PLeTs - Uma Linha de Produto de Ferramentas de Teste Baseado em Modelos

Elder de Macedo Rodrigues, Avelino Francisco Zorzo

[1]Programa de Pós-graduação em Ciência da Computação
Pontificia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brazil

{elder.rodrigues,avelino.zorzo}@pucrs.br

Abstract. *In the last years a diversity of testing tools has been developed to support the software testing activities. However, most of them have been individually and independently developed from scratch based on a single architecture. Thus, they face difficulties of integration, evolution, maintenance, and reuse. Software Product Lines (SPL) offers possibility of systematically generating software products at lower costs, in shorter time, and with higher quality. In this work we propose a SPL for generate testing tools that support MBT. Moreover, we also present two examples of use, performed in collaboration of an IT company, where four MBT testing tools are derived from our SPL.*

Resumo. *Nos últimos anos, uma grande diversidade de ferramentas de teste foi desenvolvida para apoiar as atividades de teste de software. No entanto, a maioria delas foram individualmente e independentemente desenvolvidas. Assim, elas enfrentam dificuldades de integração, evolução, manutenção e reutilização. Linhas de Produto de Software (LPS) oferece a possibilidade de gerar sistematicamente produtos de software a custos mais baixos, em menor tempo e com maior qualidade. Neste trabalho, é proposta uma LPS de ferramentas de teste que suportam Teste Baseado em Modelos. Além disso, são apresentado dois exemplos de uso, realizados em cooperação com uma indústria de TI, onde foram geradas a partir de nossa LPS quatro ferramentas de teste que suportam TBM.*

1. Introdução

Nos últimos anos, as mais diversas áreas do conhecimento tem adotado novos e mais avançados sistemas de *software*, para que sejam utilizados como ferramentas de apoio a suas atividades fim. Por este motivo, as empresas que produzem esses sistemas buscam incessantemente garantir que o *software* desenvolvido esteja funcionando de acordo com sua especificação. Portanto, *software* de alta qualidade pode ser considerado um ativo muito importante para qualquer empresa de *software* hoje em dia.

Teste de *software* é uma das atividades realizadas durante o processo de desenvolvimento com o objetivo de garantir certa qualidade e confiabilidade aos produtos de *software* desenvolvidos [Myers and Sandler 2004] [Harrold 2000]. No entanto, para que as atividades de teste contribuam para aumentar a qualidade do *software* é necessário que as mesmas sejam realizadas de forma sistemática e usando técnicas e critérios de teste bem definidos [DeMillo et al. 1978] [Myers and Sandler 2004] [Rapps and Weyuker 1985].

Apesar da grande variedade de técnicas, critérios e ferramentas disponíveis atualmente, em muitos casos a etapa de teste é conduzida de maneira *ad hoc*, onde engenheiros de *software* desenvolvem *scripts* e casos de teste manualmente e ainda precisam entender e escrever arquivos de configuração de ferramentas. Por este motivo, essas atividades são propensas a erros e dessa forma falhas podem ser inadvertidamente inseridas durante as atividades de teste.

Neste contexto, **Teste Baseado em Modelos** (**TBM**) é uma técnica de teste que tem se destacado últimamente, principalmente pelo fato de possibilitar a automatização das atividades de teste. TBM se baseia em informações de teste representadas em modelos dos sistemas testados (System Under Test - SUT) para gerar casos de teste e *scripts* de forma automatizada [Utting and Legeard 2006]. A utilização de TBM apresenta diversas outras vantagens quando comparada as demais técnicas de teste. Por exemplo, o uso de TBM pode reduzir a probabilidade de erros de interpretação dos requisitos do sistema por analista de sistemas, engenheiros de teste e testadores.

Últimamente, uma grande variedade de ferramentas foram desenvolvidas para explorar as vantagens que TBM apresenta [Hartman and Nagin 2004] [Huima 2007] [Abbors et al. 2010]. Por exemplo, a ferramenta Conformiq Qtronic [Huima 2007] deriva casos de teste automaticamente a partir de modelos comportamentais de um sistema, *e.g.*, Diagrama de Estados e QML (Qtronic Modeling Language). Por sua vez, a suíte AGEDIS [Hartman and Nagin 2004] gera casos de testes a partir de modelos AML (*AGEDIS Modeling Language*) e modelos UML, com o objetivo principal de testar sistemas distribuídos baseados em componentes. Apesar do fato de que a maioria dessas ferramentas terem sido desenvolvidas por diferentes empresas ou grupos de pesquisa, a maioria deles usa um processo semelhante e, em alguns casos, os mesmos tipos de modelos do sistema, mesmo quando gera casos de testes e *scripts* para diferentes níveis de teste.

Entretanto, apesar de existir uma diversidade de ferramentas, que compartilham um conjunto de características, com o propósito de automatizar as atividades da etapa de teste de *software*, a maioria dessas ferramentas foram implementadas independentemente. Assim, essas ferramentas apresentam diversas dificuldades quando precisam ser desenvolvidas, integradas e evoluídas.

A fim de reduzir essas dificuldades no desenvolvimento, integração e evolução de ferramentas que suportam TBM, seria relevante adotar uma estratégia para gerar automaticamente produtos específicos, *i.e.*, ferramentas de teste, baseado na reutilização de artefatos e baseados em uma arquitetura comum. O reuso de artefatos comuns para desenvolver variantes de um conjunto de produtos é um dos principais conceitos utilizados no desenvolvimento de Linhas de Produtos de *Software* (LPS). Basicamente, uma LPS é composta por um conjunto de sistemas interelacionados, que compartilham características comuns e um conjunto gerenciável de partes variáveis que representam decisões de projetos adiadas [Clements and Northrop 2001]. A adoção de LPS tem aumentado nos últimos anos e vários casos de sucesso têm sido relatados na literatura [SEI] [Bass et al. 1997] [Northrop 2002].

O objetivo principal deste trabalho é propor a aplicação dos conceitos de Linha de Produto de *Software* para sistematizar e apoiar a geração de ferramentas de Teste Baseados em Modelos. Nesse contexto, nós apresentamos e discutimos os requisitos, decisões

de projeto, identificação de variabilidade e o desenvolvimento da PLeTs (A **P**roduct **L**ine of Mod**e**l-based **T**esting Tool**s**).

Este trabalho esta organizado da seguinte forma: A Seção 2 apresenta brevemente o referencial teórico sobre os tópicos abordados no decorrer do trabalho. Na Seção 3 serão apresentados a PLeTs, os requisitos iniciais que nortearam o projeto da LPS bem como as decisões de projeto e a arquitetura da LPS. Por sua vez, na Seção 4 serão apresentados dois exemplos de uso, que descrevem como quatro ferramentas de teste que suportam TBM foram geradas a partir da PLeTs. Finalmente na Seção 5 são apresentadas conclusões e os trabalhos futuros.

2. Fundamentação Teórica

Teste baseado em modelo (TBM) é uma técnica usada para apoiar a geração automática de casos de teste/*scripts* a partir de modelos do SUT [El-Far and Whittaker 2001]. Entretanto, a adoção de uma abordagem TBM requer mais atividades do que as atividades tradicionais da etapa teste de *software* [Utting and Legeard 2006]. Por exemplo, a adoção de uma abordagem TBM requer que os engenheiros de teste invistam na formação da equipe de testes e ajustem os seus processos e suas atividades de testes. De acordo com El-Far e Whittaker [El-Far and Whittaker 2001] as principais atividades do processo de TBM podem ser definidas como: *a*) *Criar Modelo*: consiste na construção de um modelo baseado na especificação do sistema a ser testado; *b*) *Gerar Entradas Esperadas*: usa-se o modelo criado na atividade anterior para gerar entradas de teste (casos de teste, *scripts* de teste e os dados de entrada de aplicação); *c*) *Gerar Resultados Esperados*: com base em algum mecanismo se determina se os resultados de uma execução de teste estão corretos. Este mecanismo é um oráculo de teste, e é usado para determinar a correção da saída (resultados); *d*) *Executar o Teste*: se executa os *scripts* de teste e armazena os resultados. *e*) *Comparar os Resultados*: compara os resultados dos testes com resultados esperados (teste do Oracle), gerando relatórios para alertar a equipe de teste sobre as falhas; *f*) *Decidir as Ações Futuras*: com base nos resultados, é possível estimar a qualidade do *software*. Dependendo da qualidade alcançada, é possível parar o teste (qualidade alcançada), modificar o modelo para incluir mais informações para gerar novas entradas/saídas, modificar o sistema em teste (para remover falhas restantes), ou executar mais testes; *g*) *Finalizar o Teste*: se conclui o teste e se libera o *software* para o ambiente de produção.

A medida que a adoção de uma abordagem usando TBM possibilita a automatização das atividades de teste, pode-se reduzir o custo da etapa de teste de *software*, uma vez que o custo do teste está relacionado com o número de interações e de casos de teste que são executadas durante a fase de testes. Como os custos da fase de testes normalmente variam entre 30% e 60% do esforço de desenvolvimento de *software* [Myers and Sandler 2004], TBM é uma abordagem valiosa para mitigar os custos da etapa de teste [Veanes et al. 2008]. Além disso, a adoção de uma abordagem TBM pode trazer várias outras vantagens para a equipe de teste, tais como [El-Far and Whittaker 2001]:

- A etapa de teste pode ser finalizada em menor tempo e com menor custo;
- Possibilita a identificação precoce das ambiguidades na especificação do modelo do SUT;
- Melhora a comunicação entre os desenvolvedores e testadores;
- Facilidade para atualizar os casos de teste quando os requisitos da aplicação são alterados;

- Menos esforço para realizar testes de regressão;

No entanto, apesar das diversas vantagens proporcionadas pela adoção de uma abordagem TBM, a adoção de uma ferramenta de apoio é obrigatória. Atualmente, uma série de ferramentas comerciais, acadêmicas e *open-source* que suportam TBM estão disponíveis [Utting and Legeard 2006]. No entanto, não é uma tarefa fácil para um engenheiro de testes definir qual ferramenta será adotada. Além disso, a decisão de se desenvolver ou evoluir ferramentas TBM, pela própria equipe de teste, é uma atividade desafiadora e custosa, já que essas ferramentas podem ser baseadas em uma variedade de diferentes modelos, critérios de cobertura, abordagens e notações. Uma maneira de se mitigar estes problemas durante o desenvolvimento de ferramentas TBM, seria adotar uma abordagem que favorecesse o reúso dos artefatos comuns e fosse baseado em uma arquitetura comum.

Ao longo dos últimos anos, as empresas de desenvolvimento de *software* têm buscado utilizar algumas estratégias, tais como engenharia de *software* baseada em reúso, para desenvolver *software* com menor custo, em menor tempo, e com maior qualidade. Engenharia de *software* baseada em reúso é uma estratégia em que o processo de desenvolvimento está focado na reutilização de componentes de *software*, reduzindo o esforço e melhorando a qualidade do *software*. últimamente, tem sido propostas várias técnicas para apoiar o reúso de *software*, tais como, o desenvolvimento baseada em componentes, reutilização de produtos já disponíveis (Commercial of-the-shelf - COTS) e Linhas de Produto de Software [Sommerville 2011].

Linha de produtos de software é focado no desenvolvimento de uma família de aplicações baseadas em uma arquitetura comum e um conjunto compartilhado de componentes, onde cada aplicação é gerado a partir destes componentes e de acordo com os requisitos impostos por diferentes clientes [Sommerville 2011].

Nos últimos anos, LPS surgiu como uma técnica promissora para possibilitar a reutilização sistemática e ao mesmo tempo diminuir os custos de desenvolvimento e o tempo necessário para se colocar o produto no mercado [Pohl et al. 2005]. Desde o seu surgimento, diversos estudos tem sido publicados relatando casos de uso bem sucedidos [SEI] [Linden et al. 2007]. Por exemplo, a Nokia relatou que a adoção de LPS resultou em um aumento significativo na produção de novos modelos de telefones celulares, e a Hewlett Packard (HP) relatou uma melhora na produtividade e uma diminuição do tempo necessário para colocar as suas impressoras no mercado. Além disso, alguns autores, como Van der Liden [Linden et al. 2007] e Klaus Pohl [Pohl et al. 2005], advogam que a adoção de LPS por uma organização pode apresentar vários outros benefícios, tais como: *a*) *Redução dos custos de desenvolvimento*: como o desenvolvimento de cada aplicação é feita com base em um conjunto comum de artefatos, isso implica na redução do custo total de desenvolvimento [Pohl et al. 2005] [Linden et al. 2007] [Weiss and Lai 1999]. *b*) *Redução do tempo de desenvolvimento*: após o esforço inicial para desenvolver os artefatos comuns, há uma redução significativa no esforço para desenvolver as variantes do produto, sendo que estes produto chegarão rapidamente ao mercado e consequentemente resultará em um retorno mais rápido do investimento (Return of Investment - ROI). *c*) *Facilita a estimativa de custo*: baseado no fato que uma LPS é uma família de aplicações baseadas em uma arquitetura comum e com um conjunto comum de artefatos, é mais simples para uma empresa de desenvolvimento estimar o risco, e consequentemente o custo

do desenvolvimento de um novo produto de *software* a partir dessa plataforma.

Com bases nesses motivos, seria relevante aplicar os conceitos de LPS para se projetar e desenvolver ferramentas que suportam TBM. Desse modo, as equipes que desenvolvem essas ferramentas, para serem utilizadas internamente dentro de suas empresas ou para serem comercializadas, podem reutilizar sistematicamente os artefatos já desenvolvidos para apoiar a geração de novas variantes de ferramentas de teste que usam TBM. No entanto, antes de propor uma abordagem baseada em LPS para apoiar a geração de ferramentas que usam TBM, é mandatório compreender o domínio, bem como as ferramentas disponíveis que suportam TBM. Além disso, é necessário que o engenheiro de LPS seja um profundo conhecedor do domínio, para que seja possível definir os requisitos do domínio bem como identificar as suas características comuns e variáveis, como por exemplo, modelos e níveis de testes.

3. PLeTs - Uma Linha de Produto de Ferramentas de Teste Baseado em Modelos

Nessa seção iremos inicialmente apresentar os requisitos que orientaram a proposta e o projeto da PLeTs. A seguir, detalharemos as características[1] da LPS e em seguida apresentar os principais componentes de sua arquitetura.

3.1. Contextualização e Requisitos da LPS

O ponto de partida para o desenvolvimento da PLeTs foi a definição dos requisitos da LPS e a identificação de quais são comuns ou variáveis. Como já discutimos na Seção 2, a identificação de requisitos das ferramentas TBM é uma tarefa desafiadora, uma vez que há poucas contribuições atualizados na literatura sobre as funcionalidades dessas ferramenta e há um número significativo de ferramentas disponíveis [Dias Neto et al. 2007] [Shafique and Labiche 2010]. Por este motivo, foi realizada um mapeamento sistemático da literatura[2] (MSL) com o objetivo de mapear as ferramentas TBM disponíveis, bem como as suas principais características e funcionalidades. Além disso, recorremos a um Laboratório de Desenvolvimento de Tecnologia em Teste de Software (LDT) de uma empresa parceira para discutir os requisitos da LPS e validar as funcionalidades identificadas. Com base nos resultados do MSL e com a consultoria do LDT, foram definidos os requisitos básicos da nossa LPS de ferramentas TBM:

RQ1) *As ferramentas derivadas precisam suportar a geração automática de dados de teste*: as ferramentas TBM gerados a partir dos artefatos da PLeTs devem suportar a geração automática de casos de teste e/ou *scripts* de teste com base no modelo do sistema. RQ2) *As ferramentas gerados a partir da PLeTs devem possibilitar a integração com ferramentas de teste já existentes*: A geração de ferramentas TBM deve tirar proveito da integração com outras ferramentas de teste para executar os *scripts* de teste que foram gerados usando TBM. Portanto, as últimas atividades do processo de TBM (*Gerar Resultados Esperados* e *Executar teste*) devem ser projetados para suportar uma ferramenta externa, como por exemplo, LoadRunner [Hewlett Packard - HP] ou Jabuti [Vincenzi et al. 2005]. RQ3) *Os artefatos da LPS devem ser concebidos e desenvolvidos para apoiar a geração automática das ferramentas TBM*: a geração de ferramentas de

[1] Tradução do terno original em inglês: *Feature*

[2] Mais detalhes sobre o mapeamento sistemático pode ser encontrado em [Rodrigues 2013]

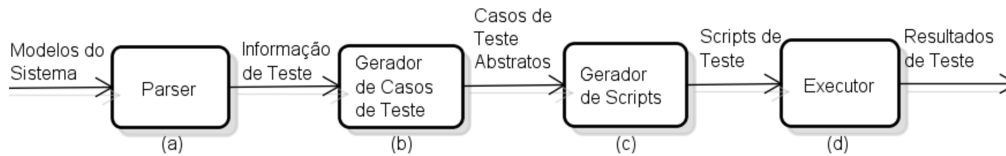

Figura 1. Processo TBM utilizado pelas ferramentas geradas pela PLeTs

TBM, a partir do conjunto de artefatos da LPS, deve requerer o mínimo de intervenção manual (codificação) por parte da equipe de teste. Além disso, caso os testadores decidam adicionar uma característica adicional a LPS, esta atividade não deve exigir nenhum conhecimento sobre estrutura interna dos artefatos da LPS, nem pode implicar em qualquer alteração no código fonte desses artefatos. RQ4) *A PLeTs não deve ser vinculada a um ambiente ou ferramentas de apoio proprietárias*: existem diversas preocupações relacionadas à utilização de ferramentas e ambientes proprietárias, *e.g.*, pure::variants [Beuche 2012] and Gears [Krueger and Clements 2012], para apoiar o processo de desenvolvimento da LPS de uma empresa, como por exemplo: as ferramentas podem não atender aos requisitos específicos da empresa ou a sua adoção pode requerer altos custos de treinamento. Além disso, a maioria das ferramentas disponíveis no mercado exigem um razoável investimento financeiro em licenças e em alguns casos, um alto investimento em consultoria. Assim, a decisão de adotar uma ferramenta de apoio a geração automática dos produtos a partir da PLeTs não deve requerer um significativo investimento adicional ou exigir das equipes de teste conhecimentos ou habilidades específicas.

3.2. Decisões de Projeto, Processo e Controle de Varialiabilidade

O Requisito *RQ1* define que as ferramentas TBM gerados a partir da PLeTs devem suportar a geração automática de casos de teste e *scripts* a partir de modelos do sistema. Embora El-far [El-Far and Whittaker 2001] tenha inicialmente apresentado e discutido o processo de TBM, nós também utilizamos os resultados da MSL a fim de identificar as características comuns e variáveis da ferramenta. Com base nesses resultados identificamos que as ferramentas TBM compartilham o seguinte conjunto de características básicas: a) extrair informações de teste a partir de modelos de sistemas; b) gerar os casos de teste/*scripts*; c) executar o teste. Motivado pelo fato que o requisito *RQ2* define que as ferramentas TBM devem tirar proveito da integração com outras ferramentas de teste, decidimos dividir a geração de casos de teste e *scripts* em duas características distintas: a) Gerador de casos de teste abstratos: gera casos de teste abstratos, que não estão relacionadas com uma tecnologia de teste. b) Gerador de scripts: instancia casos de teste abstratos em *scripts* executáveis para uma ferramenta de teste específica. Assim, uma ferramenta TBM gerada a partir da PLeTs poderia ter as seguintes características básicas: extrair informações de teste a partir de modelos de sistemas, gerar casos de teste abstratos, gerar *scripts* e executar o teste. Conforme apresentado na Figura 1, uma ferramenta TBM gerada a partir PLeTs deve extrair informações de teste de modelos (Figura 1 (a)) e com base nessas informações gerar os casos de teste abstratos (Figura 1 (b)). Depois disso, a ferramenta TBM pode concretizar os casos de teste abstratos para o formato de uma ferramenta (Figura 1 (c)) e, em seguida, executar o teste (Figura 1 (d)).

A Figura 2 apresenta o modelo de característica da PLeTs, que é composto por quatro características básicas, sendo que cada uma das característica básica representa

um passo no processo TBM (conforme a Figura 1):

- *Parser* É uma característica obrigatória que tem um relação de especialização com duas subcaracterísticas, *UmlPerf* e *UmlStruct*. A subcaracterística *UmlPerf* tem por funcionalidade extrair informações de teste de desempenho de diagramas UML enquanto a funcionalidade da subcaracterística *UmlStruct* é extrair informações de teste estrutural de diagramas UML.
- *TestCaseGenerator* é uma característica obrigatória que tem um relação de especialização com duas subcaracterísticas *AbstrTestCaseGen* e *SeqGen*. A subcaracterísticas *AbstrTestCaseGen* tem um relação de especialização com outras duas subcaracterísticas: *PerformanceTesting* e *StructuralTesting*. Por sua vez, a subcaracterísticas *SeqGen* tem um relação de especialização com duas subcaracterísticas: *FiniteStateMachineHSI* e *RandomDataGenerator*. A subcaracterísticas *AbstrTestCaseGen* é responsável pela geração dos casos de teste abstratos: para ferramentas de testes de desempenho o engenheiro da LPS precisa selecionar *PerformanceTesting* ou *StructuralTesting* para teste estrutural. A subcaracterística *SeqGen* é responsável por gerar as sequências de teste. A geração de seqüências de teste pode ser baseada no método gerador de seqüência HSI [Petrenko et al. 1993] quando a subcaracterística *FiniteStateMachineHSI* é selecionada ou com base na geração de dados de teste randômico quando *RandomDataGenerator* é selecionada.
- *ScriptGenerator* é uma característica opcional, já que algumas ferramentas podem gerar apenas casos de teste abstratos, cuja funcionalidade é instanciar os casos de teste abstratos em *scripts* executáveis para uma ferramenta de teste, que então será usada para executar o teste sobre o SUT. A característica *ScriptGenerator* tem um relação de especialização com tem cinco subcaracterísticas: *VisualStudioScript*, *LoadRunnerScript*, *JMeterScript*, *JabutiScript* e *EmmaScript*. Essas subcaracterísticas possuem a funcionalidade de instanciar os casos de teste abstratos para *scripts* de teste para as seguintes ferramentas de teste: Visual Studio [Perez and Guckenheimer 2006], LoadRunner [Hewlett Packard - HP] e Jmeter [Apache] para testes de desempenho e Emma [Roubtsov] e Jabuti [Vincenzi et al. 2005] para teste estrutural. Com base no pressuposto de que uma equipa de teste de uma empresa deve usar pelo menos uma ferramenta de teste para cada nível de teste, se espera que novas características sejam adicionadas de uma forma reativa a LPS.
- *Executor* é uma característica opcional cuja funcionalidade é executar uma ferramenta externa de teste e também iniciar a execução automática dos testes. Essa característica tem um relação de especialização com tem cinco subcaracterísticas: *VisualStudioSParameters*, *LoadRunerParameters*, *JMeterParameters*, *JabutiParamters* e *EmmaParameters*. Essas características tem por funcionalidade a execução da ferramenta de teste, o carregamento dos *scripts* na ferramenta e a execução dos mesmos nas seguintes ferramentas de teste: Visual Studio, LoadRunner e Jmeter para testes de desempenho e Emma e Jabuti para o teste estrutural.

O requisito *RQ3* define que os artefatos da PLeTs devem ser projetados e desenvolvidos para suportar a geração automátizada das ferramentas TBM. Por este motivo, tomamos as seguintes decisões de projeto:

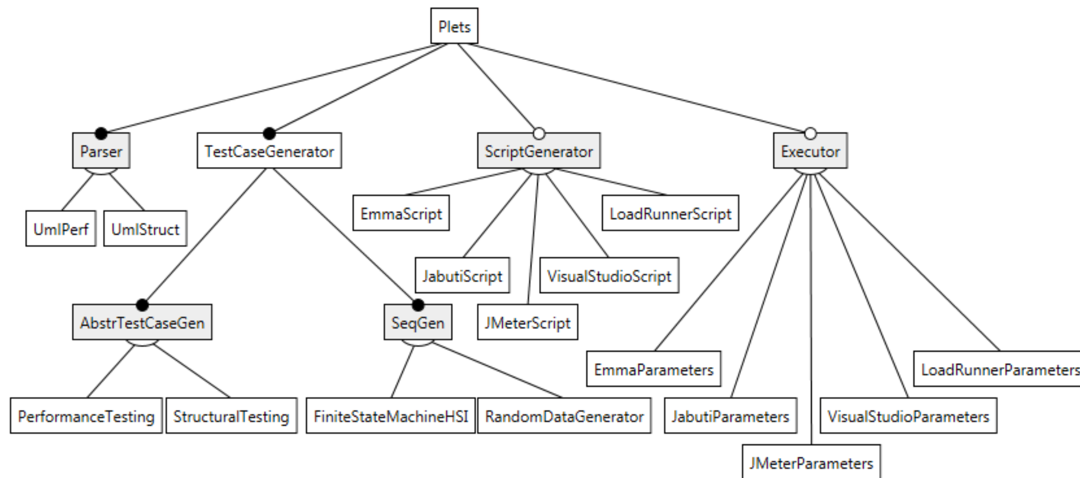

Figura 2. Modelo de características da PLeTs

- O modelo de características da PLeTs suporta a definição de características abstratas e concretas (na Figura 2 as características em retângulos brancos são concretas e em retângulos cinzas são abstratas) [Thum et al. 2011]. Na PLeTs, as características abstratas são utilizados apenas para fins de legibilidade e as características concretas devem ser mapeadas para exatamente um componente de *software*. Essa abordagem simplifica a atividade de rastreabilidade, reduz os erros, o esforço de desenvolvimento e os custos. O mapeamento é um-para-um (01:01) entre característica e componente e é realizada por correspondência dos nomes.
- Utilizar a substituição de componentes como mecanismo de variabilidade na PLeTs. A escolha desse mecanismo de variabilidade requer que o engenheiro de LPS implemente várias versões de um componente, sendo que cada versão segue uma especificação diferente. Por exemplo, podem ser desenvolvidas várias implementações da característica *Parser*, onde cada implementação extrai informações de um tipo diferente de modelo ou arquivo. Assim, um engenheiro de teste pode derivar uma nova ferramenta TBM , utilizando uma ferramenta de suporte, apenas selecionando as funcionalidades desejadas, que são então diretamente mapeados para componentes.

O requisito *RQ4* define que os artefatos da PLeTs não devem ser vinculados a um ambiente ou ferramenta de apoio proprietária. Este requisito nos levou a decisão de projetar e desenvolver um ambiente (PlugSPL) para apoiar o projeto, o desenvolvimento e a geração de produtos de um LPS, cujo mecanismo de variabilidade seja a substituição de componentes [Rodrigues et al. 2012] [Rodrigues et al. 2014]. Além disso, a adoção desse ambiente não exige habilidades específicas ou implica em custos adicionais.

3.3. Arquitetura e implementação

Conforme apresentado na Seção 3.2, uma de nossas decisões de projeto foi utilizar a substituição de componentes como mecanismo de variabilidade. Desta forma, uma ferramenta TBM derivada a partir da PLeTs é composta por um conjunto de componentes de *software* e uma base comum.

Para gerenciar as dependências entre os componentes e para representar a variabilidade na arquitetura da PLeTs, optamos por aplicar a abordagem **S**tereotype-based **M**anagement of **V**ar**i**abili**ty** (SMarty) [Oliveira et al. 2010]. A abordagem Smarty é composta por um perfil da UML e um processo para gerenciar as variabilidades de uma LPS. O perfil da UML define um conjunto de estereótipos e *tagged values* para representar a variabilidade de uma LPS a nível da arquitetura. O processo Smarty consiste em um conjunto de atividades que orientam o usuário para detectar, identificar e controlar as variabilidades em uma SPL. As motivações que nos levaram a escolher a abordagem Smarty, entre outras abordagens para gerenciar variabilidade utilizando modelos UML, são: a facilidade de se estender a notação e sua alta curva de aprendizado [Oliveira et al. 2010]. A Figura 3 apresenta o diagrama de componentes da PLeTs anotado de acordo com a abordagem Smarty, que por sua vez reflete as características descritas no diagrama de características (Figura 2). Como estamos usando um mecanismo de substituição de componentes, cada interface fornecida representa um ponto de variação e cada implementação de um componente representa uma variante. As seguintes interfaces são providas pelos componentes da PLeTs:

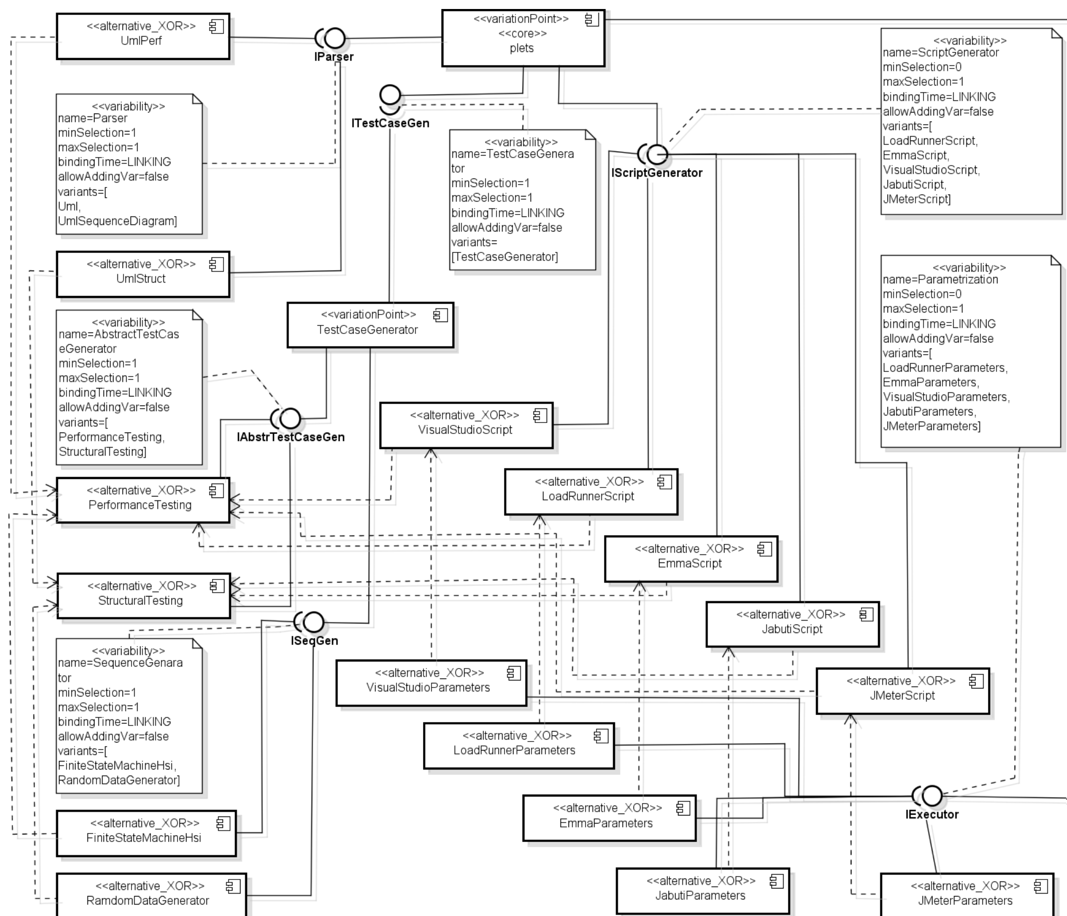

Figura 3. PLeTs UML Component Diagram with SMarty

- *IParser* é uma interface (ponto de variação) que deve ser realizada por um dos seguintes componentes (variantes): *UmlPerf* e *UmlStruct*. É importante salientar,

que a variabilidade indica que o número mínimo de variantes é um (*minSelection = 1*) e o máximo é de um (*maxSelection = 1*).

- *ITestCaseGen* é uma interface que deve ser realizada pelo componente *TestCase-Generator*. Por sua vez, este componente prove duas interfaces: *IAbstractTestGen* e *IseqGen*. A interface *IAbstractTestGen* deve ser realizada por um dos seguintes componentes: *PerformanceTesting* ou *StructuralTesting*. A interface *IseqGen* deve ser realizada por um dos seguintes componentes: *FiniteStateMachineHsi* ou *RandomDataGenerator*. O número mínimo de variantes que podem realizar a interface é um (*minSelection = 1*) e o máximo é de um (*maxSelection = 1*).
- *IScriptGenerator* é uma interface que pode ser realizada por um dos seguintes componentes: *VisualStudioScript*, *LoadRunnerScript*, *EmmaScript*, *JabutiScript* e *JmeterScript*. Assim, o número mínimo de variantes que podem ser realizados pela interface é zero (*minSelection = 0*) e o valor máximo de 1 (*maxSelection = 1*).
- *IExecutor* é uma interface que pode ser realizada por um dos seguintes componentes: *VisualStudioScript*, *LoadRunnerScript*, *EmmaScript*, *JabutiScript* e *JmeterScript*. O número mínimo de variantes que podem ser realizados pela interface é zero (*minSelection = 0*) e o valor máximo de 1 (*maxSelection = 1*).

A abordagem Smarty possibilita representar cenários onde a seleção de uma variante requer a seleção de uma outra variante (restrições entre as variantes). Por exemplo, se o componente *LoadRunnerParameters* for selecionado para compor um produto a ser gerado a partir da PLeTs, obrigatoriamente o componente *LoadRunnerScript* precisa se também selecionado, que por sua vez requer que o componente *PerformanceTesting* seja selecionado. Desta forma, uma configuração válida de um produto derivado poderia ser composta dos seguintes componentes: *plets*, *UmlParser*, *TestCaseGenerator*, *PerformanceTesting*, *FiniteStateMachineHsi*, *LoadRunnerScript*, *LoadRunnerParameters*.

4. Exemplo de Uso

Nesta seção serão apresentados dois exemplos de uso que descrevem como quatro ferramentas que usam TBM foram gerados a partir da PLeTs. Estas ferramentas foram projetadas e desenvolvidas no contexto de um projeto de colaboração entre o LDT de uma empresa de TI e nossa universidade. No primeiro exemplo de uso, será apresentado como foram geradas duas ferramentas que usam TBM para gerar casos de teste e *scripts* de desempenho, PLeTsPerfLR e PLeTsPerfVS [Rodrigues 2013] [Silveira 2012]. No segundo exemplo de uso será apresentado a geração de duas ferramentas que usam TBM para gerar dados para teste estrutural, PLeTsStructJabuti e PLeTsStructEmma [Costa 2012].

4.1. Geração de Ferramentas TBM para Teste de Desempenho

Nesta seção será demonstrada a geração, a partir da PLeTs, de duas ferramentas que usam TBM para aplicar teste de desempenho, PLeTsPerf e PLeTsVS. A PLeTsPerfLR é uma ferramenta TBM que gera *scripts* e cenários para o gerador de carga LoadRunner enquanto a PLeTsVS gera *scripts* e cenários para o Visual Studio. Essas ferramentas geram *scripts* de desempenho a partir de modelos UML, especificamente diagrama de casos de uso e de atividades. Como essas ferramentas são geradas a partir de uma LPS, compartilham algumas características comuns e neste caso se utilizam da mesma abordagem [Rodrigues et al. 2010] [Silveira et al. 2011]: recebem como entrada diagramas de caso

de uso e atividades anotados com informações de teste e derivam automaticamente os casos de teste abstratos. Depois disso, os casos de teste abstratos podem ser concretizados em *scripts* de teste de uma ferramenta para gerar carga. A seguir, os *scripts* podem ser automaticamente carregados na ferramenta de carga e o teste executado.

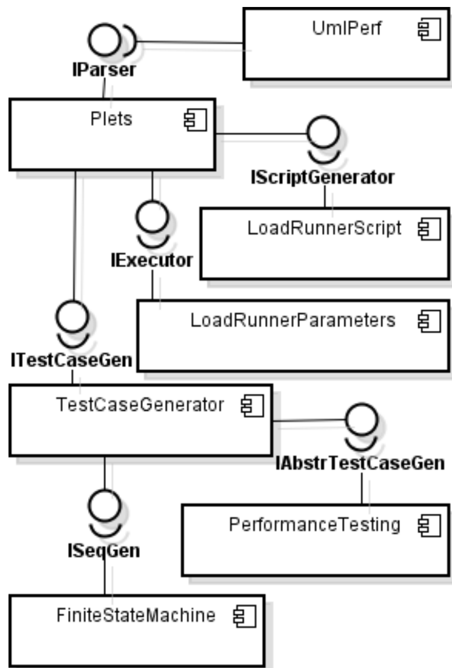

Figura 4. Arquitetura da ferramenta PLeTsPerfLR

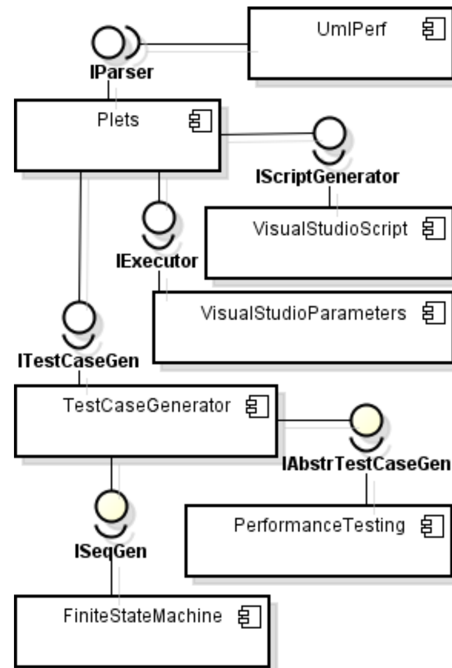

Figura 5. Arquitetura da ferramenta PLeTsPerfVS

As Figuras 4 e 5 apresentam a arquitetura das duas ferramentas, bem como os seus componentes comuns (*UmlPerf, PerformanceTesting, TestCaseGenerator e FiniteStateMachineHsi*) e específicos (*LoadRunnerScript, LoadRunnerParameters, VisualStudioScript e VisualStudioParameters*). Baseado no fato que essas ferramentas compartilham um conjunto comum de componentes e que cada passo da nossa abordagem é implementada em um componente, ambas as ferramentas aceitam modelos UML como entrada (componente *UMLPer* - ver Figura 4 e 5) e geram automaticamente os casos de teste abstratos (componentes *FiniteStateMachine, Performancetesting e TestCaseGenerator*). Por sua vez, os componentes específicos, que são relacionados a uma ferramenta de geração de carga, possuem a funcionalidade de gerar os *scripts* e realizar a execução do teste. Portanto, a fim de gerar a ferramenta PLeTsPerfLR foram desenvolvidos os componentes *LoadRunnerScript* e *LoadRunnerParameters*. Para gerar a ferramenta PLeTsPerfVS, foi necessário desenvolver dois componentes específicos: *VisualStudioScripts* e *VisualStudioParameters* (Figura 5). Em relação ao esforço necessário para desenvolver os componentes, é importante ressaltar que os componentes comuns possuem 1.646 linhas de código, os componentes específicos da PLeTsPerfLR tem 258 linhas de código e os componentes específicos da PLeTsPerfVS têm 344 linhas de código. Estes números indicam que o uso de nossa LPS para gerar ferramentas TBM pode reduzir o esforço de desenvolvimento dessas ferramentas, *e.g.* o número de linhas de código necessárias para desenvolver a PLeTsPerfVS teve uma redução de quase 80%.

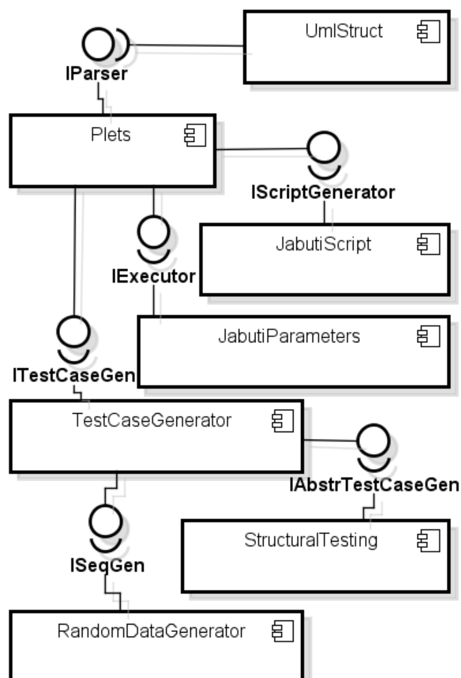

Figura 6. Arquitetura da ferramenta PletsStructJabuti

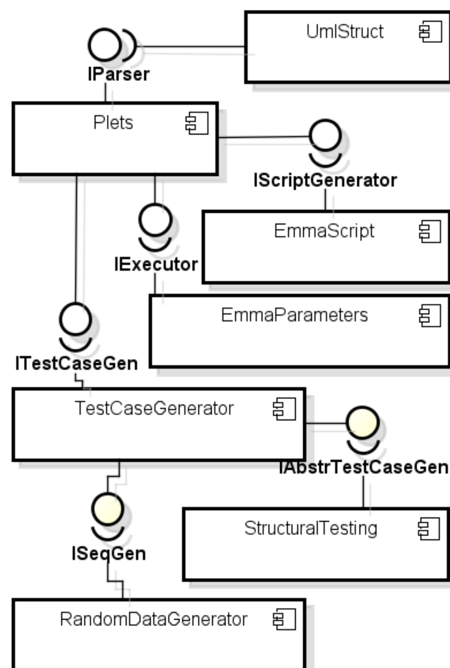

Figura 7. Arquitetura da ferramenta PletsStructEmma

4.2. Geração de Ferramentas TBM para Teste de Estrutural

Além da geração das ferramentas que usam TBM para aplicar teste de desempenho, também foram geradas duas ferramentas que usam TBM para aplicar teste estrutural: PletsStructJabuti e PletsStructEmma. A PletsStructJabuti é uma ferramenta que usa TBM para gerar dados de teste a partir de diagrama de sequência da UML, que são então executados pela ferramenta de teste estrutural Jabut. Por sua vez a PletsStructEmma gera dados para a ferramenta Emma. As Figuras 6 e 7 apresentam as arquiteturas da ferramentas e consequentemente permite indentificar os componentes comuns (*UmlStruct, TestCaseGenerator, RandomDataGenerator e StructuralTesting*) e específicos (*JabutiScript, EmmaScripts, JabutiParameters e EmmaParameters*). Assim como nas ferramentas de teste de desempenho, os componentes específicos são relacionados a uma ferramenta de teste estrutural e possuem a funcionalidade de gerar os dados de teste no formato requerido pela ferramenta externa e realizar a execução do teste.

Desta forma, foi necessário desenvolver os componentes específicos *EmmaScript, EmmaParameters* para a ferramenta PLeTsStructEmma e *JabutiScript, JabutiParameters* para a ferramenta PLeTsStructJabuti. Apesar das ferramentas de teste estrutural não possuírem componentes comuns com as ferramentas de teste de desempenho, o esforço requerido para o desenvolvimento das ferramenta TBM para teste estrtural a partir da PLeTs é significativamente menor do que desenvolver as ferramentas isoladamente. Por exemplo, os componentes comuns das ferramentas estruturais tem 731 linhas de código. Por sua vez, os componentes específicos da ferramentas PLeTsStructEmma tem apenas 355 linhas de código e os componentes específicos da PLeTsStructJabuti têm 354 linhas de código.

5. Conclusão

Neste artigo foram apresentados os requisitos básicos e as decisões de projeto no desenvolvimento de uma linha de produtos de *software* de ferramentas de testes baseados em modelos. Inicialmente, foram indentificados com base em uma MSL e em cooperação com um LDT de uma empresa de TI quais os requisitos básicos e as quais características uma ferramenta de teste que usa TBM deveria possuir. Com base nessas informações foi definido um processo MBT, bem como identificadas a variabilidade e as restrições entre as características. O Próximo passo foi a definição de um mecanismo de variabilidade, projetar a arquitetura e mapear cada característica a um artefato de software (componente). É importante destacar que a escolha do mecanismo de variabilidade e a definição da notação para representar a arquitetura foram realizados tendo por objetivo automatizar e simplificar o processo de geração das ferramentas. A simplicidade e a automatização do projeto, codificação dos componentes e derivação das ferramentas é fundamental em qualquer cenário, mas é especialmente importante no cenário onde a PLeTs tem por objetivo ser utilizada: equipes de teste que desenvolvem e evoluem suas próprias ferramentas de teste.

Com base nesse cenário foram conduzidos dois exemplos de uso, onde foram geradas quatro ferramentas, duas para executar teste de desempenho e duas para teste estrutural. Com base no esforço (linhas de código) que foi necessário para desenvolver os componentes especifico de uma das ferramentas podemos identificar um ganho significativo.

Como trabalhos futuros pode-se destacar a necessidade de se investigar de maneira mais aprofundada o esforço requerido para o desenvolvimento de novas variantes de ferramentas. Atualmente já se encontra em desenvolvimento uma ferramenta MBT para apoiar teste funcional, cujos resultados de esforço serão avaliados e comparados com os resultados apresentados nesse trabalho. É importante destacar que o LDT está atual executando um projeto piloto para avaliar a adoção de ferramentas geradas a partir da PLeTs por suas equipes de teste.

Referências

Abbors, F., Backlund, A., and Truscan, D. (2010). MATERA - An Integrated Framework for Model-based Testing. In *Proceedings of the 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 321–328.

Apache. JMeter Performance Test.

Bass, L., Clements, P., and Kazman, R. (1997). *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc.

Beuche, D. (2012). Modeling and building software product lines with pure::variants. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc.

Costa, L. T. (2012). Conjunto de Características para Teste de Desempenho: Uma Visão a Partir de Ferramentas. Master's thesis, Pontificia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil.

DeMillo, R., Lipton, R., and Sayward, F. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 1:34–41.

Dias Neto, A. C., Subramanyan, R., Vieira, M., and Travassos, G. H. (2007). A Survey on Model–based Testing Approaches: A Systematic Review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pages 31–36.

El-Far, I. K. and Whittaker, J. A. (2001). Model-based Software Testing. In Marciniak, J., editor, *Encyclopedia of Software Engineering*, pages 825–837. Wiley.

Harrold, M. J. (2000). Testing: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, page 381.

Hartman, A. and Nagin, K. (2004). The AGEDIS Tools for Model Based Testing. *SIGSOFT Software Engineering Notes*, 29:129–132.

Hewlett Packard - HP. Software HP LoadRunner.

Huima, A. (2007). Implementing Conformiq Qtronic. In Petrenko, A., Veanes, M., Tretmans, J., and Grieskamp, W., editors, *Testing of Software and Communicating Systems*, pages 1–12. Springer Berlin Heidelberg.

Krueger, C. and Clements, P. (2012). Systems and software product line engineering with BigLever software Gears. In *Proceedings of the 16th International Software Product Line Conference*.

Linden, F. J., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer–Verlag New York, Inc.

Myers, G. J. and Sandler, C. (2004). *The Art of Software Testing*. John Wiley & Sons.

Northrop, L. M. (2002). SEI's Software Product Line Tenets. *IEEE Software*, 19:32–40.

Oliveira, E. A., Gimenes, I. M. S., and Maldonado, J. C. (2010). Systematic Management of Variability in UML-based Software Product Lines. *Journal of Universal Computer Science*, 16:2374–2393.

Perez, J. and Guckenheimer, S. (2006). *Software Engineering with Microsoft Visual Studio Team System*. Pearson Education.

Petrenko, A., Yevtushenko, N., Lebedev, A., and Das, A. (1993). Nondeterministic State Machines in Protocol Conformance Testing. In *Proceedings of the 6th International Workshop on Protocol Test Systems VI*, pages 363–378.

Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer–Verlag New York, Inc.

Rapps, S. and Weyuker, E. J. (1985). Selecting Software Test Data Using Data Flow Information. *Transactions on Software Engineering*, 11:367–375.

Rodrigues, E. M. (2013). *PLeTs - A Product Line of Model-based Testing Tools*. PhD thesis, Pontificia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil.

Rodrigues, E. M., Passos, L., Teixeira, L., Oliveira, F., Zorzo, A. F., and Saad, R. (2014). On the requirements and design decisions of an in-house component-based spl automated environment. In *25th International Conference on Software Engineering and Knowledge Engineering*, pages 1–6.

Rodrigues, E. M., Viccari, L. D., and Zorzo, A. F. (2010). PLeTs - Test Automation using Software Product Lines and Model Based Testing. In *Proceedings of the 22th International Conference on Software Engineering and Knowledge Engineering*, pages 483–488.

Rodrigues, E. M., Zorzo, A. F., Oliveira Junior, E. A., Souza, I. M. G., Maldonado, J. C., and Domingues, A. R. P. (2012). Plugspl: An automated environment for supporting plugin-based software product lines. In *24th International Conference on Software Engineering and Knowledge Engineering*, pages 647–650.

Roubtsov, V. EMMA: a Free Java Code Coverage Tool.

SEI. Software Engineering Institute - Case Studies.

Shafique, M. and Labiche, Y. (2010). A Systematic Review of Model-based Testing Tool Support. Technical report, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada.

Silveira, M. B. (2012). Conjunto de características para teste de desempenho : uma visão a partir de modelos. Master's thesis, Pontificia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil.

Silveira, M. B., Rodrigues, E. M., Zorzo, A. F., Vieira, H., and Oliveira, F. (2011). Model-based Automatic Generation of Performance Test Scripts. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, pages 1–6.

Sommerville, I. (2011). *Software Engineering*. Pearson/Addison–Wesley.

Thum, T., Kastner, C., Erdweg, S., and Siegmund, N. (2011). Abstract Features in Feature Modeling. In *Proceedings of the 2011 15th International Software Product Line Conference*, pages 191–200.

Utting, M. and Legeard, B. (2006). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann.

Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., and Nachmanson, L. (2008). Model–based Testing of Object-oriented Reactive Systems with Spec Explorer. In Hierons, R. M., Bowen, J. P., and Harman, M., editors, *Formal methods and testing*, pages 39–76. Springer–Verlag.

Vincenzi, A., Maldonado, J., Wong, W., and Delamaro, M. (2005). Coverage Testing of Java Programs and Components. *Science of Computer Programming*, 56:211–230.

Weiss, D. M. and Lai, C. T. R. (1999). *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc.