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

AUTOMAÇÃO DE TESTE DE SOFTWARE ATRAVÉS DE LINHAS DE PRODUTOS E TESTES BASEADOS EM MODELOS

LEONARDO DAVI VICCARI

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

Orientador: Prof. Dr. Avelino Francisco Zorzo

Porto Alegre
2009

Dados Internacionais de Catalogação na Publicação (CIP)

V631a Viccari, Leonardo Davi
Automação de teste de software através de linhas de produtos e teste baseados em modelos / Leonardo Davi Viccari. – Porto Alegre, 2009.
85 f.

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

1. Informática. 2. Engenharia de Software. 3. Software – Avaliação. 4. UML (Informática). I. Zorzo, Avelino Francisco. II. Título.

CDD 005.1

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

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

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "Automação de Teste de Software Através de Linhas de Produtos Testes Baseados em Modelos", apresentada por Leonardo Davi Viccari, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído , aprovada em 12/03/10 pela Comissão Examinadora:

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

Prof. Dr. Marcelo Blois Ribeiro – PPGCC/PUCRS

Prof. Dr. Eduardo Augusto Bezerra – UFSC

Prof. Dr. Itana Maria de Souza Gimenes – UEM

Homologada em 15/06/2010, conforme Ata No. 010 pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

ERRATA: No título, onde se lê: "Automação de Teste de Software Através de Linhas de Produtos Testes Baseados em Modelos", leia-se: "Automação de Teste de Software Através de Linhas de Produtos e Testes Baseados em Modelos".

Prof. Avelino Francisco Zorzo
Presidente da Comissão Examinadora

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

DEDICATÓRIA

Dedico este trabalho aos meus pais, Vilso (*in memorian*) e Laurete.

AGRADECIMENTOS

À minha família, pelo exemplo e apoio dados em todos os momentos de minha vida.
À Letícia, pelo carinho, compreensão, por estar ao meu lado e sempre acreditar em mim.
Ao prof. Avelino, pelos ensinamentos, paciência e, principalmente, pela confiança.
Aos meus colegas e amigos, por todos os bons momentos.
À HP, pela bolsa e oportunidade.

AUTOMAÇÃO DE TESTE DE SOFTWARE ATRAVÉS DE LINHAS DE PRODUTOS E TESTES BASEADOS EM MODELOS

RESUMO

Com o crescimento da dependência de inúmeras atividades humanas por sistemas computacionais, cresce também a expectativa dos usuários, que querem não apenas a automação de processos, mas também crescentes qualidade e agilidade na entrega de informações e serviços.

A forma mais difundida de garantia de cumprimento dos requisitos de um sistema é o teste de *software*. Apesar de trazerem indispensáveis benefícios ao processo de construção de *software*, os testes de *software* ainda são atividades extremamente manuais, repetitivas e, por vezes, artesanais.

O uso de modelos formais para descrever o comportamento e a estrutura de um sistema são técnicas baseadas na matemática, comprovadas e aceitas, que em muito acrescentam na tarefa de caracterizar de forma precisa e não ambígua um sistema. A disciplina de testes baseados em modelos compreende uma técnica onde os testes a serem realizados são definidos automaticamente a partir do modelo da aplicação.

Uma das formas de realizar as ideias presentes na técnica de testes baseados em modelos, tirando proveito do reuso de componentes similares de *software* a fim de diminuir o esforço sem perder qualidade é a engenharia de linha de produto de *software*.

O presente trabalho busca aliar técnicas de geração de testes baseados em modelos formais a técnicas de engenharia de linha de produto de *software*, a fim de automatizar etapas manuais da geração e execução de casos de teste. Dessa forma, é proposta uma arquitetura baseada nos conceitos citados, e concretizada através da implementação de uma ferramenta baseada no conceito de *plug-ins*, a fim de validar a ideia proposta.

Palavras-chave: Teste Baseado em Modelos (MBT), Modelos Formais, UML, Linha de Produto de Software (SPL), Automação de Teste de Software.

SOFTWARE TEST AUTOMATION THROUGH PRODUCT LINES AND MODEL BASED TESTING

ABSTRACT

The use and dependency of several human activities on computer systems implicates on a parallel growth of the users' expectations. Users want not only processes automation, but also a crescent quality and agility in the delivery of services and information.

The most popular way to guarantee compliance to a system's requirements is by performing software testing. Although bringing indispensable benefits to the software development process, software testing activities are extremely manual, repetitive, and sometimes handcraft.

The use of formal models to describe the behavior and structure of a system are proven and accepted techniques, based on mathematics. These techniques help in the task of characterizing a system in a precise and non-ambiguous manner. Model-based testing comprehends a technique in which the tests to be executed are automatically defined, based on a formal model of the system.

One of the ways of implementing the ideas present in the model-based testing techniques is the software product line engineering, which takes advantage of similar software components reuse, with the intent of minimizing human effort without losing quality.

This work intends to join formal model-based testing techniques with software product line engineering, in order to automate manual steps of the test case generation and execution. It is proposed, then, a software architecture based on these concepts, realized by the implementation of a tool based in a plug-in architecture concept, in order to validate the proposed ideas.

Keywords: Model Based Testing (MBT), Formal Models, UML, Software Product Lines (SPL), Software Test Automation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

BPEL – *Business Process Execution Language*
BPMN – *Business Process Modeling Notation*
DNA – *Deoxyribonucleic Acid*
FODA – *Feature-Oriented Domain Analysis*
FSM – *Finite State Machine*
GSPN – *Generalized Stochastic Petri Net*
MBT – *Model-Based Testing*
OMG – *Object Management Group*
OO – *Object-Oriented*
SPL – *Software Product Line*
SUT – *System Under Test*
UIO – *Unique Input-Output*
UML – *Unified Modeling Language*
XMI – *XML Metadata Interchange*
XML – *eXtensible Markup Language*

SUMÁRIO

1. INTRODUÇÃO

O uso de sistemas computacionais é, mais do que uma realidade, uma necessidade em inúmeras áreas profissionais e também de conhecimento da humanidade. Com o crescimento no uso desse tipo de sistema e serviço, cresce também a expectativa de seus usuários, que querem não apenas a automação de processos, mas uma crescente qualidade e agilidade na entrega de informações e serviços. Serviços que garantam alta disponibilidade, qualidade, segurança e outros atributos têm um diferencial, principalmente se for levada em consideração a multiplicidade do número de fornecedores para cada serviço, que é uma realidade em muitos casos.

Uma das formas de garantir que um sistema computacional atenda aos seus requisitos dentro de um determinado nível de qualidade é através de aplicação de técnicas de teste de *software*. O teste de *software* pode ser definido como uma investigação empírica conduzida a fim de prover informações sobre a qualidade do produto ou do serviço sendo testado às partes interessadas [KAN06].

Normalmente, cada requisito de um sistema computacional possui um ou mais caso(s) de teste a ele associado(s). Casos de teste são conjuntos de condições que ajudarão um testador a determinar se o sistema sob teste preenche ou não um determinado requisito ou caso de uso. Na engenharia de *software* convencional, a construção de casos de teste é uma etapa que ocorre durante a especificação dos testes.

Modelos formais usados no desenvolvimento de sistemas computacionais são técnicas baseadas na matemática, com o intuito de descrever propriedades do sistema [WIN90]. O uso de um modelo para descrever o comportamento de um sistema é uma vantagem comprovada e uma técnica aceita, que os times de teste e desenvolvimento têm a seu favor. Modelos são usados para entender, especificar e desenvolver sistemas em várias disciplinas, e não só na Ciência da Computação [APF97].

O norte desta pesquisa foi, então, no sentido de maximizar a automação dos testes, a fim de reduzir o esforço gasto nesta etapa, sem comprometer a sua qualidade. Para isso, foi desenvolvida uma ferramenta de geração e execução de testes baseados em modelos, que tende a auxiliar na diminuição de esforço e em tarefas repetitivas inerentes ao teste de *software*. Tendo como base o fato de que diferentes tipos de testes baseados em modelos possuem várias tarefas em comum, como a análise e *parsing* do modelo, geração de um modelo intermediário e geração de um arquivo de saída, uma das formas encontradas para aumentar a efetividade no

desenvolvimento da ferramenta foi o aproveitamento de técnicas de linha de produto de *software* durante a sua concepção e construção.

Linhas de produto podem ser vistas como um tipo de arquitetura e construção que leva em consideração as diferenças e, principalmente, as similaridades entre diferentes produtos, no caso, *softwares* [WEI99], [CLE02]. Trata-se, em última instância, do reaproveitamento de partes do sistema que são comuns a vários módulos, ou ainda a vários produtos.

A presente dissertação apresenta um estudo no sentido de automatizar algumas partes da geração e execução de casos de teste, propondo uma ferramenta que orquestra essas etapas nos moldes de uma linha de produto, focando no reuso de componentes. A principal contribuição deste trabalho é prover uma arquitetura extensível para auxiliar na automação da definição e execução de testes baseados em modelos, de forma genérica. Dessa forma, o Capítulo 2 aborda alguns conceitos de testes de *software*, o Capítulo 3 apresenta a disciplina de testes baseados em modelo e o Capítulo 4 explica a arquitetura de linha de produto de *software* proposta. O Capítulo 5 apresenta a arquitetura proposta para uma nova ferramenta, com casos de uso, possíveis extensões, considerações finais e alguns trabalhos relacionados.

2. TESTE DE *SOFTWARE*

Teste de *software* é o processo que compreende a execução de um programa ou sistema com a finalidade de encontrar erros [MYE04]. É uma investigação empírica conduzida a fim de prover aos envolvidos informações sobre a qualidade do produto ou serviço sendo testado [KAN06]. Teste está, portanto, intimamente relacionado com garantia da qualidade. Teste de *software* pode, também, ser definido como o processo de validar e verificar que um programa, aplicação ou produto de *software* preenche os requisitos de negócio e técnicos que guiaram seu desenvolvimento, funciona conforme o esperado, e pode ser implementado com as mesmas características.

Defeitos em um *software* diferem dos defeitos da maioria dos sistemas físicos existentes. Qualquer equipamento físico ou mecânico falha de forma fixa e muito mais previsível do que um *software*. Além disso, *software* normalmente não sofre de envelhecimento, desgaste e outros males que afetam sistemas físicos. A detecção de todas as formas possíveis que um *software* pode falhar é tecnicamente inviável, a menos que o sistema em questão seja consideravelmente pequeno [PAN99].

O processo de testes não pode identificar completamente todos os defeitos existentes em uma aplicação ou sistema. Pelo contrário, o processo fornece uma crítica ou balisador, a fim de comparar o estado e comportamento do produto com oráculos (princípios ou mecanismos através dos quais alguém pode ser capaz de reconhecer um problema). Estes oráculos podem incluir (mas não são limitados a) especificações, contratos, produtos comparáveis, versões passadas do mesmo produto, inferências sobre o propósito pretendido ou esperado do sistema, expectativa dos usuários ou clientes, padrões ou normas relevantes, leis aplicáveis ou outros critérios [LEI07].

Um dos principais motivos que leva à impossibilidade de cobertura de todos os possíveis casos é a complexidade inerente à grande maioria dos sistemas computacionais. *Bugs* (ou defeitos) de *software* quase sempre existirão em qualquer módulo de *software* com tamanho moderado, não por desleixo ou irresponsabilidade dos programadores, mas porque a complexidade do *software* é normalmente intratável [PAN99]. Além disso, humanos têm uma capacidade apenas limitada de gerência de complexidade, pois são processadores de informação com capacidade finita [KAN06]. Também é verdade que, para qualquer sistema complexo, defeitos nunca serão eliminados por completo [PAN99].

Não se pode garantir que todo *software* funcione corretamente, sem a presença de erros, visto que os mesmos muitas vezes possuem um grande número de estados com fórmulas, atividades e algoritmos complexos. O tamanho do projeto a ser desenvolvido e a quantidade de pessoas envolvidas aumentam ainda mais a complexidade. Idealmente, toda permutação possível do *software* deveria ser testada. Entretanto, isso se torna impossível para a ampla maioria dos casos devido à quantidade impraticável de possibilidades. A qualidade do teste acaba se relacionando à qualidade dos profissionais envolvidos em filtrar as permutações relevantes [MYE04].

Quanto mais um *software* é testado, mais imune ele se torna aos mesmos testes, de forma análoga ao que acontece aos insetos (*bug* em inglês, a mesma palavra geralmente usada para descrever defeitos de um dado *software*) com a aplicação de pesticidas. Se o mesmo pesticida (analogamente, o mesmo teste) continuar sendo aplicado, os insetos (o *software*) tornar-se-ão imunes, e o pesticida (e o teste) não mais produzirá o efeito desejado [BEI90].

A complexidade de um sistema (e, portanto, de seus defeitos) cresce além dos limites de nossa habilidade de gerenciá-la. Eliminando os *bugs* fáceis, novas funcionalidades (e mais complexidade) são adicionadas, bem como novos defeitos, mais sutis, que devem ser eliminados para, no mínimo, manter a confiabilidade anterior. Limitar a complexidade não é um caminho que a sociedade e o mercado estão dispostos a seguir, já que o que as pessoas querem é ter mais e mais funcionalidades. Então, os usuários de qualquer sistema pressionam até a barreira da complexidade. A proximidade que se pode chegar dessa barreira, com segurança, é determinada pela eficácia e pelos resultados das técnicas que podemos usar contra defeitos cada vez mais complexos e sutis [BEI90].

Além da complexidade, portanto, a natureza dinâmica dos sistemas computacionais pode ser considerada um complicador, já que a correção de um defeito ou a adição de uma nova funcionalidade (na verdade, qualquer mudança no código) pode adicionar novos defeitos. Isso faz com que os casos de teste executados no primeiro ciclo não possam mais ser garantidos, o que implica em uma nova execução de todo o ciclo de testes (e normalmente os ciclos crescem a cada iteração). O custo dessas iterações é, normalmente, proibitivo [PAN99].

Teste de *software* não é uma disciplina madura, e ainda é uma arte, porque ainda não podemos fazer dela uma ciência [PAN99]. As técnicas e metodologias utilizadas ainda são as mesmas de algumas décadas atrás, e algumas delas correspondem a métodos manuais e heurísticas, ao invés de bons métodos de engenharia. O teste de *software* é um processo caro, mas não testar o *software* é ainda mais caro. Não se pode ter certeza de que um componente de

software está correto, e nem de que as especificações estão corretas. Nenhum sistema de verificação pode verificar todos os programas. Não podemos sequer afirmar que um sistema de verificação está correto [MYE04].

2.1. Falha, Erro e Defeito

Para abordar corretamente os conceitos e definições relacionados ao teste de *software*, é importante introduzir os conceitos de falha, erro e defeito, fundamentais para o entendimento dos diferentes processos de teste.

Os conceitos básicos das possíveis ameaças a um sistema são os mesmos aplicados largamente em outros trabalhos [WEB01], [AVI04], e são aqui apresentados juntamente com o respectivo termo em inglês, já que alguns não são facilmente traduzidos para o português. Ameaças são fatos ou eventos que podem afetar um sistema e causar uma queda na sua dependabilidade. Basicamente, podemos ter três tipos de ameaça em um sistema computacional: falhas, erros ou defeitos.

Defeito (*failure*) é um evento que ocorre quando o serviço desvia do resultado correto, ou seja, desvia da sua especificação. Isto pode ocorrer porque o serviço não está aderente à especificação funcional, ou ainda porque a especificação não descreve corretamente o funcionamento do serviço. Um sistema está em um estado errôneo, ou erro (*error*) se, a partir do estado atual, o processamento pode resultar em um defeito. Como um serviço é uma seqüência de estados externos do sistema, um defeito significa que pelo menos um dos estados externos desviou do estado correto do serviço. O erro é justamente este desvio. Falha (*fault*), ou falta, é a causa raiz, seja física ou algorítmica, determinada ou hipotética, do erro. Como exemplo clássico de falha/erro/defeito, temos o pneu furado de um carro. A falha é o furo no pneu. O erro é a perda de pressão no interior do pneu, e o defeito é o fato de que o carro não pode mais rodar devido à perda de pressão.

2.2. Aplicações de Teste de *Software*

Os próximos parágrafos descrevem as justificativas mais comuns e aceitas para o emprego de testes de *software*, o que equivale a dizer que são os principais motivos pelos quais testamos.

O principal motivo para o emprego de testes é a melhoria de qualidade que os testes podem proporcionar a um sistema. Qualidade significa conformidade com os requisitos de

projeto. A qualidade mínima esperada em qualquer projeto é, na prática, execução conforme requisitado sob as circunstâncias esperadas. A imperfeição da natureza humana faz com que seja praticamente impossível construir um programa moderadamente complexo da forma correta na primeira tentativa. Encontrar os problemas e corrigi-los é o propósito do teste e correção de defeitos durante o desenvolvimento de um sistema [KAN99].

Técnicas de teste são, também, aplicadas com o intuito de verificação e validação de um sistema. Verificação é um processo de controle de qualidade usado para avaliar se um produto, serviço ou sistema está em conformidade com as especificações ou condições impostas no início da fase de desenvolvimento. Validação é um processo de garantia de qualidade executado a fim de coletar evidências de que um produto, serviço ou sistema cumpre seus requisitos. A diferença é sutil, e pode ser feito um paralelo com as seguintes perguntas: validação normalmente significa perguntar: “você está fazendo a coisa certa?”, enquanto verificação consiste na pergunta: “você está fazendo da maneira certa?”.

A confiabilidade de *software* está intimamente relacionada com muitos de seus aspectos, incluindo a estrutura e a quantidade de testes ao qual o *software* foi submetido. Baseado em um perfil operacional (ou seja, uma estimativa da frequência relativa do uso de várias entradas no programa), testes podem servir como uma amostra estatística para ter acesso a dados para a estimativa de confiabilidade [KAN99], [LYU96].

2.3. Tipos de Teste de *Software*

As metodologias de testes são normalmente divididas em testes de caixa branca e testes de caixa preta. Elas diferem entre si quanto ao ponto de vista de um engenheiro de teste durante o projeto dos casos de teste. Se o engenheiro conhece as estruturas de dados e algoritmos internos, ou seja, se tem acesso ao código do sistema, a metodologia é denominada caixa branca. Caso contrário, se os detalhes internos não forem de conhecimento do engenheiro de testes, temos a metodologia de caixa preta [SAV08].

As próximas seções discorrem sobre os tipos, ou níveis, de teste aos quais um sistema computacional pode ser submetido [MYE04]. É importante ressaltar que existem outras formas de classificar testes, como teste de desempenho, teste de segurança, teste estatístico, entre outros [KER99], [MCG01], [NGU06].

2.3.1. Teste Unitário

Teste de módulo (ou unitário) é o processo de testar os subcomponentes individuais de um programa, tais como rotinas, procedimentos, classes, etc. Ou seja, ao invés de começar o teste pelo programa como um todo, o teste foca inicialmente nos pequenos blocos que formam o programa. As justificativas para proceder desta forma são a maior facilidade para isolar os defeitos encontrados, por termos a certeza de que se encontram em um módulo menor que o programa inteiro, a possibilidade de executar testes de vários módulos em paralelo, e por ser uma forma de gerenciar possíveis combinações de testes.

2.3.2. Teste de Integração

Teste de integração pode ser definido como qualquer tipo de teste de *software* que objetive verificar as interfaces entre componentes, em relação ao projeto do *software*. Os componentes podem ser integrados de forma iterativa ou todos de uma vez. A primeira opção normalmente é considerada uma melhor prática, por permitir que problemas de interface e interação sejam encontrados e corrigidos mais rapidamente.

2.3.3. Teste Funcional

Teste funcional é o processo de tentar encontrar diferenças entre o programa e sua especificação externa. Especificação externa é uma descrição precisa e não ambígua do comportamento do programa, do ponto de vista de seu usuário final. Exceto quando usado em programas pequenos, o teste funcional é normalmente uma atividade de caixa preta, ou seja, sem acesso aos detalhes de implementação. Isso significa que o processo de teste confia nos testes unitários previamente executados para atingir os critérios de cobertura conhecidos como caixa branca.

2.3.4. Teste de Sistema

Normalmente, o teste de sistema é o que mais tem falhas em sua interpretação, e também o que apresenta o processo mais difícil de ser seguido. Não é um processo de teste relativo às funções do sistema ou programa completo, porque isso seria redundante com o processo de teste

funcional. O teste de sistema tem como principal propósito comparar o *software* aos seus objetivos originais, e isto gera duas implicações. A primeira é que o teste de sistema não é limitada a sistemas. Se o produto sob teste é um programa, o teste de sistema é o processo que visa demonstrar como o programa, como um todo, não cumpre seus objetivos. A segunda é que o teste de sistema, por definição, é impossível de ser executado caso não haja um conjunto de objetivos escritos e mensuráveis para o produto.

2.3.5. Teste de Aceitação

Teste de aceitação é definido como o processo de comparação do programa a seus requisitos iniciais e às necessidades atuais de seus usuários finais. É um tipo de teste não usual, e normalmente é executado pelo cliente ou usuário final do programa, e não é considerado responsabilidade da organização que desenvolve o programa. Se houver um contrato em vigência, a organização contratante (usuário) realiza o teste de aceitação através da comparação da operação do programa com o contrato original.

2.3.6. Teste de Instalação (ou Implantação)

O teste de implantação não está relacionado (ao contrário de todos os demais processos de teste) a alguma fase específica do processo de projeto do *software*. É um tipo de teste não usual, pois seus propósitos não são encontrar erros de *software*, e sim encontrar erros que ocorram durante a instalação do *software*.

2.4. Considerações

É muito difícil dissociar o conceito de teste de *software* da ideia de qualidade do *software*. As atividades compreendidas nos diversos processos de teste de *software* estão intimamente relacionadas com esta busca pela melhoria na qualidade, além da diminuição dos riscos da implantação de um sistema. Tais atividades são, normalmente, custosas, e exigem um grande esforço da equipe de testes. O próximo capítulo apresenta algumas alternativas que buscam diminuir o esforço humano gasto nas atividades de teste, os testes baseados em modelos.

3. TESTES BASEADOS EM MODELOS

Testes manuais de *software* podem se tornar excessivamente tediosos e, principalmente, caros, fazendo com que a automação dos testes, por menor que possa ser, seja altamente desejada. No entanto, a automação total dos testes é uma tarefa difícil de ser alcançada. As razões incluem a sensibilidade do processo a mudanças no sistema e o fato do sistema sob teste não ser sempre claro a partir dos *scripts* ou planos de teste. A derivação de sequências apropriadas de teste é um dos obstáculos à automação. Teste baseado em modelos ou especificações é uma técnica promissora para a geração de casos de teste [KRI04].

Testadores de produto, assim como desenvolvedores, estão normalmente sob severa pressão devido aos curtos ciclos de desenvolvimento de versões de *software* esperados no mercado atual. Diversos problemas têm encorajado as organizações a procurar por técnicas que apresentem melhorias frente à tradicional maneira artesanal de elaborar casos de teste individuais. Técnicas de automação de teste oferecem muita esperança aos times de teste. A maneira mais simples de automatizar é a execução automática de testes, que permite que conjuntos de testes artesanais sirvam como testes de regressão. Todavia, execução automática de testes não trata os problemas de alto custo de desenvolvimento dos testes e cobertura incerta do domínio de entrada [DAL99].

As técnicas de testes baseados em modelos, mesmo não sendo uma panacéia, oferecem uma promessa considerável de redução do custo de geração de testes, aumentando a efetividade e encurtando os ciclos de testes. A geração dos testes pode ser especialmente eficaz em sistemas que mudam frequentemente, já que os testadores podem atualizar o modelo de dados e rapidamente gerar um conjunto de testes, evitando uma edição tediosa e sujeita a erros de todo um conjunto de testes artesanais [DAL99].

3.1. Modelos Formais

De forma simplificada: um modelo de *software* é uma representação de seu comportamento [ELF01a]. Modelos ou métodos formais são técnicas matematicamente rigorosas e ferramentas para a especificação, projeto e verificação de sistemas de *software* e *hardware*. “Matematicamente rigoroso” significa que as especificações usadas nos métodos formais são assertivas bem formadas (do ponto de vista da lógica matemática) e que as verificações formais

nesse sentido são deduções rigorosas nesta lógica. O maior valor alcançado com o uso de modelos formais é a obtenção de um meio de examinar de forma simbólica todo o espaço de estados de um projeto de *software* ou *hardware*, e estabelecer uma propriedade de corretude ou segurança que é verdadeira para todas as possíveis entradas. Contudo, estas técnicas são raramente postas em prática (exceto em componentes críticos ou sistemas dos quais dependam vidas), devido à enorme complexidade dos sistemas reais.

Modelagem é uma maneira econômica de captura de conhecimento sobre um sistema, a fim de reusar este conhecimento conforme o sistema cresce. Essa informação é muito valiosa para uma equipe de testes. Tipicamente, as informações sobre o sistema sendo testado são preservadas apenas dentro de scripts de teste, ou, no máximo, em um plano de testes. Se um modelo for construído definindo o comportamento esperado do sistema, tem-se um mecanismo que possibilita uma análise estruturada desse sistema [APF97]. Dessa forma, fica mais fácil o entendimento acerca da cobertura de testes e os planos de teste são desenvolvidos no contexto do sistema e dos recursos para ele disponíveis. Obviamente, os maiores benefícios são o reuso e a formalização do conhecimento.

O subconjunto mais importante dos modelos formais, do ponto de vista da disciplina de teste baseado em modelos, é [ELF01a]: máquinas de estados finitas, diagramas de estados, diagramas UML, cadeias de Markov e gramáticas.

3.1.1. Máquinas de Estados Finitas

Uma máquina de estados finita (*finite state machine*, FSM), ou simplesmente uma máquina de estados, é um modelo de comportamento composto por um número finito de estados, transições entre este estados, e ações. É similar a um grafo de fluxo, em que é possível inspecionar a forma pela qual a lógica executa quando certas condições são atingidas. Uma FSM é um modelo abstrato de uma máquina com uma memória interna primitiva.

FSMs são uma solução simples e elegante, especialmente para *software* de comunicação, mas também para qualquer componente de *software* que deva lidar com decisões e temporização. Para descrever uma FSM são necessários quatro elementos: conjunto de possíveis estados, conjunto de possíveis entradas, conjunto de possíveis transições de um estado para outro e o conjunto de ações realizadas a cada transição [DRU04]. A Figura 1 [LEE96] ilustra uma FSM simples, que possui três estados (S_1, S_2 e S_3) e seis transições. Em cada transição, o caracter à

esquerda da barra (/) significa a entrada consumida pela transição, e o caracter à direita da barra representa a saída produzida pela transição.

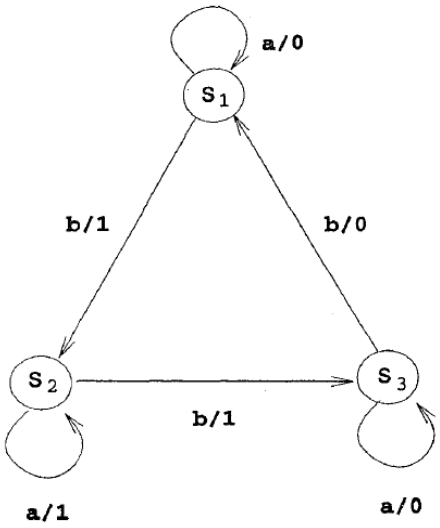

Figura 1 – Exemplo de máquina de estados finita

FSMs se aplicam a qualquer modelagem que possa ser precisamente descrita com um número finito (e normalmente pequeno) de estados específicos. Máquinas de estado existem desde antes da engenharia de *software*, e, juntamente com algumas outras variações, são o centro de uma teoria da computação madura e estável. O uso de FSMs no projeto e teste de componentes de *hardware* computacionais está estabelecido há muito tempo, e é considerado o padrão hoje.

Modelos de estados finitos são a escolha mais óbvia para o teste de *software* onde os engenheiros de teste lidam com construção de sequências e entrada que servirão de dados de teste. Máquinas de estado são modelos ideais para descrever sequências de entrada. A combinação destes elementos com algoritmos para percorrer grafos fazem a geração de testes ser menos penosa do que os testes manuais. Por outro lado, componentes complexos de *software* implicam em grandes máquinas de estados, cuja construção e manutenção não são triviais [ELF01a].

3.1.2. Diagramas de Estados

Um diagrama de estados (*statechart*) é uma representação de uma máquina de estado que modela as mudanças de comportamento dos estados. Diagramas de estados mostram os vários estados que um objeto percorre, bem como os eventos que causam transições de um estado para

outro. Os elementos presentes nesse tipo de modelo são: estados, estados iniciais e finais, transições, e ações de entrada, saída e execução.

Diagramas de estados estendem as máquinas de estados, e [HAR87] apresenta uma explicação simples sobre a forma como isso ocorre: um diagrama de estados é uma máquina de estados com profundidade, ortogonalidade e comunicação em *broadcast*.

Além de serem uma extensão das FSMs, os diagramas de estados tratam especificamente a modelagem de sistemas complexos ou de tempo real. Provêem um *framework* que permite especificar máquinas de estado hierarquizadas, em que um estado único pode ser expandido em outra máquina de estados de nível mais baixo. Também provêem máquinas de estado concorrentes, uma funcionalidade que só tem equivalente na teoria dos autômatos. Além disso, a estrutura dos diagramas de estado envolve condições que afetam a ocorrência de uma determinada transição a partir de um estado em particular, o que em muitas situações pode reduzir o tamanho do modelo sendo criado. Diagramas de estados são intuitivamente equivalentes à mais poderosa forma de autômato: a máquina de Turing. No entanto, são mais pragmáticos, enquanto mantém a mesma capacidade de expressão. Diagramas de estados são provavelmente mais fáceis de ler do que FSMs, apesar de seu manuseio não ser trivial, a ponto de requerer treinamento [ELF01a].

A Figura 2 ilustra um diagrama de estados representando o comportamento de um semáforo, com as transições entre as três lâmpadas do mesmo.

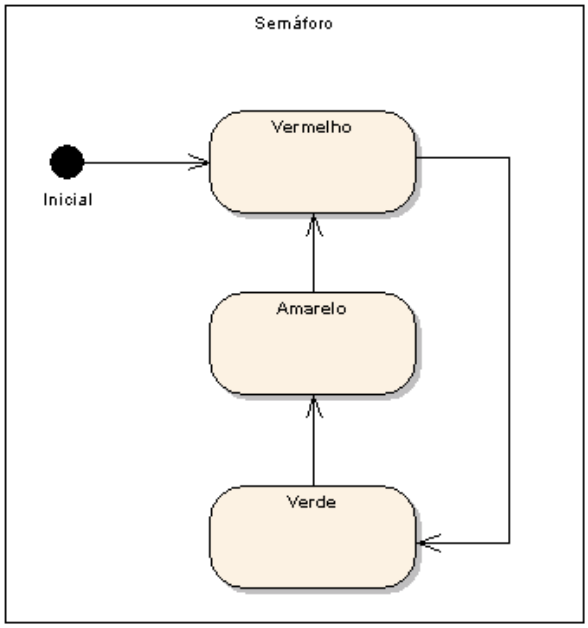

Figura 2 – Exemplo de diagrama de estados

3.1.3. Diagramas UML

A linguagem unificada de modelagem (*Unified Modeling Language*, UML) é uma família de notações gráficas, suportada por um meta-modelo, que ajuda a descrever e projetar sistemas de *software*, particularmente os construídos usando o paradigma orientado a objetos (*object-oriented*, OO) [FOW04]. A UML é um padrão relativamente aberto, controlado pelo *Object Management Group* (OMG), um consórcio aberto de companhias.

A versão atual da UML (UML 2.2) possui quatorze tipos de diagramas, divididos em duas categorias principais. Sete dos diagramas representam informações estruturais, enquanto os demais sete representam tipos de comportamento [OMG09]. Os diagramas podem ser hierarquicamente categorizados conforme a Figura 3 [OMG09]. Ou seja, há dois grandes grupos, o primeiro contendo os diagramas estruturais (abaixo de *structure diagram*), e o segundo contendo os diagramas comportamentais (abaixo de *behavior diagram*)

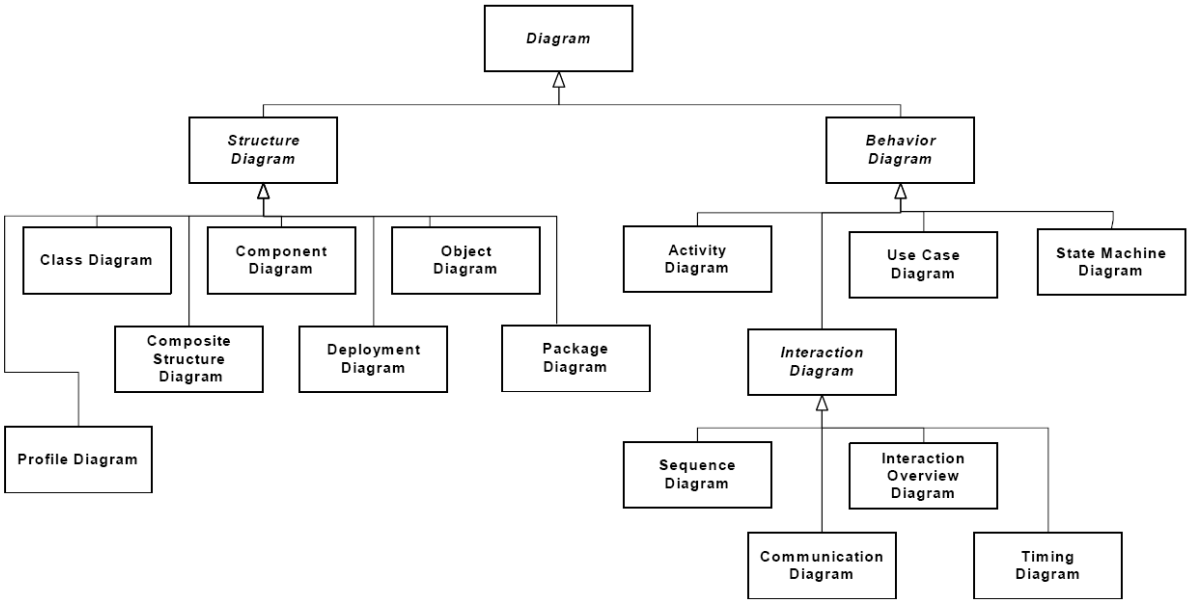

Figura 3 – Categorização dos diagramas UML

Com base no escopo deste trabalho, os diagramas mais importantes são o diagrama de casos de uso (*Use Case Diagram*) e o diagrama de atividades (*Activity Diagram*). Não por acaso ambos são diagramas de cunho comportamental, já que a geração de testes aqui proposta leva em consideração o comportamento dos sistemas sob teste.

Um diagrama de casos de uso UML é um tipo de diagrama comportamental definido e criado a partir de uma análise de casos de uso. Seu propósito é apresentar uma visão geral, gráfica, da funcionalidade provida por um sistema, em termos de seus atores, seus objetivos (representados por casos de uso), e quaisquer dependências que possam existir entre estes casos de uso. Sua principal função é mostrar quais funções do sistema são executadas por quais atores. O papel de cada ator no sistema pode ser detalhado conforme a necessidade. A Figura 4 exemplifica um diagrama simplificado de casos de uso de um sistema bancário, em que existem dois atores: “Cliente” e “Gerente”. O ator “Cliente” está associado aos casos de uso correspondentes às tarefas que ele pode realizar, como solicitar o saldo da conta corrente, solicitar um empréstimo, realizar saque ou realizar depósito. No caso de uma solicitação de empréstimo, é necessária a aprovação do gerente, o que implica na associação do caso de uso correspondente com o ator “Gerente”.

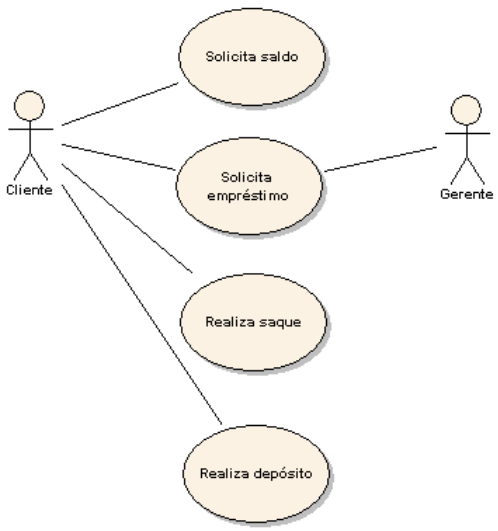

Figura 4 – Exemplo de diagrama de casos de uso

Diagramas de atividades são representações visuais mostrando fluxos de trabalho (*workflows*) de atividades e ações, com suporte a escolha, iteração e concorrência. Na UML, diagramas de atividades podem ser usados para descrever as sequências operacionais e de negócio dos componentes de um sistema. Um diagrama de atividades mostra, essencialmente, o fluxo geral de controle do sistema. A Figura 5 exemplifica o diagrama de atividades correspondente à solicitação de um empréstimo em um sistema bancário. Basicamente, o usuário se identifica no sistema e, caso seja um usuário válido e obtenha a aprovação do gerente para executar a operação, o empréstimo é concedido. Caso contrário, o empréstimo é negado.

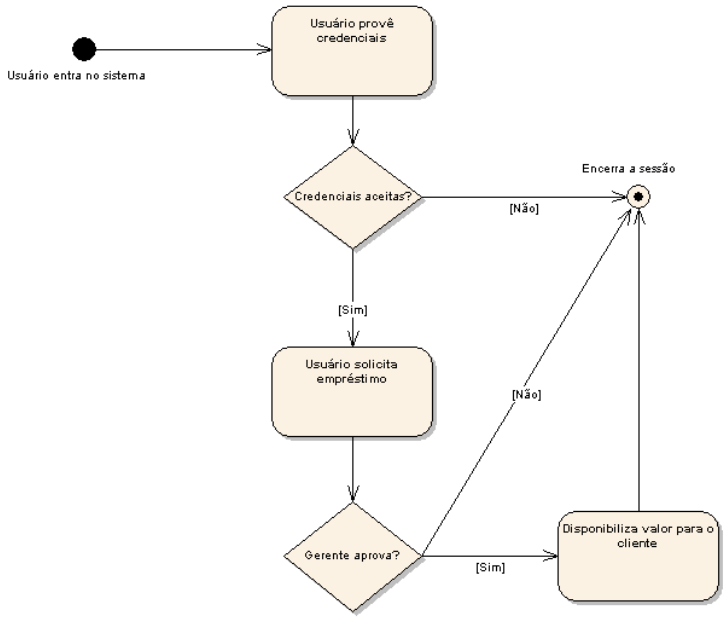

Figura 5 – Exemplo de diagrama de atividades

Dois fatores colocam a UML como principal escolha quando da aplicação de testes automatizados: vasto suporte por ferramentas comerciais e aplicação em toda a indústria. Projeto e execução de testes com o apoio da UML estão largamente difundidos em toda a academia e indústria, conforme relatado em diversos trabalhos [ABD00], [BAS00], [HAR00], [OFF99], [SKE00], [NET07].

3.1.4. Cadeias de Markov

Na matemática, uma cadeia de Markov é um processo randômico onde todas as informações sobre o futuro estão contidas no estado atual (ou seja, não é necessário examinar o passado para determinar o futuro). Para ser mais exato, o processo possui a propriedade de Markov [MAR54], que significa que os estados futuros dependem apenas do estado atual, e são independentes dos estados passados [KEM76].

Em uma cadeia de Markov, a cada passo, o sistema pode mudar do estado atual para algum outro estado (ou permanecer no mesmo estado), de acordo com uma distribuição de probabilidades. As mudanças de estado são chamadas de transições, e as probabilidades associadas com várias mudanças de estado são chamadas probabilidades de transição.

A Figura 6, adaptada de [MOL09], representa uma cadeia simples de Markov, onde cada estado é uma das bases nitrogenadas encontradas no ácido desoxirribonucleico (*deoxyribonucleic acid*, DNA). A cadeia nada mais é do que um conjunto de estados conectados por setas chamadas de transições. Cada transição tem um parâmetro de probabilidade a ela associada. A probabilidade contida em uma seta de C para G, por exemplo, representa a probabilidade de um G seguir um C. A Tabela 1 [MOL09], apresenta outra maneira de representar as probabilidades de uma base seguir outra na cadeia do DNA.

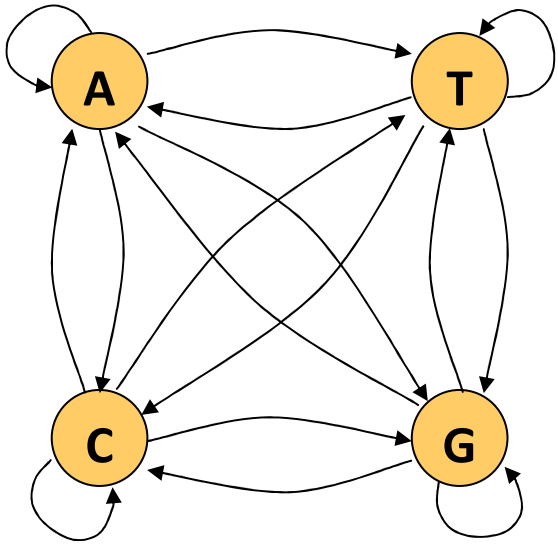

Figura 6 – Cadeia de Markov representando as bases do DNA

Tabela 1 – Matriz de probabilidades de ocorrências das bases do DNA

	A	**C**	**G**	**T**
A	0.95	0	0.05	0
C	0.2	0.5	0	0.3
G	0	0.2	0	0.8
T	0	0	1	0

Uma classe específica de cadeias de Markov, a cadeia de Markov irredutível, homogênea ao tempo, com estados finitos e parâmetros discretos, vem sendo usada para modelar o uso de *software* [ELF01a]. Elas são estruturalmente similares a FSMs, e podem ser vistas como autômatos probabilísticos. Seu propósito primário é não apenas gerar testes, mas também coletar e analisar

dados de defeitos, a fim de estimar medidas como confiabilidade e tempo médio para a falha [AVI04].

3.1.5. Gramáticas

Uma gramática é um conjunto de regras para formar palavras em uma linguagem formal. Estas regras formadoras da gramática descrevem como formar palavras (válidas do ponto de vista sintático) usando o alfabeto da linguagem. A gramática não descreve o significado das palavras, apenas seu lugar e a forma como podem ser manipuladas [CHO56], [CHO57]. Gramáticas e linguagens formais formam a teoria das linguagens formais, que está estreitamente relacionada à teoria dos autômatos.

A Figura 7 contém as regras de produção de uma gramática que, juntamente com o alfabeto {a,b}, estado inicial S, e conjunto de estados {S, A, B} aceita as palavras $\{a^n b^m, n,m \geq 1\}$.

1. $S \longrightarrow aA$
2. $A \longrightarrow aA$
3. $A \longrightarrow bB$
4. $B \longrightarrow bB$
5. $B \longrightarrow \varepsilon$

Figura 7 – Exemplo de gramática

Gramáticas são mais usadas para descrever a sintaxe de linguagens de programação e outras linguagens de entrada de dados. Sob o aspecto funcional, diferentes classes de gramáticas são equivalentes a diferentes formas de máquinas de estado. Normalmente as gramáticas são representações mais simples e compactas para modelar certos tipos de sistemas, como parsers. Também são geralmente mais fáceis de escrever, revisar e manter [ELF01a]. No entanto, no que tange à geração de testes e definição de critérios de cobertura, há algumas restrições ao uso de gramáticas, e não há muitos trabalhos publicados nesse sentido.

3.1.6. Redes de Petri

Rede de Petri é mais uma entre diversas linguagens matemáticas de modelagem para a descrição de sistemas distribuídos discretos. Uma rede de Petri é um grafo bipartido direto, no qual os nodos representam transições (por exemplo, eventos discretos, representados por barras),

lugares (por exemplo, condições, desenhadas como círculos), e arcos direcionados (descrevendo quais lugares são pré- ou pós-condições para quais transições, identificados graficamente por setas). As redes de Petri foram originalmente apresentadas por Carl Adam Petri, no ano de 1962 [PET62] apud [EHR04].

Lugares podem conter qualquer número não negativo de fichas (*tokens*, em inglês). Uma distribuição de fichas entre os lugares da rede é chamado de *marca*, e representa o estado da rede. Uma transição em uma rede de Petri pode acontecer sempre que houver uma ficha no início de cada um dos arcos de entrada da transição. Quando a transição executa, ela consome estas fichas e coloca fichas no fim de cada um de seus arcos de saída. Uma execução de uma transição é sempre atômica.

A Figura 8, adaptada de [SOW00], ilustra uma rede de Petri simples, representando um ponto de ônibus, onde estão modelados o ônibus e os passageiros, e as transições e estados correspondentes, como a chegada, espera e partida de um ônibus e a espera das pessoas pela chegada de um ônibus.

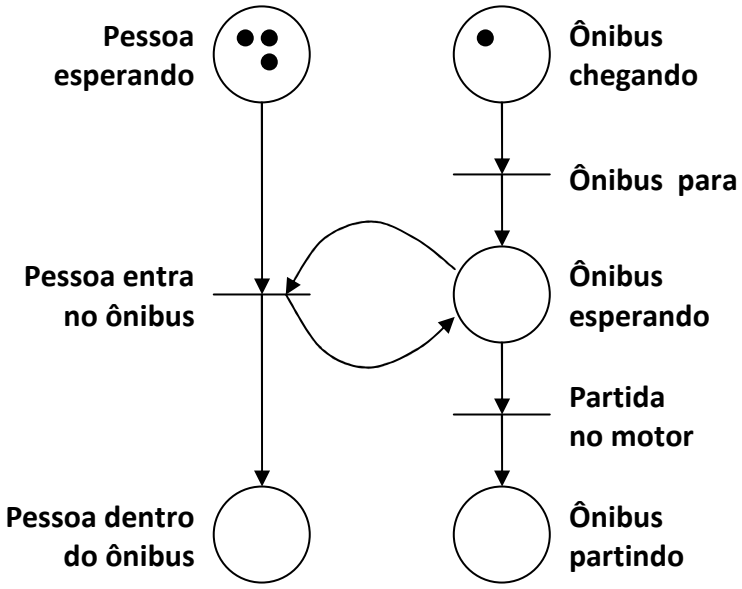

Figura 8 – Exemplo de rede de Petri

Redes de Petri possuem execução não-determinística, pois quando há a possibilidade de múltiplas transições serem executadas, qualquer uma delas pode ser executada. Se uma transição está com seus requisitos (ou seja, fichas) preenchidos, ela pode executar, mas não necessariamente executará. Como a execução é não-determinística, e múltiplas fichas podem

estar presentes em quaisquer lugares da rede (até mesmo em um único lugar), as redes de Petri são boas candidatas para a modelagem de comportamento concorrente em sistemas distribuídos.

Quando usadas como ferramentas gráficas, as redes de Petri provêem um poderoso meio de comunicação entre o usuário (tipicamente o engenheiro de requisitos) e o cliente do sistema. Especificações complexas de requisitos, ao invés de usar descrição textual ambígua ou notações matemáticas difíceis de serem entendidas pelo consumidor, podem ser representadas graficamente usando redes de Petri. Essa possibilidade, combinada com a existência de ferramentas computacionais que permitem realizar simulações gráficas de redes de Petri, dá aos engenheiros do produto uma poderosa ferramenta, que auxilia no processo de desenvolvimento de sistemas complexos [ZUR94].

3.1.7. Considerações

Existem outros modelos interessantes à área de teste de *software*, como tabelas de decisão e árvores de decisão, e alguns trabalhos, como [DAV88], apresentam comparativos entre os modelos aqui citados.

3.2. Testes Baseados em Modelos

Teste baseado em modelos (*model-based testing*, MBT) é um processo que embasa tarefas comuns de teste, tais como geração de casos de teste e avaliação dos resultados de um teste, em um modelo da aplicação sendo testada [JOR95]. Usualmente, os aspectos do sistema que são descritos pelo modelo são aspectos funcionais.

Podemos afirmar que toda e qualquer atividade de teste de *software* é considerada "baseada em modelos", já que qualquer caso de teste é projetado a partir de, pelo menos, algum modelo mental da aplicação sendo testada [HAR02]. O processo de desenvolvimento de *software* tem se beneficiado do uso de modelos (como a UML), obtendo resultados mais concisos e corretos. No entanto, o uso desse tipo de modelo para a geração de casos de teste ainda é uma área a ser muito explorada [HAR02]. As tarefas fundamentais em MBT são [ELF01a]:

- Entendimento do sistema sendo testado;
- Escolha do modelo;
- Construção do modelo;

- Geração dos testes;
- Execução dos testes;
- Coleta dos resultados dos testes;
- Uso dos resultados dos testes.

A construção de um modelo é um esforço criativo; envolve as habilidades, a experiência e a imaginação do testador, e, por isso, se assemelha à escrita de código. A escolha de um modelo em MBT é como a escolha de uma linguagem de programação para a implementação de um sistema: a decisão do modelo a ser usado é crítica para a produtividade e efetividade de todas as tarefas baseadas no modelo durante o processo de testes. Infelizmente, a maioria das boas práticas é específica a modelos em particular. Porém, os passos a seguir são sempre muito úteis para se ter em mente, com relação a modelos que serão usados em MBT [ELF01b]:

- O modelo precisa ser tão breve quanto possível, mas sem o sacrifício de informações.
- O modelo não deve conter informação que não seja pertinente aos objetivos.
- O modelo deveria excluir informação que seja pertinente, porém redundante ou inútil.
- O modelo precisa ser tão legível quanto possível, mesmo que às custas da brevidade.
- O modelo deve ter pontos de extensão e ser aberto a mudanças, caso seja possível.
- A escrita do modelo precisa considerar todos os detalhes de implementação dos testes, mesmo que o modelo não o faça.

A fim de obter resultados aceitos pela indústria, um método puramente acadêmico de aplicação de MBT não seria totalmente eficaz. Então, alguns requisitos extras são impostos. Exemplos citados são: o uso de uma especificação semiformal (como a UML), necessidade de produção de um plano de teste em estágios iniciais do processo de desenvolvimento, mesmo quando o sistema estiver apenas parcialmente modelado, e ainda a precisão do teste [BER04]. A redução do esforço de teste é normalmente obtida às custas da precisão dos resultados dos testes.

Os principais benefícios do uso de MBT são permitir que os modelos suportem análise de defeitos de requisitos e automação de testes [BLA04], [ROB99], [ELF01b]. Checagem baseada em modelos pode garantir que algumas propriedades, como a consistência, não sejam violadas. Adicionalmente, ajudam a refinar requisitos não claros ou mal-definidos. Após o refinamento dos modelos, ocorre a geração de testes. É possível, então, eliminar os defeitos do modelo

antecipadamente, antes do início da codificação do sistema, além de automatizar o projeto dos testes e gerar casos e *scripts* de teste de forma mais eficiente. Todo este conjunto resulta em significativa redução de custo e código com maior qualidade. O típico resultado esperado, quando há a aplicação de MBT, é a automação da geração de testes. Porém, há outros benefícios que normalmente só são percebidos após a aplicação de MBT, como um melhor entendimento dos requisitos, melhoria na consistência e plenitude do sistema e seus casos de teste e a antecipação da identificação e remoção de defeitos dos requisitos, que se reflete na diminuição de custos.

Algumas das principais dificuldades de implantação dos testes baseados em modelos são o aporte de investimentos substanciais, como em conhecimento, tempo e outros recursos necessários. Os testadores também precisam estar familiarizados com o(s) modelo(s) e ter um mínimo de conhecimento teórico e matemático (autômatos, linguagens formais, teoria dos grafos, estatística, etc.). Também precisam possuir experiência em ferramentas, scripts e linguagens de programação [ELF01a].

A fim de economizar recursos na etapa de testes, o uso de MBT requer um esforço inicial considerável, na escolha do tipo de modelo, na divisão das funcionalidades do sistema em partes do modelo, e na construção do modelo. O custo destas etapas pode se tornar proibitivo se não houver uma combinação de planejamento cuidadoso, ferramentas apropriadas e suporte de especialistas.

Além disso, os modelos inserem algumas dificuldades que não podem ser completamente superadas, sendo necessário algum tipo de contorno. Por exemplo, modelos de estados podem crescer além do limite gerenciável, mesmo com o uso de ferramentas apropriadas. Portanto, deve-se considerar que um processo puramente baseado em técnicas de MBT pode não ser a melhor opção, mas sim um conjunto de técnicas que inclua MBT no seu núcleo.

Levando em consideração os benefícios e dificuldades apresentados pelo processo de testes baseados em modelos, a técnica tem sido aplicada em praticamente todo tipo de sistema computacional, e como alguns exemplos dessa diversidade, podemos citar *software* automotivo [BRI08], sistemas médicos [HAS08], páginas *web* [KRI04], segurança no controle de acesso de pessoal [PRE01], telefonia [DAL98], etc.

3.3. Considerações

A técnica de testes baseados em modelos apresenta benefícios indiscutíveis quando corretamente aplicada, ajudando na automação de tarefas manuais, melhor entendimento dos

requisitos e do sistema, e maior produtividade dos envolvidos (problemas críticos na aplicação tradicional de testes manuais e exaustivos). A técnica também acrescenta uma maior certeza da obtenção de resultados, pela formalidade que agrega ao processo de desenvolvimento de *software*.

UML é a notação mais aceita pela indústria atualmente, e, portanto, a que mais tem chances de dar certo em um projeto de *software* normal, pela maior disponibilidade de recursos humanos capacitados, menor curva de aprendizado, resultados já apresentados em toda a indústria, entre outros fatores. Porém, a UML, por ser um modelo semi-formal, não apresenta a mesma facilidade de adaptação a algoritmos clássicos e bem estabelecidos, quando comparada a máquinas de estado e redes de Petri, por exemplo.

O Capítulo 5 deste trabalho apresenta uma proposta que permite a descrição de um sistema através de um modelo formal qualquer. Os exemplos de aplicação citados utilizam como modelo de entrada diagramas UML, e trabalham, internamente, com modelos mais formais (como máquinas de estado e redes de Petri) para a geração e manuseio de casos e *scripts* de teste.

Ferramentas de teste baseado em modelos podem apresentar diversas similaridades e, portanto, faz sentido o reaproveitamento de componentes já desenvolvidos, ou mesmo a adaptação do seu processo de construção para que leve em consideração as partes em comum e as diferenças entre as ferramentas. Com esse propósito, o próximo capítulo apresenta o conceito de linha de produto de *software*.

4. LINHAS DE PRODUTO DE *SOFTWARE*

Devido à similaridade de muitas das ferramentas de apoio a testes baseados em modelos, a obtenção de uma implementação que contemple o reuso de *software* e a diminuição da duplicação do esforço gasto em desenvolvimento é uma vantagem considerável. Quando o *software* é uma grande proporção (ou a totalidade) do custo de um sistema, muitas organizações examinam como podem alavancar um maior reuso dos *softwares* que elas criaram ou criarão. Normalmente, o reuso significante é obtido quando as empresas constroem produtos de *softwares* repetíveis com 90% a 100% de intersecção de funcionalidades, como em compiladores, por exemplo [MAN02]. O mesmo raciocínio se aplica a ferramentas de testes baseados em modelos, pois de uma ferramenta para outra, pode-se chegar a um número pequeno de variações.

Uma tendência crescente no desenvolvimento de *software* é a necessidade de desenvolvimento de muitos sistemas similares ao invés de um único produto individual. Há várias razões para isso. Produtos desenvolvidos para mercados internacionais devem ser adaptados às leis e costumes de cada país, além do suporte às linguagens adequadas, e adaptação de interfaces com o usuário. Devido às limitações de tempo e custo, não é possível um desenvolvimento a partir da estaca zero para cada cliente, o que aumenta a necessidade do reuso de *software* [STE04].

Uma linha de produtos de *software* é definida como um conjunto de produtos de *software* que compartilham uma série de funcionalidades em comum, satisfazendo as necessidades de um mercado em particular, porém contendo uma significante e previsível variabilidade [WEI99], [CLE02].

Linhas de produto de *software* (*Software Product Lines*, SPL) estão emergindo como um paradigma de desenvolvimento importante e viável, permitindo às companhias realizarem melhorias de ordem de magnitude em *time to market*[1], custo, produtividade, qualidade e outros aspectos de negócio. A engenharia de linha de produto também pode trazer uma rápida entrada no mercado e uma resposta flexível, provendo a possibilidade de customização em massa [SEI09a].

Várias companhias estão enxergando que a prática de construir conjuntos de sistemas relacionados a partir de ativos comuns pode, de fato, determinar melhorias quantitativas na qualidade do produto e na satisfação do cliente, suprindo de forma eficiente a demanda por customização em massa. Os benefícios tangíveis, ou seja, aqueles que podem ser mensurados, são

[1] Quantidade de tempo decorrido entre a concepção de um produto e sua disponibilidade para o mercado [ROS05].

a lucratividade, qualidade, tempo de integração e produtividade. Porém, os benefícios intangíveis (que não são facilmente medidos mas são percebidos pelos clientes) são uma menor demanda de profissionais, a aceitação da metodologia pelo time de desenvolvimento, a satisfação do time durante o desenvolvimento e a satisfação do cliente final com o produto [MAT04].

Os principais problemas, dificuldades e riscos encontrados durante a implantação de uma linha de produtos de *software* são destacados em [COH07]: os problemas que podem afetar a adoção desta abordagem são a falta de uma liderança comprometida, a falta de comprometimento da gerência, uma abordagem inapropriada (ou seja, esquecer que a linha de produto deve fazer parte da estratégia da organização), a falta de comprometimento do time, uma interação inadequada entre os times, padronização inapropriada, má adaptação, falta de desenvolvimento da abordagem (o ideal seria uma melhoria contínua) e falta de disseminação do conhecimento. De forma similar, os principais riscos identificados são a falta de clientes ou mesmo a resistência à mudança, a tendência que a organização tem em retornar à metodologia anterior, e/ou o pouco tempo destinado à implementação da linha de produtos.

A arquitetura de linha de produtos de *software* supre os requisitos necessários a um conjunto de ferramentas de testes baseados em modelos, já que estas ferramentas podem apresentar diversas funcionalidades em comum, como o *parsing* do modelo, geração de estados intermediários, geração de casos de teste, etc. Ao mesmo tempo, tais ferramentas apresentam pontos bem definidos de variabilidade, como o tipo de modelo a ser usado, e a estrutura dos *scripts* de teste a serem gerados.

As principais motivações para o uso de SPL são a redução dos custos de desenvolvimento, a melhoria da qualidade, a redução do *time-to-market*, entre outros [POH05]. As figuras 9 e 10 [POH05] retratam as diferenças em custo de desenvolvimento e *time-to-market*, quando comparamos o desenvolvimento de *software* da forma tradicional com o desenvolvimento fazendo uso de linhas de produto.

A Figura 9 ilustra a variação do custo de desenvolvimento em função do número de sistemas desenvolvidos, comparando uma abordagem tradicional de desenvolvimento com uma abordagem baseada no conceito de família de produtos. O custo inicial do uso do conceito de linha de produto é maior, mas a partir do terceiro produto desenvolvido, aproximadamente, o custo de desenvolvimento de sistemas que não são baseados em linhas de produto se torna significativamente maior, e a cada novo produto a diferença cresce ainda mais.

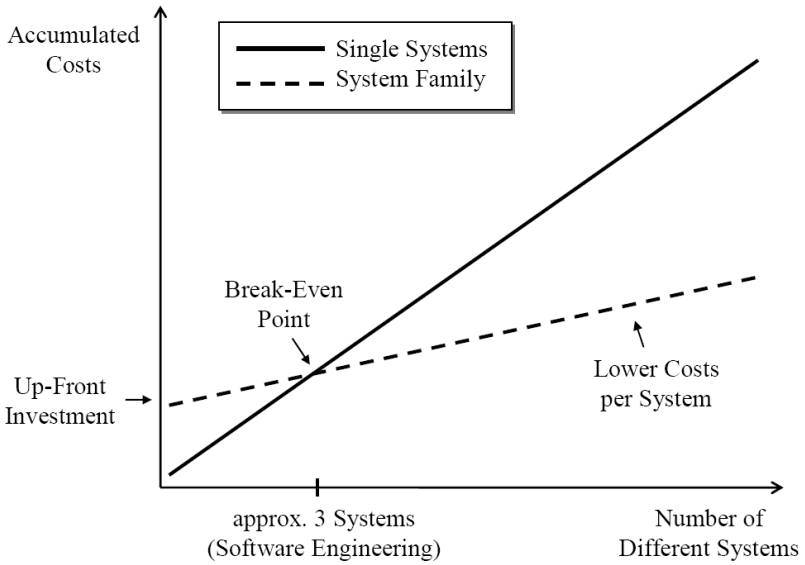

Figura 9 – Custo do uso de SPLs e da abordagem tradicional

Na Figura 10, a variação ilustrada é a do *time to market* em função do número de sistemas, quando comparados o uso ou não de linha de produto de *software*. O *time to market* do primeiro produto de uma linha de produtos é consideravelmente maior do que o de um produto concebido sem o conceito de linha de produtos. Com o aumento do número de produtos desenvolvidos e, consequentemente, com o amadurecimento da linha, é feito um maior aproveitamento do reuso de componentes, o que leva a um *time to market* por produto menor do que na abordagem tradicional.

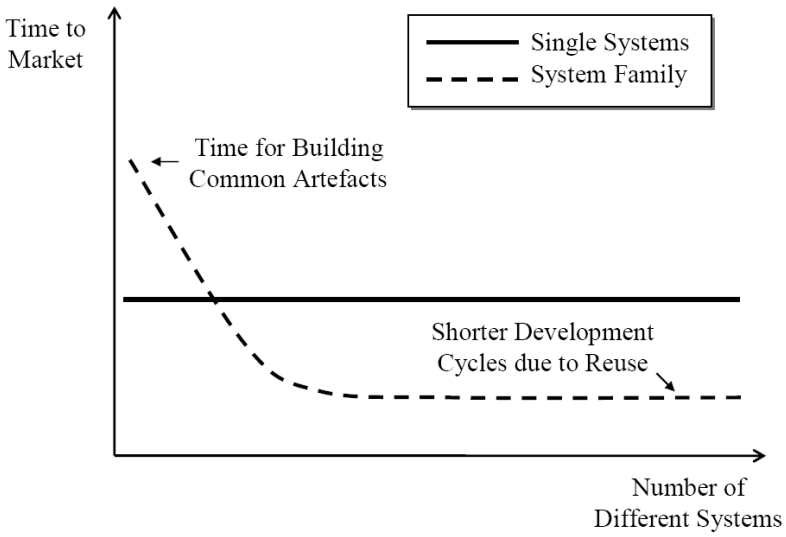

Figura 10 – *Time to market* de SPLs e da abordagem tradicional

As motivações da indústria para o uso de linhas de produto de *software* são: atingir ganhos de produtividade de grande escala, melhorar o *time to market*, manter a presença no mercado, sustentar o crescimento, alcançar uma maior agilidade de mercado, compensar uma limitação na capacidade de contratação, habilitar a customização em massa, ganhar controle de diversas configurações de produtos, melhorar a qualidade de seus produtos, aumentar a satisfação do consumidor e aumentar a previsibilidade de custo, calendário e qualidade dos produtos [SEI09a].

No mercado competitivo de hoje, onde é comum as empresas congelarem as contratações, as linhas de produto podem ajudar as organizações a superar os problemas causados pela escassez de recursos. Organizações de todos os tipos e tamanhos têm descoberto que uma estratégia de linha de produtos, quando corretamente implementada, pode trazer muitos benefícios, e mesmo dar às organizações um diferencial competitivo [SEI09a]. Em diferentes aplicações de uma linha de produto, os autores documentaram os seguintes benefícios: aumento de dez vezes na produtividade, aumento de dez vezes na qualidade, diminuição dos custos em 60%, diminuição da necessidade de mão-de-obra em 87%, diminuição do *time-to-market* em 98% e habilidade de conquistar novos mercados em meses, ao invés de anos.

Um trabalho relacionado apresenta o estudo da aplicação de linhas de produto de *software* em uma empresa de engenharia de tamanho pequeno-médio, e os resultados foram que a empresa reduziu seu tempo e esforço de desenvolvimento, e também reduziu o tempo e esforço gasto em melhorias e em manutenção. Mais especificamente, o tempo gasto para cada novo produto, após a aplicação da SPL, diminuiu em 26%, enquanto o esforço gasto em cada novo produto diminuiu cerca de 30% [SEL07].

4.1. Arquitetura Típica

Os dois conceitos básicos relacionados a SPLs são o uso de plataformas, que nada mais é do que uma coleção de artefatos reusáveis [MEY97], e a possibilidade de prover customização em massa. Para atingir tais conceitos, [POH05] discute a necessidade pelo planejamento visando o reuso em todas as etapas do desenvolvimento, além do fornecimento de customização em massa através da adição de variabilidade à arquitetura de linha de produto.

O paradigma da engenharia de linha de produto de *software* descreve dois processos [POH05], [BOE04], [WEI99], [BOE01], [LIN02]:

- Engenharia de domínio: é o processo responsável pelo estabelecimento da plataforma reusável e, portanto, pela definição dos pontos em comum e dos pontos de variabilidade da linha de produto. A plataforma aqui citada consiste de todos os tipos de artefatos de *software* (requisitos, projeto, testes, etc.).
- Engenharia de aplicação: este é o processo responsável por derivar aplicações da linha de produto a partir da plataforma estabelecida na engenharia de domínio. Explora a variabilidade da linha de produto e garante o correto uso da variabilidade de acordo com as necessidades específicas da aplicação.

A engenharia de domínio pode, também, ser definida como um método que cria uma arquitetura generalizando uma plataforma de produtos de uma organização. Os vários produtos pertencentes à mesma linha podem ser derivados da engenharia de domínio, criando a oportunidade de reusar e diferenciar produtos em uma mesma família.

A Figura 11 [SEL07] ilustra a divisão conceitual entre a engenharia de domínio e a engenharia de aplicação. Na parte superior da figura, que ilustra a engenharia de domínio, está representada a tradução do conhecimento do domínio em uma estrutura reusável, que servirá como base para todos os produtos da linha. Na parte inferior, correspondente à engenharia de aplicação, os requisitos do usuário são traduzidos em produtos da linha, por meio de reuso de componentes projetados na engenharia de domínio.

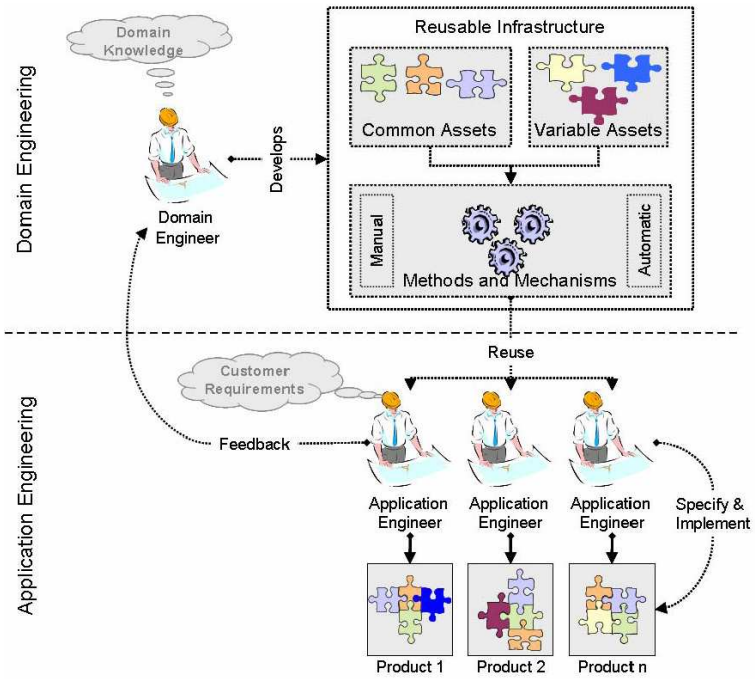

Figura 11 – Divisão conceitual entre engenharia de domínio e engenharia de aplicação

A vantagem desta divisão em engenharias de domínio e de aplicação é uma melhor separação conceitual, a fim de construir uma plataforma robusta, bem como aplicações específicas, em um curto espaço de tempo.

A separação em dois processos também indica uma separação de conceitos com relação à variabilidade. A engenharia de domínio é responsável por garantir que o grau de variabilidade disponibilizado é apropriado para a construção de aplicações, o que envolve mecanismos comuns para a derivação de uma aplicação específica. A plataforma é definida com a quantidade certa de flexibilidade em muitos artefatos reusáveis. Uma grande parte da engenharia de aplicação consiste em reusar a plataforma e fazer uso da variabilidade conforme requisitado pelas diversas aplicações da linha de produtos [POH05].

A Tabela 2 [SEI09b] enumera os principais custos (ou exigências) que devem ser levados em consideração durante o projeto e implementação de uma linha de produtos de *software*.

Tabela 2 – Principais custos na implementação de SPLs

Componentes fundamentais em SPL	**Exigência(s)**
Arquitetura	Deve suportar variações nativamente na linha de produtos.
Componentes de *software*	Devem ser projetados para serem genéricos sem perda de desempenho; devem ser construídos para suportar pontos de variação.
Planos, casos e dados de teste	Devem considerar pontos de variação e múltiplas instâncias da linha de produto.
Análises de mercado e de negócio	Devem endereçar uma família de produtos de *software*, e não apenas um produto.
Planos de projeto	Devem ser genéricos ou então precisarão ser escritos de forma extensível, a fim de prever variações de produtos.
Ferramentas e processos	Devem ser mais robustos.
Habilidades e treinamentos	Devem envolver treinamento e formalização do conhecimento sobre os ativos e procedimentos associados à linha de produto.

Nas atuais aplicações da metodologia de SPL, o alto nível de abstração obtido pela análise de domínio fica restrito à simplificação e representação dos requisitos da linha de produto, ou como um guia um tanto vago para o projeto do sistema.

Em se tratando de engenharia de linha de produto de *software*, existem diversas abordagens focando na modelagem das variabilidades de sistemas computacionais em diferentes níveis de abstração. Uma das abordagens utilizadas é o uso de arquiteturas baseadas em *plug-ins*, que possibilita aos desenvolvedores construir aplicações naturalmente extensíveis e personalizáveis às necessidades de usuários individuais. Uma aplicação com uma pequena porção

central é estendida com funcionalidades implementadas na forma de componentes que são adicionados sem muito impacto em tempo de execução [WOL08].

A insuficiência das metodologias de linha de produto é abordada em [SOC04], que tenta esclarecê-la, no que diz respeito a tentar preencher a lacuna existente entre os modelos de funcionalidades e a arquitetura. A metodologia proposta pelo trabalho envolve a adoção de uma estrutura arquitetural modularizada e baseada em *plug-ins*.

Engenharia de linha de produto e técnicas de *plug-ins* visam objetivos diferentes, porém complementares. SPLs pregam a modelagem da variabilidade de *software* em diversos níveis de abstração, enquanto sistemas baseados em *plug-ins* dão suporte à extensibilidade, personalização e evolução do *software*. A integração de ambas as áreas é benéfica, e [WOL08] apresenta a integração de uma plataforma de *plug-ins* com uma linha de produtos de *software* já existente, de forma satisfatória.

A arquitetura proposta no Capítulo 5 se baseia nos conceitos das arquiteturas tipicamente voltadas ao uso de *plug-ins* para modelar a variabilidade necessária às linhas de produto de *software*. A abordagem escolhida é uma forma de se obter o máximo de reuso, através dos conceitos de SPL, facilitando também a obtenção de extensibilidade, devido à implementação de uma arquitetura baseada em *plug-ins*. Dessa forma, fica preenchida a lacuna entre a modelagem conceitual e a realização da arquitetura [SOC04].

Dessa forma, os pontos de variabilidade expostos no modelo de funcionalidades são conceitualmente análogos aos pontos de extensibilidade de uma arquitetura baseada em *plug-ins*. Enquanto os pontos de variabilidade focam em possibilitar variações de implementação para um dado modelo conceitual, arquiteturas de *plug-ins* apresentam a possibilidade de estender uma arquitetura pela adição de novas funcionalidades.

4.2. Considerações

A estratégia de linha de produto se baseia na ideia do reuso de vários artefatos, durante grande parte do ciclo de desenvolvimento de *software*. Seu principal objetivo e sua motivação são a obtenção de diminuições drásticas no tempo e no esforço gastos no projeto, desenvolvimento, teste, documentação, planejamento e outras áreas do processo de desenvolvimento de *software*.

O retorno financeiro para uma organização que adote uma linha de produto de *software* começa a aparecer geralmente a partir do terceiro produto de uma mesma linha, desde que a arquitetura seja corretamente planejada.

Há uma lacuna que normalmente não é preenchida entre a modelagem conceitual, considerando uma abordagem de linha de produto, e a realização da arquitetura. Alguns autores propõem o uso de uma arquitetura baseada em *plug-ins* para garantir a extensibilidade e customização dos produtos de uma SPL.

No próximo capítulo, será apresentada e detalhada uma arquitetura para a geração de testes baseados em modelos, conceitualmente baseada na metodologia de linha de produto e fazendo uso de conceitos de uma arquitetura baseada em *plug-ins*. São descritos, também, exemplos de produtos da linha em questão, implementados na forma de *plug-ins* para a ferramenta apresentada.

5. FERRAMENTA PleTs

Para obtermos resultados práticos sem o desperdício de recursos, lidar com teste de *software* requer algum nível de automação, assim como o uso de técnicas que mantenham os aspectos formais e o rigor. Reaproveitamento de trabalho é, também, altamente desejável. Nesse contexto, as técnicas de testes baseados em modelos e linha de produtos de software podem ser aliadas, a fim de obter resultados tanto no meio acadêmico quanto na indústria.

A fim de prover uma ferramenta que auxilie na geração, execução e coleta de resultados de casos de teste de *software*, este trabalho apresenta a arquitetura e a implementação de uma ferramenta que busca automatizar etapas do processo de teste. A ferramenta, denominada PLeTs Tool (t*est automation using Product Lines and model-based Tests*), ajuda no processo de execução de testes baseados em modelos, e sua concepção é baseada em técnicas de linhas de produto de *software*. O objetivo da ferramenta é não apenas dar o suporte apropriado, mas também facilitar a tarefa de geração e execução de testes de *software* baseados em um modelo da aplicação, que normalmente são tarefas manuais. A arquitetura foi desenvolvida com o intuito de ser usada por engenheiros de desenvolvimento e teste de *software*, assistindo no processo de definição e criação de casos e *scripts* de teste. A ferramenta foi desenvolvida sobre o *Framework* .NET, que é considerado um dos padrões da indústria de desenvolvimento de *software* [ECM09].

5.1. Modelagem Conceitual

O primeiro passo na concepção da ferramenta foi a modelagem da engenharia de domínio, que permite definir e visualizar o escopo da ferramenta, conforme a Figura 12. Neste modelo, foram incorporadas as funcionalidades básicas necessárias a uma ferramenta de testes baseados em modelos. Essas funcionalidades foram definidas com base nas entradas e saídas desejáveis para tal tipo de ferramenta (e o conjunto tem uma interseção parcial com a lista de tarefas fundamentais em MBT, apresentadas na Seção 3.2 [ELF01a]), ou seja, englobam a modelagem e os artefatos resultantes de uma ou mais possíveis execuções da ferramenta.

Figura 12 – Funcionalidades determinadas na engenharia de domínio

Uma das formas de modelagem de linhas de produtos é a FODA (*Feature-Oriented Domain Analysis*) [KAN90]. O modelo de funcionalidades (ou *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.

O modelo de funcionalidades foi dividido em dois níveis, ilustrados pela Figura 13: no primeiro nível estão localizadas as funcionalidades da linha de produto de testes baseados em modelos, enquanto no segundo nível estão localizadas as funcionalidades que estendem cada funcionalidade do primeiro nível. Este modelo de funcionalidades muda com a adição de uma nova funcionalidade (tanto no primeiro quanto no segundo nível), e isso é saudável para a evolução da SPL.

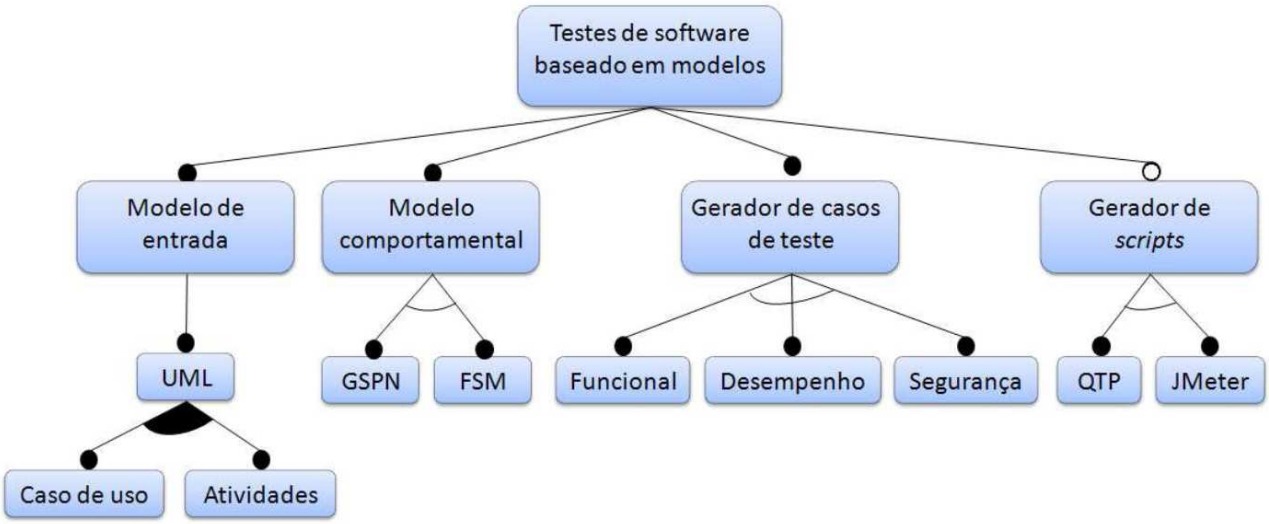

Figura 13 – Modelo de *features* (funcionalidades) da linha de produto

No segundo passo, cada funcionalidade básica do primeiro nível do modelo de funcionalidades é convertida em um passo de um *workflow* de atividades, a ser usado na ferramenta de testes baseados em modelos. Com esse procedimento, é possível converter qualquer atividade de teste em um passo de uma sequência de atividades. As funcionalidades

presentes no segundo nível podem ser vistas como pontos de variabilidade, e são convertidas em componentes de *software* durante o desenvolvimento de *plug-ins* para a ferramenta.

Em seguida, os conceitos do modelo de funcionalidades são aplicados no processo de desenvolvimento da ferramenta. A Seção 5.2 detalha a arquitetura e a construção da ferramenta e de alguns *plug-ins*.

Outros métodos para modelar funcionalidades incluem o método FORM (*Feature Oriented Reuse Method*) [KAN98], o método FeatuRSEB (*Feature Reuse-Driven Software Engineering Business*) [GRI98], e a notação de Jan Bosch [BOS01], sendo que todos estes métodos são baseados principalmente no método FODA.

5.2. Arquitetura e Implementação

Além de ser baseada nos conceitos da engenharia de linha de produto, a ferramenta segue uma arquitetura de *plug-ins*, que é a responsável por prover extensibilidade. Um *plug-in*, no escopo deste trabalho, é uma classe que herda características de um *plug-in* abstrato base (`BasePlugIn`), previamente definido. Em resumo, o *plug-in* base define a inicialização e os parâmetros iniciais necessários para a execução do *plug-in*. Adicionalmente aos membros herdados, cada *plug-in* implementará uma série de métodos estáticos e públicos, correspondentes às funcionalidades disponibilizadas por cada *plug-in*. A interface destes métodos pode variar de acordo com o ponto de vista do desenvolvedor e com as necessidades do negócio. Outro conceito importante na arquitetura proposta é o uso de *workflows*. É possível converter qualquer atividade de teste em um modelo de *workflow* [EST06] (na verdade, é possível e verdadeiro pensar em qualquer sequência de passos sendo modelada como um *workflow*). Levando isso em consideração, pode-se modelar uma sequência de geração e execução de testes baseados em modelos na forma de um *workflow*, e este *workflow* é uma das entradas da ferramenta proposta por este trabalho.

A Figura 14 apresenta um exemplo de *workflow* no formato requerido pela ferramenta, ilustrando os conceitos apresentados. O nodo raiz do trecho descrito em linguagem XML representa o *workflow*, contendo o seu nome, que será exibido pela ferramenta. A primeira geração de nós filhos do *workflow* representa o conjunto de métodos (ou atividades) do *workflow*, cada um descrevendo seu nome, tipo de retorno e os parâmetros que necessitam para executar. No exemplo da Figura 14, temos dois métodos, `ShowOneMessageMethod` e `ShowTwoMessagesMethod`.

```
<Workflow name=”Sample Workflow”>
  <Method name=”ShowOneMessageMethod” returnType=”string”>
    <Param name=”Message” type=”string” origin=”Initialize” position=”0”/>
  </Method>
  <Method name=”ShowTwoMessagesMethod” returnType=”void”
    dependsOn=”ShowOneMessageMethod”>
    <Param name=”Message1” type=”string” origin=”Initialize” position=”1”/>
    <Param name=”Message2” type=”string” origin=”ShowOneMessageMethod”/>
  </Method>
</Workflow>
```

Figura 14 – Exemplo de *workflow*

A arquitetura de alto nível da ferramenta pode ser dividida em duas partes principais: núcleo e *plug-ins*. O núcleo é a seção responsável pela carga do *workflow*, descrito em um arquivo XML (*eXtensible Markup Language*). Um *workflow*, conforme ilustrado na Figura 14, pode conter diversas atividades, cada qual esperando receber parâmetros e retornando algum tipo de valor. Desta forma, o arquivo XML precisa descrever, para cada atividade, quais são os parâmetros esperados (juntamente com seus tipos), dos quais a atividade depende para poder executar. Também precisa conter o tipo de retorno da atividade (por exemplo, considerando a sintaxe da linguagem C# (do *framework* .NET), `int`, `string` ou `void`). Outra responsabilidade do núcleo é a carga dos *plug-ins* disponibilizados, e sua disposição para que o usuário possa escolher o que melhor se aplica à atividade sendo executada. Os *plug-ins* são bibliotecas escritas sobre o framework .NET, e as classes nelas contidas são carregadas pela ferramenta usando reflexão (*Reflection*, descrita no *namespace* `System.Reflection` do *framework* .NET [MSD09]).

Portanto, cada *plug-in*, sendo uma classe, terá um número de métodos, e cada método público e estático teoricamente corresponde a uma atividade de um *workflow*, já que as atividades têm estruturas similares (um certo número de parâmetros e um tipo de retorno). Isso possibilita mais uma funcionalidade ao núcleo da ferramenta: a associação entre atividades de um *workflow* e métodos de um *plug-in*, através de uma interface de usuário baseada no conceito de arrastar e soltar (*drag-and-drop*), desde que suas assinaturas sejam compatíveis.

A Figura 15 ilustra de maneira genérica a arquitetura da ferramenta, permitindo várias possibilidades de associação entre os elementos da modelagem conceitual e os *plug-ins* implementados. O bloco em cinza escuro representa um *workflow* carregado pela ferramenta, contendo diversos pontos de extensibilidade. Cada uma das caixas ao redor do bloco em cinza é um *plug-in*. Conforme ilustrado, dependendo da situação, um dentre vários *plug-ins* pode ser o escolhido para desempenhar determinada função. Cada camada horizontal da figura corresponde

a uma das atividades do *workflow*, e essas camadas estão relacionadas às funcionalidades descritas na Figura 12.

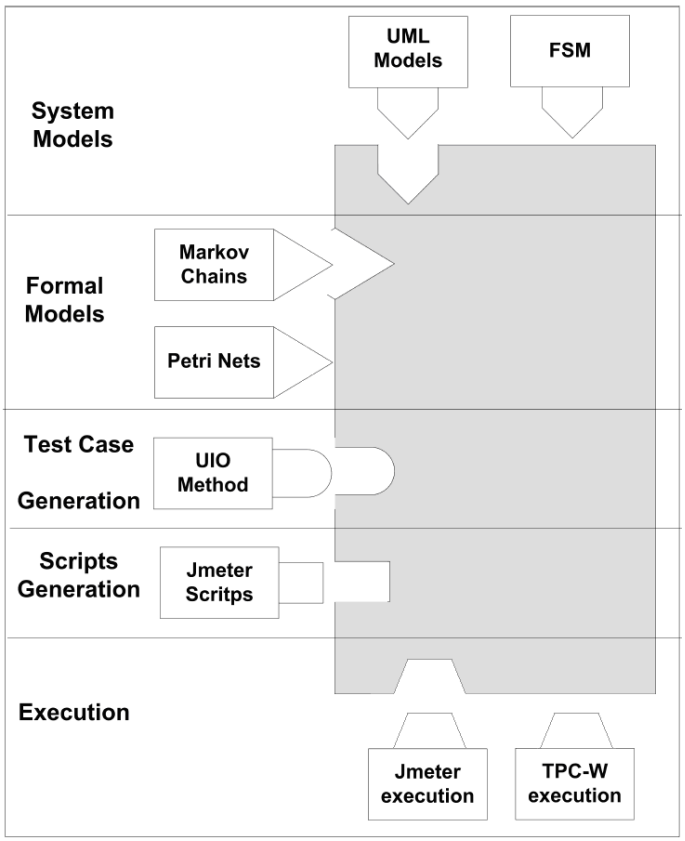

Figura 15 – Arquitetura da ferramenta e exemplos de associação

Quando o usuário clica e arrasta um método de um dos *plug-ins* carregados, apenas as atividades compatíveis (dependentes dos mesmos parâmetros, e com o mesmo tipo de retorno) com este método permanecem habilitadas no *workflow*. Dessa forma, o usuário pode "soltar" o método dentro de uma das atividades, criando uma associação entre a atividade e o método. Uma vez que todas as atividades do *workflow* carregado estejam associadas a métodos, a ferramenta permite que o usuário execute o *workflow*. Também é possível associar os métodos às atividades através de botões presentes na representação gráfica das atividades. A Figura 16 mostra uma tela obtida da ferramenta. No lado esquerdo, está uma representação gráfica do *workflow* carregado, que no caso é o de testes de segurança baseados em modelos UML. No lado direito, encontram-se os *plug-ins* carregados, com os métodos que os mesmos disponibilizam. Todas as atividades do *worfklow*, exceto pela última (que é a responsável pela geração dos casos de teste) estão associadas a métodos de algum *plug-in*.

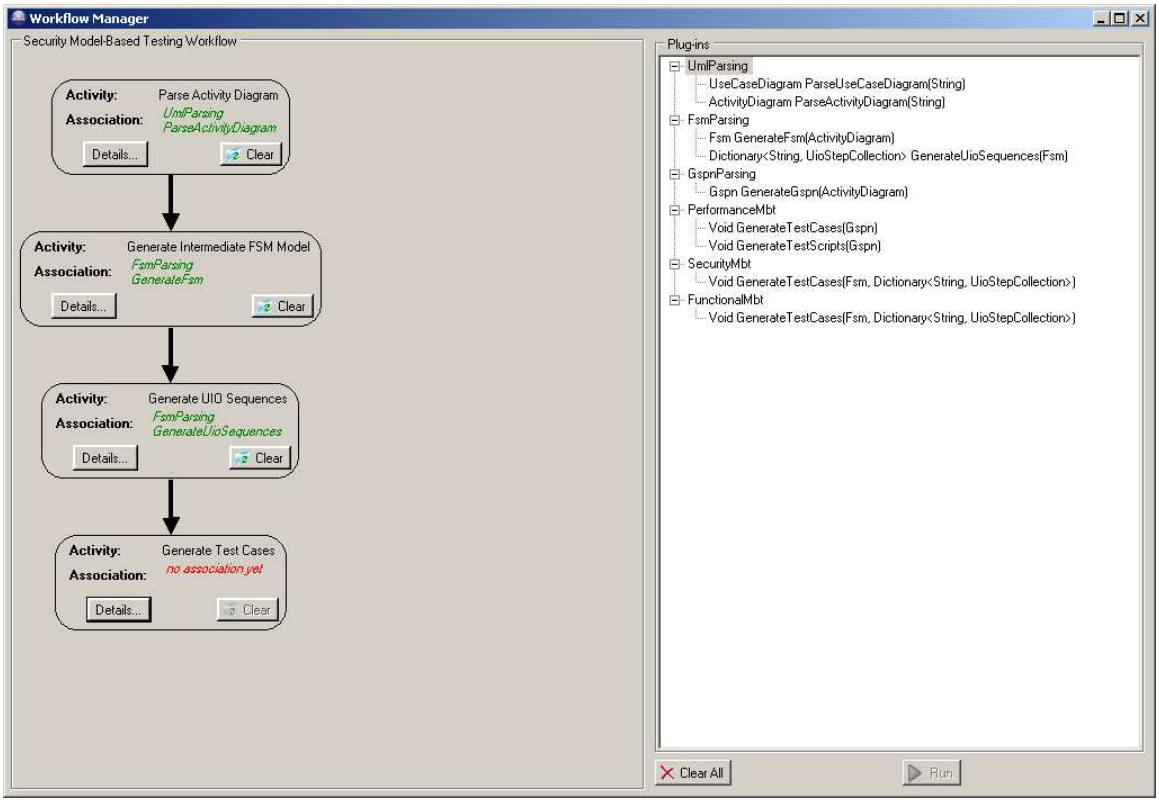

Figura 16 – Tela da ferramenta desenvolvida

Supondo que tenha-se um *plug-in* responsável pelo tratamento de teste de desempenho baseado em modelos. Este *plug-in* terá, necessariamente, métodos que lidem com a leitura e *parsing* do modelo UML e geração de casos de teste, por exemplo. Se for necessário incluir um novo *plug-in* na ferramenta, a fim de suportar testes de segurança baseados em modelos, o ideal é que não seja necessário implementar os mesmos métodos de leitura e *parsing*, que teoricamente deveriam semanticamente iguais aos do *plug-in* anterior. Na prática, a ferramenta permite que um conjunto de associações *workflow*/*plug-ins* contenha um *workflow* e métodos de diversos *plug-ins*, garantindo o reuso a nível de componente. Isto é visível na Figura 16, onde as atividades de um *workflow* estão associadas a mais de um *plug-in* (duas ao *plug-in* `FsmParsing` e uma ao *plug-in* `UmlParsing`). Como o usuário pode adicionar o número que quiser de *plug-ins*, a extensibilidade também é obtida.

A Figura 17 ilustra o diagrama de classes do núcleo da ferramenta, composto pelas classes `Program` e `WorkflowManagerForm`. A classe `TypeAliasResolver` é a responsável pela resolução de tipos declarados no arquivo XML de descrição do *workflow*, enquanto a classe `AssignedMethod` representa uma implementação concreta de um método, associada a uma atividade do *workflow*. A classe `ActivityComparer` faz o trabalho de comparação entre as assinaturas de um método e uma atividade, para permitir somente a associação entre instâncias compatíveis (ou seja, que tenham a mesma assinatura). O pacote `Controls` contém a

implementação de controles visuais utilizados pela ferramenta, e o pacote `Workflow` contém classes geradas automaticamente para realizar o *parsing* dos arquivos XML de descrição do *workflow*. O pacote `PlugIns` contém os *plug-ins* implementados durante este trabalho.

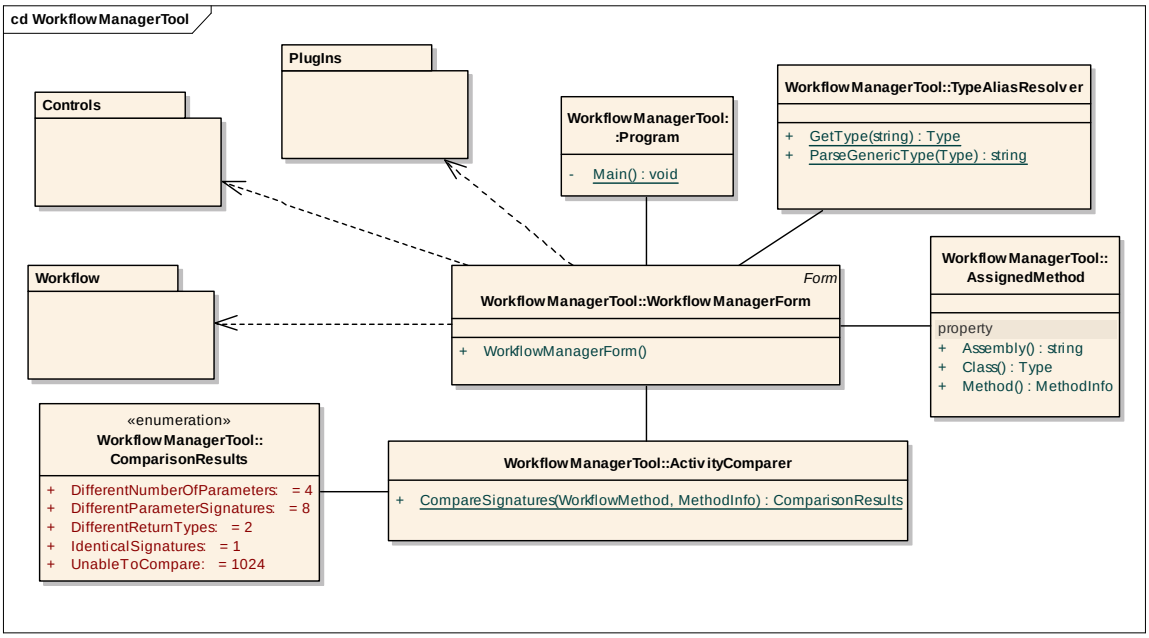

Figura 17 – Diagrama de classes do núcleo da ferramenta

A Figura 18 contém os tipos de modelos usados na implementação dos *plug-ins* desenvolvidos durante este trabalho. Nela, estão presentes os *plug-ins* de testes funcionais, de segurança e de desempenho (com suas classes associadas para a geração dos casos de teste), além da classe base `BasePlugIn`, e dos *plug-ins* referentes ao *parsing* e construção dos modelos UML, FSM e GSPN. Por motivos de clareza, os pacotes dos modelos foram detalhados em uma figura à parte, a Figura 19. Esta figura contém as classes básicas usadas para a modelagem dos três tipos de modelos. Por motivo de clareza, algumas classes, como as classes que representam estereótipos e marcações (*tags*) UML foram omitidas.

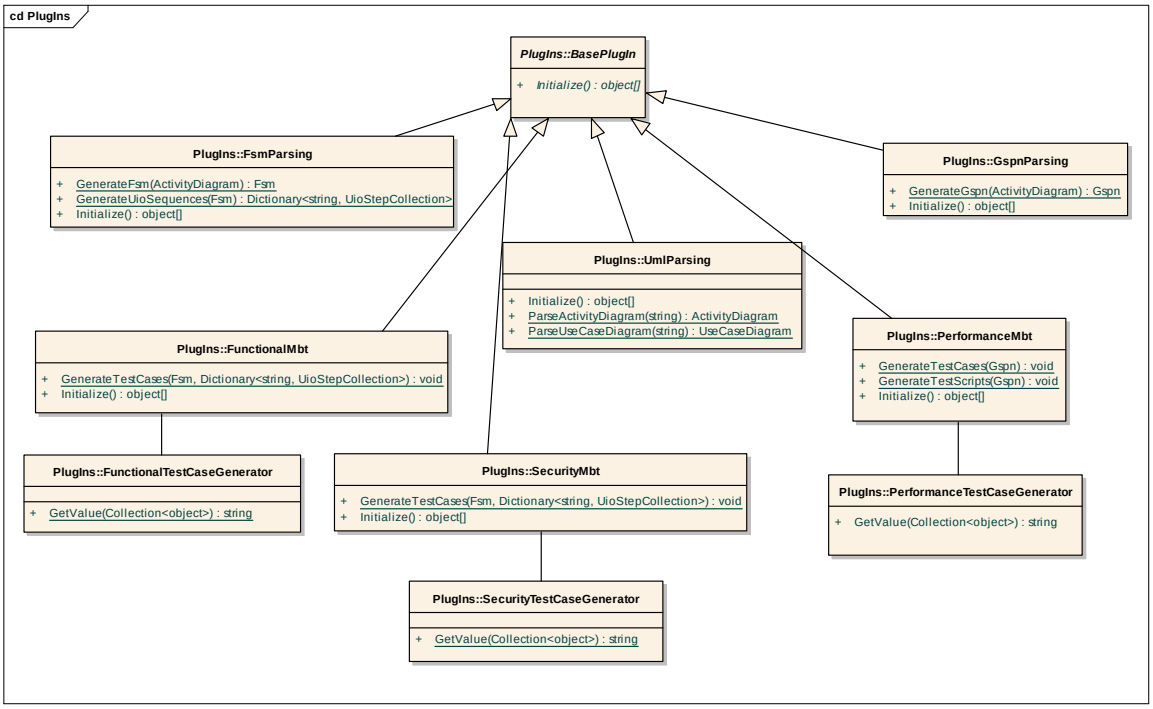

Figura 18 – Tipos de modelos usados nos *plug-ins*

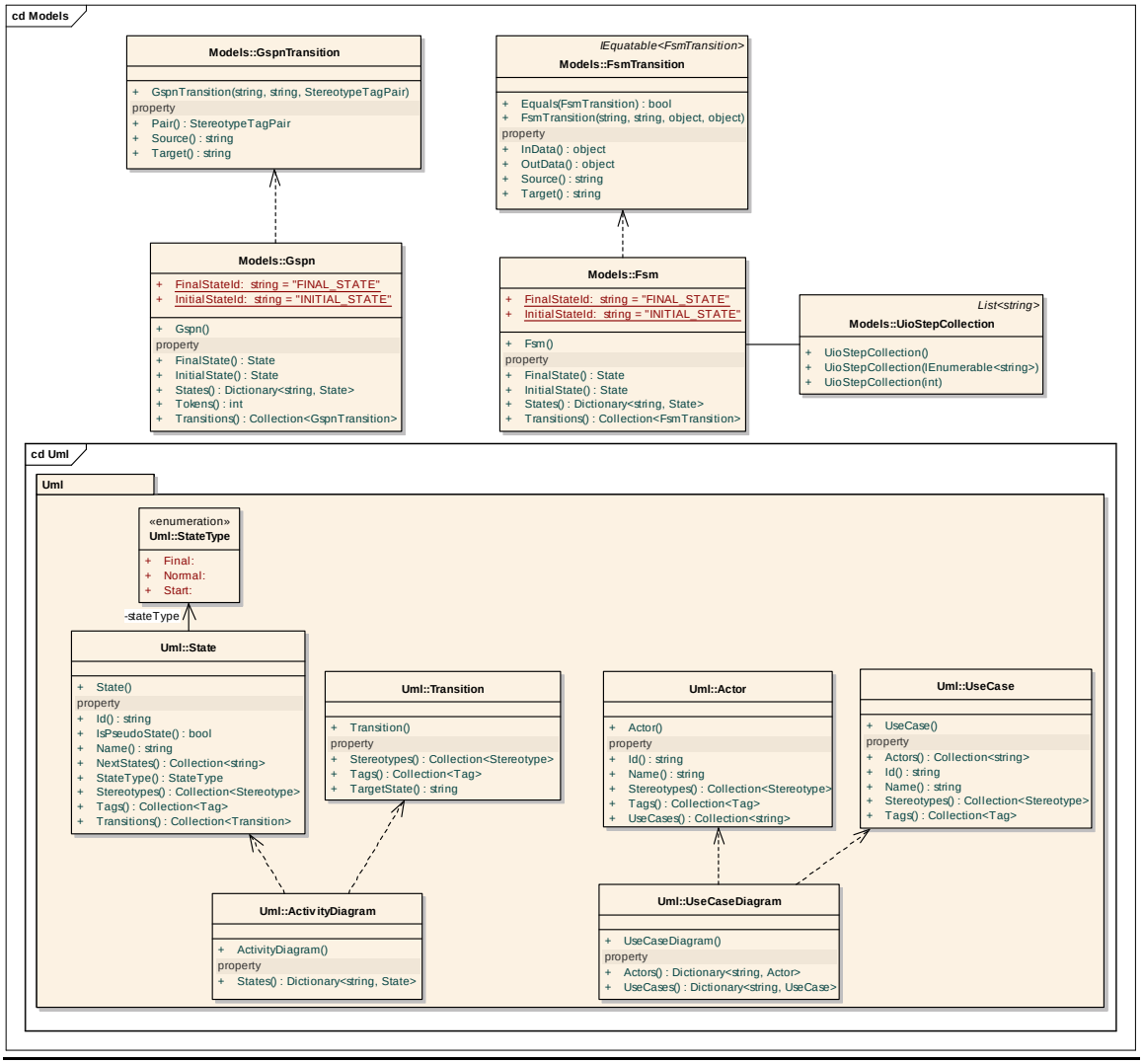

Figura 19 – Diagramas de classes representando os tipos de modelos usados

Um dos problemas detectados durante o desenvolvimento da ferramenta foi em relação à passagem de parâmetros entre uma atividade e outra. Como a ferramenta saberia qual a origem dos parâmetros necessários por cada atividade? Este problema foi resolvido com a adição de um atributo extra (`origin`) a cada declaração de parâmetro no arquivo XML de descrição do *workflow*. Esta solução pode ser vista na Figura 14. Se o atributo `origin` tiver valor igual a `Initialize`, significa que um método chamado `Initialize`, presente em todos os *plug-ins*, proverá os dados necessários à atividade. O *plug-in* base declara um método abstrato chamado `Initialize` (cuja assinatura é `public abstract object[] Initialize();`), obrigando todos os *plug-ins* derivados a implementar este método.

Originalmente, a ferramenta foi pensada para lidar apenas com *workflows* lineares, mas esta estratégia mostrou-se insuficiente. Então, um novo mecanismo para gerenciar *workflows* na forma de grafos foi adicionado à ferramenta. Isto significa que os *workflows* podem conter desvios e junções, que são representados na descrição do *workflow* da seguinte forma: se uma atividade B depende de uma atividade A, B terá um atributo `dependsOn` indicando que a mesma depende da atividade A e, portanto, deve ser executada depois de A. Se mais de uma atividade depender de A, temos um desvio após a execução de A, podendo gerar execuções em paralelo. Se uma nova atividade, C, depender de A e B, temos uma junção antes da atividade C. As Figuras 20 e 21 representam, respectivamente, o arquivo XML e o fluxo visual do exemplo descrito.

```
<Workflow name=”Sample Fork/Join Workflow”>
 <Method name=”A” returnType=”string”>
  <Param name=”SampleParam1” type=”string” origin=”Initialize” position=”0”/>
 </Method>
 <Method name=”B” returnType=”void” dependsOn=”A”>
  <Param name=”SampleParam2” type=”string” origin=”Initialize” position=”1”/>
  <Param name=”SampleParam3” type=”string” origin=”A”/>
 </Method>
 <Method name=”C” returnType=”void” dependsOn=”A,B”>
  <Param name=”SampleParam4” type=”string” origin=”A”/>
  <Param name=”SampleParam5” type=”string” origin=”B”/>
 </Method>
</Workflow>
```

Figura 20 – Arquivo XML do exemplo descrito

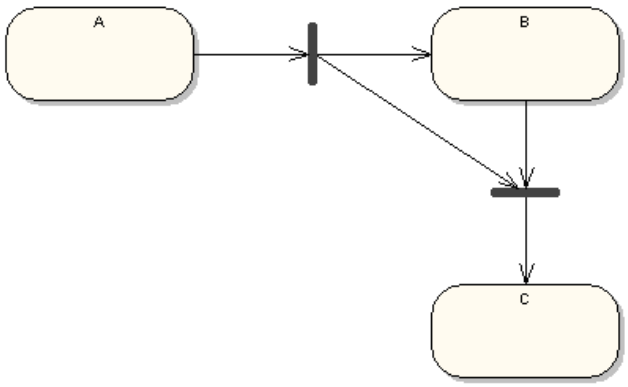

Figura 21 – Fluxo visual do exemplo descrito

No sentido de exemplificar a expansão da ferramenta, para que suporte outros aspectos de teste, as próximas seções mostram como é possível conectar outros dois trabalhos a este, tirando proveito da habilidade da ferramenta em gerenciar *workflows* e do suporte a *plug-ins* de testes baseados em modelos. Os três aspectos de teste já suportados pela ferramenta são os relativos a testes de desempenho, testes funcionais e testes de segurança, todos baseados em modelos UML.

A arquitetura proposta para a ferramenta é genérica o suficiente para ser usada em várias aplicações. O foco deste trabalho é a geração e implementação de testes baseados em modelos usando o conceito de linha de produto de *software*. Os *plug-ins* sugeridos representam bons casos de uso, já que apresentam vários dos mesmos conceitos.

A possibilidade de associação entre as funcionalidades de um ou mais *plug-ins* é um item desejável do ponto de vista de reuso e composição, e foi alcançada pelo uso da ferramenta. Devido à simplicidade do processo de desenvolvimento de um novo *plug-in*, sem ser necessária nenhuma modificação no núcleo da ferramenta, está garantida a extensibilidade da mesma.

A implementação concreta de um conjunto de técnicas de testes baseados em modelos (por exemplo, testes de segurança, desempenho e funcionais, conforme descrito nas próximas seções) pode ser feita de várias formas:

1. Implementando vários *plug-ins*, cada um suportando um *workflow*, sendo que cada *plug-in* implementa todas as atividades do *workflow* correspondente, do início a fim do processo de testes em questão.
2. Um único *plug-in*, capaz de gerenciar todos os *workflows*. Esta pode ser considerada uma implementação de testes baseados em modelos, no sentido mais amplo do conceito, capaz de executar vários tipos de MBT.

3. Vários *plug-ins*, cada um especializado em uma ou mais tarefas comuns a vários *workflows*. Por exemplo, um dos *plug-ins* seria especializado no *parsing* de diagramas UML, outro seria especializado na geração de casos de teste, e assim por diante.
4. Uma combinação dos três itens acima. Como a ferramenta suporta associações, de forma dinâmica, de *plug-ins* a *workflows*, é possível combinar mais de um dos tipos de implementação descritos acima em uma execução da ferramenta.

A estratégia adotada foi a descrita no item 3. Foram desenvolvidos *plug-ins* para:

- Lidar com o *parsing* de arquivos XMI contendo os modelos que representam o SUT.
- Criar e gerenciar o modelo intermediário (FSM ou GSPN), gerando as sequências UIO no caso das FSMs).
- Gerar os casos e/ou scripts de teste (um *plug-in* para cada tipo de teste).

5.3. Exemplos de Aplicação

As próximas três seções apresentam os três tipos de testes baseados em modelos abordados por este trabalho, discorrem sobre os processos de cada um dos tipos de teste, e apresentam o exemplo de aplicação abordado no trabalho, sob a forma de implementação dos três *plug-ins* correspondentes.

5.3.1. Teste de Desempenho Baseado em Modelos UML

O primeiro exemplo de aplicação está relacionado com uma proposta de realocação de recursos em ambientes virtualizados, para maximização do aproveitamento de recursos e do desempenho das aplicações [ROD08], [ROD09a], [ROD09b]. O trabalho tinha a necessidade de integração com uma ferramenta que possibilitasse avaliar o desempenho de aplicações. Este trabalho foi usado como base para o desenvolvimento da ferramenta para a geração de casos de teste de desempenho baseados em modelos UML da aplicação sob teste (SUT, *system under test*).

O primeiro passo do processo consiste em usar diagramas UML da aplicação, a fim de extrair informações relacionadas ao comportamento da aplicação sendo testada. Neste caso, a informação necessária será extraída de diagramas de caso de uso e diagramas de atividades. Porém, há algumas informações necessárias para a geração automática de casos de teste que não

são encontradas nos diagramas UML, e precisam ser inseridas manualmente nos modelos. Esta informação, relevante para o entendimento do comportamento do sistema, é inserida através do uso de estereótipos e *tags* UML.

Nos diagramas de caso de uso, dois tipos de informação são inseridos através dos estereótipos, `PApopulation`, que é o número de usuários que acessarão o sistema durante o teste, e `PAprob`, que é a probabilidade de execução de cada caso de uso (veja a Figura 22 [ROD09b]). A inclusão destas informações é necessária para informar quantos usuários terão de ser simulados em cada teste, e qual a probabilidade de um determinado caminho ser seguido por cada caso de uso. As informações serão usadas na construção de uma rede de Petri que representará os casos de uso em questão, e servirá como ponto de entrada para o algoritmo de geração de testes de desempenho. `PAprob` é especialmente importante no teste de aplicações *web*, já que alguns *links* são mais acessados que os outros, e este comportamento não pode ser ignorado durante testes de desempenho.

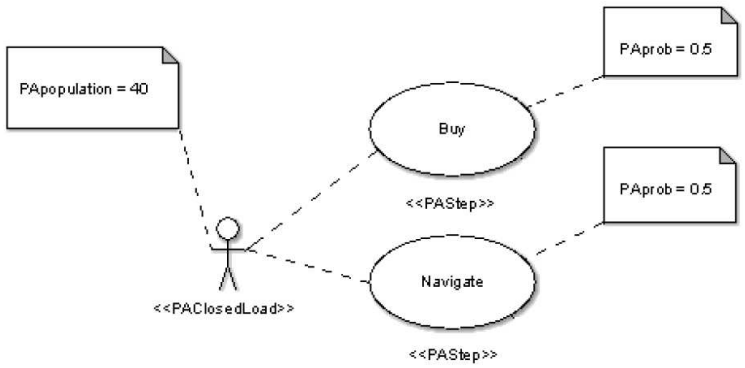

Figura 22 – Diagrama simples de casos de uso de um *e-commerce*

No caso do diagrama de atividades, os estereótipos e *tags* devem ser adicionados a todas as atividades pertencentes ao diagrama, porque a ferramenta precisa, no mínimo, simular o tempo que o usuário gasta em cada atividade (*think time*). Normalmente, mas não necessariamente, este valor é randômico.

A saída do processamento desta etapa do processo é composta por dois objetos, um representando o diagrama de atividades, e outro representando o diagrama de casos de uso.

Após o *parsing* dos diagramas de atividades e casos de uso, é gerado um modelo comportamental intermediário, na forma de uma rede de Petri estocástica generalizada (*generalized stochastic Petri net*, GSPN) [MAR84]. GSPNs são muito usadas para modelar o comportamento de sistemas no intuito de realizar testes ou coleta de resultados de desempenho,

ou modelagem de sistemas paralelos. Este tipo de abordagem é usado em vários outros trabalhos [OLI07], [LOP04], [BER02], [MAR95]. A tradução do diagrama de atividades em rede de Petri é feita tendo como base os passos descritos em [LOP04]. A rede de Petri resultante possuirá as marcações de desempenho em seus estados e transições, o que permitirá a geração de casos e *scripts* de teste.

A última etapa do processo é a geração dos casos de teste textuais, em português estruturado. Para isso, a ferramenta simula execuções da rede de Petri modelada no passo anterior, e a cada estereótipo de segurança encontrado, gera o caso de teste (ou passo do caso de teste) correspondente ao estereótipo. Também é possível gerar *scripts* de teste, de forma análoga, para a execução em ferramentas de teste de desempenho, como o Jmeter [HAL08] e o TPC-W [GAR03]. A Figura 23 dá uma visão geral do processo de geração de testes de desempenho, adaptado à arquitetura proposta.

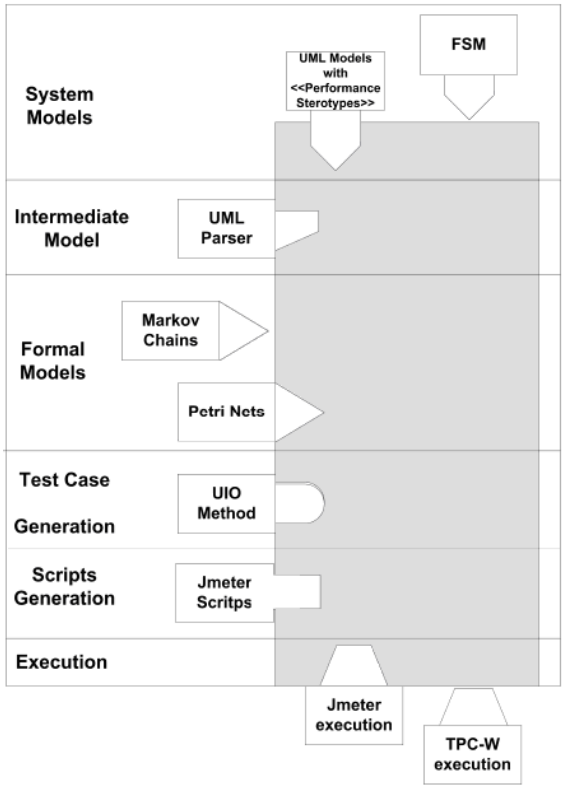

Figura 23 – Visão geral do processo de geração de testes de desempenho

5.3.2. Teste de Segurança Baseado em Modelos UML

Os trabalhos apresentados em [PER08], [PER09] introduzem, no contexto de segurança de sistemas, um conjunto de estereótipos UML para definir critérios de teste de *software*, com o

objetivo de encontrar possíveis brechas de segurança. A definição dos estereótipos foi feita com a ajuda de algumas taxonomias de segurança (especialmente a proposta em [WEB05]) e também com a informação disponibilizada por alguns repositórios de segurança, como a base de dados do projeto OWASP [OWA09]. O resultado foram seis estereótipos, conforme a lista abaixo:

- <<BufferOverflow>>: indica o número máximo de caracteres que um campo pode receber;
- <<Flooding>>: indica o número máximo de conexões simultâneas suportadas pela aplicação;
- <<Encrypt>>: indica o nome do campo que precisa (ou deveria) ser criptografado;
- <<ByPassing>>: indica papéis que podem acessar certas partes do sistema;
- <<Expiration>>: indica o tempo máximo para a expiração da sessão do usuário, caso ele não interaja com o sistema;
- <<SqlInjection>>: indica os campos que podem ser suscetíveis a ataques de injeção de SQL.

O uso destes estereótipos tem dois objetivos principais: guiar os desenvolvedores durante o projeto de *software*, a fim de evitar defeitos relacionados a segurança, e prover informação necessária para a geração de casos de teste, de forma automática.

Após o projeto dos diagramas de casos de uso e atividades, considerando a inclusão dos estereótipos próprios ao teste de segurança, as próximas etapas são similares às do *plug-in* anterior. Ou seja, a ferramenta faz o *parsing* do arquivo XMI (*XML Metadata Interchange*) contendo a descrição do modelo UML. Com esses dados em mãos, o modelo comportamental intermediário gerado é uma máquina de estados finitos (FSM), que é o modelo mais apropriado para a geração de testes [DOR05], [GON70], [HEN64], [HIE06], [SID88], [URA97], [BON08].

De posse da máquina de estados, a ferramenta aplica o método UIO (*Unique Input-Output*) [DEL07]. Uma sequência UIO para um determinado estado de um protocolo ou máquina é uma sequência de pares de entrada e saída que é única para este estado. Esta sequência UIO nada mais é do que uma sequência de passos a serem seguidos a fim de que atinja-se o estado para o qual se está calculando a sequência. Ou seja, após a aplicação do método, teremos sequências que sugerirão os passos a serem seguidos para atingir determinado estado.

Após a derivação das sequências UIO, o passo executado pela ferramenta é a geração dos casos de testes textuais, em português estruturado, com base nas sequências de passos e nos estereótipos de segurança que cada estado porventura apresenta. A Figura 24 ilustra, em alto

nível, o processo de geração de casos de teste de segurança, adaptado à arquitetura da ferramenta.

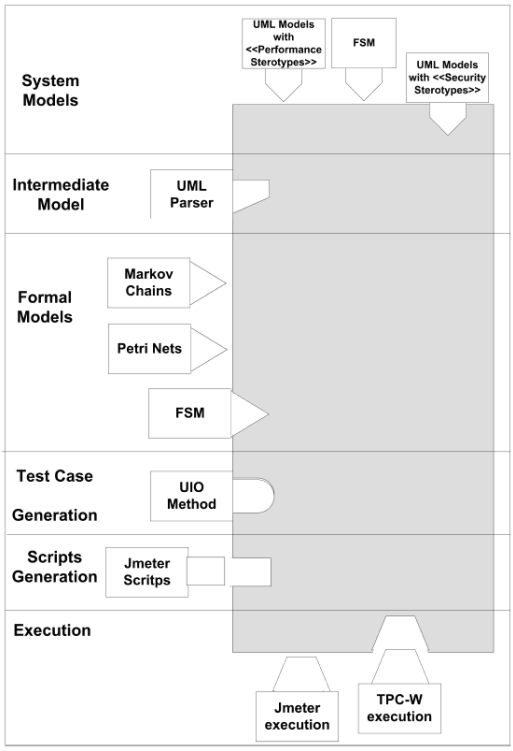

Figura 24 – Visão geral do processo de geração de testes de segurança

5.3.3. Teste Funcional Baseado em Modelos UML

Conforme descrito na Seção 2.3.3, a importância do teste funcional está em garantir a qualidade do funcionamento do sistema como um todo. Ou seja, de garantir que o sistema faz exatamente o que foi previsto e projetado em sua especificação, e garantir também que o sistema não faz algo além disso.

O terceiro exemplo de aplicação está relacionado com uma proposta para a geração de testes funcionais a partir de modelos UML, que foi seguida pelo presente trabalho, para a implementação de um *plug-in* responsável pelos testes funcionais, na ferramenta proposta [ORO09b].

A geração de testes funcionais se baseia no diagrama UML de atividades do SUT. O diagrama é percorrido, e transformado em uma máquina de estados finita, onde cada atividade do diagrama corresponde a um estado na máquina. Transições entre atividades também são preservadas, passando, na FSM, a ligar estados. A proposta apresenta o estereótipo <<FTStep>>, que tem o papel de indicar atividades que contenham as marcações indicativas de

ações ou resultados esperados para um teste funcional. Essas marcações são `FTAction` e `FTExpectedResult`, que indicam qual a ação que o usuário tem que executar, bem como o resultado esperado, respectivamente. O conteúdo das *tags* `FTAction` e `FTExpectedResult` serão convertidos, respectivamente, nas entradas e saídas dos estados da FSM.

Após a geração da FSM, o método UIO é aplicado, a fim de gerar as sequências de execução correspondentes a cada estado. Enfim, é executado o método do *plug-in* que lê as sequências UIO e realiza a geração de casos de teste em português estruturado. O trabalho relacionado a este exemplo de aplicação ainda descreve uma atividade complementar, que é a geração de *scripts* de teste para uma eventual execução automática, o que tornaria o *plug-in* mais completo [ORO09b].

Conforme apresentado, o processo de geração de testes funcionais é muito similar ao processo de geração de testes de segurança, diferindo apenas nos estereótipos e marcações usados, e nos passos finais do processo (após a geração da máquina de estados), que representam a derivação dos casos e/ou *scripts* de teste a partir das sequências UIO. Na prática, esta similaridade representou uma enorme economia de tempo de projeto e desenvolvimento do *plug-in*, que levou, para ser construído, cerca de 15% do tempo que o *plug-in* de segurança exigiu[2]. Essa é uma das vantagens obtidas com o uso de uma arquitetura focada no reuso e baseada nos conceitos de linha de produto e *plug-ins*. Outra vantagem aparente é que, no caso de necessidade de alteração no modo como é feito o *parsing* (devido, por exemplo, a alterações na especificação da UML, ou da estrutura do documento XMI exportado, as modificações devem ser realizadas em um só lugar, e passam a valer para todos os *plug-ins*.

5.4. Melhorias e Continuação

A continuação do trabalho prevê diversas melhorias na ferramenta proposta, em vários aspectos. Inicialmente, está planejada uma melhoria na forma de tratamento dos *workflows*, permitindo que diversos *workflows* sejam armazenados, em uma espécie de "favoritos". Outra ideia na área de gerenciamento de *workflows* é a possibilidade de edição gráfica de um *worfklow*, e posterior geração do arquivo XML correspondente, a fim de facilitar a sua criação e evitar erros oriundos da edição manual.

[2] O plug-in de testes de segurança consumiu cerca de 50h de desenvolvimento, enquanto que o plug-in de testes funcionais consumiu menos de 10h entre desenvolvimento e *refactoring*.

A descrição de *workflows* é, hoje, uma maneira *ad hoc* de descrever uma sequência de passos. Os planos de continuação e extensão da ferramenta incluem a mudança na forma de descrever e percorrer *workflows*, passando a usar alguma técnica mais difundida na indústria. A principal candidata é a *Business Process Modeling Notation* (BPMN), uma notação gráfica que descreve os passos de um processo de negócio, de ponta a ponta [WHI08]. A notação BPMN foi especificamente projetada para coordenar a sequência de processos e a troca de mensagens que ocorre entre diferentes participantes em um conjunto de atividades relacionadas. Alguns trabalhos apresentam técnicas de mapeamento entre BPMN e linguagens mais próximas da descrição da execução de um sistema, como BPEL (*Business Process Execution Language*), na tentativa de remover o *gap* conceitual entre esses dois tipos de linguagens [CHI09], [OUY06].

Outra melhoria planejada é a possibilidade de persistir, em disco, as associações entre *plug-ins* e *workflows* já realizadas pelo usuário, a fim de poupar trabalho em uma futura execução da ferramenta.

Ao invés de continuar com o modelo de *workflows* adotado pela ferramenta, outra opção seria o uso de alguma ferramenta comercial que já endereçe fluxos de execução nos mesmos moldes, como o Oracle Worfklow, por exemplo.

Uma funcionalidade que agregará bastante em termos de automação, e que também está nos planos futuros para a ferramenta, é a geração automática de um *stub* (ou esqueleto de código) em C# quando a ferramenta carregar um *workflow*, contendo a assinatura requerida por este *workflow*. Desta forma, basta ao usuário preencher a implementação de cada um dos métodos (e, se necessário, remover métodos que já estejam implementados em outros *plug-ins*) para ter um novo *plug-in* em conformidade com o *workflow*.

É perfeitamente possível a definição de um processo de testes que englobe desvios. Dessa forma, é importante que a ferramenta possa executar métodos de *plug-ins* de forma concorrente (ou seja, fazer uso de múltiplas *thread*s para realizar tarefas concorrentes), quando não houver dependência direta ou indireta entre essas tarefas. Hoje, todos os métodos são executados de forma sequencial.

No intuito de obter a colaboração e análise da comunidade acadêmica e da indústria, a ferramenta será disponibilizada em um repositório de livre acesso, a fim de que possa ser incrementada e melhorada.

Por fim, devido à extensibilidade da ferramenta, o projeto e construção de novos *plug-ins* é uma tarefa constante. A implementação de novos *plug-ins* faz parte da estratégia de continuação da ferramenta.

5.5. Trabalhos Relacionados

Testes baseados em modelos são abordados por inúmeros autores. A lista é extensa, e alguns trabalhos fazem uma análise comparativa de diversos trabalhos relacionados à área [NET07]. Outros analisam vários trabalhos alinhados com a geração automática de casos de teste [PRA05], ou as principais técnicas relacionadas à automação de testes [FAR08].

Um dos trabalhos analisa 78 artigos focados em MBT [NET07]. Destes, 47 se baseiam em modelos UML, enquanto 51 apresentam algum tipo de ferramenta para ajudar a automatizar as atividades de teste. O trabalho também cita a dificuldade em se fazer um levantamento ou uma análise comparativa entre essas ferramentas, pelo fato de existirem inúmeras delas em desenvolvimento, além de algumas serem ferramentas proprietárias.

A análise contida em [PRA05] vai mais a fundo em várias ferramentas propostas com o intuito de facilitar etapas em MBT. O trabalho analisou mais de dez ferramentas especializadas em geração de testes baseados em modelos.

Alan Hartman faz uma classificação de ferramentas para a geração de MBT, e as divide em ferramentas comerciais, ferramentas proprietárias e ferramentas acadêmicas, e apresenta uma análise de várias ferramentas dos três tipos [HAR02].

Analisando os trabalhos citados acima, nota-se que a grande maioria das ferramentas disponíveis para testes baseados em modelos possui alguma limitação. Apesar de existirem ferramentas bem definidas, e que cumprem integralmente seus requisitos, a maioria delas possui uma limitação de escopo de testes, sendo útil apenas para um conjunto finito e pré-estabelecido de tipos de testes, ou a limitação se encontra quanto ao tipo de modelo aceito. No segundo caso, apenas um conjunto também pré-estabelecido de tipos de modelos é aceito na ferramenta. O ideal é que a ferramenta seja genérica o suficiente para aceitar qualquer tipo de modelo e trabalhar com a geração de qualquer tipo de teste, mesmo que isso custe o desenvolvimento de um ou mais novos módulos de extensão.

O trabalho descrito em [ORO09a] apresenta um trabalho relacionado, no ambiente acadêmico, e com objetivos similares. O trabalho também está baseado nos conceitos de linha de produto de software usados em conjunto com técnicas de testes baseados em modelos. No entanto, o trabalho apresentou resultados apenas superficiais.

Além disso, a ferramenta proposta não é facilmente expansível, e está muito próxima de ser uma ferramenta construída apenas para solucionar os exemplos citados no trabalho, ao invés de ser uma ferramenta genérica e facilmente aplicável a outros tipos de teste. Embora o trabalho

descreva o reuso de componentes, também não ficou claro se esta propriedade é facilmente atingida.

É vital que uma ferramenta com este propósito seja facilmente extensível. Partes de componentes implementados para esse fim, ou mesmo componentes inteiros, devem também ser fáceis de reusar em casos similares de uso, reduzindo o esforço necessário para adicionar suporte a novos testes e/ou fluxos de execução.

5.6. Discussão

Conforme explicado nas primeiras seções do Capítulo 5, o principal objetivo da arquitetura em questão é permitir a automação de testes baseados em modelos, permitindo a extensibilidade e o reuso de componentes através de conceitos de engenharia de linha de produto de *software*.

Os *plug-ins* implementados nos exemplos de aplicação comprovaram que, em tal arquitetura, o reuso é facilmente obtido. Por exemplo, uma vez implementado o *plug-in* de testes de desempenho, foi necessária a implementação de mecanismos de *parsing* de diagramas UML, que representam em torno de 50% do código implementado para este tipo de teste. Durante a implementação do próximo *plug-in*, de testes de segurança, a parte de *parsing* UML pôde ser reaproveitada, caracterizando o reuso. Da mesma forma, durante a implementação do *plug-in* de testes funcionais, devido ao fato do modelo intermediário utilizado ser o mesmo (máquina de estados), foi possível reaproveitar em torno de 85% da implementação[3] do *plug-in* de testes de segurança, caracterizando de forma ainda mais clara o reuso.

A verificação da extensibilidade, por sua vez, é fácil de ser feita, já que uma arquitetura baseada no conceito de *plug-ins* permite uma fácil e rápida adição de funcionalidades à ferramenta.

Exemplificando o atingimento dos dois conceitos, suponhamos que seja interessante a criação de um novo *plug-in*, para executar testes de segurança. Suponhamos, também, que a única variação em relação ao *plug-in* de testes de segurança já existente seja o modelo usado, que agora passaria a ser uma gramática. Poderíamos aproveitar praticamente todo o *plug-in* já existente, sendo necessário, no novo *plug-in*, apenas um novo método para realizar o *parsing* da gramática. Obviamente, este método poderia ser aproveitado, no futuro, caso outro *plug-in*

[3] O *plug-in* de testes de segurança contém, no total, em torno de 750 linhas de código. Para obter um *plug-in* de testes funcionais, foi necessária a implementação de cerca de 100 novas linhas de código.

também viesse a trabalhar com gramáticas. Estes novos *plug-ins* poderiam ser adicionados e facilmente integrados à ferramenta.

Os pontos fortes da ferramenta implementada são a possibilidade de automatizar total ou parcialmente testes de *software*, tendo como base teoricamente qualquer tipo de modelagem, formal ou não, e sendo facilmente extensível.

Um dos pontos a melhorar é a realização de validação em um ambiente industrial, ou seja, automatizar a geração e execução de um conjunto de testes em uma aplicação comercial, em uso na indústria. Seria muito interessante a comparação de resultados (tempo e esforço gastos, número de defeitos encontrados, complexidade dos defeitos, etc.) entre uma abordagem manual e execuções da ferramenta.

Alguns dos problemas encontrados durante a concepção e construção da ferramenta são problemas já existentes em ambientes de desenvolvimento de *software*, como o controle do fluxo das atividades e a passagem dos parâmetros. Futuras versões da ferramenta poderiam incluir uma pesquisa e validação de outras formas de resolução destes problemas, já bem estabelecidas, de forma a aprimorar a arquitetura da ferramenta.

6. CONCLUSÃO

No quadro em que se encontra o processo de desenvolvimento de *software*, focado cada vez mais na qualidade, as metodologias de teste de *software* têm recebido cada vez mais atenção. Diversos pesquisadores e inúmeras empresas estão propondo abordagens baseadas nas técnicas de testes baseados em modelos, a fim de uniformizar processos, garantir uma maior formalidade, melhorar a etapa de testes, difundir mais facilmente o conhecimento e obter resultados mais previsíveis nas etapas de desenvolvimento e teste de sistemas. Essa abordagem, conhecida como testes baseados em modelos (MBT), provê todos os requisitos para a automação de grande parte da, e por vezes de toda, a etapa de testes de *software*.

Há muitas variáveis na abordagem de MBT, como o tipo de teste (por exemplo, teste funcional, teste de desempenho, teste de segurança, etc.), a etapa a ser automatizada (a geração de casos de teste, a geração de testes executáveis, a execução dos testes ou a coleta de resultados, etc.), o modelo sendo usado (redes de Petri, cadeias de Markov, máquinas de estados, etc.). Todas estas variáveis levam à descrição e desenvolvimento de diferentes abordagens de MBT, o que acaba gerando a implementação de diversas ferramentas para o auxílio na automação do processo de testes. Apesar de todas as diferenças citadas, é possível encontrar pontos em comum em vários dos trabalhos e das abordagens propostas.

A metodologia de engenharia de linha de produto de *software* (SPL) objetiva justamente reger o desenvolvimento de *software* através da implantação de uma linha de produtos. Ou seja, considera a implementação da linha propriamente dita, que vem a ser o agrupamento de funcionalidades em comum a todos os produtos da linha, e a implementação de produtos para essa linha. Estes são os responsáveis por adicionar variabilidade à linha, de forma a atender, com o mínimo esforço possível, requisitos diversos, porém relacionados ou similares.

Este trabalho propôs uma arquitetura de *software* baseada nos conceitos de engenharia de linha de produto, para lidar com o processo de testes baseados em modelos. A arquitetura faz uso do modelo de *plug-ins* para prover a extensibilidade, e também para preencher a clássica lacuna existente entre os modelos conceituais e as implementações de linhas de produto. O trabalho compreendeu, também a implementação de uma ferramenta que concretiza esta arquitetura, bem como a implementação de três *plug-ins* para a automação de testes funcionais, de segurança e de desempenho.

A ferramenta permite a adição de novos *plug-ins*, bem como a reutilização dos *plug-ins* desenvolvidos anteriormente, seja pela simples execução de um *plug-in* já existente, seja pela composição de diversos *plug-ins*, novos ou já existentes. Dessa forma, estão satisfeitos os dois principais atributos desejáveis na arquitetura, a extensibilidade e o reuso.

No sentido de continuar as propostas apresentadas, existem algumas melhorias e novas funcionalidades a serem incorporadas à ferramenta. No componente que gerencia o fluxo dos testes, ou seja, os *workflows* que são executados pela ferramenta, são propostas a possibilidade de lidar com mais de um *workflow*, a edição gráfica de um fluxo de testes (e posterior geração automática do arquivo que o descreve), e o uso de uma linguagem reconhecida pela indústria e pela academia para a descrição dos *workflows*, como BPMN. Outra possibilidade é substituir o modelo de *workflow* sugerido pela execução dos *workflows* em ferramentas bem estabelecidas no mercado, como o Oracle Workflow, por exemplo. É também proposta a geração automática de *stubs* a partir da descrição de um fluxo de execução, a fim de facilitar o desenvolvimento de um novo *plug-in*. Outra melhoria a ser implementada na gerência de *plug-ins* é a possibilidade de paralelizar a execução de tarefas concorrentes na ferramenta. E, a fim de dar continuidade à extensibilidade da ferramenta, há a proposta de implementação de novos *plug-ins* para serem executados com a ferramenta.

REFERÊNCIAS

[ABD00] Abdurazik, A.; Offutt, J. “Using UML Collaboration Diagrams for Static Checking and Test Generation”. In: Proceedings of the 3rd International Conference on the Unified Modeling Language (UML 00), 2000, York, 13p.

[APF97] Apfelbaum, L.; Doyle, J. “Model Based Testing”. In: Proceedings of the 10th International Software Quality Week (QW97), 1997, San Francisco, 14p.

[AVI04] Avizienis, A.; Laprie, J.; Randell, B.; Landwehr, C. “Basic Concepts and Taxonomy of Dependable and Secure Computing”. *IEEE Transactions on Dependable and Secure Computing,* vol. 1, issue 1, Jan 2004, 23p.

[BAS00] Basanieri, F.; Bertolino, A. “A Practical Approach to UML-based Derivation of Integration Tests”. In: Proceedings of the 4th International Software Quality Week Europe (QWE 2000), 2000, Brussels, 5p.

[BEI90] Beizer, B. “Software Testing Techniques”. New York: Van Nostrand Reinhold Co., 1990. 580p.

[BER02] Bernardi, S.; Donatelli, S.; Merseguer, J. “From UML Sequence Diagrams and Statecharts to Analyzable Petri Net Models”. In: Proceedings of 3rd International Workshop on Software and Performance (WOSP), 2002, Roma, 11p.

[BER04] Bertolino, A.; Marchetti, E.; Muccini, H. “Introducing a Reasonably Complete and Coherent Approach for Model-based Testing”. In: Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'04), 2004, Barcelona, 12p.

[BLA04] Blackburn, M.; Busser, R.; Nauman, A. “Why Model-Based Test Automation is Different and What You Should Know to Get Started”. In: International Conference on Practical Software Quality and Testing (PSQT), 2004, Washington, D.C., 16p.

[BOE01] Böckle, G.; Knauber, P.; Linden, F.; Northrop, L.; Pohl, K. “Dagstuhl Seminar on Product Family Development”. Capturado em: http://www.dagstuhl.de/no_cache/en/program/calendar/semhp/?semnr=03151, Dezembro 2009.

[BOE04] Böckle, G.; Knauber, P.; Pohl, K.; Schmid, K. “Software-Produktlinien – Methoden, Einführung und Praxis”. Heidelberg: Dpunkt, 2004, 320p.

[BON08] Bonifácio, A.; Moura, A.; Simão, A. “A Generalized Model-based Test Generation Method”. In: Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM), 2008, Cape Town, 10p.

[BOS01] Bosch, J.; Florijn, G.; Greefhorst, D.; Kuusela, J.; Obbink, H.; Pohl, K. “Variability Issues in Software Product Lines. In: International Workshop on Software Product-Family Engineering, 2001, London, 9p.

[BRI08] Bringmann, E.; Krämer, A. “Model-based Test of Automotive Systems”. In: International Conference on Software Testing, Verification, and Validation (ICST), 2008, Lillehammer, 9p.

[CHI09] Chinosi, M.; Trombetta, A. “Modeling and Validating BPMN Diagrams”. In: IEEE Conference on Commerce and Enterprise Computing (CEC), 2009, Vienna, 8p.

[CHO56] Chomsky, N. "Three Models for the Description of Language". *IRE Transactions on Information Theory*, vol. 2, issue 3, Set 1956, 12p.

[CHO57] Chomsky, N. “Syntactic Structures”. The Hague: Mouton, 1957, 117p.

[CLE02] Clements, P.; Northrop, L. “Software Product Line: Practices and Patterns”, Boston: Addison-Wesley, 2002, 608p.

[COH07] Cohen, S. “Product Line State of the Practice Report”. Capturado em: http://www.sei.cmu.edu/publications/documents/02.reports/02tn017.html, Dezembro 2009.

[DAL98] Dalal, S.; Jain, A.; Karunanithi, N.; Leaton, J.; Lott, C. “Model-based Testing of a Highly Programmable System”. In: Proceedings of the 9th International Symposium on Software Reliability Engineering (ISSRE), 1998, Paderborn, 5p.

[DAL99] Dalal, S.; Jain, A.; Karunanithi, N.; Leaton, J.; Lott, C.; Patton, G.; Horowitz, B. “Model-Based Testing in Practice”. In: Proceedings of the International Conference on Software Engineering (ICSE), 1999, Los Angeles, 10p.

[DAV88] Davis, A. “A Comparison of Techniques for the Specification of External System Behavior”. *Communications of the ACM*, vol. 31, issue 9, Set 1988, 18p.

[DEL07] Delamaro, M.; Maldonado, J.; Jino, M. “Introduction to Software Testing”. São Paulo: Elsevier, 2007, 408p.

[DOR05] Dorofeeva, R.; El-Fakih, K.; Yevtushenko, N. “An Improved Conformance Testing Method”. In: Formal Techniques for Networked and Distributed Systems (FORTE), *Lecture Notes in Computer Science, 3731*, Berlin/Heidelberg: Springer, 2005, 15p.

[DRU04] Drumea, A.; Popescu, C. “Finite State Machines and their Applications in Software for Industrial Control”. In: 27th International Spring Seminar on Electronics Technology (ISSE 2004), 2004, Sofia, 5p.

[ECM09] ECMA International. “Standard ECMA-334 - C# language specification”. Capturado em: http://www.ecmainternational.org/publications/standards/Ecma-334.htm, Dezembro 2009.

[EHR04] Ehrig, H.; Reisig, W.; Rozenberg, G.; Weber, H. “Petri Net Technology for Communication-Based Systems: Advances in Petri Nets”. *Lecture Notes in Computer Science, 2472*, Berlin/Heidelberg: Springer, 2004, 455p.

[ELF01a] El-Far, I.; Whittaker, J. “Model-Based Software Testing”. Encyclopedia on Software Engineering, vol. 1. Cap. Model-Based Software Testing. New York: Wiley, 2001. 22p.

[ELF01b] El-Far, I. “Enjoying the Perks of Model-Based Testing”. In: Proceedings of the Software Testing, Analysis, and Review Conference (STARWEST 2001), 2001, San Jose, 13p.

[EST06] Estublier, J.; Garcia, S. “Workflows and Cooperative Processes”. *Software Process Change*, *Lecture Notes in Computer Science, 3966*. Berlin/Heidelberg: Springer, 2006, 8p.

[FAR08] Farooq, A.; Dumke, R. “Evaluation Approaches in Software Testing”. Technical Report. Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2008, 88p.

[FOW04] Fowler, M. “UML Distilled: A Brief Guide to the Standard Object Modeling Language”. Boston: Addison-Wesley, 2004, 208p.

[GAR03] García, D.; García, J. “TPC-W E-Commerce Benchmark Evaluation”. *IEEE Computer*, vol. 36, issue 2, Feb 2003, 7p.

[GON70] Gönenç, G. “A Method for the Design of Fault Detection Experiments”. *IEEE Transactions on Computers*, vol. C-19, issue 6, Jun 1970, 8p.

[GRI98] Griss, M.; Favaro J.; D’Alessandro, M. “Integrating Feature Modeling with the RSEB”. In: International Conference on Software Reuse, 1998, Washington, DC, 10p.

[HAL08] Halili, E. “Apache JMeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for your Websites”. Birmingham: Packt Publishing, 2008, 140p.

[HAR87] Harel, D. “Statecharts: a Visual Formalism for Complex Systems”. *Science of Computer Programming*, vol. 8, issue 3, Jun 1987, 44p.

[HAR00] Hartmann, J.; Imoberdorf C.; Meisinger, M. “UML-based Integration Testing”. In: Proceedings of the 2000 International Symposium on Software Testing and Analysis (ISSTA 2000), 2000, Portland, 11p.

[HAR02] Hartman, A. “Model Based Test Generation Tools”. Technical Report, AGEDIS Consortium, 2002, 7p. Capturado em: http://www.agedis.de/documents/ ModelBasedTestGenerationTools_cs.pdf, Dezembro 2009.

[HAS08] Hasling, B.; Goetz, H.; Beetz, K. “Model Based Testing of System Requirements using UML Use Case Models”. In: International Conference on Software Testing, Verification, and Validation (ICST), 2008, Lillehammer, 10p.

[HEN64] Hennie, F. “Fault Detecting Experiments for Sequential Circuits”. In: 5th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT), 1964, Princeton, 16p.

[HIE06] Hierons, R. “Separating Sequence Overlap for Automated Test Sequence Generation”. *Automated Software Engineering*, vol. 13, issue 2, Mar 2006, 19p.

[JOR95] Jorgensen, P. “Software Testing: A Craftsman’s Approach”. Boca Raton: CRC Press, 1995, 272p.

[KAN90] Kang, K.; Cohen, S.; Hess, J.; Novak, W.; Peterson, A. “Feature-Oriented Domain Analysis (FODA) Feasibility Study”. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, 1990, 161p.

[KAN98] Kang, K.; Kim, S.; Lee, J.; Kim, K.; Kim, G.; Shin, E. “FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. In: *Annals of Software Engineering*, Red Bank, NJ: J. C. Baltzer AG, Science Publishers, 1998, vol. 5, p. 143–168.

[KAN99] Kaner, C.; Falk, J.; Nguyen, H. “Testing Computer Software”. New York: Wiley, 1999, 480p.

[KAN06] Kaner, C. “Exploratory Testing”. In: Quality Assurance Institute (QAI) Worldwide Annual Software Testing Conference, 2006, Orlando, 47p.

[KEM76] Kemeny, J.; Snell, J. “Finite Markov Chains”. New York: Springer-Verlag, 1976, 244p.

[KER99] Kernighan, B.; Pike, R. “The Practice of Programming”. Indianapolis: Addison-Wesley, 1999, 288p.

[KRI04] Krishnan, P. “Uniform Descriptions for Model Based Testing”. In: Proceedings of the 2004 Australian Software Engineering Conference (ASWEC’04), 2004, Melbourne, 10p.

[LEE96] Lee, D.; Yannakakis, M. “Principles and Methods of Testing Finite State Machines – A Survey”. *Proceedings of the IEEE*, vol. 84, issue 8, Ago 1996, 34p.

[LEI07] Leitner, A.; Ciupa, I.; Oriol, M.; Meyer, B.; Fiva, A. "Contract Driven Development = Test Driven Development - Writing Test Cases", In: Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07), 2007, Dubrovnik, 10p.

[LIN02] Linden, F. “Software Product Families in Europe: The ESAPS and CAFÉ Projects”, *IEEE Software*, vol. 19, issue 4, Jul-Aug 2002, 9p.

[LOP04] López-Grao, J.; Merseguer, J.; Campos, J. “From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering”. In: Proceedings of the 4th International Workshop on Software and Performance (WOSP), 2004, Redwood Shores, 12p.

[LYU96] Lyu, M. “Handbook of Software Reliability Engineering”. New York: McGraw-Hill, 1996, 850p.

[MAN02] Mannion, M. “Organizing for Software Product Line Engineering”. In: Proceedings of the 10th International Workshop on Software Technology and Engineering Practice (STEP’02), 2002, Montreal, 7p.

[MAR54] Markov, A. “Theory of Algorithms”, *Works of the Mathematical Institute*, Academy of Sciences of the USSR, vol. 42, 1954, 14p.

[MAR84] 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*, vol. 2, issue 2, Mai 1984, 30p.

[MAR95] Marsan, M.; Balbo, G.; Conte, G.; Donatelli, S.; Franceschinis, G. “Modeling with Generalized Stochastic Petri Nets”. New York: Wiley, 1995, 324p.

[MAT04] Matinlassi, M. “Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA”. In: Proceedings of the 26th International Conference on Software Engineering (ICSE), 2004, Edinburgh, 10p.

[MCG01] McGregor, J.; Sykes, D. “A Practical Guide to Testing Object-Oriented Software”. Upper Saddle River: Addison-Wesley, 2001, 416p.

[MEY97] Meyer, M.; Lehnerd. A. “The Power of Product Platforms: Building Value and Cost Leadership”. New York: Free Press, 1997, 288p.

[MOL09] Molecularsciences.org. “Hidden Markov Models”. Capturado em: http://www.molecularsciences.org/bioinformatics/hidden_markov_models. Dezembro 2009.

[MSD09] The Microsoft Developer Network (MSDN). “.NET Framework Developer Center”. Capturado em: http://msdn.microsoft.com/en-us/library/system.reflection.aspx. Dezembro 2009.

[MYE04] Myers, G. “The Art of Software Testing”. New York: Wiley, 2004, 256p.

[NET07] Neto, A.; Subramanyan, R.; Vieira, M.; Travassos, G. “A Survey on Model-Based Testing Approaches: A Systematic Review”. In: International Workshop on Empirical Assessment of Software Engineering Languages and Technologies, 2007, New York, 6p.

[NGU06] Nguyen, H.; Hackett, M.; Whitlock, B. “Global Software Test Automation: A Discussion of Software Testing for Executives”. Silicon Valley: Happy About, 2006, 164p.

[OFF99] Offutt, J.; Abdurazik, A. “Generating Test Cases from UML Specifications”. In: Proceedings of the 2nd International Conference on the Unified Modeling Language (UML 99), 1999, Fort Collins, 14p.

[OLI07] Oliveira, F.; Menna, R.; Vieira, H.; Ruiz, D. “Performance Testing from UML Models with Resource Descriptions”. In: I Brazilian Workshop on Systematic and Automated Software Testing (SAST), 2007, João Pessoa, 8p.

[OMG09] Object Management Group. “UML Superstructure Specification Version 2.2”. Capturado em: http://www.omg.org/cgi-bin/doc?formal/09-02-02.pdf, Dezembro 2009.

[ORO09a] Orozco, A. “Linha de Produtos de Testes Baseados em Modelos”. Dissertação de Mestrado. Pontifícia Universidade Católica do Rio Grande do Sul, 2009.

[ORO09b] Orozco, A.; Oliveira, K.; Oliveira, F.; Zorzo, A. “Derivação de Casos de Testes Funcionais: uma Abordagem Baseada em Modelos UML”. In: V Simpósio Brasileiro de Sistemas de Informação, 2009, Brasília, 12p.

[OUY06] Ouyang, C.; Dumas, M.; Hofstede, A.; Aalst, W. “From BPMN Process Models to BPEL Web Services”. In: IEEE International Conference on Web Services (ICWS'06), 2006, Chicago, 8p.

[OWA09] The Open Web Application Security Project. “The Ten Most Critical Web Application Security Vulnerabilities”. Capturado em: http://www.owasp.org, Dezembro 2009.

[PAN99] Pan, J. “Software Testing (Dependable Embedded Systems)”. Capturado em: http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/, Dezembro 2009.

[PER08] Peralta, K.; Orozco, A.; Zorzo, A.; Oliveira, F. “Specifying Security Aspects in UML Models”. In: ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems, 2008, Toulouse, 10p.

[PER09] Peralta, K. “Estratégia para Especificação e Geração de Casos de Teste a partir de Modelos UML”. Dissertação de Mestrado. Pontifícia Universidade Católica do Rio Grande do Sul, 2009.

[PET62] Petri, C. “Kommunikation mit Automaten”. Tese de Doutorado, Institut für Instrumentelle Mathematik, Universität Bonn, 1962.

[POH05] Pohl, K.; Böckle, G.; Linden, F. “Software Product Line Engineering: Foundations, Principles and Techniques”. Berlin: Springer, 2005, 468p.

[PRA05] Prasanna, M.; Sivanandam, S.; Venkatesan, R.; Sundarrajan, R. “A Survey on Automatic Test Case Generation”. *Academic Open Internet Journal*, vol. 15, 2005, 7p. Capturado em: http://www.acadjournal.com/2005/v15/part6/p4/, Dezembro 2009.

[PRE01] Pretschner, A.; Slotosch, O.; Lötzbeyer, H.; Aiglstorfer, E.; Kriebel, S. “Model Based Testing for Real: The Inhouse Card Case Study”. In: International Workshop on Formal Methods for Industrial Critical Systems (FMICS), 2001, Paris, 15p.

[ROB99] Robinson, H. “Graph Theory Techniques in Model-Based Testing”. In: International Conference on Testing Computer Software (ICTCS 99), 1999, Washington, D.C., 10p.

[ROD08] Rodrigues, E.; Winck, A.; Zorzo, A.; Rossi, F.; Ruiz, D. “Uso de Modelos Preditivos e SLAs para Reconfiguração de Ambientes Virtualizados”. In: Workshop de Sistemas Operacionais (WSO2008), 2008, Belém, 12p.

[ROD09a] Rodrigues, E.; Zorzo, A.; Oliveira, F.; Costa, L. “Reconfiguração de Ambientes Virtualizados através do uso de Teste Baseado em Modelos e SLAs”. In: Workshop de Sistemas Operacionais (WSO2009), 2009, Bento Gonçalves, 12p.

[ROD09b] Rodrigues, E. “Realocação de Recursos em Ambientes Virtualizados”. Dissertação de Mestrado. Pontifícia Universidade Católica do Rio Grande do Sul, 2009.

[ROS05] Rosenau, M.; Griffin, A.; Castellion, G.; Anschuetz, N. “The PDMA Handbook of New Product Development”. New York: Wiley, 2005, 625p.

[SAV08] Savenkov, R. “How to Become a Software Tester”. Sunnyvale: Roman Savenkov Consulting, 2008, 405p.

[SEI09a] Software Engineering Institute (SEI). “Software Product Lines (SPL)”. Capturado em: http://www.sei.cmu.edu/productlines/, Dezembro 2009.

[SEI09b] Software Engineering Institute (SEI). “Software Product Lines Essentials”. Capturado em: http://www.sei.cmu.edu/library/assets/spl-essentials.pdf, Dezembro 2009.

[SEL07] Sellier, D.; Mannion, M.; Benguria, G.; Urchegui, G. “Introducing Software Product Line Engineering for Metal Processing Lines in a Small to Medium Enterprise”. In: 11th International Software Product Line Conference, 2007, Kyoto, 7p.

[SID88] Sidhu, D.; Leung. T. “Experience with Test Generation for Real Protocols”. In ACM SIGCOMM Computer Communication Review: Symposium Proceedings on Communications Architectures and Protocols, 1988, New York, 5p.

[SKE00] Skelton. G.; Steenkamp, A.; Burge. C. “Utilizing UML Diagrams for Integration Testing of Object-Oriented Software”. *Software Quality Professional*, vol. 2, issue 3, Jun 2000, 11p.

[SOC04] Sochos, P.; Philippow, I.; Riebisch, M. “Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture”. *Software Product Lines, Lecture Notes in Computer Science, 3263*. Berlin/Heidelberg: Springer, 2004, 15p.

[SOW00] Sowa, J. “Processes and Causality”. Capturado em: http://www.jfsowa.com/ontology/causal.htm, Dezembro 2009.

[STE04] Steger, M.; Tischer, C.; Boss, B.; Müller, A.; Pertler, O.; Stolz, W.; Ferber, S. “Introducing PLA at Bosch Gasoline Systems: Experiences and Practices”. In: Proceedings of the Software Product Line Conference, 2004, Boston, 17p.

[URA97] Ural, H.; Wu, X.; Zhang, F. “On Minimizing the Lengths of Checking Sequences”. *IEEE Transactions on Computers*, vol. 46, issue 1, Jan 1997, 7p.

[WEB01] Weber, T. “Tolerância a Falhas: Conceitos e Exemplos”. Capturado em: http://www.inf.ufrgs.br/~taisy/disciplinas/textos/ConceitosDependabilidade.PDF, Dezembro 2009.

[WEB05] Weber, S.; Karger, P.; Paradkar, A. “A Software Flaw Taxonomy: Aiming Tools at Security”. In: Software Engineering for Secure Systems – Building Trustworthy Applications (SESS), 2005, Saint Louis, 7p.

[WEI99] Weiss, D.; Lai, C. “Software Product Line Engineering: A Family-Based Software Development Process”. Reading: Addison-Wesley, 1999, 448p.

[WHI08] White, S.; Miers, D. “BPMN Modeling and Reference Guide”. Lighthouse Pt: Future Strategies Inc., 2008, 226p.

[WIN90] Wing, J. “A Specifier’s Introduction to Formal Methods”. *Computer*, vol. 23, issue 9, Set 1990, 17p.

[WOL08] Wolfinger, R.; Reiter, S.; Dhungana, D.; Grünbacher, P.; Prähofer, H. “Supporting Runtime System Adaptadion through Product Line Engineering and Plug-in Techniques”. In: Proceedings of the 7th International Conference on Composition-Based Software Systems, 2008, Madrid, 10p.

[ZUR94] Zurawski, R.; Zhou, M. “Petri Nets and Industrial Applications: A Tutorial”. *IEEE Transactions on Industrial Electronics*, vol. 41, issue 6, Dez 1994, 17p.

APÊNDICE A – CRIAÇÃO DE UM NOVO *PLUG-IN*

Neste apêndice, são apresentados os passos sugeridos para a criação de um novo *plug-in* para a ferramenta proposta. Os principais pré-requisitos são: ter uma cópia da ferramenta instalada em uma máquina local e ter o .NET *Framework* instalado na mesma máquina, em uma versão igual ou superior à 2.0. Os passos serão apresentados de forma textual e bastante intuitiva, e consideram que está sendo usado o Microsoft Visual Studio .NET 2005 ou superior para o desenvolvimento do *plug-in*, que é o recomendado para o desenvolvimento de *plug-ins* para a ferramenta.

1 – Criar uma nova solução ou abrir uma solução já existente;

2 – Criar um novo projeto do tipo "Class Library";

3 – Nomear o projeto (sugere-se o sufixo `PlugIn` para o nome do projeto. Por exemplo: `ExamplePlugIn`);

4 – Adicionar (como link) a chave provida junto com a solução, a fim de assinar o binário (C:\...\WorkflowManager\WorkflowManagerTool\WorkflowManagerKey.snk);

5 – Editar as propriedades do projeto:

5.1 – Seção *Application*: alterar o *Default Namespace* para `PlugIns`;

5.2 – Seção *Build*: marcar a opção *XML documentation file*, mantendo o nome padrão;

5.3 – Seção *Signing*: marcar a opção *Sign the assembly*, e selecionar, na caixa ao lado, a chave adicionada no passo 4.

6 – Renomear, no *Solution Explorer*, o arquivo Class1.cs para um nome conveniente (por exemplo: ExamplePlugIn.cs), confirmando a busca e alteração de todas as referências ao tipo em questão (o Visual Studio pergunta se o usuário deseja fazer isso ou não).

7 – Adicionar, no projeto, uma referência ao binário BasePlugIn.dll, que se encontra na pasta da ferramenta;

8 – Renomear o *namespace* da classe adicionada para `PlugIns`;

9 – Fazer a classe (no exemplo, `ExamplePlugIn`) herdar de `BasePlugIn`;

10 – Prover implementação ao(s) membro(s) abstrato(s) de `BasePlugIn` (método `Initialize`);

11 – Implementar os métodos projetados para o *plug-in*, tomando o cuidado de mantê-los como públicos e estáticos, para que a ferramenta os reconheça como métodos válidos;

12 – Adicionar cabeçalhos XML padrão a todos os elementos públicos e protegidos, a fim de gerar corretamente a documentação das classes;

13 – Compilar a solução;

14 – Copiar o binário gerado (no exemplo, seria ExamplePlugIn.dll) para a pasta dos binários da aplicação (a mesma pasta referenciada no passo 7)

15 – Rodar a ferramenta. O novo *plug-in* aparecerá na árvore mostrada no lado direito da ferramenta e estará pronto para receber associações na mesma.