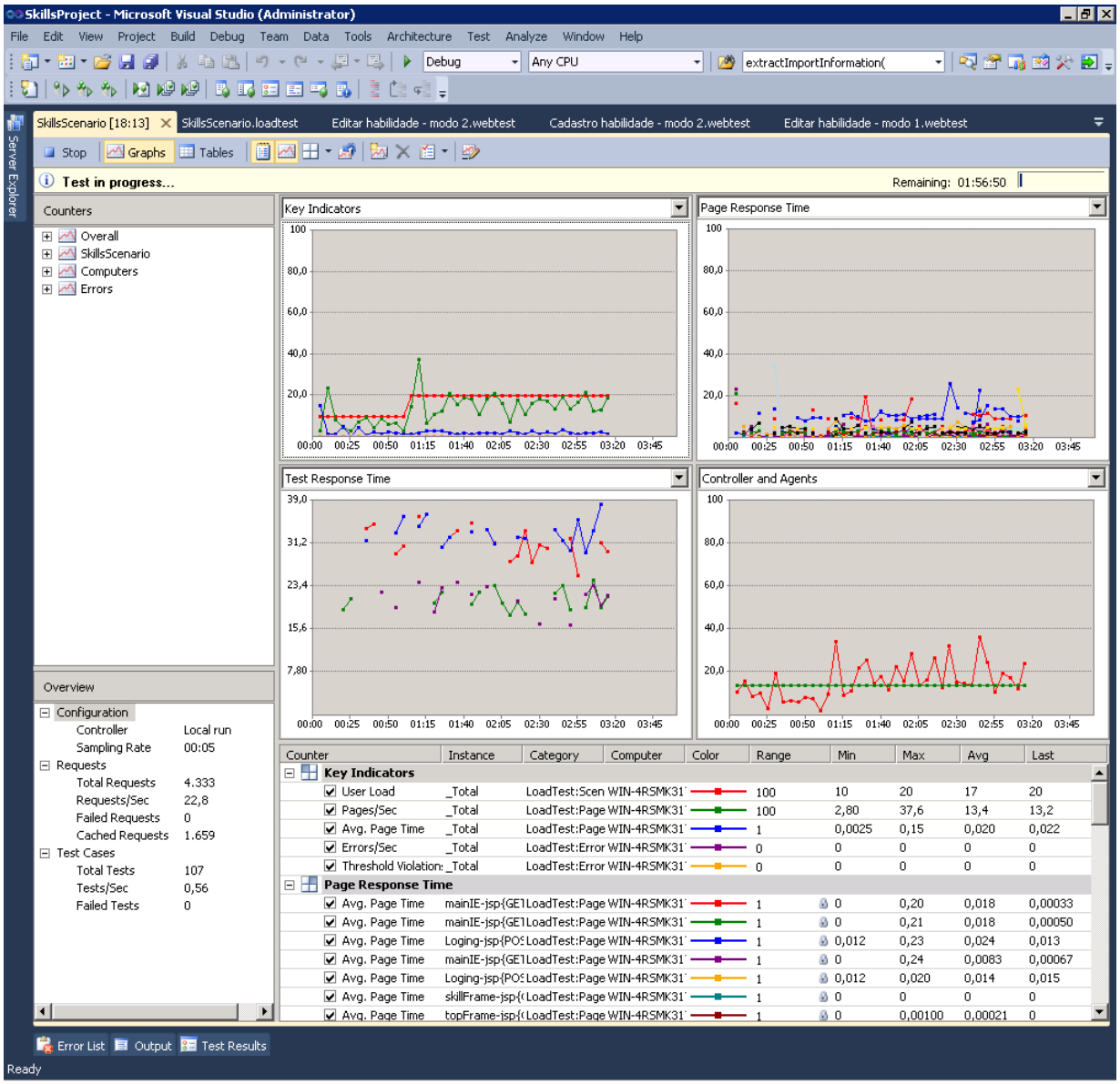

Figura 5.13: Execução do teste de desempenho com Visual Studio

Os contadores podem ser divididos em grupos e/ou categorias. Por exemplo, criar um grupo chamado IIS Web Servers e adicionar todos os contadores importantes para Web Server que executem com IIS (*Internet Information Services*). Essa organização facilita a experiência de monitoramento dos contadores, principalmente, ajudando os engenheiros de teste que não estão acostumados com essa atividade.

Ao término da execução, o Visual Studio apresenta um relatório com o resumo da execução. Informações consideradas padrões são fornecidas para o determinado tipo de teste. Para obter maiores detalhes dos resultados de cada contador de desempenho monitorado, a tecnologia permite exportar os dados para o Excel. É possível, também, gerar relatórios de comparação, *i.e.*, comparar resultados de diferentes execuções dos cenários de teste.

Vale ressaltar que o Visual Studio registra em banco de dados os resultados coletados em cada execução do teste. Para cenários de teste de desempenho que demandem um volume de carga maior, *e.g.* milhares de usuários virtuais, a aplicação permite configurar um controlador de carga (*Load Controller*) e vários agentes geradores de carga (*Load Agent*). Dessa forma, o *Load Controller* centraliza a captura dos dados monitorados, registrando todas as transações e métricas coletadas em um único banco de dados.

Uma das limitações do Visual Studio é sua capacidade de gerar relatórios de análise dos dados coletados. Da mesma forma, os produtos gerados pela $\mathrm{PL}e\mathrm{T}s$, não compreendem um módulo de análise dos resultados validando-os com um oráculo de teste, sendo essa uma das atividades de trabalhos futuros, permitindo uma integração maior com as tecnologias de teste de desempenho.

Com a definição desses casos de teste para o produto $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$, foi possível evidenciar que os aspectos funcionais do conjunto de características para teste de desempenho suportam os dados necessário para automatização desse tipo de teste. E, dessa forma, demonstrar que é possível automatizar casos de teste de desempenho com base nas informações contidas nos modelos de *software*.

5.9 Discussão

Apesar deste capítulo ter apresentado uma abordagem para representar os cenários e casos de teste abstratos, independente da tecnologia do gerador de carga, ainda há espaço para melhorar a representação destes cenários e casos de teste abstratos, *e.g.*, adição de informações dos recursos que são acessados no instante em que forem concretizados, *i.e.*, durante a geração dos cenários e *scripts* de teste. Vale a pena mencionar que o formato sugerido para os cenários e casos de teste abstratos é extensível e modular, e pode ser adaptado para diferentes contextos de teste de desempenho independente da tecnologia do gerador de carga.

Algumas alternativas de como representar os cenários e casos de teste abstratos, a fim de melhorar e validar a abordagem proposta foram discutidas durante a pesquisa, para citar algumas: a) Criação de uma gramática, *e.g.*, representação por meio de uma BNF (*Backus-Naur Form*) [89]; b) Representação através da notação BPMN (*Business Process Model Notation*) [90] [91]; c) Representação por meio de XML, além da criação de uma DTD (*Document Type Definition*) para definir a estrutura do XML e permitir sua validação.

A representação de cenários e casos de teste abstratos está sendo considerada de interesse particular por parte da indústria. Isto se deve em razão das particularidades que apresentam as características dos diferentes geradores de carga (ferramenta de desempenho, *e.g.*, HP LoadRunner).

Assim, os cenários e *scripts* de teste podem ser simplificados e generalizados em um modelo abstrato que capture as características fundamentais para o teste de desempenho. Este é um dos motivos do uso de modelos UML para a modelagem do teste de desempenho por meio dos estereótipos e rótulos da notação.

À exemplo, pode-se citar os diagramas UML modelados para o estudo de caso, que produziram interessantes cenários e casos de teste abstratos. Como não há tanta dificuldade aparente na geração desta representação para modelos mais complexos (*e.g.*, com elementos UML *Fork* e *Join*), um trabalho futuro é adicionar mais informações ao contexto para o cenário de teste abstrato. O objetivo é permitir a criação de cenários e casos de teste abstratos que evidenciem problemas (funcionais e não-funcionais) nas fases iniciais do projeto de *software*. Outra preocupação que merece atenção, é como relacionar aos cenários e casos de teste abstratos mais informações sobre a infraestrutura do SUT, *e.g.*, para citar alguns o uso de ambientes virtualizados ou computação em nuvem (*cloud computing*).

Todavia, adicionar estas informações do teste de desempenho aos modelos de teste é um trabalho manual. Esta atividade de elaboração do modelo de teste é uma das dificuldades em se aplicar a abordagem MBT, se caracterizando como uma desvantagem para todo o processo MBT. Em contrapartida, durante a pesquisa, uma das funcionalidades identificadas que a maioria das ferramentas de teste de desempenho, intitulados geradores de carga, apresentam é o recurso *Record & Playback* - uma técnica que consiste na gravação de todas as interações realizadas pelo usuário com uma aplicação. Os geradores de carga utilizam esta técnica para a geração dos *scripts* de teste automatizados.

Neste contexto, uma das lacunas de pesquisa identificadas é fazer uso do recurso *Record & Playback* para implementar um processo de "Engenharia Reversa", *i.e.*, capturar as interações do usuário para geração dos *scripts* de teste, para então converter estes *scripts* em um formalismo para teste de desempenho. Desta forma, permitiria a criação dos modelos de forma automatizada, restando ao engenheiro de teste ajustar e normalizar o modelo de teste a fim de que atenda aos requisitos de desempenho desejados.

Uma vez gerados os modelos de teste, a próxima atividade é a geração dos casos de teste abstratos, que neste trabalho foi desenvolvido utilizando a abordagem baseada em MEFs. Com relação a geração de casos de teste baseado em MEF, pode-se afirmar que os diferentes métodos existentes, permitem a geração das sequências de teste para MEFs com diferentes propriedades, atendendo diferentes classes de defeitos. Todavia, baseado nos resultados obtidos com as sequências de teste geradas para o estudo de caso, pode-se evidenciar que para o escopo de teste de desempenho, MEFs não sejam a alternativa mais adequada. No entanto, não inviabiliza sua aplicação, mas sim pelo fato de que a sequência gerada não representa, sequencialmente, o modelo de teste proposto para o SUT, *i.e.*, o modelo para teste de desempenho deve gerar sequências de teste determinísticas. Neste sentido, uma boa alternativa seria a implementação do formalismo ESG (*Event Sequence Graphs*), que inicialmente foi proposta para teste de *interfaces* [92], poderia ser experimentada para a modelagem de teste de desempenho.

6. CONSIDERAÇÕES FINAIS E CONTRIBUIÇÕES

Este capítulo apresenta um resumo da dissertação, bem como as considerações finais relacionadas à pesquisa. Destaca ainda, as contribuições científicas e acadêmicas desenvolvidas, assim como as perspectivas para trabalhos futuros a partir dos resultados obtidos.

6.1 Resumo

Este trabalho apresentou a proposta de um "conjunto de características para teste de desempenho" para geração dos cenários e casos de teste abstratos em aplicações *web*. Para isso, conduziu-se um mapeamento sistemático em MBT para analisar as características dos diferentes modelos aplicados em MBT. Baseado nas informações oriundas dos trabalhos selecionados, pode-se analisar as características necessárias para o teste de desempenho, sob uma perspectiva dos modelos de teste. Além disso, a pesquisa norteou a investigação de modelos e métodos para geração de sequências de teste. Desta forma, apresentou a abordagem de geração de casos de teste baseado em MEFs, e ainda, descreveu o processo de geração das sequências de teste através do método HSI.

A partir do "conjunto de características para teste de desempenho" proposto, foi demonstrada a modelagem da aplicação *Skills* como um estudo de caso real, utilizando o perfil UML SPT, que interpretou as características da abordagem para a criação do modelo de teste de desempenho. Por sua vez, estes modelos gerados foram utilizados para desenvolver o produto para teste de desempenho com o Visual Studio, intitulada $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$, ferramenta esta que aplica a abordagem MBT derivada da $\mathrm{PL}e\mathrm{T}s$ PL, a qual implementa uma linha de produto de *software* para teste baseado em modelos.

Neste contexto, a ferramenta $\mathrm{PL}e\mathrm{T}s - \mathrm{UFPVS}$ gerada instanciou a implementação do novo *plug-in* da $\mathrm{PL}e\mathrm{T}s$ PL baseado na técnica de teste baseado em MEFs. Este *plug-in* proposto implementou o método HSI para geração das sequências de teste que compõem o conjunto de testes. Em seguida, gerado este conjunto de testes, cada uma das sequências de teste foram instrumentalizadas nos chamados casos de teste abstratos, e então concretizados em *scripts* de teste para a ferramenta de teste de desempenho Visual Studio. Ao final, foi apresentada uma discussão a respeito das vantagens e desvantagens na representação dos cenários e casos de teste abstratos e sua relação com o conjunto de características para teste de desempenho.

6.2 Contribuições

Com o advento de novas tecnologias e serviços para a Internet, torna-se eminente a expansão de soluções e aplicações *web* que possuam uma grande quantidade de usuários. Diversas técnicas e métodos vêm sendo propostos com o intuito de aplicar modelos e formalismos existentes para modelagem de aplicações ao teste de desempenho [49] [93] [94]. Desta forma, aumenta a necessidade

de verificar e validar a qualidade dos sistemas desenvolvidos, em especial a validação de desempenho destes sistemas. Entretanto, com esta pesquisa foi possível evidenciar que a indústria não adota um padrão de modelagem para o domínio de teste de desempenho, em razão da grande quantidade de modelos e formalismos existentes.

Durante o trabalho desenvolvido nesta dissertação de mestrado, foi possível estudar diversas questões relacionadas ao tema de modelagem de teste de desempenho. Os modelos UCML, UML SPT, UML Marte e MEF estudados mostraram-se insuficientes para atender às expectativas do teste de desempenho, principalmente no que tange o escopo de aplicações *web*. Uma vez que alguns modelos atendem algumas características, e outros atendem diferentes características. Desta forma, foi possível comprovar, baseado no melhor do conhecimento adquirido, que os modelos e formalismos estudados não permitem sua aplicação direta, e não atendem satisfatoriamente as necessidades do teste de desempenho.

Pesquisas futuras podem ser desenvolvidas seguindo as proposições deste trabalho. Desta forma, é possível expandir tal estudo, o qual foi aplicado e validado em um domínio específico de linha de produto de *software* para teste baseado em modelos por meio da $\mathrm{PL}e\mathrm{T}s$. Com o propósito de formalizar este estudo para que outras pessoas consigam aplicá-lo, e não restrito somente à $\mathrm{PL}e\mathrm{T}s$ e ao conhecimento adquirido de maneira empírica. Portanto, pode-se descrever e formalizar este conhecimento a fim de que possa ser utilizado em outros ambientes, não apenas para o domínio de linha de produto, mas aplicado em outros contextos, *e.g. Model-Driven Development* (MDD), *Test Driven Development* (TDD). Desta forma, resultaria na geração de novos conhecimentos e/ou até mesmo na elaboração de um livro baseado nos conceitos investigados durante esta nova pesquisa.

A principal contribuição deste estudo foi a análise das características dos modelos e formalismos, gerando o resultado denominado "conjunto de características para teste de desempenho". Baseado na análise realizada na perspectiva dos modelos de teste, o conjunto de características para teste de desempenho pode ser utilizado para o desenvolvimento de uma notação, formalismo, ou linguagem específica, como por exemplo, uma DSL (*Domain Specific Language*) [95] para teste de desempenho. Desta forma, torna-se fundamental um padrão de modelagem, *i.e.*, buscar e propor um formalismo que atenda as necessidades do teste de desempenho, bem como sua aplicação na área de teste baseado em modelos.

Para a implementação desta proposta de formalismo, alguns aspectos a fim de tratar este desafio devem ser levados em consideração: a) Representar as características do teste de desempenho; b) Identificar objetivos do teste de desempenho; c) Destacar os indicadores de desempenho mensurados pelo teste; d) Modelar os diferentes perfis de usuários e seu comportamento.

Em relação às contribuições científicas desenvolvidas no projeto de pesquisa, além dos relatórios técnicos internos escritos e não publicados, vale ressaltar a publicação dos artigos: 1) "*Generation of Scripts for Performance Testing Based on UML Models*" na Conferência Internacional do SEKE (*Software Engineering and Knowledge Engineering*) [3]. O objetivo deste artigo foi descrever um estudo de caso que mostrasse como implementar o processo de MBT para automatizar a geração e execução de *scripts* de teste em um contexto do mundo real. 2) "*Generating Performance Test*

Scripts and Scenarios Based on Abstract Intermediate Models" na Conferência Internacional do SEKE (*Software Engineering and Knowledge Engineering*) [96]. O artigo apresenta um formato independente de tecnologia para modelagem de cenários e casos de teste abstratos, doravante denominados modelos abstratos. Além disso, descreve o processo para concretizar *scripts* de teste específicos (*e.g.*, *scripts* LoadRunner e Visual Studio) baseados nos modelos abstratos.

REFERÊNCIAS

[1] I. K. El-Far e J. A. Whittaker. "Model-based Software Testing". New York, NY, USA: Wiley, 2001, pp. 825–837.

[2] M. Utting, A. Pretschner e B. Legeard. "A taxonomy of model-based testing". The University of Waikato, Hamilton, New Zealand, Tech. Rep., 2006, 18p.

[3] M. B. da Silveira, E. de M. Rodrigues, A. F. Zorzo, L. T. Costa, H. V. Vieira e F. M. de Oliveira. "Generation of Scripts for Performance Testing Based on UML Models". In: *23rd International Conference on Software Engineering and Knowledge Engineering*, Miami, FL, USA, 2011, pp. 258–263.

[4] E. de M. Rodrigues, A. F. Zorzo, E. Y. Nakagawa, I. Gimenez, J. C. Maldonado e F. M. de Oliveira. "A Software Product Line for Model-Based Testing Tools" [Submitted]. *Journal of Systems and Software*, 2012, pp. 1–26.

[5] L. T. Costa. "Conjunto de Características para Teste de Desempenho: Uma Visão a partir de Ferramentas" [no prelo]. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2012, 113p.

[6] A. Avizienis, J.-C. Laprie, B. Randell e C. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". *IEEE Transactions on Dependable Secure Computing*, vol. 1–1, Jan 2004, pp. 11–33.

[7] M. Young e M. Pezzè. "Software Testing and Analysis: Process, Principles and Techniques". New York, NY, USA: John Wiley & Sons, 2005, 488p.

[8] A. Romanovsky e A. F. Zorzo. "On distribution of coordinated atomic actions". *SIGOPS Operation System Review*, vol. 31–4, 1997, pp. 63–71.

[9] D. Powell. "Failure mode assumptions and assumption coverage". In: *22nd International Symposium on Fault-Tolerant Computing. Digest of Papers.*, 2002, pp. 386–395.

[10] A. Romanovsky e A. F. Zorzo. "Coordinated atomic actions as a technique for implementing distributed gamma computation". *Journal of System Architecture*, vol. 45–15, Set 1999, pp. 1357–1374.

[11] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins e D. Powell. "Fault injection for dependability validation: a methodology and some applications". *IEEE Transactions on Software Engineering*, vol. 16–2, Fev 2002, pp. 166–182.

[12] J.-C. Laprie. "Dependability Evaluation of Software Systems in Operation". *IEEE Transactions on Software Engineering*, vol. SE-10–6, Maio 2009, pp. 701–714.

[13] G. J. Myers e C. Sandler. "The Art of Software Testing". New York, NY, USA: John Wiley & Sons, 2004, 256p.

[14] B. Beizer. "Software System Testing and Quality Assurance". New York, NY, USA: Van Nostrand Reinhold, 1984, 358p.

[15] M. E. Delamaro, J. C. Maldonado e M. Jino. "Introdução ao Teste de Software". Rio de Janeiro, RJ, Brazil: Elsevier, 2007, 408p.

[16] A. Hartman. "Model Based Test Generation Tools". Capturado em: <http://www.agedis.de/documents/ModelBasedTestGenerationTools.pdf>, AGEDIS Consortium, Tech. Rep., 2002, 7p.

[17] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann e L. Nachmanson. "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer". In: *Formal Methods and Testing*, R. Hierons, J. Bowen e M. Harman, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 39–76.

[18] A. Stefanescu, S. Wieczorek e A. Kirshin. "MBT4Chor: A model-based testing approach for service choreographies". In: *5th European Conference on Model Driven Architecture-Foundations and Applications*, vol. 5562. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 313–324.

[19] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch e T. Stauner. "One Evaluation of Model-based Testing and Its Automation". In: *27th International Conference on Software Engineering*, New York, NY, USA: ACM, 2005, pp. 392–401.

[20] M. Popovic e I. Velikic. "A Generic Model-Based Test Case Generator". In: *12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, Washington, USA: IEEE, 2005, pp. 221–228.

[21] P. Clements, L. Northrop e L. M. Northrop. "Software Product Lines: practices and patterns". Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, 608p.

[22] H. Gomaa. "Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures". Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004, 736p.

[23] Software Engineering Institute (SEI), "Software Product Lines (SPL)," Capturado em: <http://www.sei.cmu.edu/productlines/>, setembro 2010.

[24] M. Woodside e D. Petriu. "Capabilities of the UML Profile for Schedulability Performance and Time". In: *Workshop on the usage of the UML Profile for Scheduling, Performance and Time*, 2004, pp 1–4.

[25] OMG. "UML Profile for Schedulability, Performance, and Time Specification - OMG Adopted Specification Version 1.1, formal/05-01-02," Capturado em: <http://www.omg.org/spec/SPTP/1.1/PDF>, 2005.

[26] S. Bernardi, J. Merseguer e D. C. Petriu. "Adding Dependability Analysis Capabilities to the MARTE Profile". In: *11th International Conference on Model Driven Engineering Languages and Systems*, Berlin, Heidelberg: Springer-Verlag, 2008, pp. 736–750.

[27] OMG. "UML Profile for Modeling and Analysis of Real-Time and Embedded Systems - MARTE specification v.1.0 (2009-11-02)". 2009.

[28] E. de M. Rodrigues, L. D. Viccari, A. F. Zorzo e I. M. Gimenes. "PLeTs Tool - Test Automation using Software Product Lines and Model Based Testing". In: *22th International Conference on Software Engineering and Knowledge Engineering*, Redwood City, CA, USA, 2010, pp. 483–488.

[29] W. C. Hetzel e B. Hetzel. "The Complete Guide to Software Testing". New York, NY, USA: John Wiley & Sons, 1991, 296p.

[30] IEEE. "IEEE Standard Glossary of Software Engineering Terminology: Std 610.12-1990," IEEE, Tech. Rep., 1991, 84p.

[31] D. Galin. "Software Quality Assurance: From Theory to Implementation". Harlow, UK: Addison-Wesley, 2004, 590p.

[32] B. Beizer. "Black-box Testing: Techniques for Functional Testing of Software and Systems". New York, NY, USA: John Wiley & Sons, 1995, 320p.

[33] J. Rubin. "Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests". New York, NY, USA: John Wiley & Sons, 1994, 330p.

[34] C. Kaner, J. L. Falk e H. Q. Nguyen. "Testing Computer Software". New York, NY, USA: John Wiley & Sons, 1999, 480p.

[35] M. Woodside, G. Franks e D. C. Petriu. "The future of software performance engineering". In: *Future of Software Engineering*, Washington, DC, USA: IEEE, 2007, pp. 171–187.

[36] L. Chung e J. C. S. do Prado Leite. "On Non-Functional Requirements in Software Engineering". In: *Conceptual Modeling: Foundations and Applications*, vol. 5600 LNCS. Springer, 2009, pp. 363–379.

[37] C. U. Smith. "Software Performance Engineering". *Encyclopedia of Software Engineering*, New York, NY, USA: John Wiley & Sons, 2002.

[38] J. Meier, C. Farre, P. Bansode, S. Barber e D. Rea. *Performance Testing Guidance for Web Applications: Patterns & Practices*. Microsoft Press, 2007, 221p.

[39] Y. Jing, Z. Lan, W. Hongyuan, S. Yuqiang e C. Guizhen. "JMeter-based aging simulation of computing system". In: *International Conference on Computer, Mechatronics, Control e Electronic Engineering*, vol. 5, Ago 2010, pp. 282–285.

[40] Hewlett Packard - HP, "*Software* HP LoadRunner," Capturado em: <https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17ô_4000_100__>, setembro 2010.

[41] D. Chadwick, C. Davis, M. Dunn, E. Jessee, A. Kofaldt, K. Mooney, R. Nicolas, A. Patel, J. Reinstrom, K. Siefkes, P. Silva, S. Ulrich e W. Yeung. "Using Rational Performance Tester Version 7". Riverton, NJ, USA: IBM Redbooks, 2008, 572p.

[42] BCM Software. "Performance Benchmarking Kit Using Incident Management with Silk-Performer". BMC Software, Tech. Rep., 2007.

[43] Bai, Xin. "Testing the Performance of an SSAS Cube Using VSTS". In: *7th International Conference on Information Technology: New Generations*, Washington, USA: IEEE, 2010, pp. 986–991.

[44] T. S. Chow. "Testing Software Design Modeled by Finite-State Machines". *IEEE Transactions on Software Engineering*, vol. 4–3, Maio 1978, pp. 178–187.

[45] M. D. Beaudry. "Performance-related reliability measures for computing systems". *IEEE Transactions on Computers*, vol. C-27–6, Jun 1978, pp. 540–547.

[46] B. Plateau e K. Atif. "Stochastic automata network of modeling parallel systems". *IEEE Transactions on Software Engineering*, vol. 17–10, Out 1991, pp. 1093–1108.

[47] C.-S. Chang. "Stability, queue length, and delay of deterministic and stochastic queueing networks". *IEEE Transactions on Automatic Control*, vol. 39–5, Maio 1994, pp. 913 –931.

[48] M. K. Molloy. "Performance Analysis Using Stochastic Petri Nets". *IEEE Transactions on Computers*, vol. 31–9, Set 1982, pp. 913–917.

[49] S. Barber. "User Community Modeling Language (UCML) for performance test workloads". Capturado em: <http://www.ibm.com/developerworks/rational/library/5219.html>, 2003.

[50] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton e B. M. Horowitz. "Model-Based Testing in Practice". In: *International Conference on Software Engineering*, Los Angeles, CA, USA: ACM, 1999, pp. 285–294.

[51] L. Apfelbaum e J. Doyle. "Model Based Testing". In: *Software Quality Week Conference*, San Francisco, CA, USA, 1997, pp. 296–300.

[52] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker e A. Pretschner. "Model-Based Testing of Reactive Systems: Advanced Lectures". Secaucus, NJ, USA: Springer, 2005, 667p.

[53] P. Krishnan. "Uniform Descriptions for Model Based Testing". In: *15th Australian Software Engineering Conference*, Washington, DC, USA: IEEE Computer Society, 2004, pp. 96–105.

[54] A. C. Dias Neto, R. Subramanyan, M. Vieira e G. H. Travassos. "A Survey on Model-Based Testing Approaches: A Systematic Review". In: *1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*. New York, NY, USA: ACM, 2007, pp. 31–36.

[55] M. Utting e B. Legeard. "Practical Model-Based Testing: A Tools Approach". San Francisco, CA, USA: Morgan Kaufmann, 2006, 456p.

[56] M. Utting, A. Pretschner e B. Legeard. "A taxonomy of model-based testing approaches". *Software Testing, Verification and Reliability*, 2011, published online. Paper version to appear.

[57] B. A. Kitchenham, T. Dybå e M. Jørgensen. "Evidence-based Software Engineering". In: *26th International Conference on Software Engineering*, Washington, DC, USA: IEEE, 2004, pp. 273–281.

[58] K. Petersen, R. Feldt, S. Mujtaba e M. Mattsson. "Systematic Mapping Studies in Software Engineering". In: *12th International Conference on Evaluation and Assessment in Software Engineering*, vol. 17–1, 2008, pp. 1–10.

[59] B. Kitchenham e S. Charters. "Guidelines for performing Systematic Literature Reviews in Software Engineering". Keele University and Durham University Joint Report, Tech. Rep. EBSE 2007-001, 2007, 65p.

[60] D. Budgen, M. Turner, P. Brereton e B. Kitchenham. "Using Mapping Studies in Software Engineering". In: *Psychology of Programming Interest Group*, Lancaster University, 2008, pp. 195–204.

[61] W. Afzal, R. Torkar e R. Feldt. "A systematic review of search-based testing for non-functional system properties". *Information and Software Technology*, vol. 51–6, Jun 2009, pp. 957–976.

[62] K. Gallagher e B. Shea. "Annual Load Test Market Summary and Analysis". Newport Group, 2001, 16p.

[63] D. Lee e M. Yannakakis. "Principles and Methods of Testing Finite State Machines-A Survey". *Proceedings of the IEEE*, vol. 84–8, Ago 1996, pp. 1090–1123.

[64] A. Gill. "Introduction to the Theory of Finite State Machines". New York: McGraw-Hill, 1962, 207p.

[65] A. En-Nouaary, R. Dssouli e F. Khendek. "Timed Wp-method: testing real-time systems". *IEEE Transactions on Software Engineering*, vol. 28–11, Nov 2002, pp. 1023–1038.

[66] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou e A. Ghedamsi. "Test Selection Based on Finite State Models". *IEEE Transactions on Software Engineering*, vol. 17–6, Jun 1991, pp. 591–603.

[67] S. Naito e M. Tsunoyama. "Fault Detection for Sequential Machines by Transitions Tours". In: *11th IEEE Fault Tolerant Computing Conferece*, IEEE Computer Society Press, 1981, pp. 238–243.

[68] G. Gonenc. "A Method for the Design of Fault Detection Experiments". *IEEE Transactions on Computer*, vol. C-19–6, Jun 1970, pp. 551–558.

[69] K. Sabnani e A. Dahbura. "A protocol test generation procedure". *Computer Networks and ISDN Systems*, vol. 15–4, Set 1988, pp. 285–297.

[70] S. Pimont e J.-C. Rault. "A software reliability assessment based on a structural and behavioral analysis of programs". In: *2nd International Conference on Software Engineering*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 486–491.

[71] A. Petrenko, N. Yevtushenko, A. Lebedev e A. Das. "Nondeterministic State Machines in Protocol Conformance Testing". In: *6th IFIP WG 6.1 International Workshop on Protocol Test Systems*, Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1993, pp. 363–378.

[72] G. Luo, A. Petrenko e G. v. Bochmann. "Selecting test sequences for partially-specified nondeterministic finite state machines". In: *7th IFIP WG 6.1 International Workshop on Protocol Test Systems*, London, UK: Chapman &Hall, Ltd., 1994, pp. 95–110.

[73] N. Yevtushenko e A. Petrenko. "Test Derivation Method for an Arbitrary Deterministic Automaton". *Automatic Control and Computer Sciences*, vol. 24–5, 1990, pp. 65–68.

[74] A. Petrenko e N. Yevtushenko. "Testing from partial deterministic FSM specifications". *IEEE Transactions on Computers*, vol. 54–9, Set 2005, pp. 1154–1165.

[75] A. Simão, A. Petrenko e N. Yevtushenko. "Generating Reduced Tests for FSMs with Extra States". In: *21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 129–145.

[76] A. Simão. *Teste Baseado em Modelos*. Rio de Janeiro, Brazil: Elsevier, 2007, pp. 27–45.

[77] J. Hopcroft, R. Motwani e J. Ullman. "Introduction to Automata Theory, Languages, and Computation". Boston, MA, USA: Addison Wesley, 1979, 418p.

[78] G. H. Mealy. "A Method for Synthesizing Sequential Circuits," *Bell System Technical Journal*, vol. 34–5, 1955, pp. 1045–1079.

[79] E. F. Moore. "Gedanken-Experiments on Sequential Machines". In: *Automata Studies*, C. Shannon e J. McCarthy, Eds. Princeton, NJ: Princeton University Press, 1956, pp. 129–153.

[80] R. Dorofeeva, K. El-Fakih, S. Maag, A. Cavalli e N. Yevtushenko. "Experimental evaluation of FSM-based testing methods". In: *3rd IEEE International Conference on Software Engineering and Formal Methods*, 2005, pp. 23–32.

[81] G. Kim, H. Moon, G. P. Song e S. K. Shin. "Software Performance Testing Scheme Using Virtualization Technology". In: *4th International Conference on Ubiquitous Information Technologies & Applications*, 2009, pp. 1–5.

[82] J. Mayer, I. Melzer e F. Schweiggert. "Lightweight plug-in-based application development". In: *Objects, Components, Architectures, Services, and Applications for a Networked World*, Berlin, Heidelberg: SpringerVerlag, 2003, pp. 87–102.

[83] K. P. Peralta, A. M. Orozco, A. F. Zorzo e F. M. de Oliveira. "Specifying Security Aspects in UML Models". In: *1st International Workshop on Modeling Security In ACM/IEEE 11th International Conference on Model-Driven Engineering Languages and Systems*, vol. 1, 2008, pp. 1–10.

[84] F. M. de Oliveira, R. da S. Menna, H. V. Vieira e D. D. Ruiz. "Performance Testing from UML Models with Resource Descriptions". In: *1st Brazilian Workshop on Systematic and Automated Software Testing*, 2007.

[85] M. R. Blaha e J. R. Rumbaugh. "Object-Oriented Modeling and Design with UML". Upper Saddle River, NJ, USA: Prentice Hall, 2005, 477p.

[86] OMG. "XML Metadata Interchange (XMI) specification, v. 2.4, Beta 2 (2010-12-06)," Capturado em: <http://www.omg.org/spec/XMI/2.4/Beta2/PDF>, 2010.

[87] A. A. Andrews, J. Offutt e R. T. Alexander. "Testing Web applications by modeling with FSMs". *Software and System Modeling*, vol. 4–3, Jul 2005, pp. 326–345.

[88] R. Jhala e R. Majumdar. "Software Model Checking". *ACM Computing Surveys*, vol. 41–4, Out 2009, pp. 21:1–21:54.

[89] J. Backus. "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs". In: *ACM Turing award lectures*. New York, NY, USA: ACM, 2007, pp. 613–641.

[90] OMG. "Business Process Model and Notation (BPMN) - OMG Adopted Specification Version 2.0, formal/2011-01-03," Capturado em: <http://www.omg.org/spec/BPMN/2.0/PDF>, 2011.

[91] M. Weske. "Business process management: concepts, languages, architectures". Berlin Heidelberg: Springer-Verlag, 2007, 368p.

[92] F. Belli, C. J. Budnik e L. White. "Event-based modelling, analysis and testing of user interactions: approach and case study: Research Articles". *Software Testing, Verification & Reliability*, vol. 16–1, Mar 2006, pp. 3–32.

[93] S. Pakin. "The Design and Implementation of a Domain-Specific Language for Network Performance Testing". *IEEE Transactions on Parallel and Distributed Systems*, vol. 18–10, Out 2007, pp. 1436–1449.

[94] L. Zhu, Y. Liu, N. B. Bui e I. Gorton. "Revel8or: Model Driven Capacity Planning Tool Suite". In: *29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 797–800.

[95] N. Bui, L. Zhu, I. Gorton e Y. Liu. "Benchmark Generation Using Domain Specific Modeling". In: *18th Australian Software Engineering Conference*, 2007, pp. 169–180.

[96] L. T. Costa, R. M. Czekster, F. M. de Oliveira, E. de M. Rodrigues, M. B. da Silveira e A. F. Zorzo. "Generating Performance Test Scripts and Scenarios Based on Abstract Intermediate Models". In: *24rd International Conference on Software Engineering and Knowledge Engineering*, Redwood City, CA, USA, 2012, pp. 1–6.

[97] A. Avizienis, J.-C. Laprie, B. Randell e C. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". *IEEE Transactions on Dependable Secure Computing*, vol. 1–1, Jan 2004, pp. 11–33.

[98] A. Grasselli e F. Luccio. "A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks". *IEEE Transactions on Electronic Computers*, vol.EC-14–3, Jun 1965, pp.350–359.

A. MODELOS UML DA APLICAÇÃO SKILLS

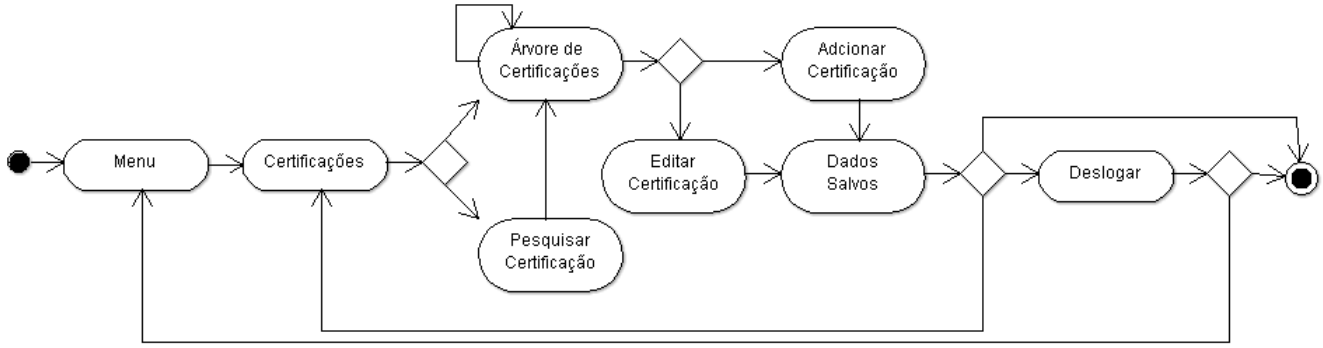

Figura A.1: Diagrama de atividades do caso de uso "Gerenciar Certificação"

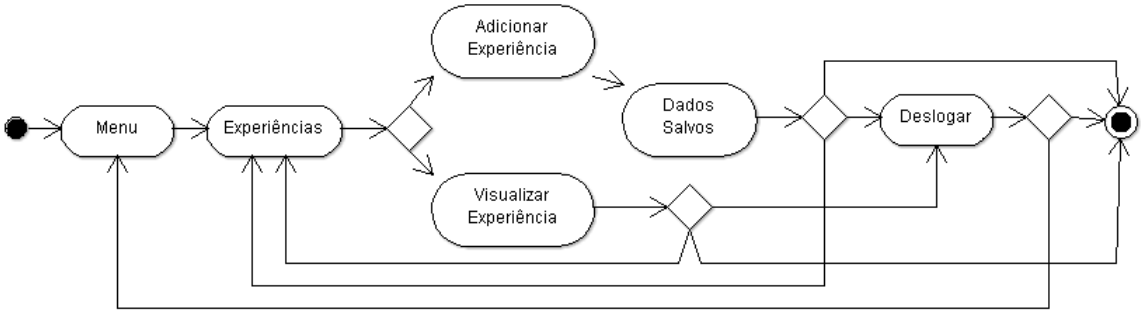

Figura A.2: Diagrama de atividades do caso de uso "Gerenciar Experiência"

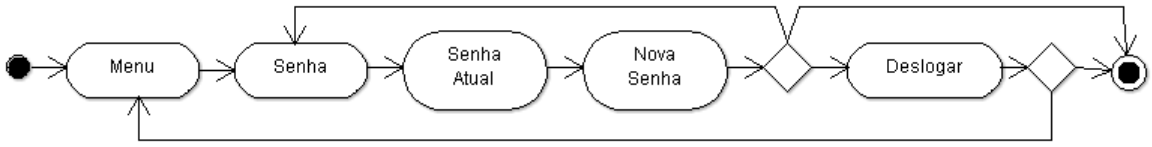

Figura A.3: Diagrama de atividades do caso de uso "Alterar Senha"

B. MEFs DA APLICAÇÃO SKILLS

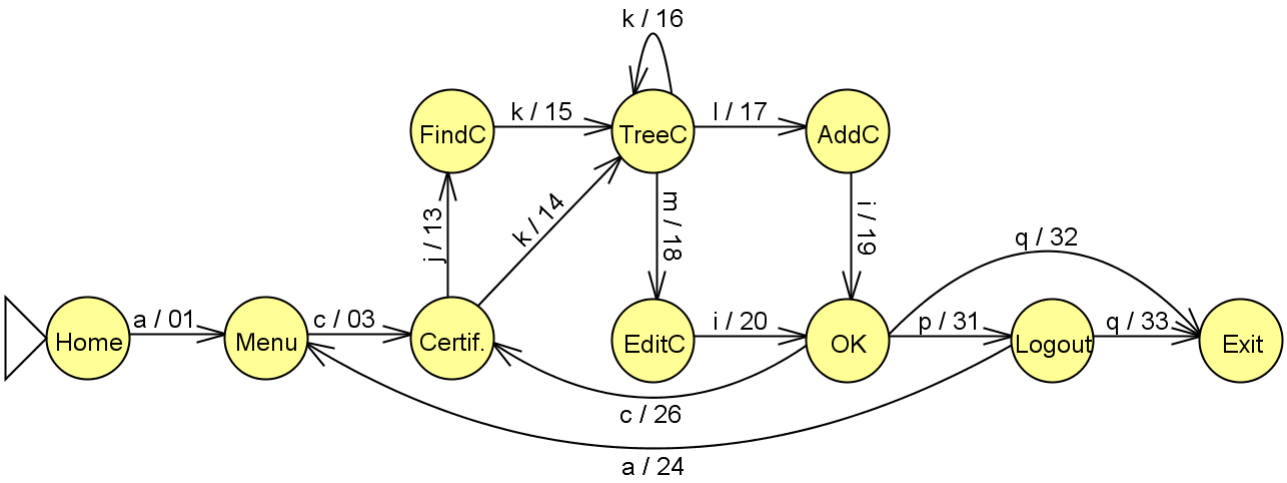

Figura B.1: MEF gerada a partir do diagrama de atividades da Figura A.1

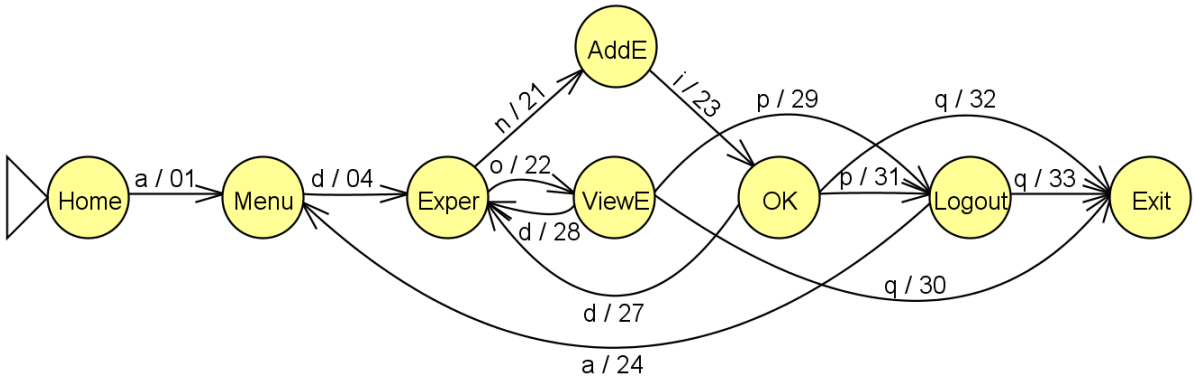

Figura B.2: MEF gerada a partir do diagrama de atividades da Figura A.2

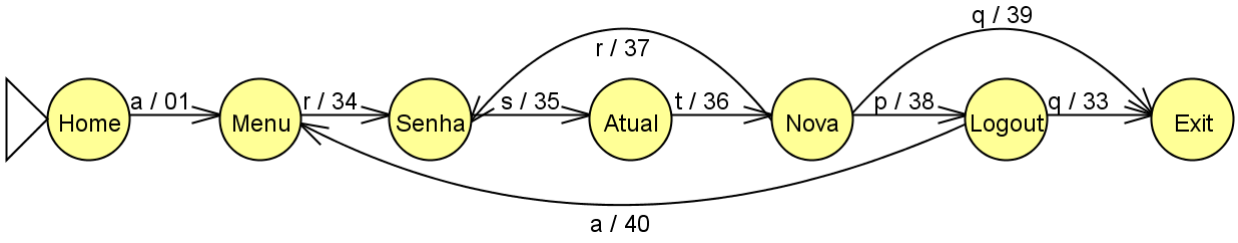

Figura B.3: MEF gerada a partir do diagrama de atividades da Figura A.3