Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Linha de produtos de testes baseados em modelos

Alex Mulattieri Suarez Orozco

Porto Alegre
2009

Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Linha de produtos de testes baseados em modelos

Alex Mulattieri Suarez Orozco

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Avelino F. Zorzo

Porto Alegre
2009

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 **"Linha de Produtos de Testes Baseados em Modelos"**, apresentada por Alex Mulattieri Suarez Orozco, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 23/12/08 pela Comissão Examinadora:

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

Prof. Dr. Eduardo Augusto Bezerra - — PPGCC/PUCRS

Prof. Dr. Toacy Cavalcante de Oliveira – — University of Waterloo

Homologada em 05/05/09, conforme Ata No. 007 pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
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

Dados Internacionais de Catalogação na Publicação (CIP)

O74l Orozco, Alex Mulattieri Suarez
Linha de produtos de testes baseados em modelos / Alex Mulattieri Suarez Orozco. – Porto Alegre, 2009.
101 f.

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

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

CDD 005.1

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

"Victories often occur after you see no way to succeed but refuse to give up anyway."

Dave Weinbaum

Para minha família.

Agradecimentos

À minha família, pelo exemplo dado, apoio e auxílio sempre prestados.

Ao Prof. Dr. Toacy Cavalcante de Oliveira, pela oportunidade oferecida para ingressar no Mestrado e pelo auxílio prestado como orientador, pois sem ele este trabalho não teria começado.

Ao Prof. Dr. Avelino Francisco Zorzo, por aceitar orientar este trabalho, dando continuidade ao trabalho iniciado pelo Prof. Toacy. A conclusão deste trabalho é fruto do seu esforço e paciência.

Ao Prof. Dr. Flávio Moreira de Oliveira, pelo exemplo de profissionalismo e pela dedicação mesmo quando ausente da obrigação.

Aos colegas Elder Macedo de Oliveira e Karine de Pinho Peralta, pessoas essenciais para que este trabalho fosse concluído com sucesso.

À HP P&D Brasil, pela oportunidade e bolsa concedida.

Resumo

Com a evolução das metodologias de desenvolvimento de software, novos desafios vão surgindo junto com este avanço. Um dos desafios fica a cargo de como garantir a qualidade do software desenvolvido. Diversos pesquisadores desenvolvem continuamente diversas abordagens de teste de software para acompanhar esta evolução. Uma das abordagens que visa prover o aumento da qualidade é através de testes de software baseados em modelos. Existem diversas formas de realização de testes de software baseados em modelos, garantindo um amplo leque de possibilidades. Uma forma de integrar estas possibilidades é através da aplicação dos conceitos de linha de produtos de software. Este trabalho propõe a realização de uma arquitetura de linha de produtos de software para integrar as diferentes técnicas de testes baseados em modelos.
Palavras-chave: Teste baseado em modelos, UML, Linha de produtos de software, Arquiteturas de software, Programação baseada em componentes.

Abstract

The continuing evolution of software development methodologies results in challenges emerging simultaneously with this advance. One of these challenges is how to guarantee the quality of software. Several researchers continuously develop different approaches for software testing following this evolution. One approach that aims to provide the increased quality is through model-based testing. There are various techniques of model-based software testing, ensuring a broad range of possibilities. One way to integrate these possibilities is applying the concepts of software product lines. In this work we propose the design of a software product line architecture to integrate the different techniques of model-based software testing.
Keywords: Model-based testing, UML, Software product line, Software architectures, Component-based programming.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

DNA	*DeoxyriboNucleic Acid*
FODA	*Feature-oriented Domain Analysis*
FSM	*Finite State Machine*
GSPN	*Generalized Stochastic Petri Net*
IDE	*Integrated Development Environment*
UML	*Unified Modeling Language*

Sumário

1 Introdução

A etapa de testes é vital no processo de desenvolvimento de software, onde busca identificar potenciais defeitos de software que podem resultar em perda de recursos financeiros, propriedades, clientes ou vidas [1]. Teste é uma atividade essencial em Engenharia de Software. Em termos simples, é a observação da execução de um sistema de software para validar se a execução comporta-se como o esperado e identificar potenciais defeitos [2]. Testes são amplamente usados na indústria para medida de qualidade: sem dúvida, ao examinar diretamente o software em execução, são providas informações realísticas do seu comportamento, entretanto é necessário utilizar outras técnicas de análise complementares.

Através da aparente simplicidade ao checar um exemplo em execução, testes englobam uma variedade de atividades, técnicas e atores, possuindo muitos desafios complexos. Certamente, com a complexidade, pervasividade e criticalidade do crescimento incessante dos softwares, garantir que os softwares comportam-se de acordo com os níveis desejáveis de qualidade torna-se mais crucial, sendo fator do aumento da dificuldade e do custo de desenvolvimento. Estudos estimam que os testes podem consumir cinqüenta por cento, ou até mais, do custo de desenvolvimento [3] [4].

Teste de software é um termo genérico que engloba um vasto espectro de diferentes atividades, desde o teste de uma pequena parte do código realizado pelo desenvolvedor até a validação do usuário de um complexo sistema de informação. Em vários estágios, os casos de teste podem ser planejados em direção a diferentes objetivos, como expor o tangenciamento dos requisitos do usuário, estimar a conformidade de uma especificação padrão, avaliar a robustez para condições de cargas estressantes ou para entradas maliciosas ou medir determinados atributos, tais como desempenho ou usabilidade, ou estimar a confiabilidade operacional, entre outros. Além disso, a atividade de teste pode ser portada de acordo com um procedimento informal e *ad hoc*, ou formal controlado (exigindo um planejamento e documentação rigorosos) [2].

Como uma conseqüência desta variedade de foco e escopo (conforme citado anteriormente), existe uma multiplicidade de significados para o termo "teste de software", gerando muitos desafios peculiares de pesquisas [2]. Para organizar os desafios em uma visão unificada, é apresentada uma classificação de problemas comuns para os muitos significados sobre teste de software. O primeiro conceito seria qual é o denominador comum, se ele existir, entre todas as possíveis "faces" de teste diferentes. Um denominador comum pode ser a visão muito abstrata que, dado uma parte de software (não importa qual seja sua tipologia, tamanho e domínio) o teste sempre consiste da observação de uma amostra de execuções, dando um veredito através desta observação [2]. Iniciando dessa visão muito generalista, pode-se então concretizar

diferentes instâncias, pela distinção de aspectos específicos que podem caracterizar a amostra de observações, tais como: porque se constroem as observações (onde a preocupação é com o objetivo do teste); quais amostras devem ser observadas, e como escolhê-las; quão grande é uma amostra; o que é executado (pode-se observar a execução ou do sistema como um todo ou focando somente em uma parte dele); onde efetuar a observação (*in vitro*, *in vivo*) e quando é, no ciclo de vida, que será efetuada a observação. Estas questões provêm uma caracterização de um esquema muito simples e intuitivo de atividades de testes de software, que podem ajudar na organização de um plano para os desafios das pesquisas [2].

A automação de testes de software pode reduzir drasticamente o esforço requerido para as atividades de teste. Através da automação, os testes podem ser realizados em minutos ao invés de demorarem horas para serem executados manualmente, podendo alcançar uma diminuição de esforço em mais de 80% [5]. Algumas organizações podem não reduzir gastos ou esforços diretamente, mas a automação dos testes permite produzir softwares de melhor qualidade mais rapidamente do que seria possível através de testes manuais.

Um regime maduro de automação de testes permite ao "toque de um botão" que testes sejam realizados durante toda a noite, período onde a infra-estrutura normalmente estaria sem uso. Testes automatizados permitem até a menor alteração no sistema ser completamente re-testada com um mínimo de esforço, eliminando tarefas consideradas repetitivas e desgastantes.

Uma solução de automação de testes é através da geração de casos de testes a partir de modelos, conhecida como testes baseados em modelos. Existem diferentes técnicas de testes baseados em modelos para diferentes tipos de teste, onde estas técnicas compartilham algumas características em comum (tais como a necessidade de possuir um modelo do sistema em teste, a simulação da execução deste modelo) e apresentam outras características específicas (como o algoritmo de geração dos casos de teste). Esta caracterização se enquadra perfeitamente na metodologia de desenvolvimento de software conhecida como linha de produtos de software. O tradicional modo de desenvolvimento de software é desenvolver sistemas isolados, ou seja, desenvolver cada sistema individualmente. Para linha de produtos de software, a metodologia de desenvolvimento é ampliada para considerar uma família de sistemas de software. Esta abordagem envolve analisar as características que são comuns a todos os membros da família e as que são específicas a cada elemento da família.

Sendo assim, este trabalho pretende propor uma arquitetura de linha de produtos de testes de software baseados em modelos, organizando as diferentes características de forma a facilitar a construção de ferramentas de testes baseados em modelos.

No Capítulo 2 é apresentado uma revisão da literatura sobre testes de software, introduzindo assim o tema em questão. No Capítulo 3 é exibida uma revisão da literatura sobre testes de software baseados em modelos, apresentado as diferentes abordagens sobre o tema. No Capítulo 4 é apresentada uma revisão da literatura sobre linha de produtos de software, exibindo os conceitos a serem empregados na proposta deste trabalho. No Capítulo 5 é apresentada a proposta deste trabalho, assim como um estudo de caso. Por fim, no Capítulo 6 são realizadas

as considerações finais e descritos os trabalhos futuros a serem realizados.

2 Testes de Software

Teste é uma atividade desempenhada para avaliar a qualidade do produto, e para aprimorar o produto identificando defeitos e problemas [6]. Esta definição descreve superficialmente os objetivos de teste. Em maiores detalhes, teste de software consiste na verificação **dinâmica** do comportamento de um software em um conjunto **finito** de casos de teste, convenientemente **selecionados** a partir de um domínio normalmente infinito, tomando como base o comportamento **esperado** do software [6].

Na definição acima, as palavras em negrito são chaves para identificar o conhecimento relacionado ao teste de software. Sendo elas: **Dinâmica**, onde este termo significa que o teste implica na execução do software com entrada de dados. Para ser preciso, o dado de entrada sozinho não é sempre suficiente para determinar um teste, pois um sistema pode apresentar um comportamento diferente para uma mesma entrada, dependendo do estado do sistema (sendo o sistema não-determinístico). Neste caso, o termo "entrada" significa também incluir uma especificação de um estado, quando for necessária tal especificação. **Finita**, pois mesmo em programas simples, teoricamente tantos são os casos de testes possíveis que testes exaustivos poderiam requerer meses ou anos para serem executados. Devido a esta situação, o domínio pode ser considerado infinito. Desta forma é conveniente definir o conjunto de casos de testes baseado nos recursos disponíveis para sua execução (como cronograma, equipamentos, entre outros). **Selecionados**, onde as muitas técnicas de teste diferem essencialmente em como selecionar os conjunto de teste. Como identificar o critério de seleção mais adequado em determinadas condições é um problema altamente complexo. **Esperado**, pois o comportamento observado do resultado da execução de um programa pode ser comparado com as expectativas do usuário (validação), com a especificação (verificação), ou finalmente com o comportamento antecipado por requisitos implícitos ou expectativas razoáveis (aceitação) [6].

A definição superficial de testes de software (citada inicialmente) apresenta teste como uma atividade focada em identificar defeitos. Um defeito ocorre quando o serviço entregue pelo sistema não corresponde ao serviço especificado, normalmente apresentando um resultado incorreto, implicando que o serviço esperado seja descrito por uma especificação ou um conjunto de requisitos [7]. Um erro é parte do estado do sistema que é responsável por conduzi-lo para um defeito [7]. Uma falha é a causa identificada ou hipotetizada de um erro [8], também conhecida como *bug*. Desta forma, um defeito ocorre devido a um erro produzido por uma falha [9].

2.1 Processo de Testes de Software

Teste não é visto como uma atividade que inicia somente depois que a fase de codificação foi completada, com a proposta limitada em detectar defeitos. Teste de software é visto como uma atividade que deve cobrir todo o processo de desenvolvimento e manutenção, sendo parte importante da construção de um produto. Sendo assim, o planejamento dos testes deve começar nos estágios iniciais do processo de requisitos, os planos de teste devem ser sistematicamente e continuamente desenvolvidos e, se possível, refinados à medida que o desenvolvimento do software evolui. O fluxo do processo de desenvolvimento de software pode ser sumarizado em sete passos [10], ilustrados na Figura 1:

Figura 1 – Processo de desenvolvimento de software.

1. O programa precisa estar traduzido em um conjunto de requisitos. Estes são os objetivos para o produto.
2. Os requisitos são traduzidos em objetivos específicos através de estimativa de viabilidade e custo, resolvendo requisitos conflitantes e estabilizando prioridades.
3. Os objetivos são traduzidos em uma especificação de produto precisa, visualizando o produto como uma caixa preta e considerando somente sua interface e as interações com o usuário final.
4. No caso de um sistema, este passo particiona o sistema em programas individuais, componentes, ou subsistemas, e define suas interfaces.

5. A estrutura do módulo, ou programa, é projetada pelas especificações das estruturas dos componentes, pela estrutura hierárquica dos componentes e pela interface entre os componentes.

6. Uma especificação precisa é desenvolvida, que define a interface para cada componente e a função de cada componente.

7. Através de um ou mais subpassos, a especificação do componente de interface é traduzida para o código fonte de cada componente.

Dada a premissa que os sete passos do ciclo de desenvolvimento envolvem comunicação, compreensão e tradução da informação, e a premissa que muitos erros de software originam de colapsos na manipulação das informações, existem três abordagens complementares para prevenir e/ou detectar esses erros [10]. Primeiro, pode-se introduzir mais precisão em um processo de desenvolvimento para prevenir muitos dos erros. Segundo, pode-se introduzir, no fim de cada passo, uma etapa de verificação em separado para localizar quantos erros forem possíveis antes de prosseguir para o próximo passo. Esta abordagem é ilustrada na Figura 2.

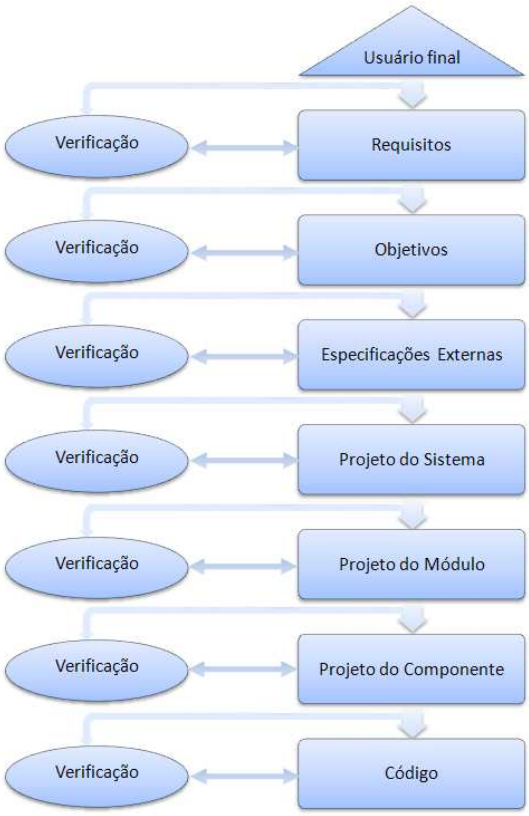

Figura 2 – Processo de desenvolvimento de software com passos de verificação intermediários.

A terceira abordagem é orientar processos de testes distintos voltados a processos de desenvolvimento distintos. Ou seja, focar cada processo de teste em um passo de tradução em particular, então focando em uma classe particular de erros. Esta abordagem é ilustrada na Figura 3. Este ciclo de teste foi estruturado para modelar o ciclo de desenvolvimento [10]. Em

outras palavras, estabilizar uma correspondência de um para um entre o processo de desenvolvimento e teste. Por exemplo, a proposta de um teste unitário é procurar discrepâncias entre o componente e sua especificação de interface. A proposta de um teste funcional é verificar se um programa não faz nada mais que sua especificação externa. A proposta de um teste de sistema é verificar se um produto está inconsistente com os objetivos originais.

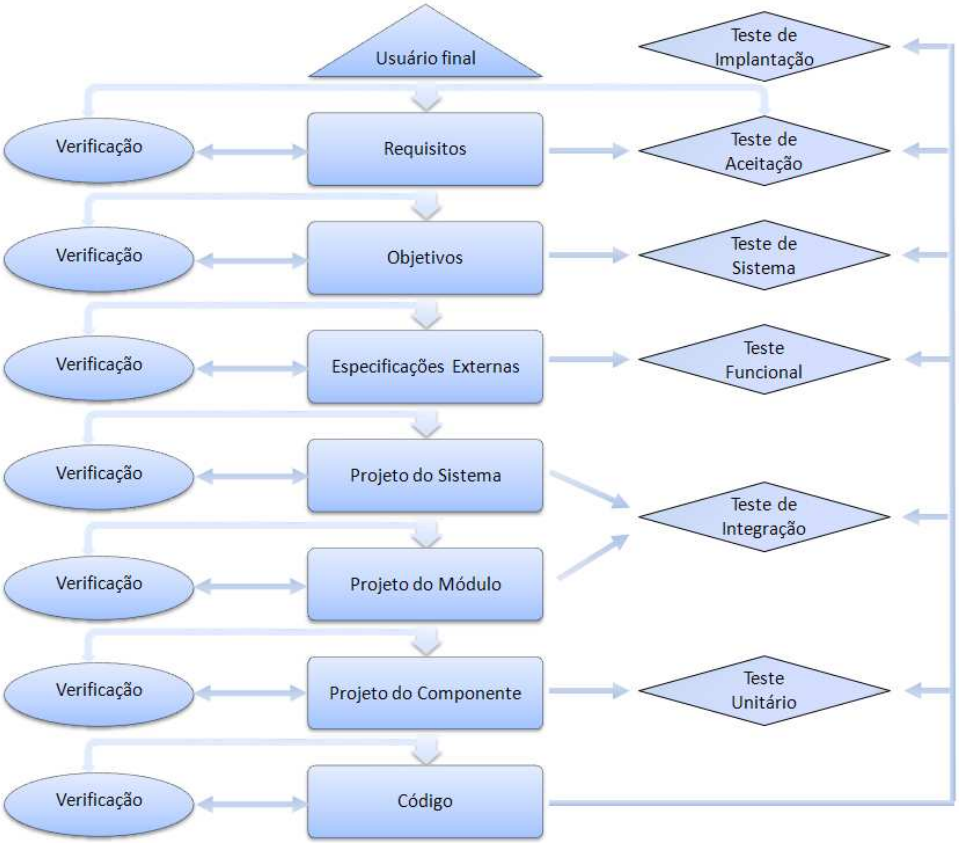

Figura 3 – Correspondência entre o processo de desenvolvimento e o processo de teste.

A sequência do processo de teste da Figura 3 não implica necessariamente em uma seqüência temporal. Por exemplo, o teste de sistema não é definido como "uma parte do teste a ser executada depois do teste funcional", mas sim como um tipo de teste focado em uma classe distinta de erros, podendo ser parcialmente sobreposta no tempo com outras etapas do processo de teste.

2.2 Tipos de Testes de Software

A seguir, serão discutidas os processos de testes unitário, funcional, de sistema, de aceitação e de instalação apresentados na Figura 3.

2.2.1 Teste Unitário

Teste unitário é o processo de teste no qual se tem acesso ao código do programa. Este teste é conhecido também como teste de caixa branca. Este teste busca localizar falhas no código do programa. Existem dois tipos de testes unitários: testes estáticos e dinâmicos [11].

Teste unitário estático é o processo de, cuidadosamente e metodicamente, revisar o código procurando por falhas sem executar o software. Este processo é conhecido também como análise estrutural. O objetivo de aplicar um teste de caixa branca estático é localizar falhas o mais cedo possível e localizar falhas que seriam difíceis isolar nas outras fases de teste.

Teste unitário dinâmico é o processo de testar um programa em execução e, durante a execução, examinar o código e como o código é executado. O teste unitário dinâmico não limita-se somente ao que o código faz. Também pode diretamente testar em baixo nível, os métodos, funções, procedimentos, subrotinas ou bibliotecas, pode acessar variáveis do programa e informação sobre o estado do software para determinar se os testes estão fazendo o que foi planejado para ser feito, pode forçar o software a realizar coisas que seriam difíceis de serem testadas normalmente e pode medir quanto do código e qual parte do código é testado quando os testes são executados, permitindo o ajuste dos testes para remover casos de testes redundantes ou adicionar etapas de forma a tornar o teste mais robusto.

2.2.2 Teste de Integração

Teste de integração é o processo que visa detectar deficiências no relacionamento entre os componentes do sistema. Este relacionamento é realizado de algumas formas, tais como a substituição gradual de cada conexão entre dois componentes, uma integração de forma cruzada, onde cada componente é integrado com o seu componente adjacente na arquitetura e esta integração é testada, a integração entre os componentes na forma *bottom-up*, ou seja, partindo da integração dos componentes básicos até a integração dos módulos compostos por estes componentes básicos, a integração na forma *top-down*, partindo da integração dos módulos na forma mais alto-nível até a integração entre os componentes básicos, e por fim a integração conhecida como "*big bang*" onde é testada a integração quando todos os componentes estão conectados [12].

2.2.3 Teste Funcional

Teste funcional é um processo de tentativa para localizar discrepâncias entre o programa e as especificações externas. Uma especificação externa é uma descrição precisa do comportamento do programa do ponto de vista do usuário final. Para prover um teste funcional, as especificações são analisadas para derivarem um conjunto de casos de testes.

Exceto quando utilizado em pequenos programas, o teste funcional é uma atividade conhecida como caixa preta, ou seja, se confia que um processo de teste unitário anterior alcançou um critério de cobertura lógica aceitável.

2.2.4 Teste de Sistema

Teste de sistema não é o processo de testar as funções do sistema completo, porque isso seria redundante com o processo de teste funcional. O teste de sistema tem um objetivo particular: comparar o sistema com os seus objetivos originais. Dada esta proposta, surgem duas implicações:

1. O teste de sistema não é limitado para sistemas. Se o produto é um programa, o teste de sistema é o processo de tentar demonstrar como o programa, como um todo, não preenche os seus objetivos.

2. Teste de sistema, por definição, é impossível de ser realizado, caso não exista um conjunto de objetivos claramente definidos e limitados para o produto.

Ao procurar por discrepâncias entre o programa e os seus objetivos, foca-se nos erros de tradução gerados durante o processo de definição das especificações externas. Isto torna o teste de sistema um processo de teste vital, pois em termos do produto, o número e a severidade dos erros encontrados são descobertos durante esta fase do ciclo de desenvolvimento. Isto implica também que, diferente do teste funcional, as especificações externas não podem ser usadas como base para derivar os casos de teste de sistema, pois subverteria a proposta do teste. Da mesma forma, somente o documento de requisitos não pode ser usado para formular casos de teste, ao menos que, por definição, contenha descrições precisas das interfaces externas do programa. Resolve-se este dilema usando a documentação do usuário, definindo o teste de sistema pela análise dos requisitos e formulando os casos de teste pela análise da documentação do usuário.

Dado o enunciado dos requisitos, não há metodologia identificável que renda um conjunto de casos de teste, exceto o vago, mas usual, guia de escrita de casos de teste para tentar mostrar que o programa é inconsistente com cada sentença dos enunciados dos objetivos.

Então, uma abordagem diferente para o projeto de casos de teste pode ser realizada da seguinte maneira: ao invés de descrever uma metodologia, categorias distintas de casos de testes de sistema são discutidas. Devido à ausência de uma metodologia, o teste de sistema requer uma quantidade substancial de criatividade, inteligência e experiência. A seguir são discutidas as categorias de testes de sistemas utilizados neste trabalho. Uma visão mais completa sobre o tema é apresentado em [10]:

Teste de Segurança

Devido ao fato da sociedade apresentar interesse pela privacidade, muitos sistemas possuem objetivos específicos de segurança. O teste de segurança é o processo de tentar desenvolver casos de testes que contestem as verificações de segurança. Por exemplo, pode-se tentar formular casos de teste que esquivem o mecanismo de proteção de memória de um sistema operacional. Pode-se tentar contestar um mecanismo de segurança de dados de um sistema de gerência de banco de dados. Outro caminho para planejar os casos de testes é estudar conhecidos problemas de segurança em sistemas similares e gerar os casos de testes que tentem demonstrar problemas similares no sistema o qual se deseja testar. Por exemplo, fontes publicadas em revistas, salas de bate-papo, ou notícias freqüentemente exibem *bugs* conhecidos em sistemas operacionais ou outros sistemas de software.

Teste de Desempenho

Muitos sistemas têm objetivos específicos de desempenho ou eficiência, declarando estas propriedades como tempos de resposta e taxas de transferência dentro de certas condições de cargas de trabalho e configuração. Novamente, como a proposta de um teste de sistema é demonstrar que o programa não alcança os seus objetivos, os casos de teste devem ser projetados para mostrar que o sistema não satisfaz os objetivos de desempenho.

2.2.5 Teste de Aceitação

Retornando ao modelo de desenvolvimento exibido na Figura 3, pode ser visto que o teste de aceitação é o processo de comparar os requisitos iniciais e a necessidade do usuário final. Este é um tipo incomum de teste onde é realizado normalmente pelo comprador do sistema ou pelo usuário final, e normalmente não é considerado responsabilidade da organização que desenvolve o software. No caso de um desenvolvimento de software contratado, a organização contratante (usuário) realiza o teste de aceitação comparando a operacionalidade do programa com o contrato original. Como é o caso para outros tipos de teste, a melhor maneira de realizar o teste é projetá-lo de forma que mostre que o programa não cumpre o contrato. Se o teste falhar, o programa é aceito.

2.2.6 Teste de Implantação

O último teste exibido na Figura 3 é o teste de implantação. Sua posição na Figura 3 é um tanto rara, pois todos os outros testes estão vinculados a uma fase específica do processo de desenvolvimento. Este é um tipo raro de teste porque sua proposta não é localizar erros de software, mas localizar erros que ocorreram durante o processo de implantação.

Muitos eventos ocorrem quando se implanta um sistema de software, por exemplo, os usuários necessitam selecionar uma variedade de opções, os arquivos e bibliotecas precisam ser alocados e carregados, uma configuração de hardware válida precisa estar presente e os programas podem necessitar de conectividade para conectarem-se com outros programas.

2.3 Considerações

Para garantir a qualidade de um software, diversas visões sobre o tema testes de software emergem. Estas visões exigem da equipe de teste um esforço substancial para cobrir todas as possíveis formas de testar um sistema. No capítulo seguinte, será apresentada uma revisão da literatura sobre uma forma de simplificar e diminuir os esforço da atividade de teste, forma esta denomindada testes de software baseados em modelos.

3 Testes de Software Baseados em Modelos

Modelos são usados para entender, especificar e desenvolver sistemas em muitas disciplinas. Das pesquisas de DNA [13] até o desenvolvimento do mais recente avião de guerra [14], modelos são usados para promover o entendimento e prover um *framework* reusável para o desenvolvimento de produtos. No processo de engenharia de software, modelos são aceitos como parte da abordagem de análise e projeto orientado a objetos. Artigos e livros têm sido escritos sobre a aplicação de modelos para o desenvolvimento de testes e análise de confiabilidade nas últimas duas décadas [15].

Modelar tem um significado muito econômico, pois tem como objetivo capturar conhecimento sobre um sistema e então reusar este conhecimento enquanto o sistema cresce [15]. Para uma equipe de teste, este conhecimento é como ouro, pois qual a porcentagem de uma tarefa de um engenheiro de teste é gasta tentando entender o que o sistema em teste deveria fazer? Uma vez que esta informação é compreendida, como a informação é preservada para o próximo engenheiro, a próxima versão, ou ordem de alteração? Com sorte estará no plano de testes, mas tipicamente codificado em um *script* de teste. Pela construção de um modelo que define o comportamento desejado do sistema, uma equipe agora tem um mecanismo para uma análise estruturada do sistema, pois os cenários de teste são descritos como uma seqüência de ações para o sistema (sendo especificados no modelo), com as respostas esperadas do sistema sendo também especificadas. A cobertura dos testes e os planos de teste são facilmente entendidos e desenvolvidos, além dos recursos disponíveis e a cobertura que pode ser entregue são estimados com uma maior segurança. O maior benefício está no reuso, pois todo este trabalho não é perdido. O próximo ciclo de testes pode começar onde o atual ciclo parou. Se o o produto apresenta a necessidade de novas funcionalidades, tais funcionalidades podem ser incrementalmente adicionadas ao modelo. Se a qualidade deve ser aperfeiçoada, o modelo pode ser refinado e a quantidade e qualidade dos testes aumentará. Se há novas pessoas na equipe, estas pessoas podem rapidamente compreender o projeto, bastando apenas revisar o modelo do projeto.

3.1 Modelos

De forma simplificada, um modelo de software é uma descrição da estrutura e do comportamento do software [16]. Para um modelo poder ser utilizado por equipes de testadores e por múltiplas tarefas de teste, o modelo necessita ser entendido por além dos que dominam

o conhecimento sobre para que supostamente o software é desenvolvido, devendo ser escrito em uma forma de fácil entendimento. É preferível, geralmente, que o modelo seja tão formal quanto prático. Através dessas propriedades, o modelo torna-se uma descrição compartilhável, reusável e precisa do sistema em teste.

Existem numerosos tipos de modelos, e cada um descreve aspectos diferentes do comportamento do software. Por exemplo, fluxo de controle, fluxo de dados, e gráficos de dependência do programa expressam como a implementação comporta-se através da representação da estrutura do código-fonte. Redes de Petri e máquinas de estados, por outro lado, são usadas para descrever os comportamentos externos conhecidos como caixa preta, ou seja, testes onde a estrutura do código-fonte não é considerada [17]. A seguir, serão descritos alguns tipos de modelos utilizados em testes de software.

3.1.1 Modelo de Máquinas de Estados Finitos

Considere o seguinte cenário de teste: um testador aplica uma entrada e então avalia o resultado. O testador então seleciona outra entrada, dependendo do resultado anterior, e outra vez reavalia o próximo conjunto de possíveis entradas. Em um dado instante, um testador tem um conjunto específico de entradas para escolher. Este conjunto de entradas varia dependendo do "estado" exato do programa. Esta característica faz o modelo baseado em estados uma ótima opção para teste de software: software está sempre em um estado específico e o estado corrente de uma aplicação governa que conjunto de entradas os testadores podem selecionar. Máquinas de Estados Finitos são aplicáveis para qualquer modelo que esteja perfeitamente descrito com um número finito de estados específicos [16]. Máquinas de Estados Finitos é uma teoria da computação madura e estável, pois modelos de máquinas de Estados Finitos têm sido usados no projeto e teste de componentes de hardware há muito tempo e é considerada uma prática padrão hoje em dia [16].

O modelo Máquina de Estados Finitos é uma opção quando os testadores possuem o fardo de construir seqüências de entrada para suprir os dados de teste. Máquinas de estados (grafos dirigidos) é um modelo ideal para descrever seqüências de entrada. Isto, combinado com uma abundância de algoritmos para percorrer grafos, torna a geração de testes menos onerosa que o teste manual. Por outro lado, softwares complexos implicam em grandes máquinas de estados, que não são triviais para construir e prover manutenção.

Na definição de Máquina de Estados Finitos (FSM - *Finite State Machine*), uma FSM é uma 5-tupla (I, S, T, F, L), onde I é o alfabeto de entrada, S é o conjunto finito de estados, $T \subset S \times I$ é a função que determina que uma transição ocorre quando uma entrada I é aplicada em um estado S, $F \subset S$ é o conjunto de estados finais e $L \subset S$ é o estado inicial [16].

Outra definição apresenta uma Máquina de Estados Finitos como uma 6-tupla (S,I,A,R,Δ,T), onde S é o conjunto finito de estados, $I \subset S$, é conjunto de estados iniciais, A é o alfabeto finito

de símbolos de entrada e R é o conjunto de possíveis saídas ou respostas. O conjunto $\Delta \subset S \times A$ é o domínio da relação de transição T, que é uma função de Δ para $S \times R$. [18]. Esta máquina é conhecida como Máquina de Estados Finitos com entradas e saídas.

A relação de transição descreve como a máquina reage ao receber entrada $a \in A$ quando $s \in S$, assumindo que $(s,a) \in \Delta$. Interpreta-se $(s, a) \notin \Delta$ como: o símbolo de entrada não pode ser aceito no determinado estado. Quando T(s,a)=(s',r), o sistema move-se para o novo estado s' e responde como saída r. Se T não é uma função, mas mais exatamente uma relação que associa cada par de estado de entrada com um conjunto não vazio de pares de estados de saida, é dito que o autômato finito é *não determinístico*, e é interpretado como o conjunto de possíveis respostas para um estímulo de entrada em um certo estado.

O modelo de Estados Finitos pode ser representado como um grafo, também chamado diagrama de transição entre estados, com nodos representando estados, arcos representando transições, e arcos com nomes representando entradas causando as transições. Normalmente, os estados finais e o inicial recebem uma marcação especial. A máquina de estados pode também ser representada como uma matriz, chamada matriz de transições entre estados. Existem duas formas usuais de matrizes de transições entre estados que são ilustrados nas Tabelas 1 e 2, no exemplo "Interruptor de lâmpadas"apresentado a seguir, junto com o correspondente diagrama de transição entre estados.

Exemplo

Considere um simulador hipotético de interruptor de lâmpadas simples. As lâmpadas podem estar ligadas ou desligadas sendo alternadas através de uma entrada. A intensidade da luz pode ser ajustada usando duas entradas para diminuir ou aumentar a intensidade. Existem três níveis de intensidade de luz: fraca, normal e brilhante. Se as lâmpadas estão brilhantes, ao aumentar a intensidade não deverá afetar a intensidade. O caso é similar para a lâmpada estar fraca e tentar diminuir a intensidade. O simulador inicia com as lâmpadas apagadas. Finalmente, quando as lâmpadas estão ligadas, a intensidade é normal por padrão, não levando em consideração a intensidade da luz quando a lâmpada foi desligada anteriormente.

O simulador pode estar em somente um dos quatro estados distintos a qualquer tempo: as lâmpadas estarem desligadas, com a iluminação fraca, normal ou brilhante. Uma maneira para modelar isto é usando uma Máquina de Estados Finitos que é definida a seguir:
"Interruptor de lâmpadas"= (I, S, T, F, L), onde:
S = { [desligada], [fraca], [normal], [brilhante] }
I = [desligada]
A = { <ligar>, <desligar>, <aumentar a intensidade>, <diminuir a intensidade> }
T = { <ligar> altera [desligada] para [normal],
<desligar> altera [fraca], [normal], ou [brilhante] para [desligada],
<aumentar a intensidade> altera [fraca] para [normal] e [normal] para [brilhante],

<diminuir a intensidade> altera [brilhante] para [normal] e [normal] para [fraca],

As entradas não afetam o estado do sistema em nenhuma condição não descrita acima }

F = { <desligada> }

L = <desligada>

Tabela 1 – Matriz de transições entre estados.

	desligada	fraca	normal	brilhante
desligada			ligar	
fraca	<desligar>		<aumentar a intensidade>	
normal	<desligar>	<diminuir a intensidade>		<aumentar a intensidade>
brilhante	<desligar>		<diminuir a intensidade>	

Tabela 2 – Outro formato de matriz de transições entre estados.

	<ligar>	<desligar>	<aumentar a intensidade>	<diminuir a intensidade>
desligada	normal			
fraca		desligada	normal	
normal		desligada	brilhante	fraca
brilhante		desligada		normal

Na Figura 4 é apresentada a máquina de estados equivalente em forma de um grafo dirigido.

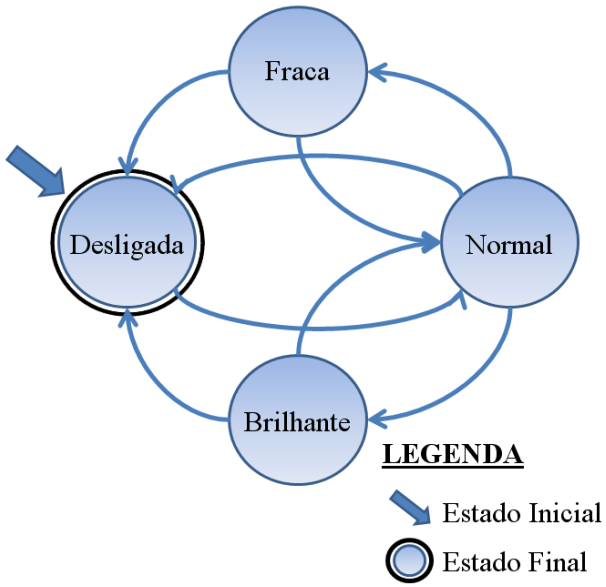

Figura 4 – Máquina de estados em forma de grafo dirigido.

3.1.2 UML

Nesta etapa do trabalho é apresentada uma rápida revisão sobre UML [19], onde não serão abordados todos os artefatos existentes, somente os interessantes para o teste baseado em modelos. Trabalhos como [20], [21] apresentam o uso de modelos de teste baseados em UML.

Modelo de Caso de Uso

Os casos de uso são normalmente aplicados na fase inicial de análise para definir que serviços (ou "casos de uso") o sistema irá prover. Um simples exemplo é exibido na Figura 5 (extraída de [17]), indicando que um ator do tipo *X* está envolvido em um caso de uso, descrevendo alguma funcionalidade *S4* do sujeito *y:Y*.

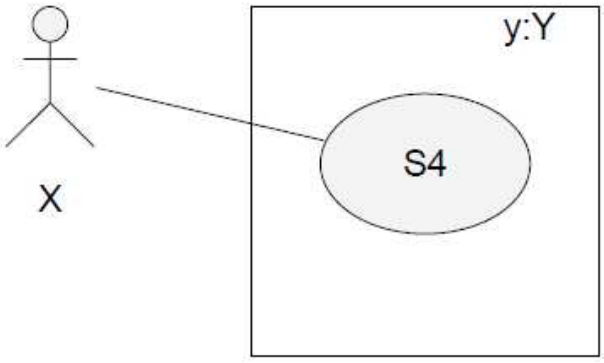

Figura 5 – Diagrama de caso de uso.

O responsável pela modelagem apresenta a descrição da funcionalidade em uma prosa estruturada [17]. Desta forma, o caso de uso *S4* poderia ser algo como:
Atores: *X*;
Pré-condições: *Y* precisa estar pronto;
Caso Normal: *X* enviará a *Y* um sinal *m1* e então *Y* retornará um sinal *m4*;
Casos Excepcionais: Nenhum.
A descrição dos casos de uso são normalmente usados pelos responsáveis em definir os comportamentos em termos mais precisos.

Modelo de Atividades

Modelos de atividades são usados para descrever comportamentos em altos ou baixos níveis de abstração. Tipicamente, em uma modelagem de alto nível, os diagramas de atividades são freqüentemente usados, e também em projetos de implementações muito detalhados ou nos efeitos comportamentais das transições de máquinas de estados.

A Figura 6 (extraída de [17]) exibe um diagrama de atividades muito simples. Similar à Máquina de Estados Finitos, este diagrama possui nodos inicial e final, representando o início e o fim do comportamento pelo círculo preenchido e pelo círculo preenchido inscrito em outro círculo, respectivamente. Os símbolos com nomes *Retorna 1* e *Retorna -1* são símbolos de atividades onde o nome refere-se a alguma atividade a ser realizada. O nodo em forma de losango significa um símbolo de decisão, e os colchetes significam as situações a serem verificadas.

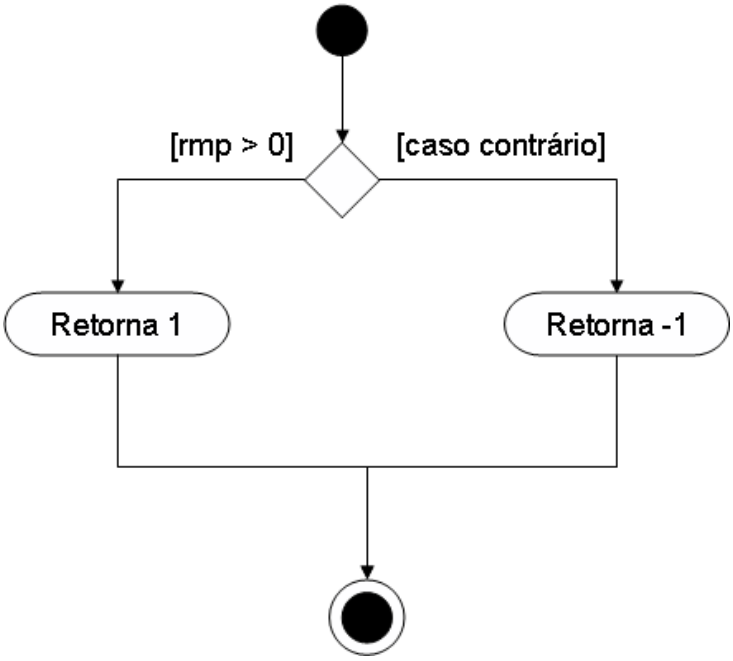

Figura 6 – Diagrama de atividades.

3.1.3 Modelos de Cadeias De Markov

Uma cadeia de Markov é um processo estocástico caracterizado por estados formadores de um espaço de estados discretos (ou seja, contáveis ou contavelmente infinitos) [22]. Um processo estocástico é definido como uma família de variáveis aleatórias $\{X_t : T \in T\}$ onde cada variável aleatória X_t é indexada pelo parâmetro $t \in T$, sendo este parâmetro normalmente o tempo, se $T \subseteq R_+ = [0, \infty)$. O conjunto de todos os valores possíveis de X_t (para cada $t \in T$) é conhecido como o espaço de estados do processo estocástico. Se o espaço temporal é também discreto, então a cadeia de Markov é chamada de Cadeia de Markov em tempo discreto. Caso o espaço temporal seja contínuo (incontável), a cadeia de Markov é chamada de Cadeia de Markov em tempo contínuo [23].

A cadeia de Markov é um processo estocástico onde seu comportamento dinâmico é tal que é que os estados anteriores são irrelevantes para a predição dos estados seguintes, desde que o estado atual seja conhecido [24].

Uma maneira simples de visualizar um tipo específico de cadeia de Markov é através de uma Máquina de Estados Finitos. Se você está no estado y no tempo n, então a probabilidade de que você se mova para o estado x no tempo $n + 1$ não depende de n, e somente depende do estado atual y em que você está. Assim em qualquer tempo n, uma cadeia de Markov finita pode ser caracterizada por uma matriz de probabilidades cujo elemento (x, y) é dado por $P(X_{n+1} = x|X_n = y)$ e é independente do tempo n. Estes tipos de cadeia de Markov finitas e discretas podem também ser descritas por meio de um grafo dirigido, onde cada aresta é rotulada com a probabilidade de transição de um estado a outro sendo estes estados representados como

os nós conectados pelas arestas [23].

Um exemplo de cadeia de Markov representado na forma de grafo dirigido é apresentado na Figura 7, referente a seguinte matriz de probabilidades: $P^1 = \begin{bmatrix} 0,75 & 0,25 \\ 0,5 & 0,5 \end{bmatrix}$

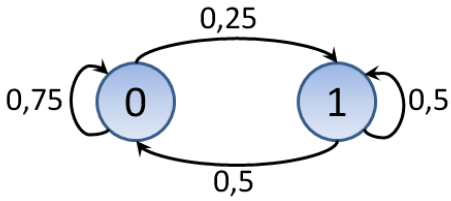

Figura 7 – Cadeia de Markov na forma de grafo dirigido.

Desta forma, existe uma probabilidade de 25% para a transição do estado "0" para o estado "1" e uma probabilidade de 75% para continuar no estado "0".

3.1.4 Modelos de Redes de Petri

As Redes de Petri foram originalmente apresentadas por Carl Adam Petri, em 1962 [25]. O modelo de Rede de Petri consiste de estados e transições entre estados, onde o estado é chamado de *marca* [26]. Uma marca muda quando uma transição ocorre. Formalmente, uma Rede de Petri básica tem a estrutura na forma *N = (P,T,F)*, onde *P* representa o conjunto de posições, *T* o conjunto de transições e $F \subseteq (P \times T) \cup (T \times P)$ é o conjunto de arcos direcionados. Uma marca μ da estrutura *N* é um mapeamento $\mu : P \rightarrow N$. Redes de Petri possuem uma representação gráfica como exemplificada na Figura 8. As posições são representadas por círculos e as transições são representadas por curtos segmentos de linha. Os arcos direcionados são representados por setas entre as posições e as transições. A Figura 8 representa uma Rede de Petri com $P = p_1, p_2, p_3, p_4, p_5$, $T = t_1, t_2, t_3, t_4$ e F = $(t_4, p_1), (p_1, t_1), (p_2, t_1), (p_3, t_1), (t_1, p_4), (p_4, t_3), (p_4, t_2), (t_3, p_5), (t_2, p_5), (t_2, p_2)$.

É interessante considerar a marca μ como um mapeamento e como um vetor. O vetor marca é definido como $[\mu(p_1), \mu(p_2), ..., \mu(p_n)]^T$, onde $p_1, p_2, ..., p_n$ são as posições da rede. Na literatura, uma forma equivalente de representar que a posição *p* possui a marca $\mu(p)$ é que p possui $\mu(p)$ fichas. A marca pode também ser representado graficamente. A Figura 8 apresenta a ficha como pequenos círculos pretos. Será denotado $\mu(p_i)$ como μ_i, então, na Figura 8, μ_1 = 1, μ_2 = 0, μ_3 = 1, μ_4 = 1 e μ_5 = 2. Assim, o vetor marca é $\mu = [1, 0, 1, 1, 2]^T$. A estrutura de uma Rede de Petri pode ser descrita por pré/pós-operações, onde uma pré-operação de uma posição *p* é o conjunto de transições de entrada de $p : \bullet p = t \in T : (t, p) \in F$. Uma pós-operação de uma posição *p* é o conjunto de transições de saída de $p : p\bullet = t \in T : (t, p) \in F$. Definições similares são aplicadas em transições, podendo ser estendidas para conjuntos de posições ou transições. Por exemplo, se $A \subseteq P$, então $\bullet A = \cup_{p \in A} \bullet p$ e $A\bullet = \cup_{p \in A} p\bullet$. Como exemplo, na

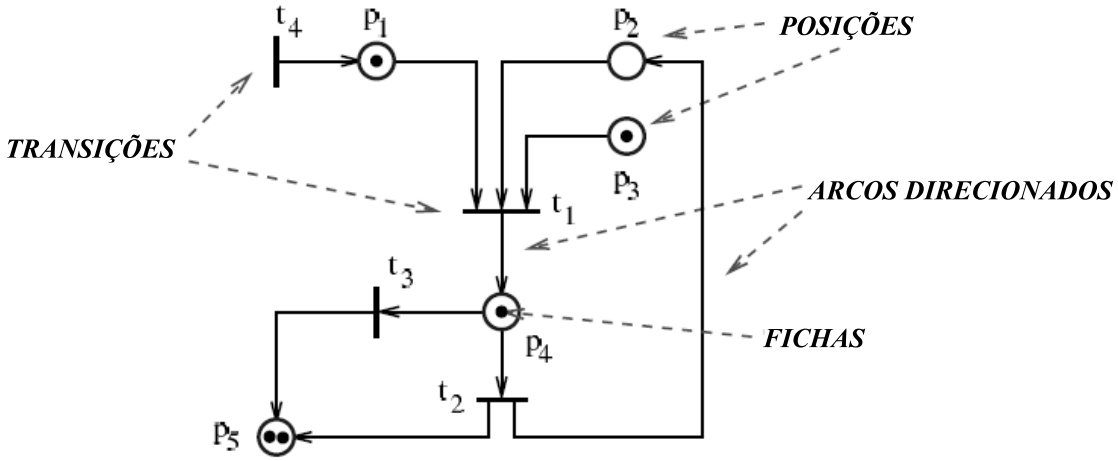

Figura 8 – Representação gráfica de uma Rede de Petri.

Figura 8 têm-se $\bullet p_1 = \{t_4\}$, $p_4\bullet = \{t_2, t_3\}$, $t_2\bullet = \{p_2, p_3\}$ e $\bullet\{p_2, p_3\} = \{t_2\}$. Como $\bullet p_3 = \emptyset$, $\bullet t_4 = \emptyset$ e $p_5\bullet = \emptyset$. É dito que *p* é uma *posição origem* se $\bullet p = \emptyset$ e uma *posição destino* se $p\bullet = \emptyset$. Transições origem e destino são similarmente definidas.

A operação de uma Rede de Petri é realizada da seguinte forma: a marca μ habilita a transição *t* se $\forall p \in \bullet t : \mu(p) \geq 1$. Quando μ habilita *t* e *t* dispara, a marca é alterada. Sendo $\mu^{'}$ a nova marca obtida pelo disparo de *t*. A marca $\mu^{'}$ satisfaz:

$$\mu^{'}(p) = \begin{cases} \mu(p) + 1 & \text{se} \quad p \in t\bullet \setminus \bullet t, \\ \mu(p) - 1 & \text{se} \quad p \in \bullet t \setminus \text{t}\bullet, \\ \mu(p) & \text{em outros casos.} \end{cases}$$

A notação $\mu \xrightarrow{t} \mu^{'}$ é usada para expressar que ao disparar *t* em μ resulta em uma nova marca $\mu^{'}$. Como um exemplo, as transições habilitadas na Figura 8 são t_2, t_3 e t_4. A transição t_1 não é habilitada, devido a μ_2=0 e $p_2 \in \bullet t_1$. Exemplos de disparos entre as transições podem ser vistos na Figura 9.

Uma modelagem de uma Rede de Petri é apresentada a seguir. Considerando uma rede de computadores na qual usuários podem enviar arquivos para imprimir em um servidor de impressoras central. As requisições de impressão são armazenadas em uma fila de impressão, que é lida pelo servidor de impressão. O servidor pode utilizar duas impressoras: LP1 e LP2. Cada uma das duas impressoras pode processar somente um serviço de impressão por vez. Quando o servidor começa a processar uma requisição, ele espera alguma impressora tornar-se disponível ou seleciona uma das impressoras e espera a impressora tornar-se disponível. Quando a impressora selecionada está disponível, a requisição é enviada à impressora. Finalmente, o servidor notifica o usuário quando o serviço de impressão está completo. Assume-se que o servidor de impressão processa, no máximo, duas requisições ao mesmo tempo.

O modelo de Rede de Petri é exibido na Figura 10. A Rede de Petri é apresentada na marca inicial, para que o servidor e as impressoras estejam ociosas, esperando por requisições. Pode-

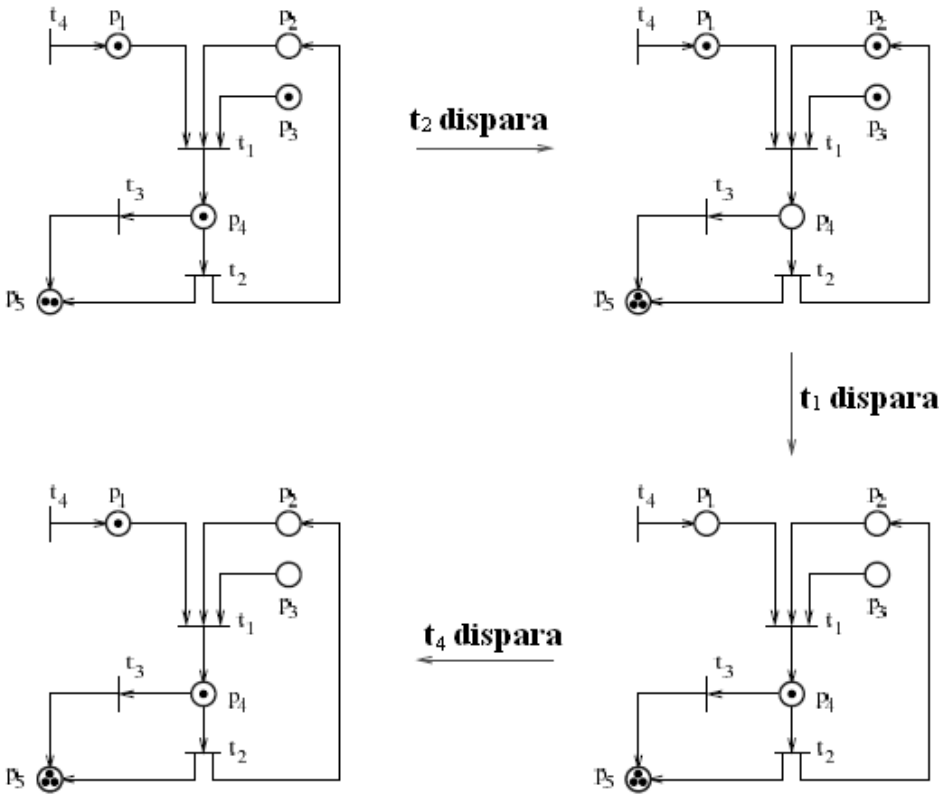

Figura 9 – Exemplos de disparos entre transições.

se notar que μ_1=2, indicando que o servidor pode somente processar duas requisições ao mesmo tempo.

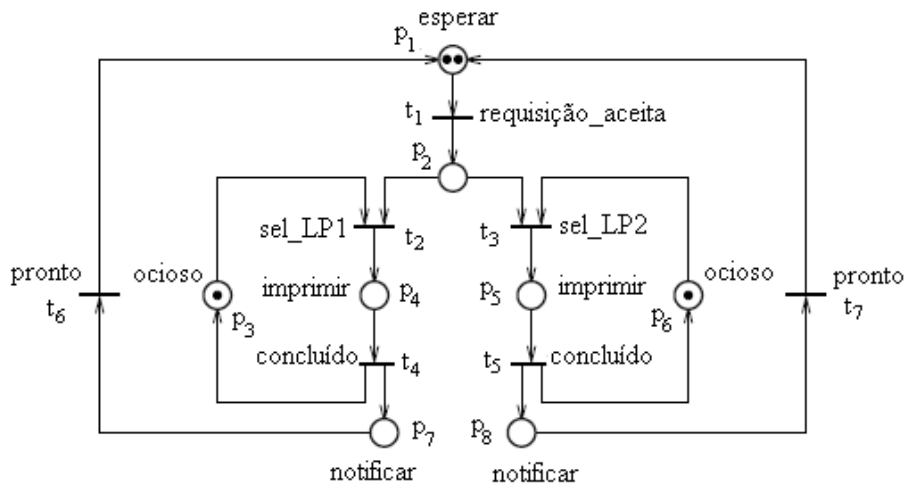

Figura 10 – Modelagem em Rede de Petri de um servidor de impressão.

Em alguns problemas é interessante possuir múltiplos arcos entre uma posição e uma transição. Situações como esta é necessário modelar em uma Rede de Petri generalizada, introduzindo uma função **peso**. Tal função atribui a cada arco um peso diferente, como exemplificado na Figura 11. Com isso, a estrutura de uma Rede de Petri torna-se uma quádrupla *N = (P,T,F,W)*,

onde *P* é o conjunto de posições, *T* é o conjunto de transições, $F \subseteq (P \times T) \cup (T \times P)$ é o conjunto de arcos entre as transições e $W : F \rightarrow N \setminus 0$ é a função peso.

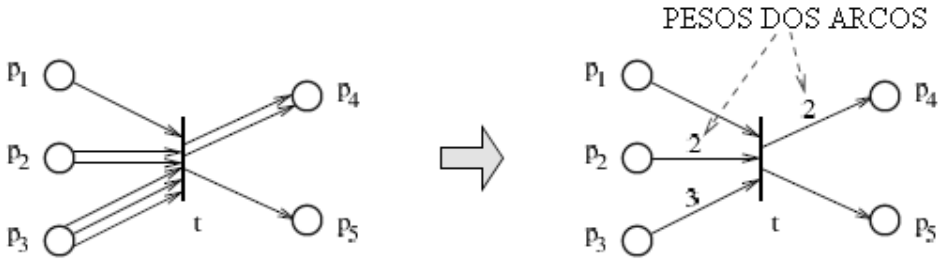

Figura 11 – Arcos com pesos.

3.1.5 Redes de Petri Estocásticas Generalizadas

As Redes de Petri provaram ser extremamente útil para o estudo de propriedades qualitativas ou lógicas de sistemas, exibindo comportamentos concorrentes e assíncronos. Porém, algumas situações exigem um atraso entre a transição entre os estados, assim como os pesos das transições simularem situações aleatórias. Sendo assim, em 1982, Michael K. Molloy propôs inserir na Rede de Petri características estocásticas, tornando a Rede de Petri isomórfica aos processos Markovianos homogêneos [27]. Após, em 1984, Aljmore Marsan, Balbo e Conte estenderam as Redes de Petri estocásticas para Redes de Petri estocásticas generalizadas (GSPN), onde uma transição possui a possibilidade de sofrer um atraso no disparo, podendo este atraso também ser estocástico. [28]. Com isso, as Redes de Petri estocásticas generalizadas assumem características determinísticas ou estocásticas, bastando inserir pesos nas transições com valores probabilísticos. Com isso a Rede de Petri assume as características determinísticas / não determinísticas / híbridas e temporais / não temporais. Desta forma, uma Rede de Petri pode então simular características como desempenho e dependência [29].

3.2 Processo de Testes de Software Baseados em Modelos

Teste baseado em modelos é definido como a automação do projeto de testes de caixa preta [30]. A diferença do teste de caixa preta usual é que ao invés de escrever manualmente os casos de teste baseado na documentação de requisitos, cria-se um **modelo** do comportamento do sistema em teste, modelo este que captura alguns dos requisitos. Então uma ferramenta de testes baseada em modelos é utilizada para gerar automaticamente os testes para este sistema em teste.

O processo de teste baseado em modelos pode ser dividido em cinco passos principais [30],

como mostrado na Figura 12:

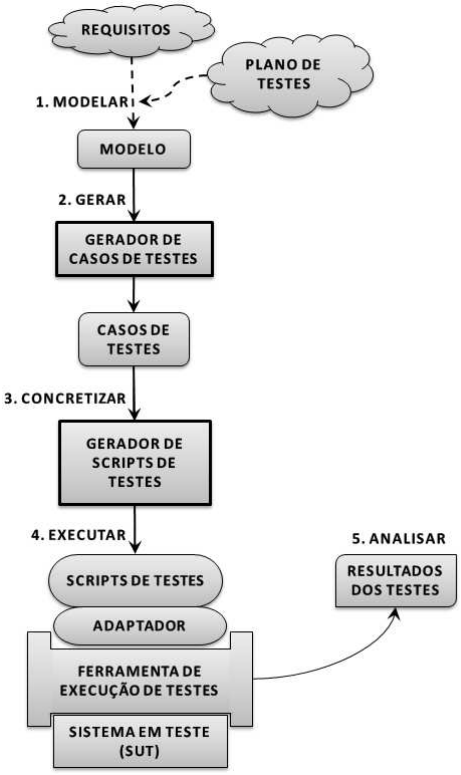

Figura 12 – Processo de teste baseado em modelos (ferramentas de teste estão nos retângulos com linhas grossas).

1. *Modelar* o SUT e/ou seu ambiente.

2. *Gerar* testes abstratos a partir do modelo.

3. *Concretizar* os testes abstratos para torná-los executáveis.

4. *Executar* os testes no SUT.

5. *Analisar* os resultados dos testes.

Os dois primeiros passos diferenciam o teste baseado em modelos dos outros tipos de teste. Em ferramentas *online* de testes baseados em modelos, do passo 2 até o passo 4, são unidos em um único passo, já em testes baseados em modelos de forma *offline*, são normalmente separados. Mas para critério de entendimento, é interessante explicar esses passos separadamente.

O primeiro passo de testes baseados em modelos é escrever um modelo abstrato do sistema que se deseja testar. Denomina-se modelo abstrato porque deveria ser muito menor e simples que o sistema em teste. O modelo deveria focar somente nos aspectos chave que se deseja testar e deveria omitir muitos dos detalhes do sistema. Durante a criação do modelo, pode-se

também incluir anotações com identificadores de requisitos para documentar claramente o relacionamento entre os requisitos informais e o modelo formal. Após a escrita do modelo, é conveniente usar ferramentas para verificar se o modelo está consistente e possui o comportamento desejado.

O segundo passo é a geração de testes abstratos a partir do modelo. Deve-se definir alguns critérios de seleção de testes, para definir quais testes deseja-se gerar a partir do modelo, pois são normalmente um número infinito de testes possíveis. O resultado principal deste paso é um conjunto de testes abstratos, os quais são seqüências de operações a partir do modelo. Como o modelo apresenta uma visão simplificada do SUT, os testes abstratos são carentes de alguns dos detalhes necessários do sistema em teste e não são diretamente executáveis. Muitas ferramentas de testes baseados em modelos também produzem uma matriz de rastreabilidade de requisitos ou vários outros relatórios de cobertura como saídas adicionais deste passo. A matriz de rastreabilidade de requisitos traça a conexão entre os requisitos funcionais e os casos de testes gerados. Os relatórios de cobertura fornecem algumas indicações de quão bem o conjunto de testes gerados exercitam todos os comportamentos do modelo. Estes relatórios apresentam dados sobre a cobertura do modelo, não sobre o sistema em teste, pois ainda não foram executados os testes neste sistema.

O terceiro passo transforma os testes abstratos em testes concretos executáveis. Esta transformação pode ser realizada utilizando uma ferramenta de transformação que utilizam vários *templates* e mapeamentos para traduzir cada teste abstrato em um *script* de teste executável, ou escrevendo um código que realiza esta tarefa.

O quarto passo é executar os testes concretos no sistema em teste. Com o teste online, os testes serão executados como foram produzidos, então a ferramenta de testes baseados em modelos gerenciará o processo de execução de testes e gravará os resultados. Com o teste *offline*, já se possui o conjunto de *scripts* concretos de testes gerados em alguma linguagem, então pode-se continuar utilizando a ferramenta de execução de testes, executá-los regularmente e gravar os resultados.

O quinto passo é analisar os resultados da execução do teste e aplicar as ações corretivas. Para cada teste que reportar um erro, deve-se determinar a falha que causou este erro. Esta falha pode ser devido a uma falha na geração do caso de teste ou no modelo ou talvez nos documentos de requisitos.

3.3 Considerações

Alguns trabalhos propõem a realização de testes baseados em modelos, por exemplo, testes funcionais a partir de máquinas de Estados Finitos estendidas [31], testes de desempenho a partir de modelos UML de atividades [32], testes de integração a partir de diagramas UML de estados (*StateChart Diagrams*) [21]. Outro trabalho analisa setenta e oito artigos sobre testes

baseados em modelos, sendo quarenta e sete deles baseados em modelos UML [33]. Como este último trabalho apresenta, grande parte das metodologias de testes baseados em modelos utilizam UML para a descrição dos modelos.

Em testes baseados em modelos as características dos modelos podem ser classificadas como determinísticas, não-determinísticas, temporizadas, não-temporizadas, discretas, híbridas e contínuas [34]. Tal classificação pode ser vista como uma linha de produtos, pois as metodologias são agrupadas conforme suas principais características. No capítulo seguinte será apresentado os conceitos sobre linha de produtos de software, conceitos necessários para a integração de testes de software baseados em modelos e linha de produtos de software, sendo esta integração o objetivo deste trabalho.

4 Linha de Produtos de Software

O sonho de um reuso de software massivo é tão antigo quanto a própria Engenharia de Software. Foram realizadas várias tentativas ou iniciativas de reusar software, mas o sucesso alcançado foi muito baixo. Estas iniciativas de reuso foram principalmente baseadas em uma abordagem focada em um reuso de pequena escala e *ad hoc*, sendo tipicamente em nível de código. O conceito de focar em um domínio específico como uma base para desenvolver artefatos reusáveis foi somente introduzido no início da década de 80 [35], sendo focado este contexto quase exclusivamente no desenvolvimento em um único domínio, baseado em ferramentas de geração automatizada de software.

O conceito de linha de produtos de software foi introduzido no início da década de 90. Uma das primeiras contribuições foi a descrição da análise de domínio orientada a *feature*, conhecida como método FODA (*Feature-oriented Domain Analysis*) [36]. Nesta mesma época, diversas empresas começaram a demonstrar um interesse pelo tema. Por exemplo, a empresa Philips apresentou o método de montagem em blocos [37].

Existem diferentes motivos que levaram as empresas a investirem na abordagem de linha de produtos de software, desde diminuição dos custos e tempo de desenvolvimento a um aumento da qualidade e da confiabilidade. Esta diminuição de custo e tempo de desenvolvimento está relacionada com o suporte ao reuso em larga escala durante o ciclo de desenvolvimento de software. Comparada às abordagens tradicionais de reuso [38], a economia pode ser de até 90% [39].

Para alcançar esta economia, é necessário inicialmente um investimento extra, pois a abordagem de linha de produtos exige a construção de artefatos reusáveis, uma mudança cultural na organização, etc. Existem diferentes estratégias para realizar este investimento, desde a abordagem conhecida como big-bang até uma estratégia incremental [39]. Como mostra a Figura 13 (extraída de [39]), o investimento começa a apresentar resultados a partir do terceiro produto da linha.

Normalmente, além da redução de custos de desenvolvimento, é alcançado também uma redução de custo de manutenção. Vários aspectos contribuem para esta redução, sendo principalmente o fato que a quantidade de código e documentação que precisa de manutenção é dramaticamente reduzido através do reuso. Outra implicação da abordagem de linha de produtos está no aumento da qualidade do software. Uma aplicação nova consiste (em um sistema complexo) em uma grande quantidade de requisitos e conseqüentemente, em uma grande quantidade de defeitos durante o seu desenvolvimento. Como na linha de produtos o foco está no reuso, os componentes arquiteturais já foram utilizados anteriormente, sendo seus defeitos detectados

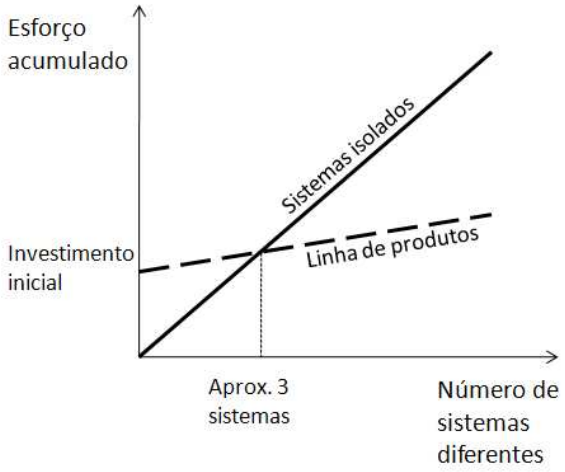

Figura 13 – Economia da linha de produtos de software.

e corrigidos durante o desenvolvimento da aplicação original destes componentes arquiteturais, logo conduzindo os produtos novos da linha a uma maior confiabilidade e segurança.

Outra vantagem da linha de produtos está na interface com o usuário. Ao invés de desenvolver uma interface para cada sistema, na linha de produtos todos os sistemas compartilham de uma interface similar, provendo uma maior usabilidade. Em alguns casos, a demanda deste tipo de vantagem é a base para a introdução da abordagem de linha de produtos no desenvolvimento de sistemas [40].

Uma linha de produtos pode ser resumida como um conjunto de *features*, onde algumas *features* são comuns a todos os produtos da linha e algumas *features* podem variar de um produto para outro. Uma *feature* é vista como uma abstração, podendo ser um requisito, uma funcionalidade, uma parte do produto, enfim, qualquer característica de um produto. Por exemplo, numa linha de produtos de automóveis, uma *feature* pode ser um pneu, uma determinada potência do motor, uma determinada pressão em um parafuso, enfim, qualquer característica relacionada ao produto em diferentes níveis de abstração.

Uma linha de produtos é abordada de duas maneiras, do ponto de vista de um produto específico e da linha como um todo. A abordagem que se foca na linha como um todo é conhecida como engenharia de domínio e a que foca exclusivamente no produto é chamada de engenharia da aplicação. A engenharia de domínio é o processo da linha de produtos de software no qual as características comuns e variáveis são definidas e concretizadas. A engenharia da aplicação é o processo da linha de produtos de software no qual os produtos da linha são montados através do reuso de artefatos do domínio e da exploração da variabilidade da linha de produtos [41]. A engenharia de domínio é composta por todo tipo de artefatos da linha (requisitos, projeto, implementação, testes, etc.).

A variabilidade da linha de produtos define o que pode variar na linha, ou seja, quais *features* a linha de produtos oferece. Esta variabilidade ocorre através da introdução de pontos de variabilidade, ou seja, locais onde é controlada a escolha de uma determinada *feature* em um

produto, definindo restrições e dependências entre as *features*.

Para obter um melhor entendimento da variabilidade da linha de produtos é interessante projetar alguma forma de modelagem. Uma maneira de modelar as *features* e a variabilidade da linha de produtos é através do modelo de *features* provido pelo método FODA [36]. O modelo de *features* aborda os conceitos e as propriedades de estruturas comuns e variáveis no domínio de interesse através de um modelo em árvore. Um modelo de *features* do método FODA consiste de um diagrama de *features*, exibindo a decomposição hierárquica das *features* com relacionamentos obrigatórios, alternativos e opcionais. Este método pode consistir também de uma descrição de cada *feature*, além de regras de composição onde são indicadas quais combinações de *features* são válidas e quais não são, além de uma lógica para selecionar ou não uma *feature*. A Figura 14 ilustra um exemplo de modelo de *features* para uma linha de montagem de automóveis.

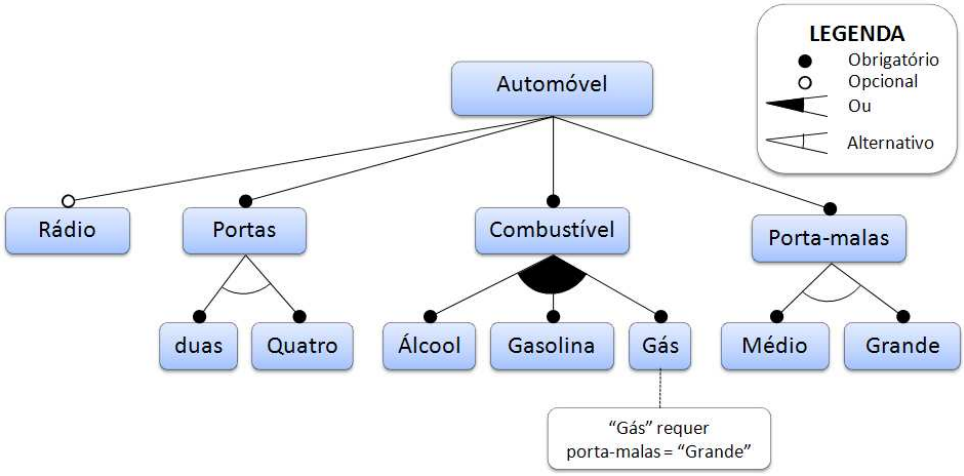

Figura 14 – Modelo de *features* parcial de uma linha de produtos de automóveis.

Outros métodos para modelar *features* incluem o método FORM (*Feature Oriented Reuse Method*) [42], o método FeatuRSEB (*Feature Reuse-Driven Software Engineering Business*) [43], e a notação de Jan Bosch [44], sendo que todas estes métodos são baseados principalmente no método FODA.

4.1 Arquitetura de Software

A arquitetura é uma parte fundamental de qualquer sistema de software não trivial. Resumidamente, a arquitetura limita o escopo das funcionalidades que o sistema pode manipular, determinando a qualidade dos atributos do sistema [41]. Uma arquitetura comum é essencial para um conjunto de produtos poder compartilhar eficientemente grande parte de sua implementação.

Alguns pontos são importantes em uma visão mais aprofundada de arquitetura de software.

Primeiramente, a arquitetura define os elementos de software [45]. A arquitetura engloba as informações de como os elementos arquiteturais se relacionam, ou seja, significa que a arquitetura omite certas informações sobre os elementos que não pertencem às suas interações. Logo, uma arquitetura é praticamente uma abstração de um sistema que suprime os detalhes de elementos que não afetam como tais elementos usam, são usados, se relacionam ou interagem com outros elementos. Os elementos interagem entre si através de interfaces que particionam um elemento em partes públicas e privadas. A arquitetura se concentra no lado público da divisão. Detalhes privados (como os que se preocupam somente com a implementação interna) não são considerados detalhes arquiteturais.

Em segundo lugar, a definição de arquitetura torna claro que os sistemas abrangem mais que uma estrutura e que nenhuma estrutura pode declarar-se como a arquitetura. Por exemplo, a maioria dos projetos complexos são particionados em unidades de implementação, sendo que estas unidades adquirem responsabilidades específicas e são freqüentemente a base para que uma equipe de desenvolvimento possa atuar. Este tipo de elemento engloba programas e dados que os programas em outras unidades de implementação irão chamar ou acessar, sendo que estes programas e dados são privados. Em um grande projeto, estes elementos provavelmente serão subdivididos e repassados para equipes específicas. Este é o tipo de estrutura que é freqüentemente usado para descrever um sistema, pois foca em como as funcionalidades do sistema são divididas para que as equipes possam atuar.

Outras estruturas são muito mais focadas em como os elementos interagem entre si em tempo de execução para que uma função do sistema seja executada. Supondo que o sistema deve ser implementado como um conjunto de processos sendo executados em paralelo. Os processos que existirão em tempo de execução, as várias unidades de implementação que são amarradas seqüencialmente para formar cada processo e a relação de sincronização entre os processos formam outro tipo de estrutura freqüentemente usada para descrever um sistema.

Nenhuma dessas estruturas sozinhas são a arquitetura do sistema, embora todas as estruturas possuam informações arquiteturais. A arquitetura consiste destas estruturas, além de muitas outras.

Em terceiro lugar, a definição de arquitetura implica que todo sistema possua uma arquitetura, visto que todo o sistema pode ser representado como elementos e a relação entre estes elementos. Embora todo sistema possua uma arquitetura, isto não significa que esta arquitetura seja conhecida por todos os envolvidos no sistema. Talvez todas as pessoas envolvidas no sistema tenham se desligado do projeto, a documentação tenha sido perdida (ou talvez nunca produzida) e o código fonte não tenha sido entregue junto com o sistema. Isto diferencia a arquitetura de um sistema com a representação da arquitetura. A arquitetura pode existir independente de sua descrição ou especificação, ressaltando assim a importância da documentação.

Por fim, o comportamento de cada elemento é parte da arquitetura a medida que o comportamento pode ser observado pelo ponto de vista de um outro elemento. Sendo assim, este comportamento precisa ser projetado tendo em vista o sistema como um todo, logo este com-

portamento é parte da arquitetura do sistema.

4.2 Arquitetura de Linha de Produtos de Software

Quando uma linha de produtos de software compartilha a mesma arquitetura, esta arquitetura é conhecida como arquitetura de referência [41]. A arquitetura de referência descreve uma arquitetura genérica que provê uma solução para uma variedade de produtos da linha, contendo a variabilidade de *features* que são oferecidas para instanciar um determinado produto, embora nem todas as *features* sejam visíveis em nível arquitetural.

A arquitetura é essencialmente importante na linha de produtos, pois uma arquitetura de referência simplifica o compartilhamento de artefatos entre diferentes produtos, tais como modelos, documentação, código, entre outros. Os artefatos reusáveis são criados baseados na arquitetura de referência, fazendo com que os engenheiros possam presumir as situações onde os artefatos, que estão sendo criados ou desenvolvidos, serão utilizados. Desta forma, a criação de artefatos apresenta uma redução de complexidade e conseqüentemente, de custos.

Os elementos da arquitetura são compostos por componentes [46] [47] [48] [49], onde um componente é definido como uma parte de um código com funcionalidades bem definidas e que pode ser conectado a outros componentes através de sua interface [50]. A tecnologia de componentes possibilita que os desenvolvedores componham uma aplicação em partes fracamente acopladas, ou seja, os componentes podem ser desenvolvidos, compilados, conectados e carregados separadamente. Somente em tempo de execução os componentes serão combinados em um sistema funcional. Com isso os desenvolvedores podem focar em na complexidade de um componente em particular, tornando o componente uma caixa preta.

Sendo assim, um componente é compostos por sua implementação interna e sua interface, como ilustrado na Figura 15.

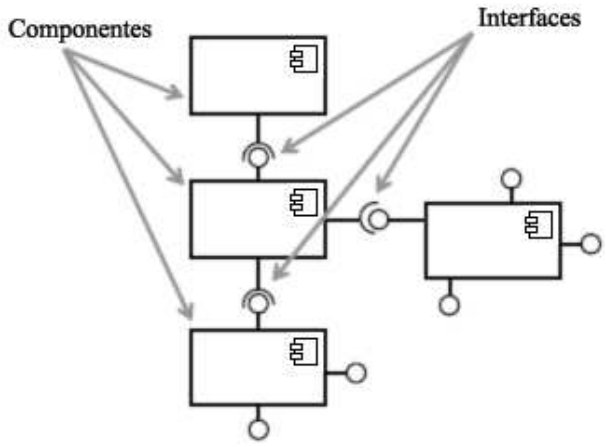

Figura 15 – Estrutura dos componentes.

Através do uso de componentes, a arquitetura de referência precisa implementar todas as possíveis variabilidades expostas no modelo de *features*, sendo que em alto-nível, exites três técnicas para prover a variabilidade na arquitetura: adaptação, substituição e extensão [41].

Na técnica de adaptação, existe uma única implementação disponível para um certo componente, mas existem diferentes interfaces para ajustar o comportamento deste componente. Esta interface pode possuir a forma de um arquivo de configuração, de parametrização ou caminhos para o código-fonte do componente para nomear as opções. Na técnica de substituição, são disponíveis várias implementações de um componente. Cada implementação adere a especificação do componente como descrito na arquitetura. Na engenharia de aplicação, uma das implementações disponíveis é escolhida, ou uma implementação específica ao produto é desenvolvida através da adaptação, seguindo as especificações da arquitetura.

A técnica de extensão exige que a arquitetura ofereça interfaces que permitam adicionar novos componentes na arquitetura. Os componentes adicionados podem ou não serem específicos de um produto. A extensão difere-se da substituição porque na extensão somente interfaces genéricas são disponíveis para adicionar componentes, enquanto na substituição a interface especifica exatamente o que o componente deve fazer, somente variando como o componente faz a sua função. Através da extensão, componentes podem ser selecionados através da mesma interface, enquando na substituição um componente é substituído por outro componente. A Figura 16 (estraída de [41]) ilustra estas três técnicas básicas.

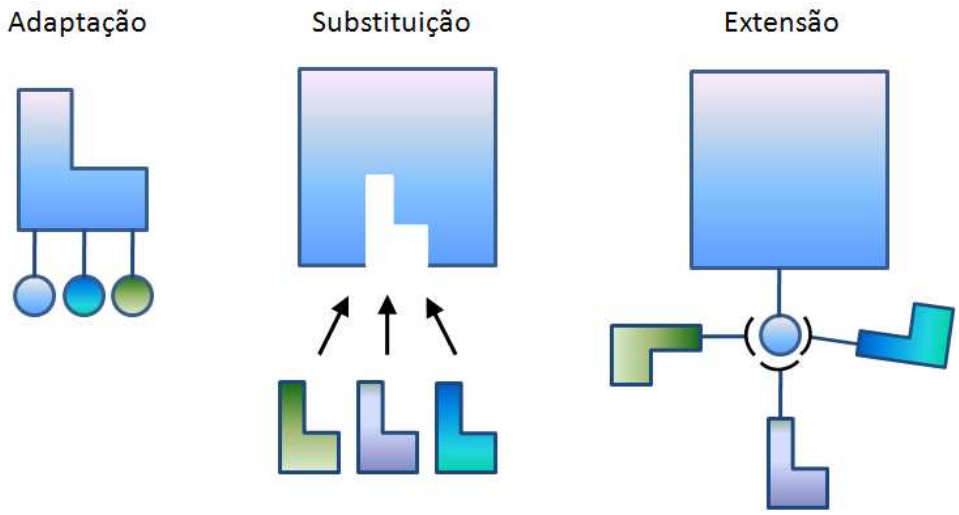

Figura 16 – Três técnicas básicas de implementar variabilidades em uma arquitetura.

4.3 Ferramentas Existentes

O projeto AMPLE [51] avaliou as três principais ferramentas existentes que possibilitam a concretização de uma linha de produtos de software, sendo elas a pure::variants [52], Gears [53]

e fmp2rsm [54].

4.3.1 Pure::variants

Pure::variants [52] é uma ferramenta comercial que provê uma representação explícita para os conjuntos de variantes de componentes (sendo estes conjuntos partes dos modelos da família de produtos) e de *features*. Os modelos da família são criados a partir de artefatos concretos (normalmente arquivos), da declaração de relacionamentos entre os artefatos e associando as implementações existentes aos componentes. O relacionamento entre as *features* e os componentes é descrito através de restrições baseadas em lógica e através de tabelas. Os modelos de descrição das variantes são definidos para representar um conjunto de modelos de configuração e de transformação das variantes. Cada variante possui um modelo de descrição de variante associado. As transformações ocorrem através da substituição de fragmentos de arquivos que representam os artefatos.

4.3.2 Gears

Gears [53] é uma ferramenta comercial que permite a definição de *features*, de pontos de variabilidade e de produtos através da seleção de variantes.

Um artefato na ferramenta Gears é um grupo de vários formatos de arquivos relacionados com dados de manutenção, testes e do código-fonte, provendo uma noção de pontos de variabilidade que representam partes de artefatos que podem ser configurados de acordo com as *features* existentes.

A ferramenta Gears provê uma linguagem de propósito geral para definir como diferentes *features* podem modificar um artefato. Esta linguagem basicamente busca testar a relação entre as *features* e os artefatos. Já as *features* são definidas através de uma linguagem que busca caracterizar conjuntos de *features* que são simultaneamente aplicáveis no produto ou são mutualmente exclusivas.

4.3.3 fmp2rsm

A ferramenta fmp2rsm é uma implementação de *templates* de modelos baseados em *features* para as ferramentas de modelagem IBM Rational Software Modeler [55] e IBM Rational Software Architect [56]. A fmp2rsm integra o *plugin* de modelagem de *features* (fmp) [57] com o Rational Software Modeler, habilitando a modelagem de linha de produtos em UML e a derivação automática de produtos. O fmp é um *plugin* para a IDE Eclipse [58] desenvolvido

para editar e configurar modelos de *features*.

A fmp2rsm provê uma representação explícita para os conjuntos de famílias de modelos [59]. Uma família de modelos é representada por um modelo de *features* e um modelo de *templates* [51]. O modelo de *features* define as restrições das *features*, já o modelo de *templates* contém a união de todas as instâncias de *templates* existentes para gerar a linha de produtos. A família de modelos responde ao conjunto de instâncias de *templates*.

A ferramenta fmp2rsm é de uso livre, mas somente funciona em conjunto com as ferramentas comerciais Rational Software Modeler e Rational Software Architect.

4.4 Considerações

A abordagem de linha de produtos de software tenta prover um reuso massivo em todo o ciclo de desenvolvimento do software, mas este reuso não é alcançado de forma simples. Este reuso é alcançado através de um esforço substancial de todos os envolvidos no desenvolvimento da linha, provendo a longo prazo um retorno atraente. As ferramentas atuais para estruturar uma linha de produtos são comerciais ou vinculadas a alguma dependência comercial, provendo barreiras na sua utilização.

No capítulo a seguir será descrita a arquitetura de linha de produtos proposta para o desenvolvimento de ferramentas de testes baseados em modelos, linha esta embasada nos capítulos apresentados até o momento.

5 Arquitetura de Linha de Produtos Proposta

Atualmente existem uma série de abordagens diferentes para a realização de testes baseados em modelos. Cada uma dessas abordagens apresenta um nível de complexidade para transformar esta abordagem em uma ferramenta concreta para derivar os casos de teste. No momento em que procura-se desenvolver estas ferramentas de forma a suprir as necessidades de diferentes fases do processo de teste, várias características de uma determinada abordagem é encontrada em outras abordagens. Ao desenvolver estas ferramentas individualmente, estas características em comum são implementadas novamente para cada ferramenta, gerando um retrabalho desnecessário.

Uma solução para reduzir o retrabalho é através do projeto de uma arquitetura de linha de produtos de software, onde após um estudo das diferentes abordagens de testes baseados em modelos, as características comuns destas abordagens são agrupadas, provendo um reuso destas características de forma a evitar o retrabalho.

Após o estudo das diferentes abordagens de testes baseados em modelos apresentados em [33], concluiu-se que as abordagens se resumem em um manipulador do modelo do sistema em teste, um modelo comportamental que simula o comportamento deste modelo de entrada e por fim um gerador de casos de teste, responsável por executar esse modelo simulado e desta execução extrair as informações para a montagem dos casos de teste. Este gerador pode ser visto de duas maneiras: como um gerador de casos de teste e como um gerador de *scripts* de teste. No gerador de casos de testes são abordadas as características em nível conceitual, enquanto no gerador de *scripts* de teste são abordadas as características tecnológicas dos casos de teste.

A Figura 17 exibe o modelo de *features* elaborado baseado nesse estudo. Este modelo de *features* é composto de quatro categorias: modelo de entrada, modelo comportamental, gerador de casos de teste e gerador de *scripts* de teste. No modelo de entrada são inseridas as *features* responsáveis por extrair e fornecer as informações providas pelos modelos utilizados para representar o sistema em teste. No modelo comportamental, são providas duas *features*: Rede de Petri estocástica generalizada (representada na Figura 17 como *GSPN*) e uma Máquina de Estados Finitos com entradas e saídas (representada na Figura 17 como *FSM*). No decorrer deste capítulo será apresentado o motivo da existência dessas *features*. Nos geradores de casos de teste e scripts são inseridas as *features* responsáveis pela geração de casos de teste e *scripts* para os diferentes tipos de testes que irão compor a linha de produtos concreta, respectivamente.

Baseado neste estudo, foi definida uma arquitetura de referência para uma linha de produtos de testes de software baseados em modelos. A Figura 18 ilustra esta arquitetura básica, sendo composta por cinco categorias: A categoria responsável por manipular o modelo de entrada (1),

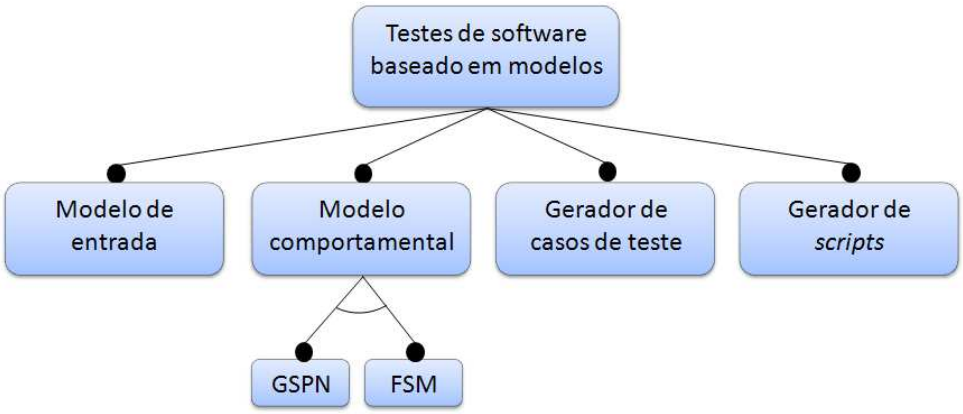

Figura 17 – Modelo de *features* base para a linha de produtos proposta.

a categoria responsável por simular o comportamento deste modelo de entrada (2), a categoria responsável pela geração dos casos de teste para um determinado tipo de teste (4), a categoria responsável pela geração dos *scripts* de um tipo de teste para uma determinada tecnologia (5) e a categoria formada por conectores responsáveis por interligar todas as categorias (3).

Para garantir um critério de cobertura relevante para a representação da simulação do comportamento das abordagens, a categoria 2 provê duas possibilidades: uma Máquina de Estados Finitos com entradas e saídas (indicada na Figura 18 como *FSM*) e uma Rede de Petri estocástica generalizada (indicada na Figura 18 como *GSPN*), cobrindo assim características determinísticas (*FSM*), estocásticas, híbridas e temporais (*GSPN*).

A categoria do manipulador do modelo de entrada (categoria 1 da Figura 18) é responsável por carregar o modelo de entrada e extrair as informações dos elementos deste modelo, provendo para a categoria de simulação comportamental (categoria 2) as informações necessárias para a sua criação.

A categoria de simulação do modelo comportamental é encarregada de simular a execução do modelo de entrada juntamente com as informações contidas neste modelo de entrada relevantes para o determinado tipo de teste a ser realizado. Ou seja, se o teste a ser realizado é um teste de desempenho, por exemplo, o modelo de entrada deve descrever os passos a serem realizados no teste, assim como as características deste teste, tais como dados temporais, restrições, etc.

A categoria de geração de casos de teste (categoria 4 da Figura 18) visa absorver as informações providas pelas categoria 2 e 3 e aplicar algoritmos de geração de casos de teste, assim como estruturar estes casos de teste de forma legível e inteligível.

A categoria de geração de *scripts* de teste (categoria 5 da Figura 18) tem o mesmo objetivo da categoria de geração de casos de teste, mas apresenta também a adequação à tecnologia a ser utilizada para a automação dos testes. Ou seja, adequar as informações providas pela simulação do modelo comportamental e pela aplicação de algoritmos de geração de *scripts* de teste ao formato que a tecnologia, onde este *scripts* vão ser executados, exige.

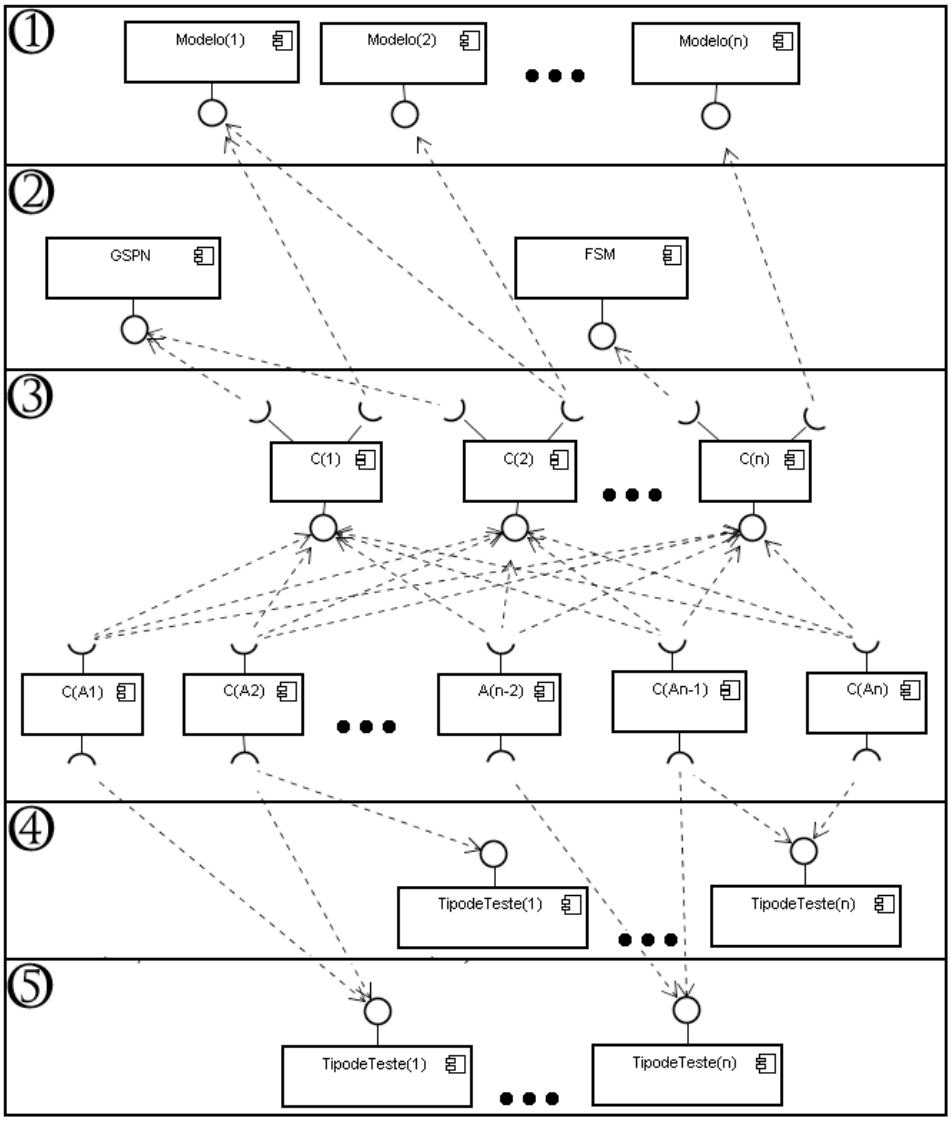

Figura 18 – Arquitetura básica das abordagens de teste baseados em modelos.

Por fim, a categoria 3 é formada por conectores, responsáveis por interligar as demais categoria da arquitetura. Estes conectores são divididos em dois grupos, os conectores responsáveis por interligar as categorias 1 e 2 (representados na Figura 18 por C(1)...C(n) e aqui denominados Grupo A) e os conectores responsáveis por interligar as categoria 4 e 5 com os conectores do grupo anteriormente descrito (representados na Figura 18 por C(A1)...C(An) e aqui denominado como Grupo B). Cada abordagem de teste possui um conector específico do grupo A, pois este conector é encarregado de extrair da categoria 1 as informações pertinentes ao tipo de teste a ser realizado e fornecer estas informações à categoria 2, além de prover o acesso da categoria 2 aos conectores do grupo B, que são encarregados de realizar a execução da simulação comportamental e fornecer as informações extraídas da simulação às categorias 4 e 5.

Cabe ressaltar que esta arquitetura de referência é uma arquitetura genérica, onde a especialização desta arquitetura é realizada através das informações providas pelo modelo de *features*

da linha de produtos a ser concretizada.

Para a instanciação de um produto em particular, é necessário selecionar quais elementos destas categorias são necessários para compor este produto, determinados pelas informações contidas no modelo de *features* da linha de produtos.

Para uma situação inicial, onde objetiva-se a criação de uma linha de produtos de testes baseados em modelos, deve-se seguir o seguinte *workflow*:

- Realizar a engenharia de domínio;
- Realizar a engenharia de aplicação;
- Criar o modelo de *features* de cada produto;
- Criar o modelo de *features* da linha de produtos;
- Para cada *feature*, inserir na respectiva categoria da arquitetura de referência um componente responsável por concretizar esta *feature*;
- Para cada relacionamento entre *features*, criar os conectores da categoria 3 da arquitetura de referência responsáveis por concretizar este relacionamento;

Para instanciar um produto da linha deve-se:

- De posse do modelo de *features* do produto, do modelo de *features* da linha e da arquitetura de referência, verificar quais são os componentes da arquitetura que concretizam as *features* escolhidas para compor o produto;
- Verificar na categoria 3 da arquitetura de referência quais conectores realizam e respeitam os relacionamentos entre as *features* escolhidas para compor o produto;
- Gerar o produto, isto é, instanciar os conectores e os componentes relativos as *features* escolhidas, formando assim a ferramenta de testes baseados em modelos;

Para a seleção das *features* é interessante observar uma possível ordem temporal, de acordo com os seguintes passos:

- Selecionar primeiramente as *features* do modelo comportamental, desta forma definindo o modo de manipulação dos testes, se de forma estocástica ou determinística;
- Selecionar as *features* relativas a geração de *scripts* de casos de teste, definindo assim o tipo de teste a ser realizado e quais tipos de teste estão de acordo com o modelo comportamental escolhido;
- Selecionar os modelo de entrada que respeitam as restrições impostas pelas *features* anteriores.

De posse destas informações, é possível instanciar um produto da linha de uma forma inicial. Mas à medida que novas *features* necessitam ser inseridas na linha de produtos, alguns critérios precisam ser analisados. A inserção de novas *features*, basicamente seguem dois princípios: a inserção de um novo tipo de teste de software baseado em modelos e a inserção de uma nova *feature* nos tipos de testes já existentes na linha de produtos. Para a inserção de um novo tipo de teste, inicialmente deve-se analisar qual a natureza deste teste, se é estocástica ou determinística. De posse desta informação, verifica-se se as atuais *features* de modelos comportamentais (Rede de Petri EstocásticaGeneralizada e Máquina de Estados Finitos) são capazes de realizar o que este novo tipo de teste se propõe a fazer. Em caso afirmativo, aplica-se o reuso destas *features*, caso contrário, uma nova *feature* é inserida no modelo de *features* e um novo componente é inserido na categoria 3 da arquitetura de referência.

De posse das informações de como os modelos de entrada serão manipulados, é necessário determinar quais serão estes modelos de entrada. Se estes modelos já existem na linha de produtos, é aplicado o reuso, caso contrário deve-se inserir estas *features* na categoria de modelos de entrada, assim como concretizar estas *features* com componentes na categoria 1 da arquitetura de referência. Em sequida, deve-se inserir as *features* relativas a geração dos casos de teste e de *scripts* deste novo tipo de teste no modelo de *features* e concretizá-las nas categorias 4 e 5 da arquitetura de referência, visto que, como é um novo tipo de teste, dificimente existirá na atual linha de produtos, não cabendo aqui a possibilidade de reuso. Por fim, são definidos os relacionamentos entre as *features* deste novo tipo de teste e das demais *features* existentes, concretizando na categoria 3 da arquitetura de referência os conectores responsáveis por realizar estes relacionamentos.

Para a inserção de uma nova *feature* nos tipos de teste já existentes na linha de produtos, é interessante verificar se alguma *feature* não realiza a função que esta nova *feature* se propõe a realizar. Em caso negativo, deve-se verificar o impacto desta nova *feature* na linha de produtos, ou seja, em quais relacionamentos entre *features*, restrições, etc., a adição desta nova *feature* impacta. Após esta análise, a nova *feature* é inserida no modelo de *features*, a sua concretização é inserida na arquitetura de referência e os relacionamentos impactados no modelo de *features* são ajustados, assim como os conectores da categoria 3 da arquitetura de referência.

Na seção seguinte será apresentado um estudo de caso de uma linha de produtos concreta, a qual possui uma arquitetura de referência composta pela arquitetura aqui proposta e pelas informações oferecidas pelo modelo de *features* deste estudo de caso específico.

5.1 Estudo de Caso

Para um melhor entendimento da linha de produtos proposta, foi realizado um estudo de caso de uma linha de produtos para três tipos de abordagens de testes baseados em modelos: teste funcional, teste de desempenho e teste de segurança. Estes tipos de teste não foram escolhidos

ao acaso, fazem parte, assim como o tema deste trabalho, a um projeto de pesquisa do Centro de Pesquisa em Computação Aplicada da Pontifícia Universidade Católica do Rio Grande do Sul em colaboração com a empresa Hewlett-Packard. O teste de desempenho baseado em modelos visa auxiliar na configuração de realocação de recursos em ambientes virtualizados com SLA [60], o teste de segurança procura auxiliar na geração de testes com requisitos não-funcionais [61] e o teste funcional busca ampliar o critério de cobertura dos testes, assim como automatizar a geração dos casos e *scripts* de teste [62]. Todos as abordagens aqui descritas utilizam UML como modelo de entrada. Para a execução de *scripts*, foram definidas as ferramentas HP Quick Test Professional [63] e Apache JMeter [64], para testes funcionais e de desempenho, respectivamente. Para a execução de *scripts* de testes de segurança não foi identificada nenhuma ferramenta, sendo então gerados somente os casos de teste.

Após a realização da engenharia de domínio, ou seja, o estudo das necessidades comuns entre os produtos da linha, foi desenvolvido o modelo de *features* da linha de produtos, apresentado na Figura 19. Nas seções 5.1.1, 5.1.2 e 5.1.3 serão apresentados os produtos da linha individualmente, definindo o porquê da existência das *features* apresentadas neste modelo de referência.

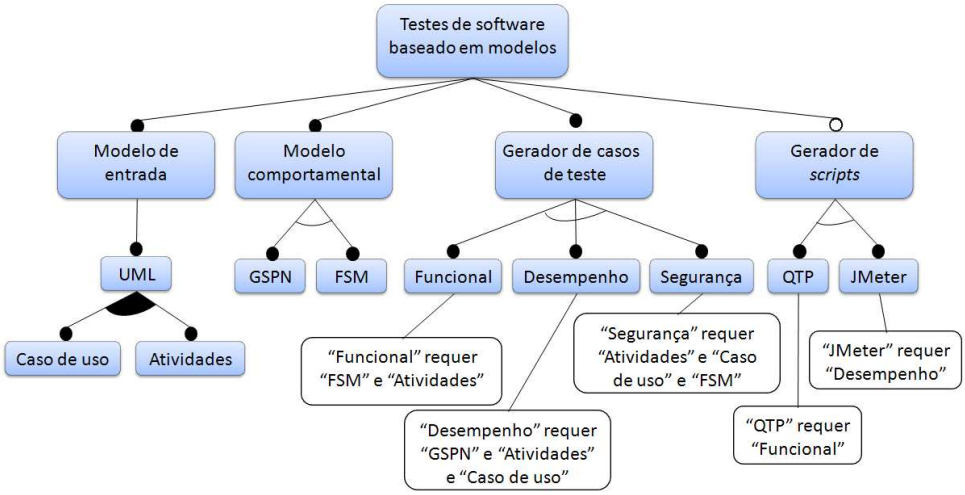

Figura 19 – Modelo de *features* da linha de produtos proposta.

Baseado nas exigências dos produtos existentes nesta linha, exigências estas apresentadas nas Seções 5.1.1, 5.1.2 e 5.1.3, e na arquitetura base exibida na Figura 18, foi definida uma arquitetura de referência, exibida na Figura 20.

Esta arquitetura de referência é composta pela categoria responsável pelos manipuladores do modelos de entrada (categoria 1 na Figura 20), sendo os modelos de entrada descritos em UML, podendo ser um modelo de caso de uso, um modelo de atividades ou ambos, pela categoria de simulação do modelo comportamental (categoria 2 da Figura 20), podendo ser uma Máquina de Estados Finitos com entradas e saídas (representado na Figura 20 como *FSM*) ou uma Rede de Petri estocástica generalizada (representado na Figura 20 como *GSPN*), pela categoria de geração de casos de teste (categoria 4), pela categoria de geração de *scripts* de teste (catego-

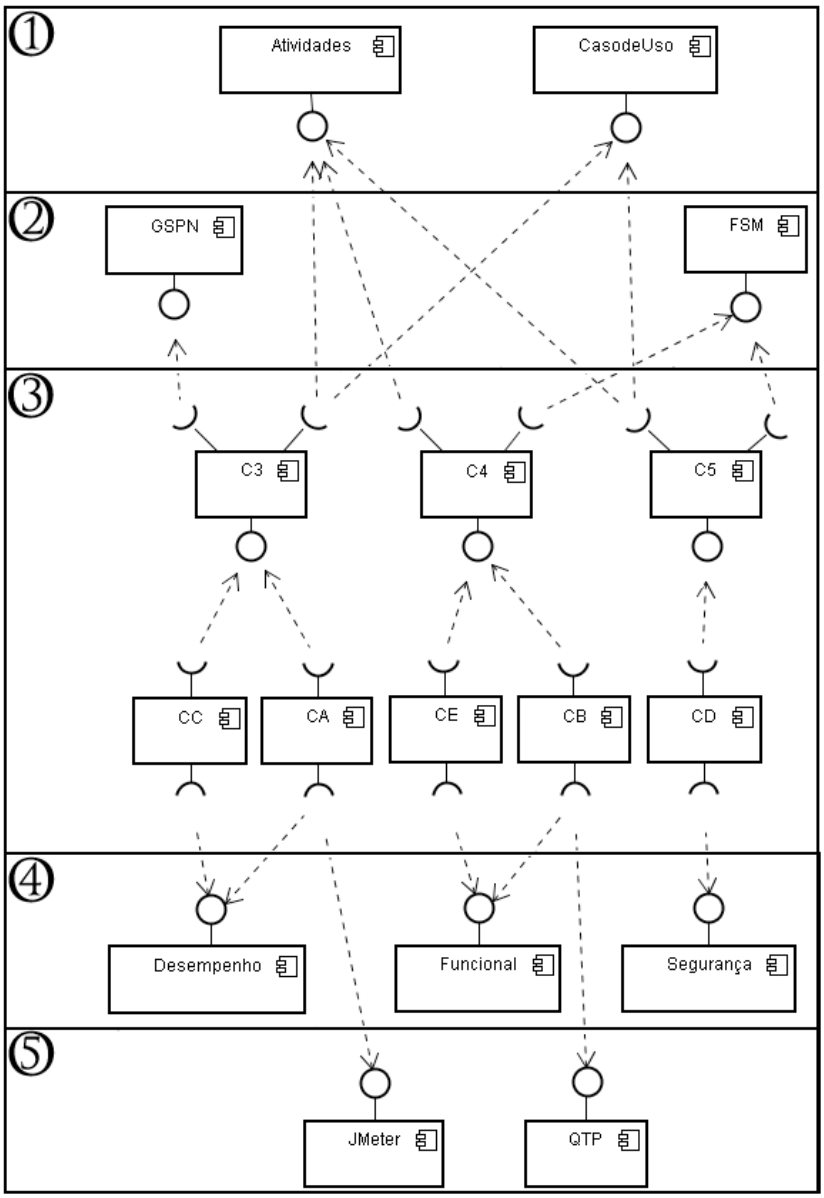

Figura 20 – Arquitetura de referência para a linha de produtos proposta.

ria 5) e pela categoria de conectores responsáveis pela interligação entre as demais categorias (categoria 3). Um produto da linha pode possuir um gerador de casos de teste e um gerador de *scripts* de teste ou somente um gerador de casos de teste. Cada componente arquitetural é selecionado a partir das restrições e exigências impostas pelo modelo de *features* da Figura 19.

A Figura 21 exibe o diagrama de classes da linha de produtos implementada neste estudo de caso.

As principais classes do diagrama da Figura 21 são as classes FSM e GSPN, onde é fornecida a base para a elaboração dos casos de teste. A Classe FSM provê funcionalidades para a criação e manipulação de uma Máquina de Estados Finita com Entradas e Saídas, tais como adicionar um novo estado, uma nova transição entre estados, inserir em um determinado estado algum conteúdo relevante para a geração dos testes, entre outras. A Figura 22 exemplifica o

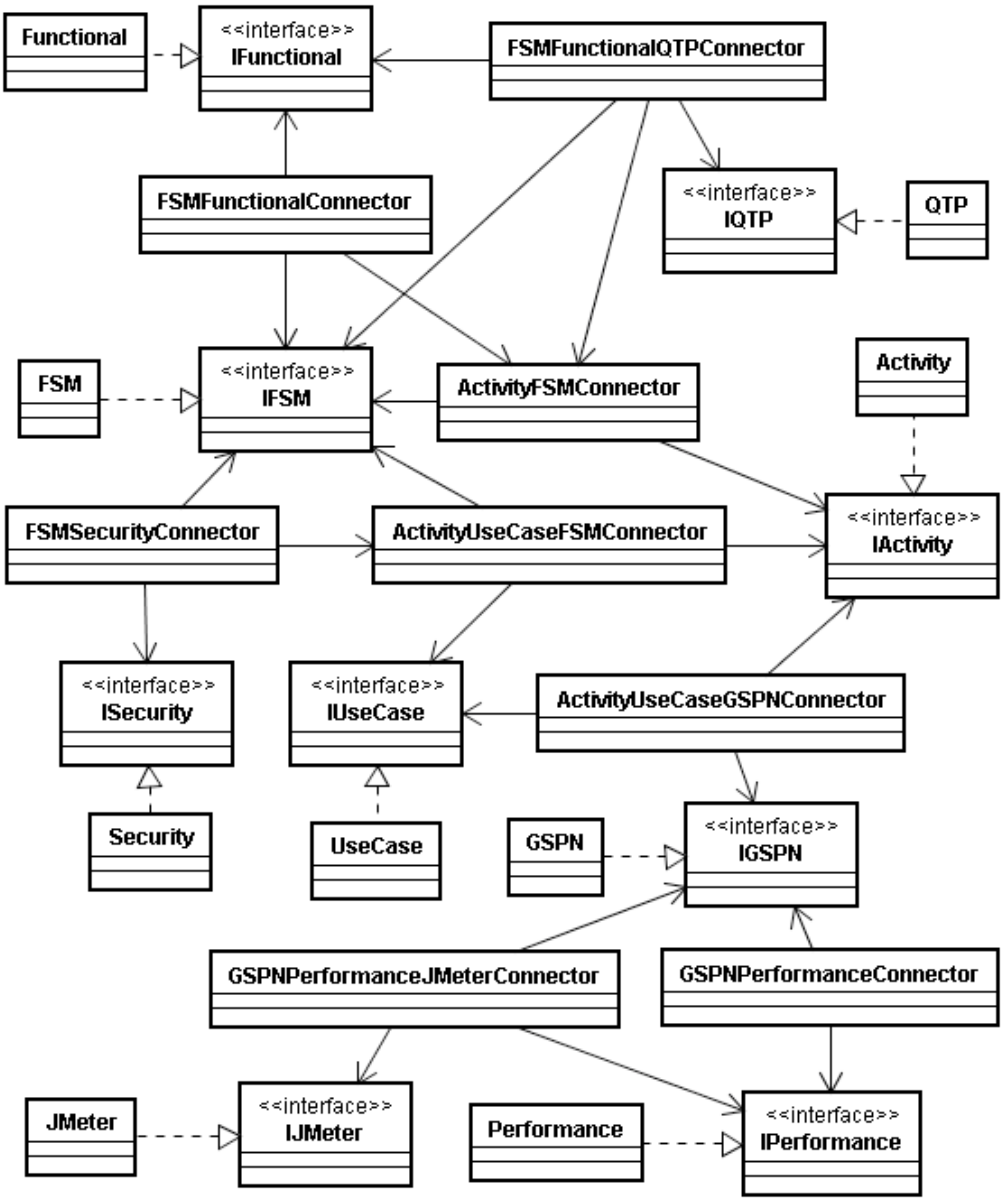

Figura 21 – Diagrama de classes da linha de produtos proposta.

código descrito na linguagem Java [65] para adicionar um novo estado na máquina de estados finita. Como parâmetros são fornecidos um nome de identificação do estado e o conteúdo relevante ao teste a ser anexado a este estado. Este conteúdo está indicado como do tipo *T*, representando uma classe parametrizada. Com base nos parâmetros fornecido, é verificado se já não existe nenhum estado com o nome de identificação indicado, em seguida adiciona o novo estado juntamente com conteúdo anexado e adiciona o novo estado nas matrizes de transições, entradas e saídas para uma indicação posterior de relacionamento entre estados. O código completo desta classe, assim como de sua interface, encontra-se no Apêndice A.2. Esta classe é utilizada por todos os produtos que necessitem de uma Máquina de Estados Finita para simular o comportamento do seu modelo de entrada, ou seja, produtos que exijam um determinismo na geração dos seus casos de teste.

```
/**
 * Cria um novo estado na FSM.
 * @param name nome do estado.
 * @param data informação a ser anexada ao estado.
 */
public void setNewState(String name, T data) {
  if (!this.hasMoreStatesAssociated(name)) {
    this.data.put(name, data);
    this.transitions.addRow();
    this.transitions.setValue(name);
    this.in.addRow();
    this.in.setValue(name);
    this.out.addRow();
    this.out.setValue(name);
  }
}
```

Figura 22 – Trecho de código responsável por criar um novo estado na FSM.

Já a classe GSPN é encarregada de prover funcionalidades para criar e manipular uma Rede de Petri Estocástica Generalizada, como criar um novo estado, uma nova transição entre estados, o conteúdo a ser anexado a um estado, para deslocar um *token* entre estados, entre outras. A Figura 23 exemplifica a classe responsável por criar uma nova transição entre estados. São fornecidos como parâmetros o estado origem, o estado destino e a probabilidade de um *token* atingir o estado destino, a partir do estado origem. Com base nesses parâmetros, o estado origem é localizado nas matrizes de transições e de probabilidades. Em seguida, a linha indicada pelo estado origem é percorrida, em ambas as matrizes, com o objetivo de localizar uma célula vazia para referenciar o estado destino e a probabilidade de acesso. Com isso, cada estado origem possui um vetor de estados destino e um outro vetor de probabilidades de acesso a estes estados destino. O código completo da classe responsável pela Rede de Petri Estocástica Generalizada, assim como o código de sua interface, encontra-se no Apêndice A.1. Esta classe é utilizada por todos os produtos que necessitem de uma Rede de Petri Estocástica Generalizada para simular o comportamento do seu modelo de entrada, ou seja, produtos que exijam uma forma probabilística de geração dos seus casos de teste.

```
/**
 * Cria uma nova transição
 * @param source
 *          estado origem
 * @param target
 *          estado destino
```

```
   * @param targetProbability
   *          probabilidade de acessar o estado
   *          destino.
   */
  public void setNewTransition(String source,
      String target, String targetProbability) {
    this.transitions.setPositionFromValueAndCol(
        source, 1);
    this.probabilities
        .setPositionFromValueAndCol(source, 1);
    for (int i = 2; i <= this.transitions
        .getCols(); i++) {
      this.transitions.setCol(i);
      this.probabilities.setCol(i);
      if (this.transitions.getValue() == null) {
        this.transitions.setValue(target);
        this.probabilities
            .setValue(targetProbability);
        return;
      }
    }
    this.transitions.addCol();
    this.probabilities.addCol();
    this.transitions.setValue(target);
    this.probabilities
        .setValue(targetProbability);
    return;
  }
```

Figura 23 – Trecho de código responsável por criar uma nova transição na Rede de Petri Estocástica Generalizada.

5.1.1 Teste Funcional Baseado em Modelos UML

A ferramenta de teste funcional baseado em modelos UML foi desenvolvida baseada na abordagem descrita em [62], onde o objetivo é gerar casos de teste e *scripts* de teste através das informações existentes no modelo de atividades do sistema em teste. Somente as informações existentes no modelo original do sistema em teste não é suficiente para a derivação de casos de teste funcionais, sendo necessário complementar o diagrama com as informações relevantes. Estas informações são incluídas no modelo através de estereótipos e marcações. Cada atividade do modelo de atividades recebe um estereótipo que identifica o teste funcional (identificado por *<FTStep>*) e recebe marcações que indicam qual a ação do usuário no sistema a ser testado (identificada por *FTAction*) e qual o resultado esperado desta ação (identificado por *FTExpec-*

tedResult). A partir destas informações extras, cada atividade do diagrama possui uma ação de entrada (*FTAction*) e uma informação de saída (*FTExpectedResult*), sendo necessário agora um método para derivar casos de teste a partir deste modelo de entrada.

O método *UIO* (*Unique Input/Output*) é um método para a geração de um conjunto de seqüências de entrada para testar uma Máquina de Estados Finitos com entradas e saídas [66]. Para a geração destas seqüências, este método utiliza-se de seqüências *UIO*. As seqüências *UIO* são utilizadas para verificar se a Máquina de Estados Finitos está em um determinado estado em particular. Sendo assim, cada estado da Máquina de Estados Finitos poderá possuir uma seqüência *UIO* distinta.

Desta forma, como definido em [66], "Para cada transição de um estado s_i para s_j, $f_s(s_i, \chi)$, com algum χ, é definida uma seqüência que conduz do estado inicial a s_i, aplica-se o símbolo de entrada e, em seguida, aplica-se a seqüência *UIO* do estado que deveria ser atingido. Sendo assim, cada seqüência é da forma $P(s_i) \cdot \chi \cdot UIO(s_j)$, sendo que $P(s_i)$ é uma seqüência que leva a Máquina de Estados Finitos do estado inicial ao estado s_i e $UIO(s_j)$ é uma seqüência para s_j".

Para aplicar este algoritmo de geração de casos de teste, o modelo de atividades é convertido em uma Máquina de Estados Finitos com entradas e saídas, onde os estados são as atividades, o alfabeto de entrada é formado pelos conteúdos da marcação *FTAction*, o alfabeto de saída é formado pelos conteúdos da marcação *FTExpectedResult* e o estado inicial e final são os mesmos do modelo de atividades. Esta Máquina de Estados Finitos segue o modelo de Mealy, ou seja, as entradas e saídas são vinculadas às transições e não aos estados.

Para a geração de *scripts* de teste, o mesmo algoritmo é empregado, sendo estes *scripts* gerados para serem executados na *engine* de teste HP Quick Test Professional, sendo assim necessária uma formatação dos *scripts* de acordo com o padrão da *engine*.

Baseado nestas informações, foi realizada a engenharia de aplicação, ou seja, o estudo das generalizações e especializações do produto, resultando no modelo de *features* exibido na Figura 24.

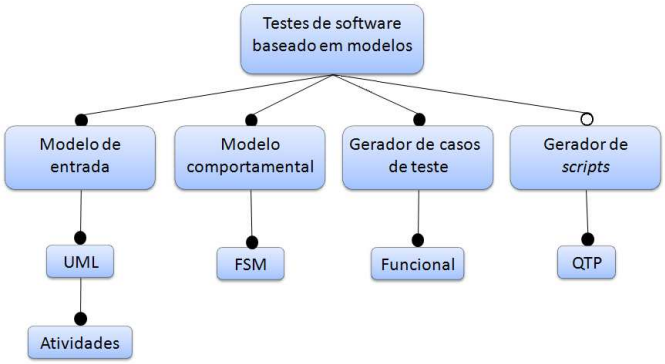

Figura 24 – Modelo de *features* do produto de teste funcional.

Com base neste modelo de *features*, a arquitetura resultante da engenharia de aplicação é apresentada na Figura 25.

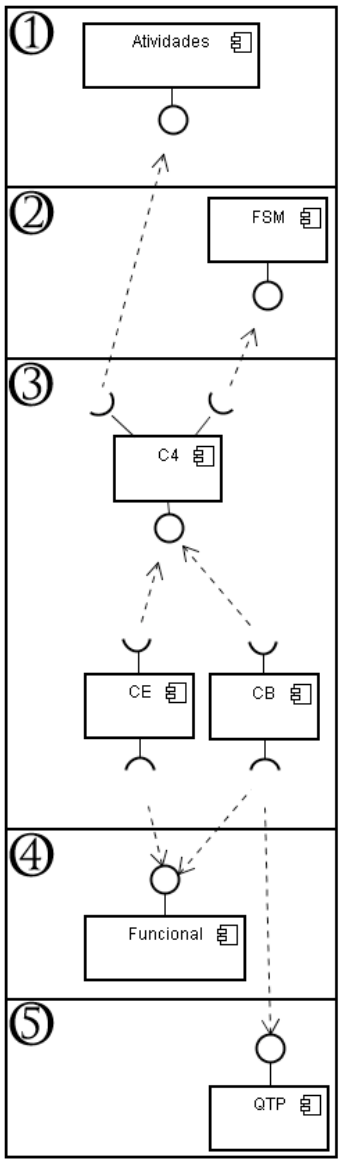

Figura 25 – Arquitetura do produto de teste funcional.

Esta arquitetura, fruto da arquitetura de referência e dos modelos de *features* da linha de produtos e da aplicação, provê dois subprodutos, uma ferramenta de derivação de casos de testes funcionais baseados em modelos UML ou uma ferramenta de derivação de casos de testes e *scripts* de testes funcionais baseados em modelos UML, dependendo da seleção da *feature* opcional de geração de *scripts*, conforme o modelo de *features* da Figura 24. Desta forma, o componente conector *CB* da categoria 3 é responsável pela *feature* opcional, pois a presença deste conector provê o acesso ao gerador de casos de teste (representado na Figura 25 pelo componente denominado *Funcional*) e ao gerador de *scripts* de teste para o Quick Test Professional (representado na Figura 25 pelo componente denominado *QTP*). Caso a *feature* opcional não seja selecionada para compor o produto, o componente conector *CE* fica responsável pela conexão com a categoria de geração de casos de teste.

Maiores informações sobre o funcionamento da abordagem de testes funcionais baseados

em modelos UML utilizada como exemplo neste trabalho podem ser obtidas em [62].

5.1.2 Teste de Segurança Baseado em Modelos UML

A ferramenta de testes de segurança baseados em modelos UML foi baseada nos trabalhos apresentados em [61] e [67], onde o objetivo é derivar casos de teste de segurança baseado nas especificações de segurança inseridos nos modelos de caso de uso de atividades do sistema sob teste, sendo este sistema composto por páginas *web*. Estas especificações são definidas nos modelos através de estereótipos que identificam diferentes estratégias de ataques à segurança do sistema. Cada estereótipo é composto de marcações que apresentam as informações necessárias para caracterizar cada tipo de ataque, tais como o endereço *web* a ser acessado, o limite de usuários simultâneos que o sistema deve suportar, o tamanho máximo de caracteres que um determinado campo de preenchimento de informações do sistema deve suportar, etc.

Como todos os diferentes tipos de ataques especificados devem ser traduzidos em casos de teste, o método de geração de casos de testes utilizado nesta abordagem é o mesmo da ferramenta de teste funcional, ou seja, o método *UIO* [66]. Como este método é aplicado em máquinas de Estados Finitos com entradas e saídas, é necessário extrair as informações pertinentes ao teste de segurança dos modelos de atividades e caso de uso, gerando uma Máquina de Estados Finitos (*FSM*) com entradas e saídas equivalente. O comportamento desta *FSM* é então simulado e nesta simulação é aplicado o método *UIO*. Através das entradas aplicadas nesta *FSM* e nas respectivas saídas são gerados os casos de teste. Diferente da ferramenta apresentada anteriormente, esta ferramenta não possui uma *engine* de teste, não sendo possível, até este momento, a geração de *scripts* de teste.

Baseado na análise dos requisitos desta abordagem, foi especificado o modelo de *features* apresentado na Figura 26.

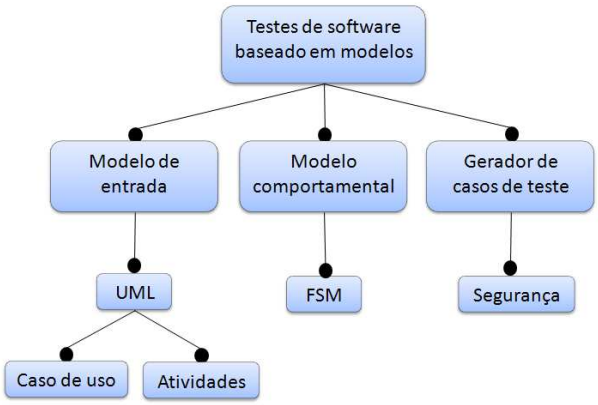

Figura 26 – Modelo de *features* do produto de teste de segurança.

Através deste modelo de *features* e dos requisitos da abordagem foi definida a arquitetura para a ferramenta, sendo exibida na Figura 27.

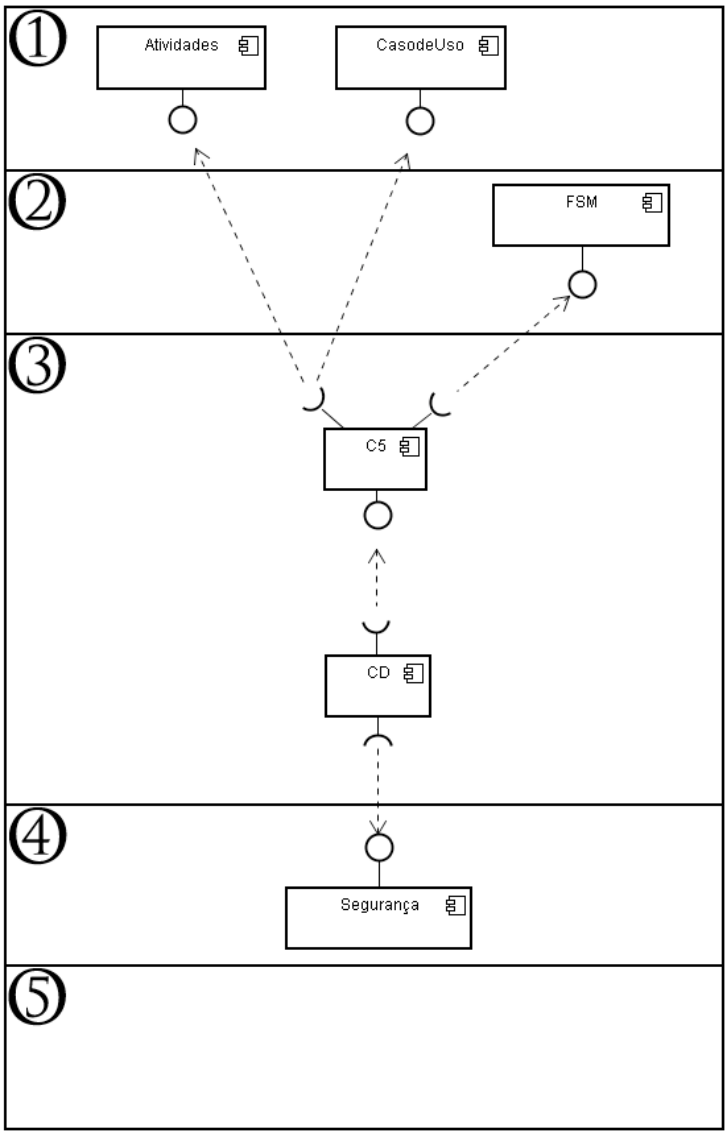

Figura 27 – Arquitetura do produto de teste de segurança.

Como esta arquitetura é parte da arquitetura de referência, ela é dividida em categorias: a categoria do modelo de entrada (1) composta pelos manipuladores dos modelos de caso de uso e atividades, a categoria da simulação comportamental do modelo de entrada (2), sendo neste caso uma Máquina de Estados Finitos com entradas e saídas (representada na Figura 27 como *FSM*), geradores de casos de teste de segurança (4), responsáveis por aplicar o método UIO e organizar os casos de teste de forma legível e inteligível e os conectores responsáveis por unir estas categorias (3). Como não foram encontradas *engines* de teste de segurança, não existem componentes responsáveis por gerar *scripts* de teste (categoria 5).

Maiores informações sobre o funcionamento da abordagem de testes de segurança baseados em modelos UML utilizada como exemplo neste trabalho podem ser obtidas em [67] e [61].

5.1.3 Teste de Desempenho Baseado em Modelos UML

A ferramenta de testes de desempenho baseados em modelos UML foi baseado nos trabalhos apresentados em [32], [68] e [60], onde o objetivo é derivar casos de teste de desempenho baseado nas especificações de desempenho apresentadas nos modelos de atividades e caso de uso do sistema sob teste. Esta abordagem necessita de um comportamento estocástico e temporizado, com o objetivo de simular o comportamento de um usuário atuando no sistema. Para prover este comportamento, a abordagem utiliza uma Rede de Petri Estocástica Generalizada (*GSPN*), criada a partir das informações extraídas dos modelos UML exigidos. Após a construção desta *GSPN*, é realizada a simulação deste modelo comportamental, onde os caminhos a serem percorridos pelas fichas na *GSPN* são definidos de acordo com as probabilidades definidas nos modelos UML para o acesso a cada transição. Durante o percurso destas fichas pela *GSPN*, são definidos de forma aleatória os tempos que uma determinada ficha espera para passar para a posição seguinte, simulando o comportamento de um usuário acessando o sistema, ou seja, tempos relativos ao preenchimento de um formulário, clicar em um botão, etc.

Através das informações extraídas dos modelos UML e providas pela simulação da GSPN, são derivados os casos de teste, assim como os scripts de teste, sendo estes scripts organizados no formato exigido pela *engine* de teste de desempenho Apache JMeter.

De acordo com os requisitos impostos pela abordagem, foi elaborado um modelo de *features*, modelo este exibido na Figura 28.

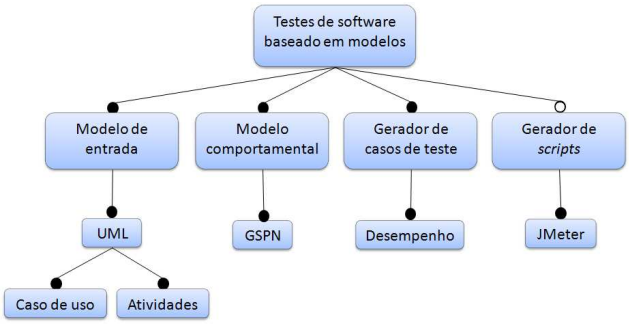

Figura 28 – Modelo de *features* do produto de teste de desempenho.

Um detalhe a ser considerado neste modelo de *features* é a *feature* opcional gerador de *scripts*, fazendo com que a arquitetura proveja as duas opções, possuir um gerador de casos de teste e um gerador de *scripts*, ou somente o gerador de casos de teste.

Através deste modelo de *features* e dos requisitos da abordagem foi definida a arquitetura para a ferramenta, sendo exibida na Figura 29.

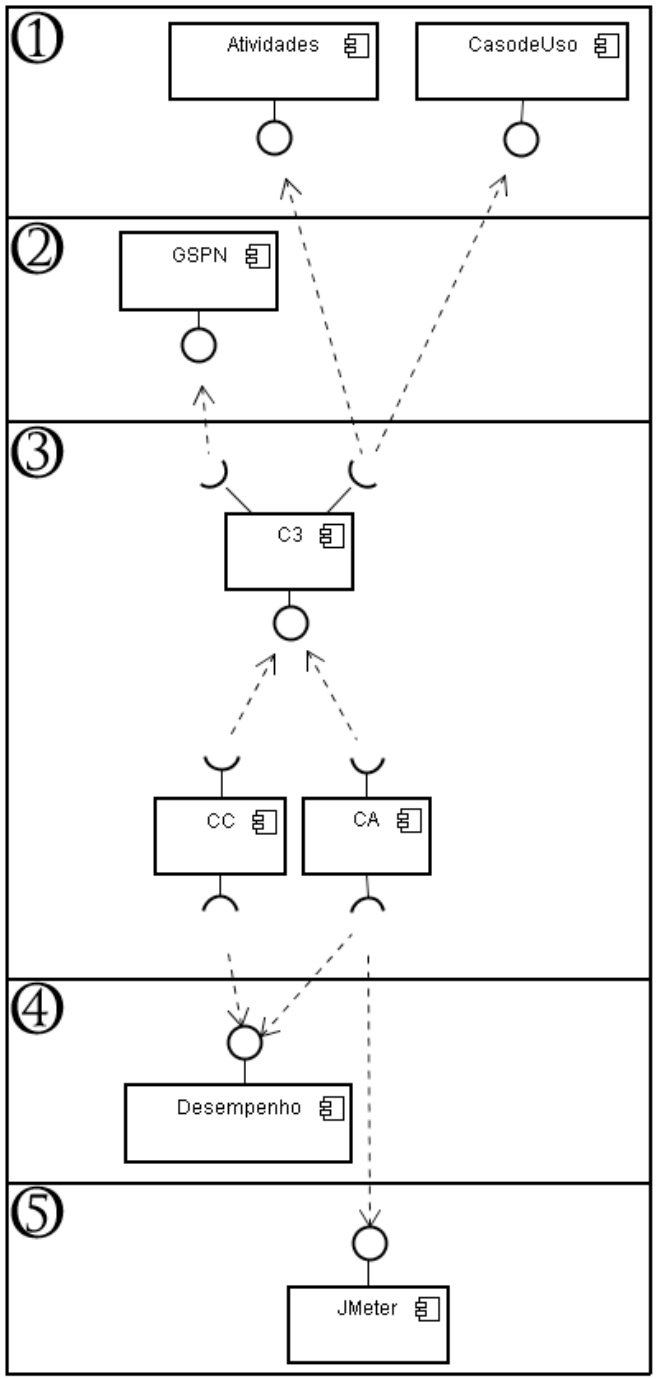

Figura 29 – Arquitetura do produto de teste de desempenho.

Como esta arquitetura faz parte da arquitetura de referência, ela é dividida em categorias: a categoria do modelo de entrada (1) composta pelos manipuladores dos modelos de caso de uso e atividades, a categoria da simulação comportamental do modelo de entrada (2), sendo neste caso uma Rede de Petri Estocástica Generalizada (representada na Figura 29 como *GSPN*), gerador de casos de teste de desempenho (4), responsável por extrair as informações providas pela categoria 2 e organizar os casos de teste de forma legível e inteligível (representado na Figura 29 como *Desempenho*), gerador de *scripts* de teste (5), responsável por extrair as informações

providas pela categoria 2 e organizar os *scripts* no formato exigido pela ferramenta Apache JMeter (representado na Figura 29 como *JMeter*) e a categoria de conectores (3), responsável por interligar as demais categorias. O conector descrito como *CA* na Figura 29 é o conector responsável pela *feature* opcional exibida na Figura 28, sendo encarregado de prover os geradores de casos de teste e de *scripts*.

Maiores informações sobre o funcionamento da abordagem de testes de desempenho baseados em modelos UML utilizada como exemplo neste trabalho podem ser obtidas em [32] e [60].

5.1.4 Ferramenta desenvolvida

Para concretizar a arquitetura proposta, foi desenvolvida uma ferramenta de geração automatizada da linha de produtos. Esta ferramenta baseia-se no modelo de *features* dos produtos específicos e no modelo de *features* da linha de produtos como um todo. Cada *feature* é mapeado para a implementação de um componente, conforme apresentado no início deste capítulo. A Figura 30 ilustra esta ferramenta desenvolvida.

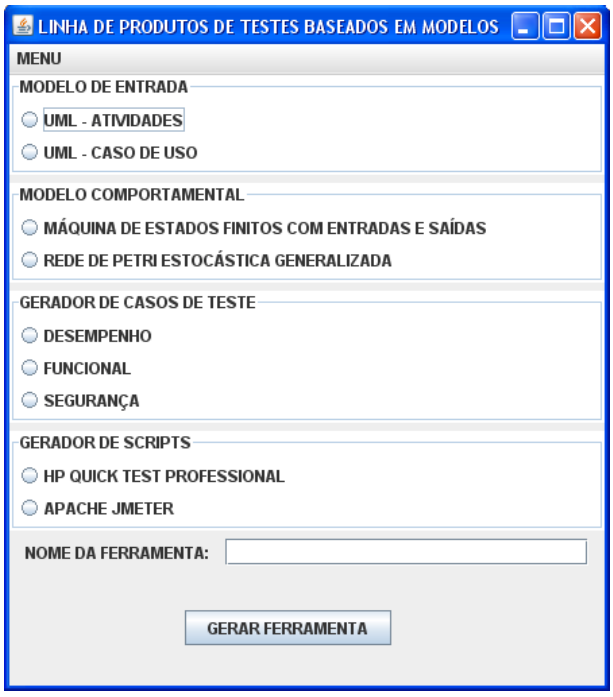

Figura 30 – Ferramenta desenvolvida.

Como exibe a Figura 30, as *features* são divididas em quatro categorias: modelo de entrada, modelo comportamental, gerador de casos de teste e gerador de *scripts* de teste, equivalendo respectivamente às categorias 1, 2, 4 e 5 da arquitetura de referência exibida na Figura 18. A categoria 3 da arquitetura de referência é representada na ferramenta através de relacionamentos

entre as *features*, onde cada relacionamento entre as *features* das categorias 1 e 2 é concretizado através de um conector, assim como o relacionamento entre as categorias 2, 4 e 5. A Figura 31 exemplifica o relacionamento entre as *features* da Figura 19 Atividades, Caso de uso, e *FSM* através do componente descrito como *ActivityUseCaseFSMConnector* e entre as *features FSM* e Segurança através do componente descrito como *FSMSecurityConnector*.

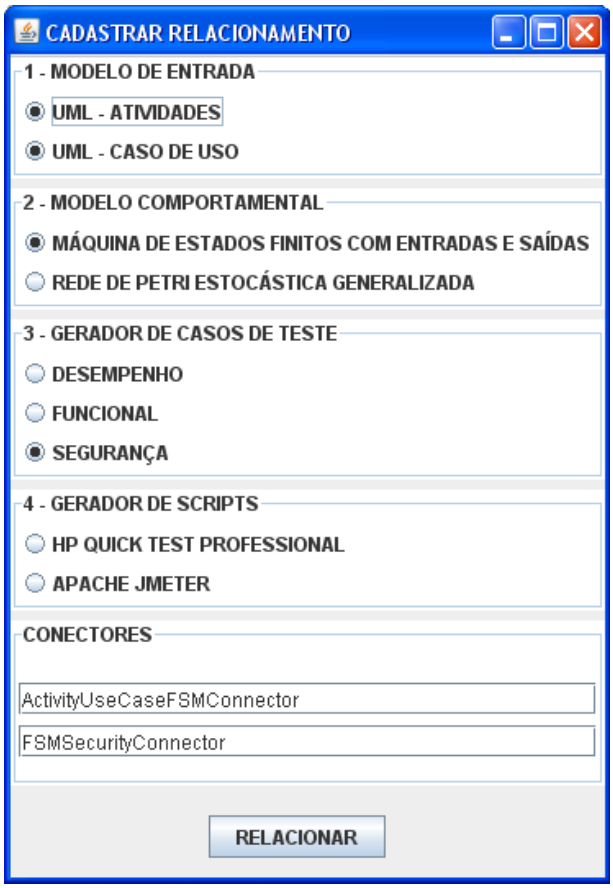

Figura 31 – Relacionamento entre *features*.

A partir das diferentes relações entre as *features*, a ferramenta possibilita ou não a seleção de uma determinada *feature* para compor um determinado produto. Passada esta etapa de seleção, a ferramenta exige a inserção de uma denominação do produto a ser criado, proporcionando assim a geração do produto. Esta geração é proporcionada através de *templates*, onde são inseridos o nome do produto e os componentes que o constituem. Após é executada a ferramenta Apache Ant [69], responsável por gerar um arquivo executável deste produto. A ferramenta Apache Ant exige um arquivo de configuração onde são especificadas as bibliotecas necessárias para compor o produto, qual a classe principal do produto, onde o produto deve ser salvo, entre outros requisitos de configuração. Este arquivo fica localizado na pasta do projeto, sendo necessário que este arquivo possua o nome de build.xml. A Figura 32 ilustra o arquivo build.xml configurado para compor este estudo de caso.

```
<project name="ModelBasedTesting" basedir="." default="executar">

    <property name="classes" location="classes"/>

    <target name="dir">
        <mkdir dir="classes" />
    </target>

    <target name="limpar">
        <delete dir="classes"/>
    </target>

    <target name="compilar" depends="dir">
        <javac srcdir="${basedir}/lib/src"
        classpath="${classes}"
        destdir="${classes}" />
    </target>

    <target name="empacotar" depends="compilar">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/ModelBasedTesting.jar"
            basedir="${basedir}/classes">
            <manifest>
                <attribute name="Main-Class" value="Main.Main"/>
            </manifest>
        </jar>
    </target>

    <target name="executar" depends="empacotar, limpar">
        <java jar="build/jar/ModelBasedTesting.jar" fork="true"/>
    </target>

</project>
```

Figura 32 – Arquivo build.xml utilizado para configurar a ferramenta Apache Ant no estudo de caso realizado.

Este arquivo build.xml define características como o nome do projeto (indicado através da marcação *<project name = ModelBasedTesting*), a criação de uma pasta temporária onde o projeto será organizado para a compilação (indicada por *<mkdir dir="classes"/>*), a instrução que o compilador *java* deve executar para compilar o projeto (indicado pela marcação *<javac srcdir="$basedir/lib/src"classpath="$classes"destdir="$classes"/>*), qual o arquivo principal do produto a ser gerado (indicado pela marcação *<attribute name="Main-Class"value="Main.Main"/>*), entre outras características.

A Figura 33 ilusta um exemplo de produto de teste funcional baseado em modelos UML

gerado pela ferramenta.

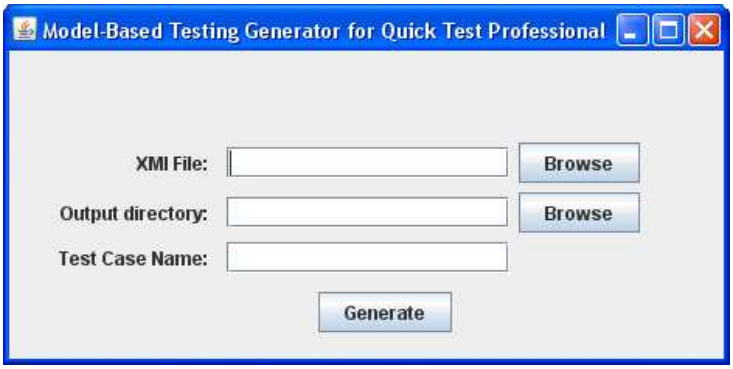

Figura 33 – Exemplo de produto gerado.

5.2 Considerações

Como pode ser analisado nos exemplos de produtos providos pelo estudo de caso, a arquitetura de referência exibida na Figura 20 (ou exibida de forma genérica na Figura 18) busca prover o reuso dos componentes arquiteturais, facilitando assim o desenvolvimento de novas técnicas de testes baseados em modelos. De acordo com os produtos individuais apresentados, todos compartilham o uso do componente da categoria 1 denominado *Atividades* e dois dos produtos compartilham o uso do componente denominado *CasodeUso*. Dois dos produtos compartilham também o componente da categoria 2 denominado *FSM*, sendo os demais componentes das categorias 2, 4 e 5 utilizados de forma individual, devido às exigências de cada tipo de teste. Caso haja o interesse de inserir outra abordagem do mesmo tipo de teste já existente na linha de produtos, basta apenas o desenvolvimento dos conectores próprios para esta nova abordagem, inserindo nestes conectores as exigências impostas pela abordagem, como por exemplo quais informações devem ser extraídas dos modelos de entrada. Caso o interesse seja em que a linha de produtos proveja outro tipo de teste, deverão ser implementados, além dos conectores, os devidos geradores de casos e *scripts* de teste, inserindo estes componentes na arquitetura de referência. Desta forma, estes geradores poderão ser reusados para o desenvolvimento de outras abordagens do mesmo tipo de teste.

Este estudo de caso foi ao encontro ao apresentado em [39], onde uma linha de produtos começa a apresentar lucros normalmente a partir de uma linha com três produtos. Inicialmente, a abordagem de linha de produtos exige um grande esforço, visto que cada produto deve ser projetado tendo em vista a generalisação e o reuso, pois a elaboração visa que outros produtos utilizem os elementos desenvolvidos para este primeiro produto. A experiência deste estudo de caso mostrou que a medida que outros produtos vão sendo inseridos na linha, estes produtos vão reusando os elementos já desenvolvidos anteriormente, tornando cada vez mais simples a inserção destes novos produtos na linha.

6 Considerações Finais

A partir de uma visão, que se torna cada vez mais complexa, do desenvolvimento de software focado na qualidade, diversos pesquisadores trabalham em técnicas para facilitar o processo de teste de software. Uma das abordagens propostas para a simplificação do processo de testes é através do uso dos modelos do sistema sob teste, para proporcionar informações sobre como este teste deve ser realizado, provendo assim uma oportunidade de automatização da geração de casos de teste. Esta abordagem é conhecida como testes baseados em modelos.

Devido a grande quantidade de visões sobre o que deve ser testado no sistema sob teste, diversas abordagens de testes baseados em modelos surgem diariamente. Embora as abordagens apresentem características diferentes, alguns requisitos básicos são comuns nos diferentes tipos de teste de software.

Neste trabalho foi proposto uma aglutinação dos requisitos comuns das abordagens de teste baseados em modelos através de uma arquitetura de linha de produtos de software, proporcionando assim o reuso dos elementos arquiteturais comuns destas abordagens, facilitando a implementação de ferramentas, diminuindo o tempo de desenvolvimento e aumentando a qualidade destas ferramentas, sendo estas vantagens providas pelos conceitos de linha de produtos de software.

Para alcançar uma maior compreensão do domínio do problema, foi realizado um estudo de caso, sendo este estudo um projeto do Centro de Pesquisa em Computação Aplicada da Pontifícia Universidade Católica do Rio Grande do Sul em cooperação com a empresa Hewlett-Packard. Este estudo de caso foi realizado através de uma linha de produtos composta por três abordagens de teste de software baseados em modelos UML. A partir da análise individual destas abordagens, junto com o estudo das demais abordagens de testes baseados em modelos existentes na literatura, foi desenvolvida a arquitetura de linha de produtos de testes de software baseados em modelos UML. Através desta arquitetura de referência, cada ferramenta de teste foi desenvolvida, focando sempre no reuso de software.

Como trabalhos futuros, pretende-se inserir outras abordagens de teste, como por exemplo testes em arquiteturas orientadas a serviços. Pretende-se também tornar esta arquitetura dinâmica, através da descrição da arquitetura em uma linguagem de descrição de arquitetura (ADL - *Architecture Description Language*) assim como utilizar os benefícios da programação orientada a aspectos para classificar os diferentes produtos da linha em nível de código, podendo assim expandir a quantidade de produtos da linha de forma clara e simples.

Referências

[1] MCGREGOR, J. *Testing a Software Product Line*. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, Carnegie Mellon University. 2001.

[2] BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: *Future of Software Engineering, collocated with 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society Press, 2007. p. 85–103.

[3] BEIZER, B. *Software testing techniques*. 2nd. ed. New York, NY, USA: Van Nostrand Reinhold Co., 1990. 580 p.

[4] SHEPARD, T.; LAMB, M.; KELLY, D. More testing should be taught. *Communications of the ACM*, ACM, New York, NY, USA, v. 44, n. 6, p. 103–108, 2001.

[5] FEWSTER, M.; GRAHAM, D. *Software test automation: effective use of test execution tools*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999. 596 p.

[6] BOURQUE, P.; DUPUIS, R. *Guide to the Software Engineering Body of Knowledge*. Los Alamitos, CA, USA: IEEE Computer Society, 2004. 202 p.

[7] LYU, M. *Software Fault Tolerance*. New York, NY, USA: John Wiley & Sons, Inc., 1995. 354 p.

[8] LYU, M. ; HE, Y. Improving the n-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability*, IEEE Computer Society Press, New York, NY, USA, v. 42, n. 2, p. 179–189, 1993.

[9] PULLUM, L. *Software fault tolerance techniques and implementation*. Norwood, MA, USA: Artech House, Inc., 2001. 360 p.

[10] MYERS, G. *The Art of Software Testing*. 2nd. ed. Hoboken, NJ, USA: John Wiley & Sons, 2004. 260 p.

[11] PATTON, R. *Software Testing*. 2nd. ed. Indianapolis, IN, USA: Sams Publishing, 2005. 408 p.

[12] JORGENSEN, P.; ERICKSON, C. Object-oriented integration testing. *Communications of the ACM*, ACM, New York, NY, USA, v. 37, n. 9, p. 30–38, 1994.

[13] BARASH, Y. et al. Modeling dependencies in protein-dna binding sites. In: *International Conference on Research in Computational Molecular biology*. New York, NY, USA: ACM, 2003. p. 28–37.

[14] REED, J.; FOLLEN G.; AFJEH, A. Improving the aircraft design process using web-based modeling and simulation. *ACM Transactions on Modeling and Computer Simulation*, ACM, New York, NY, USA, v. 10, n. 1, p. 58–83, 2000.

[15] APFELBAUM, L.; DOYLE, J. Model-based testing. In: *10th International Software Quality Week*. San Francisco, CA, USA: Software Research Institute, 1997. p. 13–27.

[16] EL-FAR, I.; WHITTAKER, J. Model-based software testing. In: *Encyclopedia of Software Engineering*. San Francisco, CA, USA: Wiley InterScience, 2002. v. 1, p. 825–837.

[17] BAKER, P. et al. *Model-Driven Testing - Using the UML Testing Profile*. New York, NY, USA: Springer Berlin Heidelberg, 2007. 184 p.

[18] FRIEDMAN, G. et al. Projected state machine coverage for software testing. In: *international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2002. p. 134–143.

[19] OBJECT MANAGEMENT GROUP. Unified Modeling Language. Disponível em <http://www.uml.org/>. Acesso em: 17 nov. 2008.

[20] BRIAND, L.; LABICHE, Y. A uml-based approach to system testing. In: *International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. London, UK: Springer-Verlag, 2001. p. 194–208.

[21] HARTMANN, J.; IMOBERDORF, C.; MEISINGER, M. Uml-based integration testing. In: *international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2000. p. 60–70.

[22] TRIVEDI, K. *Probability and statistics with reliability, queuing and computer science applications*. 2nd. ed. Chichester, UK, UK: John Wiley and Sons Ltd., 2002. 830 p.

[23] BOLCH, G. et al. *Queueing Networks and Markov Chains*. 2nd. ed. New York, NY, USA: Wiley-Interscience, 2005. 878 p.

[24] NORRIS, J. *Markov Chains*. Cambridge, UK: Cambridge University Press, 1998. 237 p. (Cambridge Series in Statistical and Probabilistic Mathematics).

[25] PETRI, C. *Kommunikation mit Automaten*. Tese (Doutorado) — Universität Bonn, Germany, 1962.

[26] IORDACHE, M.; ANTSAKLIS, P. *Supervisory Control of Concurrent Systems: A Petri Net Structural Approach (Systems & Control: Foundations & Applications)*. Cambridge, MA, USA: Birkhäuser Basel, 2006. 281 p.

[27] MOLLOY, M. Performance analysis using stochastic petri nets. *IEEE Transactions on Computers*, IEEE Computer Society, Los Alamitos, CA, USA, v. 31, n. 9, p. 913–917, 1982.

[28] MARSAN, M.; CONTE, G.; BALBO, G. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, ACM, New York, NY, USA, v. 2, n. 2, p. 93–122, 1984.

[29] LINDEMANN, C. *Performance modelling with deterministic and stochastic petri nets*. New York, NY, USA: John Wiley & Sons, 1998. 422 p.

[30] UTTING, M.; LEGEARD, B. *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. 456 p.

[31] PRETSCHNER, A. et al. One evaluation of model-based testing and its automation. In: *International conference on Software engineering*. Saint Louis, MO, USA: ACM Press, 2005. p. 392–401.

[32] OLIVEIRA, F. et al. Performance testing from UML models with resource descriptions. In: *Brazilian Workshop on Systematic and Automated Software Testing*. João Pessoa, PB, BR: Brazilian Computer Society, 2007. p. 47–54.

[33] NETO, A. et al. A survey on model-based testing approaches: a systematic review. In: *international workshop on Empirical assessment of software engineering languages and technologies*. New York, NY, USA: ACM Press, 2007. p. 31–36.

[34] UTTING, M.; PRETSCHNER, A.; LEGEARD, B. *A Taxonomy of Model-Based Testing*. Technical Report 04/2006, Department of Computer Science, The University of Waikato, New Zealand. 2006.

[35] NEIGHBORS, J. The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, IEEE Computer Society Press, New York, NY, USA, v. 10, n. 5, p. 564–574, 1984.

[36] KANG, K. et al. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University. 1990.

[37] VAN DER LINDEN, F.; MÜLLER, J. Creating architectures with building blocks. *IEEE Software*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 12, n. 6, p. 51–60, 1995.

[38] POULIN, J. *Measuring software reuse: principles, practices, and economic models*. Boston, MA, USA: Addison-Wesley, 1996. 224 p.

[39] SCHMID, K.; VERLAGE, M. The economic impact of product line adoption and evolution. *IEEE Software*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 19, n. 4, p. 50–57, 2002.

[40] BIRK, A. et al. Product line engineering: The state of the practice. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 20, n. 6, p. 52–60, 2003.

[41] VAN DER LINDEN, F.; SCHMID, K.; ROMMES, E. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Secaucus, NJ, USA: Springer-Verlag, 2007. 334 p.

[42] KANG, K. et al. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, J. C. Baltzer AG, Science Publishers, Red Bank, NJ, USA, v. 5, n. 1, p. 143–168, 1998.

[43] GRISS, M.; FAVARO J.; D' ALESSANDRO, M. Integrating feature modeling with the rseb. In: *International Conference on Software Reuse*. Washington, DC, USA: IEEE Computer Society, 1998. p. 76–85.

[44] BOSCH J. et al. Variability issues in software product lines. In: *International Workshop on Software Product-Family Engineering*. London, UK: Springer-Verlag, 2001. p. 13–21.

[45] CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practices*. 2nd. ed. Boston, MA, USA: Addison-Wesley, 2003. 528 p.

[46] LEAVENS, G.; SITARAMAN, M. *Foundations of component-based systems*. New York, NY, USA: Cambridge University Press, 2000. 312 p.

[47] SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. New York, NY, USA: ACM Press and Addison-Wesley, 1998. 589 p.

[48] CRNKOVIC, I. *Building Reliable Component-Based Software Systems*. Norwood, MA, USA: Artech House, Inc., 2002. 458 p.

[49] HEINEMAN, G.; COUNCILL, W. *Component-Based Software Engineering: Putting the Pieces Together*. New York, NY, USA: Addison-Wesley Professional, 2001. 880 p.

[50] WANG, J. Towards component-based software engineering. In: *Midwestern conference on Small colleges*. Valparaiso, IN, USA: Consortium for Computing Sciences in Colleges, 2000. p. 177–189.

[51] POHL, C. et al. Survey of existing implementation techniques with respect to their support for product lines. AMPLE deliverable D2.1. Disponível em <http://ample.holos.pt/gest_cnt_upload/editor/File/public/Deliverable%20D3.1.pdf>. Acesso em: 17 nov. 2008.

[52] PURE-SYSTEMS. Pure::variants. Disponível em <http://www.pure-systems.com/Variant_Management.49.0.html>. Acesso em: 17 nov. 2008.

[53] BIGLEVER SOFTWARE. Gears. Disponível em <http://www.biglever.com/solution/ product.html>. Acesso em: 17 nov. 2008.

[54] CZARNECKI, K.; EISENECKER, U. *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. 864 p.

[55] IBM. Rational software modeler. Disponível em <http://www-01.ibm.com/software/awdtools/modeler/swmodeler/index.html>. Acesso em: 17 nov. 2008.

[56] IBM. Rational software architect. Disponível em <http://www-01.ibm.com/software/awdtools/architect/swarchitect/>. Acesso em: 17 nov. 2008.

[57] ANTKIEWICZ M.; CZARNECKI, K. Featureplugin: feature modeling plug-in for eclipse. In: *Workshop on Eclipse Technology Exchange, Collocated with 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2004. p. 67–72.

[58] ECLIPSE FOUNDATION. Eclipse. Disponível em <http://www.eclipse.org/>. Acesso em: 17 nov. 2008.

[59] ANTKIEWICZ M.; CZARNECKI, K. Mapping features to models: A template approach based on superimposed variants. In: GLÜCK R.; LOWRY, M. (Ed.). *International Conference on Generative Programming and Component Engineering*. Tallinn, Estonia: Springer, 2005. (Lecture Notes in Computer Science, v. 3676), p. 422–437.

[60] RODRIGUES, E. *Alocação de recursos em ambientes virtualizados*. Dissertação (Mestrado) — Faculdade de Informática. Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, RS, BR, 2008.

[61] PERALTA, K. *Uma Estratégia para Especificação e Geração de Casos de Teste de Segurança usando Modelos UML*. Dissertação (Mestrado) — Faculdade de Informática. Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, RS, BR, 2008.

[62] OROZCO A. et al. Derivação de casos de testes funcionais a partir de modelos UML. In: *Submetido ao Simpósio Brasileiro de Sistemas de Informação*. Brasília, DF, BR: Anais do Simpósio Brasileiro de Sistemas de Informação, 2008.

[63] HEWLETT-PACKARD. Quick test professional. Disponível em <https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24%5E1352_4000_100__>. Acesso em: 17 nov. 2008.

[64] APACHE SOFTWARE FOUNDATION. Apache JMeter. Disponível em <http://jakarta.apache.org/jmeter/index.html>. Acesso em: 17 nov. 2008.

[65] HORSTMANN, C. *Big Java*. 3th. ed. Hoboken, NJ, USA: John Wiley & Sons, 2008. 1204 p.

[66] DELAMARO, M.; MALDONADO, J.; JINO, M. *Introdução ao teste de software*. Rio de Janeiro, RJ, BR: Editora Campus, 2007. 408 p.

[67] PERALTA, K. et al. Specifying Security Aspects in UML Models. In: *Workshop on Modeling Security*. Toulouse, FR: Central Europe workshop proceedings, 2008. v. 413, p. 11–20.

[68] RODRIGUES, E. et al. Uso de Modelos Preditivos e SLAs para Reconfiguração de Ambientes Virtualizados. In: *Workshop de Sistemas Operacionais*. Belém, PA, BR: Anais da Sociedade Brasileira de Computação, 2008. p. 147–158.

[69] APACHE SOFTWARE FOUNDATION. Apache Ant. Disponível em <http://ant.apache.org/>. Acesso em: 17 nov. 2008.

Apêndice A – Código da linha de produtos desenvolvida

A.1 Classes responsáveis por prover Redes de Petri Estocásticas Generalizadas

A.1.1 Interface IGSPN

```
package GSPN;

public interface IGSPN<T> {

  /**
   * Insere um novo estado na GSPN.
   * @param name
   *          O ID do estado.
   * @param data
   *          A informação a ser inserida no
   *          estado.
   */
  public void setNewState(String name, T data);

  /**
   * Cria uma nova transição
   * @param source
   *          estado origem
   * @param target
   *          estado destino
   * @param targetProbability
   *          probabilidade de acessar o estado
   *          destino.
   */
  public void setNewTransition(String source,
      String target, String targetProbability);

  /**
   * Cria uma nova transição com probabilidade de
   * 100% de acesso ao estado destino
   * @param source
   *          estado origem.
   * @param target
   *          estado destino.
   */

  public void setNewTransition(String source,
      String target);

  /**
   * Insere o número de tokens existentes na GSPN.
   * @param totalTokens
```

```
   *          o número de tokens.
   */
  public void setTokens(int totalTokens);

  /**
   * Seta o próximo estado como final e com
   * probabilidade de 100% de acesso.
   */
  public void setFinalState();

  /**
   * Verifica se o estado existe na GSPN.
   * @param stateID
   *          o ID do estado.
   * @return TRUE se o estado existe e FALSE, caso
   *         contrário.
   */
  public boolean hasState(String stateID);

  /**
   * Retorna o número de tokens existentes na
   * GSPN.
   * @return o número de tokens existentes.
   */
  public int getTokens();

  /**
   * Retorna as informações externas existentes no
   * estado atual.
   * @return as informações no formato
   *         parametrizado.
   */
  public T getData();

  /**
   * Verifica se o estado atual é final.
   * @return TRUE caso final e FALSE, caso
   *         contrário.
   */
  public boolean isFinalState();

  /**
   * Seta o token para a posição seguinte,
   * escolhendo uma das possíveis posições baseado
   * nas probabilidades de acesso.
   */
  public void setNext();

}
```

A.1.2 Classe GSPN

```
package GSPN;

import java.util.HashMap;
import java.util.Random;
import Grid.*;
```

```
public class GSPN<T> implements IGSPN<T> {

  private HashMap<String, T> data;
  private int tokens;
  private String initialState;
  private IGrid<String> transitions;
  private IGrid<String> probabilities;
  private String actualState;
  public static final String INITIAL = "INITIALSTATE";
  public static final String FINAL = "FINALSTATE";

  public GSPN() {
    this.transitions = new Grid<String>();
    this.transitions.setCols(2);
    this.probabilities = new Grid<String>();
    this.probabilities.setCols(2);
    this.data = new HashMap<String, T>();
    this.initialState = GSPN.INITIAL;
    this.actualState = GSPN.INITIAL;
    this.data.put(this.initialState, null);
    this.transitions.addRow();
    this.transitions.setValue(GSPN.INITIAL);
    this.probabilities.addRow();
    this.probabilities.setValue(GSPN.INITIAL);
    this.tokens = 1;

  }

  /**
   * Insere um novo estado na GSPN.
   * @param name
   *          O ID do estado.
   * @param data
   *          A informação a ser inserida no
   *          estado.
   */
  public void setNewState(String name, T data) {
    if (!this.hasState(name)) {
      this.data.put(name, data);
      this.transitions.addRow();
      this.transitions.setValue(name);
      this.probabilities.addRow();
      this.probabilities.setValue(name);
    }
  }

  /**
   * Cria uma nova transição
   * @param source
   *          estado origem
   * @param target
   *          estado destino
   * @param targetProbability
   *          probabilidade de acessar o estado
   *          destino.
   */
  public void setNewTransition(String source,
```

```
    String target, String targetProbability) {
  this.transitions.setPositionFromValueAndCol(
      source, 1);
  this.probabilities
      .setPositionFromValueAndCol(source, 1);
  for (int i = 2; i <= this.transitions
      .getCols(); i++) {
    this.transitions.setCol(i);
    this.probabilities.setCol(i);
    if (this.transitions.getValue() == null) {
      this.transitions.setValue(target);
      this.probabilities
          .setValue(targetProbability);
      return;
    }
  }
  this.transitions.addCol();
  this.probabilities.addCol();
  this.transitions.setValue(target);
  this.probabilities
      .setValue(targetProbability);
  return;
}

/**
 * Cria uma nova transição com probabilidade de
 * 100% de acesso ao estado destino
 * @param source
 *          estado origem.
 * @param target
 *          estado destino.
 */

public void setNewTransition(String source,
    String target) {
  this.transitions.setPositionFromValueAndCol(
      source, 1);
  this.probabilities
      .setPositionFromValueAndCol(source, 1);
  for (int i = 2; i <= this.transitions
      .getCols(); i++) {
    this.transitions.setCol(i);
    this.probabilities.setCol(i);
    if (this.transitions.getValue() == null) {
      this.transitions.setValue(target);
      this.probabilities.setValue("1");
      return;
    }
  }
  this.transitions.addCol();
  this.probabilities.addCol();
  this.transitions.setValue(target);
  this.probabilities.setValue("1");
  return;
}

/**
 * Insere o número de tokens existentes na GSPN.
```

```
 * @param totalTokens
 *          o número de tokens.
 */
public void setTokens(int totalTokens) {
  this.tokens = totalTokens;
}

/**
 * Seta o próximo estado como final e com
 * probabilidade de 100% de acesso.
 */
public void setFinalState() {
  this.transitions.setCol(2);
  this.probabilities.setCol(2);
  for (int x = 1; x <= transitions.getRows(); x++) {
    this.transitions.setRow(x);
    this.probabilities.setRow(x);
    if (this.transitions.getValue() == null) {
      this.transitions.setValue(GSPN.FINAL);
      this.probabilities.setValue("1");
    }

  }
  this.transitions.addRow();
  this.transitions.setValue(GSPN.FINAL);
  this.probabilities.addRow();
  this.probabilities.setValue(GSPN.FINAL);
}

/**
 * Verifica se o estado existe na GSPN.
 * @param stateID
 *          o ID do estado.
 * @return TRUE se o estado existe e FALSE, caso
 *         contrário.
 */
public boolean hasState(String stateID) {
  return this.data.containsKey(stateID);
}

/**
 * Cria uma GSPN baseada em informações
 * pré-existentes.
 * @param data
 *          Dados externos a serem inseridos no
 *          estado.
 * @param transitions
 *          Matriz de transições
 * @param probabilities
 *          Matriz de probabilidades
 * @param totalTokens
 *          total de tokens existentes na GSPN
 * @param initialState
 *          ID do estado inicial.
 */
public GSPN(HashMap<String, T> data,
    IGrid<String> transitions,
    IGrid<String> probabilities,
```

```
    int totalTokens, String initialState) {
  this.data = data;
  this.transitions = transitions;
  this.probabilities = probabilities;
  this.tokens = totalTokens;
  this.initialState = initialState;
  this.actualState = initialState;

  this.setFinalState();
}

/**
 * Cria uma GSPN baseada em informações
 * pré-existentes, com estado inicial default.
 * @param data
 *          Dados externos a serem inseridos no
 *          estado.
 * @param transitions
 *          Matriz de transições
 * @param probabilities
 *          Matriz de probabilidades
 * @param totalTokens
 *          total de tokens existentes na GSPN
 */
public GSPN(HashMap<String, T> data,
    IGrid<String> transitions,
    IGrid<String> probabilities, int totalTokens) {
  this.data = data;
  this.transitions = transitions;
  this.probabilities = probabilities;
  this.tokens = totalTokens;
  this.initialState = GSPN.INITIAL;
  this.actualState = GSPN.INITIAL;
  this.setFinalState();
}

/**
 * Cria uma GSPN baseada em informações
 * pré-existentes, com 1 token e estado inicial
 * default.
 * @param data
 *          Dados externos a serem inseridos no
 *          estado.
 * @param transitions
 *          Matriz de transições
 * @param probabilities
 *          Matriz de probabilidades
 */
public GSPN(HashMap<String, T> data,
    IGrid<String> transitions,
    IGrid<String> probabilities) {
  this.data = data;
  this.transitions = transitions;
  this.probabilities = probabilities;
  this.tokens = 1;
  this.initialState = GSPN.INITIAL;
  this.actualState = GSPN.INITIAL;
  this.setFinalState();
```

```
}

/**
 * Gera um número aleatorio entre 0 e 1.
 * @return o número em formato double.
 */
private Double getRandomNumber() {
  Random random = new Random();
  Long temp;
  temp = random.nextLong();
  random.setSeed(temp);
  return random.nextDouble();
}

/**
 * Retorna o número de tokens existentes na
 * GSPN.
 * @return o número de tokens existentes.
 */
public int getTokens() {
  return this.tokens;
}

/**
 * Retorna as informações externas existentes no
 * estado atual.
 * @return as informações no formato
 *         parametrizado.
 */
public T getData() {
  if (this.actualState == this.initialState) {
    this.setNext();
    return this.getData();
  }
  return this.data.get(actualState);
}

/**
 * Verifica se o estado atual é final.
 * @return TRUE caso final e FALSE, caso
 *         contrário.
 */
public boolean isFinalState() {

  if (this.actualState == GSPN.FINAL) {
    return true;
  }
  return false;
}

/**
 * Seta o token para a posição seguinte,
 * escolhendo uma das possíveis posições baseado
 * nas probabilidades de acesso.
 */
public void setNext() {

  if (this.isFinalState()) {
```

```
  if (this.tokens == 1) {
    this.tokens--;
  }
  if (this.tokens > 1) {
    this.tokens--;
    this.actualState = GSPN.INITIAL;
  }
  return;
}

this.transitions.setPositionFromValueAndCol(
    this.actualState, 1);
this.probabilities
    .setPositionFromValueAndCol(
        this.actualState, 1);
this.transitions.setNextCol();
this.probabilities.setNextCol();

while (this.probabilities.getValue() != null) {
  if (this.getRandomNumber() < Double
      .valueOf(this.probabilities.getValue())) {
    this.actualState = this.transitions
        .getValue();
    if (this.isFinalState()) {
      if (this.tokens == 1) {
        this.tokens--;
      }
      if (this.tokens > 1) {
        this.tokens--;
        this.actualState = GSPN.INITIAL;
      }
    }
    return;
  }
  this.probabilities.setNextCol();
  this.transitions.setNextCol();
  if (this.probabilities.getValue() == null) {
    this.transitions
        .setPositionFromValueAndCol(
            this.actualState, 1);
    this.probabilities
        .setPositionFromValueAndCol(
            this.actualState, 1);
    this.transitions.setNextCol();
    this.probabilities.setNextCol();
    String winnerTransition = this.transitions
        .getValue();
    Double bigger = Double
        .valueOf(this.probabilities
            .getValue());
    while (this.probabilities.getValue() != null) {
      this.transitions.setNextCol();
      this.probabilities.setNextCol();
      if (this.probabilities.getValue() != null) {
        if (bigger < Double
            .valueOf(this.probabilities
                .getValue())) {
          bigger = Double
```

```
                    .valueOf(this.probabilities
                        .getValue());
                winnerTransition = this.transitions
                    .getValue();
              }
            }
          }
          this.actualState = winnerTransition;
          if (this.isFinalState()) {
            if (this.tokens == 1) {
              this.tokens--;
            }
            if (this.tokens > 1) {
              this.tokens--;
              this.actualState = GSPN.INITIAL;
            }
          }
          return;
        }
      }
    }
}
```

A.2 Classes responsáveis por prover Máquinas de Estados Finitas com Entradas e Saídas

A.2.1 Interface IFSM

```
package FSM;

import java.util.ArrayList;

import Grid.IGrid;

public interface IFSM<T> {

/**
 * Cria um novo estado na FSM.
 * @param name nome do estado.
 * @param data informação a ser anexada ao estado.
 */
  public void setNewState(String name, T data);

  /**
   * Cria uma nova transição na FSM.
   * @param source estado origem.
   * @param target estado destino.
   * @param in elemento de entrada.
   * @param out elemento de saída
   */
  public void setNewTransition(String source,
      String target, String in, String out);

  /**
   * Identifica que o próximo estado é final.
```

```
     */
    public void setFinalState();

    /**
     * Verifica se existem mais estados relacionados ao estado indicado.
     * @param stateID ID do estado.
     * @return TRUE se existem estados associados e FALSE, caso contrário.
     */
    public boolean hasMoreStatesAssociated(
        String stateID);

    /**
     * Fornece o alfabeto de entrada.
     * @return um ArrayList com o alfabeto de entrada.
     */
    public ArrayList<String> getInputAlphaBet();

    /**
     * Fornece as informações anexadas ao estado atual.
     * @return as informações anexadas ao estado.
     */
    public T getData();

    /**
     * Verifica se o estado atual é final.
     * @return TRUE se o estado é final e FALSE, caso contrário.
     */
    public boolean isFinalState();

    /**
     * Avança para o estado que exija um determinado elemento de entrada.
     * @param in o elemento de entrada do estado.
     * @return o elemento de saída do estado
     */
    public String setNext(String in);

    /**
     * Fornece o conjunto de estados.
     * @return Um ArrayList com os estados existentes na FSM.
     */
    public ArrayList<String> getStateList();

    /**
     * Fornece o ID do estado atual.
     * @return o ID do estado.
     */
    public String getActualStateID();

    /**
     * Fornece a matriz de transições.
     * @return a matriz de transições.
     */
    public IGrid<String> getTransitionMatrix();

    /**
     * Move o cursor para o estado inicial.
     */
    public void reset();
```

```
}
```

A.2.2 Classe FSM

```
package FSM;

import java.util.HashMap;
import Grid.*;
import java.util.ArrayList;
import java.util.HashSet;

public class FSM<T> implements IFSM<T> {

  private HashMap<String, T> data;
  private String initialState;
  public IGrid<String> transitions;
  public IGrid<String> in;
  public IGrid<String> out;
  private String actualState;
  public static final String INITIAL = "INITIALSTATE";
  public static final String FINAL = "FINALSTATE";

  public FSM() {
    this.transitions = new Grid<String>();
    this.transitions.setCols(2);
    this.in = new Grid<String>();
    this.in.setCols(2);
    this.out = new Grid<String>();
    this.out.setCols(2);
    this.data = new HashMap<String, T>();
    this.initialState = FSM.INITIAL;
    this.actualState = FSM.INITIAL;
    this.data.put(this.initialState, null);
    this.transitions.addRow();
    this.transitions.setValue(FSM.INITIAL);
    this.in.addRow();
    this.in.setValue(FSM.INITIAL);
    this.out.addRow();
    this.out.setValue(FSM.INITIAL);
  }

  /**
   * Cria um novo estado na FSM.
   * @param name nome do estado.
   * @param data informação a ser anexada ao estado.
   */
  public void setNewState(String name, T data) {
    if (!this.hasMoreStatesAssociated(name)) {
      this.data.put(name, data);
      this.transitions.addRow();
      this.transitions.setValue(name);
      this.in.addRow();
      this.in.setValue(name);
      this.out.addRow();
      this.out.setValue(name);
    }
  }
```

```
  /**
   * Cria uma nova transição na FSM.
   * @param source estado origem.
   * @param target estado destino.
   * @param in elemento de entrada.
   * @param out elemento de saída
   */
  public void setNewTransition(String source,
      String target, String in, String out) {
    this.transitions.setPositionFromValueAndCol(
        source, 1);
    this.in.setPositionFromValueAndCol(source, 1);
    this.out
        .setPositionFromValueAndCol(source, 1);
    for (int i = 2; i <= this.transitions
        .getCols(); i++) {
      this.transitions.setCol(i);
      this.in.setCol(i);
      this.out.setCol(i);
      if (this.transitions.getValue() == null) {
        this.transitions.setValue(target);
        this.in.setValue(in);
        this.out.setValue(out);
        return;
      }
    }
    this.transitions.addCol();
    this.in.addCol();
    this.out.addCol();
    this.transitions.setValue(target);
    this.in.setValue(in);
    this.out.setValue(out);
    return;
  }

  /**
   * Identifica que o próximo estado é final.
   */
  public void setFinalState() {
    this.transitions.setCol(2);
    this.in.setCol(2);
    for (int x = 1; x <= transitions.getRows(); x++) {
      this.transitions.setRow(x);
      this.in.setRow(x);
      this.out.setRow(x);
      if (this.transitions.getValue() == null) {
        this.transitions.setValue(FSM.FINAL);
        this.in.setValue("0");
        this.out.setValue(null);
      }

    }
    this.transitions.addRow();
    this.transitions.setValue(FSM.FINAL);
    this.in.addRow();
    this.in.setValue(FSM.FINAL);
    this.out.addRow();
```

```
  this.out.setValue(FSM.FINAL);
}

/**
 * Verifica se existem mais estados relacionados ao estado indicado.
 * @param stateID ID do estado.
 * @return TRUE se existem estados associados e FALSE, caso contrário.
 */
public boolean hasMoreStatesAssociated(
    String stateID) {
  return this.data.containsKey(stateID);
}

/**
 * Fornece o alfabeto de entrada.
 * @return um ArrayList com o alfabeto de entrada.
 */
public ArrayList<String> getInputAlphaBet() {
  ArrayList<String> result = new ArrayList<String>();
  HashSet<String> partialResult = new HashSet<String>();
  this.in.setCol(2);
  this.in.setRow(1);
  for (int j = 1; j <= in.getRows(); j++) {
    this.in.setRow(j);
    for (int i = 2; i <= in.getCols(); i++) {
      this.in.setCol(i);
      if (this.in.getValue() != null) {
        partialResult.add(this.in.getValue());
      }
    }
  }
  String[] temp = new String[partialResult
      .size()];
  partialResult.toArray(temp);
  for (int k = 0; k < temp.length; k++) {
    result.add(temp[k]);
  }
  return result;
}

/**
 * Fornece as informações anexadas ao estado atual.
 * @return as informações anexadas ao estado.
 */
public T getData() {
  if (this.actualState == this.initialState) {
    return null;
  }
  return this.data.get(actualState);
}

/**
 * Verifica se o estado atual é final.
 * @return TRUE se o estado é final e FALSE, caso contrário.
 */
public boolean isFinalState() {

  if (this.actualState == FSM.FINAL) {
```

```
      return true;
    }
    return false;
  }

  /**
   * Avança para o estado que exija um determinado elemento de entrada.
   * @param in o elemento de entrada do estado.
   * @return o elemento de saída do estado
   */
  public String setNext(String in) {

    this.transitions.setPositionFromValueAndCol(
        this.actualState, 1);
    this.in.setPositionFromValueAndCol(
        this.actualState, 1);
    this.out.setPositionFromValueAndCol(
        this.actualState, 1);
    this.transitions.setNextCol();
    this.in.setNextCol();
    this.out.setNextCol();

    while (this.in.getValue() != null) {
      if (this.in.getValue().equals(in)) {
        this.actualState = this.transitions
            .getValue();
        return this.out.getValue();
      } else {
        this.transitions.setNextCol();
        this.in.setNextCol();
        this.out.setNextCol();
      }
    }
    throw new NullPointerException();
  }

  /**
   * Fornece o ID do estado atual.
   * @return o ID do estado.
   */
  public String getActualStateID() {
    return this.actualState;
  }

  /**
   * Fornece o conjunto de estados.
   * @return Um ArrayList com os estados existentes na FSM.
   */
  public ArrayList<String> getStateList() {
    ArrayList<String> states = new ArrayList<String>();
    this.transitions.setCol(1);
    for (int i = 1; i < this.transitions
        .getRows(); i++) {
      this.transitions.setRow(i);
      states.add(this.transitions.getValue());
    }
    return states;
  }
```

```
  /**
   * Fornece a matriz de transições.
   * @return a matriz de transições.
   */
  public IGrid<String> getTransitionMatrix() {

    IGrid<String> result = new Grid<String>();
    result.setCols(4);

    for (int i = 1; i <= this.transitions
        .getRows(); i++) {

      this.transitions.setRow(i);
      this.transitions.setCol(1);

      this.in.setRow(i);
      this.in.setCol(2);

      this.out.setRow(i);
      this.out.setCol(2);

      String temp = this.transitions.getValue();

      this.transitions.setNextCol();

      while (this.transitions.getValue() != null) {
        result.setCol(1);
        result.addRow();
        result.setAndMove(temp);
        result.setAndMove(this.in.getValue());
        result.setAndMove(this.out.getValue());
        result.setValue(this.transitions
            .getValue());
        this.in.setNextCol();
        this.out.setNextCol();
        this.transitions.setNextCol();
      }
    }
    return result;
  }

  /**
   * Move o cursor para o estado inicial.
   */
  public void reset() {
    this.actualState = this.initialState;
    return;
  }
}
```