Uma Abordagem para Geração de Casos de Teste Estrutural Baseada em Modelos

Leandro T. Costa, Flávio M. Oliveira, Elder M. Rodrigues, Maicon B. da Silveira, Avelino F. Zorzo

[1]Programa de Pós-Graduação em Ciência da Computação
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Caixa Postal 1.429 – 90.619-900 – Porto Alegre – RS – Brasil

leandro.teodoro@acad.pucrs.br, bernardino@acm.org,

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

Abstract. *Structural testing, or white-box testing, is a technique to generate test cases based on analysis of an application source code. Currently, there are a lot of tools that perform this type of test, e.g. JaBUTi and Poke-Tool. However, despite the benefits of these tools, some tasks still have to be performed manually, e.g. the description of the test cases. This makes the test process inefficient and prone to faults. Thus, this paper describes a technology-independent test case format, i.e., an abstract structure for white-box testing. We describe also how this abstract structure is inserted into a process that uses it to, automatically, generate and execute concrete test cases in a specific structural testing tool*[1][2].

Resumo. *O teste estrutural, ou teste caixa-branca, é uma técnica para gerar casos de teste a partir da análise do código fonte. Atualmente, existem muitas ferramentas baseadas neste tipo de teste, e.g. JaBUTi e Poke-Tool. No entanto, apesar dos benefícios destas ferramentas, é necessário executar algumas tarefas manualmente, e.g. a elaboração de casos de teste. Isto torna o processo de teste ineficiente e suscetível a falhas. Desta forma, este artigo apresenta um formato para casos de testes independente de tecnologia, i.e., uma estrutura abstrata para teste estrutural. O artigo descreve, também, como esta estrutura abstrata é inserida em um processo que a usa para, automaticamente, gerar e executar casos de teste concretos em uma tecnologia de teste estrutural específica.*

1. Introdução

A evolução e incremento da complexidade dos sistemas computacionais existentes têm tornado o processo de teste destes sistemas uma atividade tão ou mais complexa que o processo de desenvolvimento em si. Para contornar os problemas decorrentes do aumento da complexidade dos sistemas, e aumentar a eficiência no processo de geração de testes das aplicações, diversas ferramentas foram desenvolvidas com o objetivo de automatizar os processos de avaliação e verificação de *software*. Atualmente, existem diversas ferramentas que avaliam a funcionalidade, desempenho, disponibilidade e escalabilidade das

[1]A ordem dos autores é meramente alfabética.

[2]Study developed by the Research Group of the PDTI 001/2012, financed by Dell Computers of Brazil Ltd. with resources of Law 8.248/91.

aplicações. Dentre elas citam-se Apache JMeter [Halili 2008], IBM Rational Performance Tester (RPT) e JaBUTi (*Java Bytecode Understanding and Testing*) [Eler et al. 2009].

No entanto, apesar dos benefícios advindos do uso de ferramentas para automatização da execução dos testes, ainda é necessária a execução de algumas atividades de maneira manual ou semi-automatizada como, por exemplo, a etapa de elaboração dos casos de teste. Esta geração manual ou semi-automatizada torna o processo de teste ineficiente e suscetível à inserção de falhas[3] nestes casos de teste até mesmo por profissionais experientes. Uma alternativa para melhorar o uso destas ferramentas é automatizar o processo de geração de casos de teste por meio de técnicas de teste de *software*. Uma das técnicas que auxiliam o processo de teste de *software* é o Teste Baseado em Modelos *(Model-Based Testing* - MBT) [Krishnan 2004]. Esta técnica consiste na geração dos casos de teste e/ou *scripts* de teste baseado no modelo do *software*. Modelo este que, inclui a especificação das características que serão testadas [Krishnan 2004]. Além disso, o uso de MBT apresenta várias vantagens, *e.g.* a probabilidade de redução da má interpretação dos requisitos do sistema por um engenheiro de teste e/ou a redução do tempo do teste.

Neste contexto, este artigo apresenta uma abordagem relacionada à automatização de casos de teste estrutural para diversas tecnologias a partir de diagramas de sequência UML. Esta abordagem consiste em três etapas: (a) anotação dos diagramas de sequência com as informações das classes e métodos que se deseja testar; (b) geração automática de uma estrutura abstrata de teste estrutural, que possui um formato genérico e descreve as informações do teste extraídas dos diagramas de sequência criados; (c) gerar automaticamente casos de teste para uma tecnologia de teste específica a partir das informações presentes na estrutura abstrata de teste estrutural.

Uma das vantagens desta abordagem está relacionada à capacidade de reuso das informações de teste. Assim, as informações descritas na estrutura de teste, originada a partir desta abordagem, podem ser utilizadas para gerar casos de teste para diversas tecnologias, *e.g.* acadêmicas: JaBUTi [Eler et al. 2009], Poke-Tool [Chaim 1991]; comerciais [Yang et al. 2006]: Semantic Designs Test Coverage, IBM Rational PurifyPlus. Desta forma, uma empresa pode se adaptar mais rapidamente a uma eventual migração de tecnologia de teste, pois devido ao fato de seus engenheiros de teste não terem a necessidade de criar casos de teste manualmente, não necessitam de um conhecimento aprofundado referente à nova tecnologia. Outra vantagem de nossa abordagem está relacionada à utilização de modelos UML para a geração e execução de casos de teste. Modelos provêem uma representação das informações de teste em alto nível, facilitando o entendimento por parte do analista, ou dos responsáveis pela implementação e execução dos casos de teste.

Com base nesta abordagem foi desenvolvida uma ferramenta capaz de gerar de forma automatizada a estrutura abstrata a partir de diagramas de sequência e, instanciar as informações presentes nesta estrutura para automaticamente gerar e executar casos de teste concretos para uma tecnologia de teste específica, a JaBUTi. É importante ressaltar que a ferramenta apresentada neste artigo foi gerada a partir de uma linha de produto denominada PLeTs PL [Rodrigues et al. 2010]. A PLeTs PL é uma linha de produto de ferramentas de teste capaz de gerar produtos (ferramentas) que automatizam a geração e execução de casos de teste aplicando MBT.

[3]Neste artigo utilizamos falha (*fault*), erro (*error*) e defeito (*failure*) segundo [Avizienis et al. 2004].

Este artigo está estruturado da seguinte forma: a Seção 2 faz uma breve introdução dos conceitos de teste baseado em modelos, teste estrutural e linha de produto de *software*. A Seção 3 apresenta detalhadamente a abordagem deste trabalho, bem como as funcionalidades da ferramenta que implementa a abordagem descrita. A Seção 4 apresenta a definição de alguns casos de teste para a nossa ferramenta utilizando uma aplicação que gerencia habilidades de uma empresa em uma empresa de TI como exemplo de uso. Ao final, a Seção 5 apresenta as conclusões e trabalhos futuros.

2. Fundamentação Teórica

Nesta seção serão apresentados os conceitos relativos a teste baseado em modelos, teste estrutural e linha de produto de *software*. Além disso, será apresentada a descrição da linha de produto de *software* que está diretamente relacionada com o tema abordado neste trabalho, *i.e.* PLeTs PL [Rodrigues et al. 2010].

2.1. Teste Baseado em Modelos

Teste Baseado em Modelos (*Model-Based Testing* - MBT) é uma técnica para a geração automática de artefatos de teste com base em modelos extraídos de artefatos de *software* [Dalal et al. 1999]. O principal objetivo de MBT é a criação de artefatos de teste que descrevam os requisitos e comportamento do próprio sistema, visando automação do processo de teste de *software*. Assim, o teste de *software* baseado em MBT requer a elaboração de modelos de teste para determinar parte de esforço do teste. Engenheiros de teste, quando elaboram os casos de testes do sistema, implicitamente, constroem estes modelos mentalmente [Sarma et al. 2010].

Modelos de teste possibilitam encapsular o comportamento e a estrutura do sistema, a fim de que sejam compartilhados e reutilizados pela equipe de teste. Desta forma, a partir destes modelos, é possível extrair as informações anotadas para a geração de novos artefatos de teste, tais como casos de teste, *scripts* e cenários de teste [El-Far and Whittaker 2001]. Uma vez desenvolvido o modelo, ele pode ser utilizado de várias formas e por várias etapas do processo de desenvolvimento de *software*, tais como: especificação dos requisitos, geração de código, análise de confiabilidade do sistema e execução de testes [Broy et al. 2005].

O custo do teste de *software* está relacionado ao número de interações e casos de teste que são executados durante o processo de desenvolvimento. Como ele é um dos processos mais onerosos e caros do desenvolvimento de *software* [Krishnan 2004], MBT é uma ótima abordagem para mitigar este problema, automatizando o processo de geração de casos de teste ou *scripts* com o objetivo de reduzir tempo e custo do processo de teste. Entretanto, no contexto da automatização da geração de casos de teste raramente empresas adotam MBT em suas atividades, permanecendo a maioria delas no processo manual do teste [Sarma et al. 2010].

A adoção da abordagem MBT exige a criação de modelos formais baseados nos requisitos especificados por engenheiros e analistas de testes. O objetivo é que estes modelos possam detectar informações que na maioria das vezes estão implícitas em documentos tradicionais de especificação, anotando comentários e estereótipos ao modelo que enriquecem a qualidade da especificação. Desta forma, estas informações incrementadas ao modelo servirão para a criação de novos artefatos, ou ainda, permitindo a

automatização de outros processos para melhorar a comunicação da equipe e a qualidade dos artefatos desenvolvidos. Esta modelagem pode ser implementada em vários contextos, pois a abordagem de MBT se aplica a diversos tipos e/ou níveis de teste de *software* [Dias Neto et al. 2007].

Os primeiros estudos que abordam modelagem de teste de *software* se limitavam à técnica funcional, *i.e.* teste caixa preta, descrevendo os aspectos funcionais a serem testados. Atualmente, os modelos são capazes de abstrair inúmeras outras informações, *e.g.* parâmetros e dados de entrada do teste, permitindo ser aplicada a técnica MBT para realização de outros tipos de testes, *e.g.* o teste estrutural [Delamaro et al. 2007].

2.2. Teste Estrutural

O teste estrutural é uma técnica que tem por objetivo a geração de casos de teste a partir da análise do código fonte. Busca avaliar os detalhes internos da implementação, tais como teste de condição, caminhos lógicos, etc. Por este motivo, é também chamado de teste orientado à lógica ou teste caixa-branca (*white-box*) [Delamaro et al. 2007]. Como citado anteriormente, a técnica de teste estrutural se baseia na análise da estrutura do programa para a geração dos casos de teste. Esta técnica define um conjunto de critérios estruturais, os quais se baseiam em diferentes elementos de um programa para definir requisitos de teste. Segundo [Delamaro et al. 2007], estes critérios tem por objetivo a execução de componentes e partes elementares de um programa e, são classificados basicamente em critérios baseados no(a):

- *Fluxo de controle:* utiliza análise do fluxo de controle do programa para gerar casos de teste. Utiliza os aspectos de controle do programa, tais como, laços, desvios ou condição para derivar os casos de teste;
- *Fluxo de dados:* utiliza análise do fluxo de dados do programa para gerar casos de teste. Casos de testes são derivados a partir de associações entre definições de variáveis e o uso dessas variáveis;
- *Complexidade:* fundamentam-se em informações de complexidade do programa para levantamento dos requisitos de teste. O critério de teste de caminho é um dos mais conhecidos critérios de complexidade.

Todavia, a maioria dos testes é baseada nas especificações do *software*, o que não é o caso do teste estrutural, em que são avaliados os diferentes caminhos que o programa pode executar, a partir de sua implementação. Muitas vezes, apesar de testados os possíveis caminhos mapeados de execuções de um programa, isso não é garantia de que o programa não possa vir a apresentar defeitos, pois algum caminho pode não ter sido percorrido. E ainda, a execução de um comando ou programa com falha, infelizmente, pode não resultar na geração de defeitos percebidos pelo usuário final [Pezzè and Young 2008]. Entretanto, o teste estrutural é relevante para fatores de qualidade do *software* como manutenabilidade, estrutura e confiança, pois inclui casos de teste que não são avaliados através de testes funcionais, *e.g.* identificar se todo o código foi executado.

2.3. Linha de Produto de *Software*

Uma Linha de Produto de *Software* (*Software Product Line - SPL*) é definida como um conjunto de sistemas que compartilham características comuns e gerenciáveis com o intuito de satisfazer as necessidades de um domínio específico, podendo este ser um segmento de mercado ou missão [Clements and Northrop 2001]. O objetivo é explorar as

semelhanças entre os sistemas visando gerenciar os aspectos relativos à variabilidade entre eles e, dessa forma, determinar uma maior reusabilidade dos componentes de *software*. Por meio desta capacidade de reutilização de componentes, uma SPL possibilita criar um conjunto de sistemas similares, reduzindo assim o tempo de comercialização, custo e com isso, obter maior produtividade e melhoria da qualidade.

A parte mais importante de uma SPL diz respeito ao núcleo de artefatos, o qual forma a base de uma SPL e pode ser formado por componentes reusáveis, modelos de domínios, requisitos da SPL, casos de teste e modelo de características (*feature models*). O modelo de características apresenta todas as características de uma linha de produto e a relação entre os componentes. O modelo de características de uma SPL é responsável por representar os aspectos relacionados à variabilidade, a qual pode estar vinculada a diferentes níveis de abstração como código fonte e documentação. As variabilidades são representadas por pontos de variabilidades (*variation points*) e variantes (*variants*), onde um ponto de variabilidade pode conter uma ou mais variantes. Em uma linha de produto de telefones móveis, por exemplo, um ponto de variabilidade poderia ser o protocolo de comunicação e suas variantes poderiam ser GSM (*Global System for Mobile Communications*) ou UMTS (*Universal Mobile Telecommunication System*).

No contexto de teste, atualmente, está em desenvolvimento no Centro de Pesquisa em Engenharia de Sistemas da PUCRS uma linha de produto denominada PLeTs PL [Rodrigues et al. 2010]. A PLeTs PL é uma SPL que busca facilitar a derivação de ferramentas de teste baseado em modelos, com os quais é possível criar e executar casos de teste de forma automatizada. O objetivo desta SPL não é apenas gerenciar a reutilização de artefatos e componentes de *software*, mas também tornar mais fácil e rápido o desenvolvimento de uma nova ferramenta que, como citado anteriormente, tem por objetivo otimizar a criação e execução de casos de teste. Para melhor entendimento, é apresentado na Figura 1 o modelo de características atual da PLeTs PL. Atualmente, o modelo de características da PLeTs PL é constituído de quatro características em seu primeiro nível:

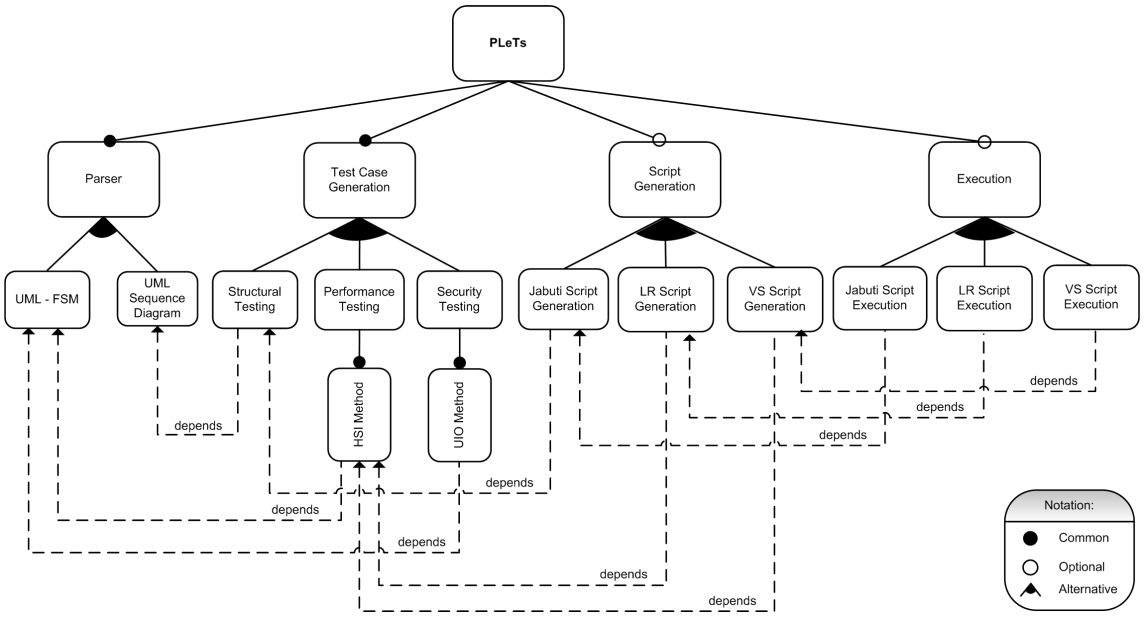

Figura 1. Modelo de características da PLeTs PL [Rodrigues et al. 2010]

1) *Parser*: esta é uma característica ou ponto de variabilidade obrigatório e seu objetivo é extrair as informações contidas, por exemplo, em um arquivo UML (Diagrama de Sequência, Diagrama de Atividades) para então gerar um modelo formal na característica `Test Case Generation`. Atualmente, esta característica possui duas variantes mutuamente excludentes: `UML - FSM (Finite State Machine)` e `UML - Sequence Diagram`, onde apenas uma das duas variantes é selecionada; 2) *Test Case Generation*: também é uma característica obrigatória. É responsável por gerar sequências de teste a partir das informações contidas no modelo formal. Podem ser geradas sequências para teste estrutural, desempenho ou segurança. Este ponto de variabilidade possui três variantes (`Structural Testing`, `Performance Testing` e `Security Testing`) e obrigatoriamente, assim como na característica `Parser`, apenas uma delas pode ser selecionada. Como pode-se observar na Figura 1, as três variantes possuem uma relação de dependência (*depends*) com as variantes do `Parser`. Indicando que a seleção de uma variante é dependente de outra. Por exemplo, caso a variante `Structural Testing` seja escolhida, obrigatoriamente a variante `UML - Sequence Diagram` deve ser selecionada; 3) *Script Generation*: consiste na geração de *scripts* de teste para ferramentas de automação de teste. Também possui três variantes, sendo elas: `Jabuti Script Generation`, `LR Script Generation` e `VS Script Generation`, as quais representam a implementação de *scripts* e cenários de teste[4], respectivamente, para as ferramentas JaBUTi, LoadRunner e Visual Studio; 4) *Execution*: representa a execução da ferramenta e também a execução do teste sob a aplicação a ser testada, utilizando uma determinada ferramenta com os *scripts* e cenário de teste gerados na etapa anterior. Possui três variantes e cada uma possui uma relação de dependência com as variantes do ponto de variabilidade `Script Generation`.

A relação da PLeTs PL com este trabalho diz respeito ao fato de que um dos produtos gerados por esta SPL refere-se à ferramenta abordada neste artigo. Portanto, nas seções seguintes serão apresentadas de forma detalhada as funcionalidades desta ferramenta, bem como a abordagem que a originou.

3. Geração de Casos de Teste Estrutural

Em trabalhos anteriores [Silveira et al. 2011] [Rodrigues et al. 2010] foram apresentadas abordagens com as quais é possível automatizar a geração e execução de casos de teste de desempenho de *software*. Nesse trabalho será apresentada a aplicação de uma abordagem que, diferentemente das anteriores, tem por objetivo automatizar a geração e execução de casos de teste estrutural a partir de diagramas de sequência.

A aplicação de nossa abordagem é dividida em três etapas (ver Figura 2). Inicialmente, se faz necessária a anotação do diagrama de sequência que descreve as classes e métodos da aplicação que se deseja testar (ver Figura 2 - a). É importante que este diagrama esteja "completo", contendo informações referentes aos parâmetros (nome, tipo) destas classes e métodos, bem como o tipo de retorno de cada método. Caso estas informações não estejam explícitas, o diagrama deve ser anotado adicionando tais informações. Desta forma, o próximo passo diz respeito à anotação deste diagrama com informações que serão usadas para a etapa seguinte de nossa abordagem (maiores detalhes a respeito de como o diagrama é anotado serão descritos na Seção 4). Para anotar

[4]Cenários de teste contém um ou mais *scripts* e possuem informações referentes à configuração do teste.

estas informações é utilizado o seguinte conjunto de *tags*:

<<**TDexternalLibray**>>: especifica o caminho das bibliotecas da aplicação sob teste;
<<**TDclassPath**>>: especifica o caminho das classes que serão testadas;
<<**TDtechnologyPath**>>: especifica informações particulares à tecnologia de teste escolhida, *e.g.*, diretório de instalação e o caminho de seu arquivo executável;
<<**TDimportList**>>: especifica uma lista de classes importadas (*import, package*).

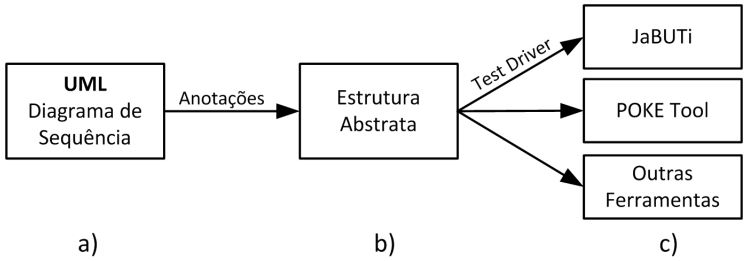

Figura 2. Abordagem para a geração de casos de testes estrutural

Após este processo de anotação do diagrama, é gerado um arquivo no formato *XMI* contendo as informações descritas no diagrama de sequência anotado. A segunda etapa de nossa abordagem consiste em submeter o arquivo *XMI* a um *parser*, o qual é responsável por extrair os dados presentes neste arquivo para então gerar uma estrutura abstrata de teste (ver Figura 2 - b). Esta estrutura, a qual chamamos de `Estrutura Abstrata`, se trata de um arquivo de texto que descreve em um formato genérico e sequencial todo o fluxo de dados das classes e métodos da aplicação que se deseja testar.

A vantagem de utilizar esta estrutura está relacionada à capacidade de reuso das informações nela descritas por diversas tecnologias de execução de teste estrutural, *e.g.* JaBUTi [Eler et al. 2009], Poke-Tool [Chaim 1991]. Neste sentido, caso uma empresa de TI decida, devido a uma estratégia administrativa, migrar para uma tecnologia de teste estrutural diferente da que possui, poderá aproveitar os casos de teste previamente definidos. Além de facilitar uma migração de tecnologia, a estrutura abstrata apresenta as informações do teste em um formato claro, tornando simples e fácil o entendimento dos casos de teste definidos.

A última etapa de nossa abordagem diz respeito à instanciação da estrutura abstrata para determinada tecnologia e a configuração de informações particulares à tecnologia de teste escolhida (ver Figura 2 - c). Esta etapa será descrita com maiores detalhes nas próximas seções, onde será apresentado um exemplo real com a estrutura abstrata sendo instanciada para gerar casos de teste para determinada aplicação. Ainda que nossa abordagem possa suportar diversas tecnologias, para este trabalho foi utilizada a ferramenta JaBUTi. Portanto, será apresentada a nossa ferramenta, a qual é capaz de gerar e executar automaticamente casos de teste para a JaBUTi a partir de informações descritas em diagramas de sequência.

3.1. Geração de Casos de Teste para a JaBUTi

Desenvolvida pelo grupo de Engenharia de *Software* da USP-São Carlos, a JaBUTi [Eler et al. 2009] (*Java Bytecode Understanding and Testing*) é uma ferramenta utilizada para execução de teste estrutural em aplicações Java. Ao contrário de outras ferramentas de teste estrutural, a JaBUTi é capaz de executar testes sem que seja necessária uma

análise do código fonte, pois a execução de testes pela ferramenta é realizada a partir da análise do Java *Bytecode*. Para a análise de cobertura das classes e componentes dos programas a serem testados, a Jabuti se baseia em oito critérios de teste estrutural, sendo quatro critérios de análise de fluxo de dados e outros quatro para análise de fluxo de controle [Vincenzi. et al. 2005].

Para tornar possível a geração automática de casos de teste com a JaBUTi foi necessária uma análise para entender o funcionamento interno da ferramenta. Para a execução de casos de teste com a JaBUTi é necessário criar um arquivo de projeto, cuja extensão é `.jbt`. Todas as informações referentes ao *bytecode* das classes que serão testadas e o caminho dessas classes são armazenadas neste arquivo. Neste arquivo também são descritos os caminhos referentes a todas as bibliotecas que pertencem à aplicação a ser testada, bem como o *bytecode* da classe `TestDriver`, que contém informações que serão utilizadas para testar as classes e métodos da aplicação sob teste. Com base nas informações do *bytecode* das classes, a JaBUTi constrói o grafo definição-uso (*Def-Use Graph* - DUG) para cada uma dessas classes.

Após a criação do arquivo de projeto, a JaBUTi realiza a instrumentação das classes a serem testadas. Ela executa os casos de teste, descritos no arquivo `TestDriver.java`, por meio da chamada do método `probe.DefaultProber.probe` que armazena as informações do programa que será testado. Em seguida, outro método é chamado (`probe.DefaultProber.dump`) e todos os dados coletados na chamada do método anterior são armazenados em um arquivo de rastro (`.trc`). Os dados gravados no arquivo de rastro correspondem aos caminhos percorridos pelo programa durante a execução do teste. A JaBUTi extrai as informações deste arquivo de rastro, atualiza os dados do teste e recalcula as informações de cobertura.

De posse desse conhecimento, foi possível verificar que nossa ferramenta é capaz de automatizar a geração e execução de casos de teste com a JaBUTi por meio da criação de dois arquivos. Estes arquivos se tratam da `TestDriver.java` e do arquivo de projeto da ferramenta, sendo que ambos são gerados automaticamente a partir das informações contidas na estrutura abstrata. Uma vez que estes dois arquivos são gerados, a execução dos casos de teste consiste na chamada interna dos métodos `probe.DefaultProber.probe` e `probe.DefaultProber.dump`. Ao final, a nossa ferramenta cria um processo Java para executar a JaBUTi contendo as informações de cobertura já atualizadas.

4. Exemplo de Uso

Nesta seção será apresentada a definição de um conjunto de casos de teste para serem gerados e executados a partir da aplicação de nossa abordagem. Para isto, será utilizada como exemplo de uso uma aplicação denominada Skills, a qual tem por objetivo a gerência de perfis profissionais de funcionários de uma empresa. O objetivo é verificar e avaliar os aspectos estruturais a partir da ferramenta que implementa a abordagem apresentada na seção anterior.

4.1. Definição dos Casos de Teste

A aplicação utilizada como exemplo de uso para este trabalho é denominada Skills (*Workforce Planning: Skill Management Prototype Tool*) [Silveira et al. 2011] e tem por obje-

tivo gerenciar os perfis profissionais de funcionários de uma dada empresa. Esta aplicação foi desenvolvida por um grupo de pesquisa da PUCRS em colaboração com uma empresa de tecnologia e tem como principal funcionalidade o gerenciamento do cadastro de habilidades, certificações e experiências de funcionários.

Com o intuito de verificar os aspectos funcionais de nossa abordagem, foi definido um caso de teste para ser executado pela nossa ferramenta. Para isso, foi definido um caso de teste para a análise estrutural de alguns métodos da aplicação Skills. O caso de teste definido descreve o processo referente à busca de informações de determinada certificação por parte do usuário. As informações referentes ao caso de teste são representadas por um diagrama de sequência, o qual foi definido e implementado de forma manual (ver Figura 3). Como é possível visualizar neste diagrama, a interação do usuário com a aplicação inicia por meio da inserção de informações referentes ao nome da certificação e do fornecedor (`provider`). Em seguida, são chamados três métodos pertencentes à classe `ServletCertification`.

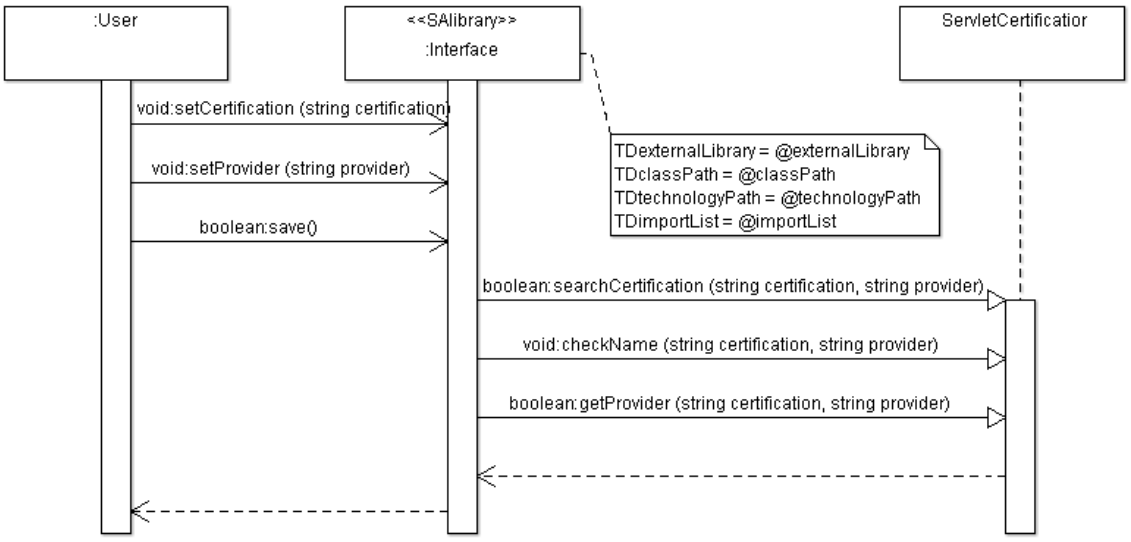

Figura 3. Diagrama de sequência para busca de certificação

O primeiro método (`searchCertification`) é responsável por acessar a base de dados e retorna se a certificação e `provider` informados pelo usuário foram cadastrados no sistema. O segundo método (`checkName`) faz a concatenação dos nomes da certificação e `provider`, enquanto o terceiro método (`getProvider`) verifica a existência de um `provider` dado um nome de certificação que é também composto pelo nome do `provider` (por exemplo, procura-se a certificação `Certified Scrum Master` do `provider` `CSM`, quando, na realidade, o nome da certificação é `CSM - Certified Scrum Master`). Ao final, são apresentadas ao usuário as informações referentes à data de aquisição e descrição da certificação informada previamente por ele.

Conforme descrito na Seção 3, o diagrama de sequência pode ser anotado com quatro *tags*: `TDexternalLibray`, `TDclassPath`, `TDtechnologyPath` e `TDimportList`. Para o nosso exemplo, estas quatro *tags* foram anotadas no diagrama de sequência, sendo que cada uma possui seu respectivo parâmetro, `@externalLibrary`, `@classPath`, `@technologyPath` e `@importList`. A

partir do diagrama de sequência é gerado um arquivo no formato *XMI*, o qual servirá como entrada para a execução de nossa ferramenta. Durante sua execução, a ferramenta submete o arquivo *XMI* a um *parser*, extraindo as informações dos métodos e da classe a serem testados para gerar a estrutura abstrata.

É importante destacar que nossa abordagem consiste na geração e execução automatizada de casos de teste onde somente são analisados os métodos das classes internas do sistema. Portanto, nenhum método chamado a partir da interação do usuário será analisado. Neste contexto, somente as informações referentes aos métodos `searchCertification`, `checkName` e `getProvider` da classe `ServletCertification` serão utilizadas para gerar automaticamente a estrutura abstrata. Como pode ser visto na Figura 4, a estrutura abstrata possui informações utilizadas para a geração de casos de teste, sendo dividida em três grupos: a) *Configuração da Tecnologia*: caracteriza o parâmetro `@technologyPath`, o qual especifica as informações da tecnologia a ser utilizada para o teste; b) *Configuração do Teste*: caracteriza os parâmetros `TDclassPath`, `@externalLibrary` e `@importList`, os quais definem as informações utilizadas para um caso de teste específico; c) *Configuração do Fluxo Sequencial*: caracteriza o fluxo sequencial dos métodos e classe a serem testados;

```
Estrutura abstrata: Busca Certificação
## Configuração da Tecnologia ##
Informações da Tecnologia : <<TDtechnologyPath: @technologyPath>>

## Configuração do Teste ##
Bibliotecas Externas : <<TDexternalLibray: @externalLibrary>>
Caminho das Classes : <<TDclassPath: @classPath>>
Classes Importadas  : <<TDimportList: @importList>>

## Configuração do Fluxo Sequencial ##
1. ServletCertification
1.1. searchCertification(string certification, string provider): boolean
1.2. checkName(string certification, string provider): void
1.3. getProvider(string certification, string provider): boolean
```

Figura 4. Estrutura abstrata

Cada parâmetro possui uma referência para um arquivo de dados contendo informações dos valores que serão utilizados para instanciar os casos de teste para determinada tecnologia. Portanto, quando a estrutura abstrata for instanciada para gerar os casos de teste para a JaBUTi, a classe `TestDriver.java` e o arquivo de projeto (`.jbt`) da ferramenta serão gerados com os valores descritos neste arquivo de dados. Este mesmo arquivo de dados também descreve um conjunto de informações que serão utilizadas como entrada dos métodos a serem testados. Na abordagem proposta nesse trabalho, a geração das informações destes dados de entrada é realizada, automaticamente, aplicando a técnica de geração aleatória de dados [Hamlet and Taylor 1990] a partir de um domínio finito de entradas. Esta técnica foi utilizada devido a sua praticidade, por ser menos custosa e mais simples de automatizar. Contudo, outras técnicas de geração de dados de teste são apresentadas na literatura, *e.g.* geração com execução simbólica [Lin et al. 2011] e geração com execução dinâmica [Dara et al. 2009].

Quanto à utilização de um arquivo de dados, uma de suas vantagens refere-se ao fato de que não é necessário alterar no diagrama de sequência, os valores dos parâmetros dos métodos a serem testados. Desta forma, para gerar casos de teste com diferentes

valores de entrada para os parâmetros dos métodos basta aplicar novamente a técnica de geração aleatória de dados de teste. Para exemplificar, a Figura 5 possui algumas informações referentes aos valores dos parâmetros presentes neste arquivo.

```
@technologyPath = C:\Jabuti\bin; C:\Jabuti\lib\bcel-5.2.jar; C:\Jabuti\lib\capi.jar; ...
@externalLibrary = C:\Apache Software Foundation\Tomcat 6.0\lib\jsp-api.jar; ...
@classPath = C:\Temp\Workspace\CmTool_SkillsTest\web\WEB-INF\classes; ...
@importList = servlets.*; java.io.*; java.util.StringTokenizer
1. ServletCertification
1.1. searchCertification("ActiveX", "BrainBench")
1.2. checkName("ActiveX", "BrainBench")
1.3. getProvider("ActiveX", "BrainBench")
```

Figura 5. Arquivo de dados

A partir das informações descritas na estrutura abstrata, são gerados o arquivo de projeto da JaBUTi e a classe `TestDriver.java`. Para gerar o arquivo de projeto, nossa ferramenta cria um processo Java que executa a classe `br.jabuti.cmdtool.CreateProject` da JaBUTi, onde as informações de tecnologia descritas no arquivo de dados são utilizadas como parâmetro para a execução desta classe. Por outro lado, a criação do arquivo `TestDriver.java` é menos complexa, sendo gerada a partir da estrutura abstrata com os valores extraídos do arquivo de dados. Entretanto, a `TestDriver.java` necessita ser compilada, pois como citado anteriormente a JaBUTi realiza a análise estrutural apenas a partir do *bytecode* das classes a serem testadas. Portanto, nossa ferramenta cria um processo Java para compilar este arquivo. Para exemplificar, as Figuras 6 e 7 apresentam, respectivamente, o arquivo `TestDriver.java` e o arquivo de projeto da JaBUTi.

```
import servlets.*;
import java.io.*;
import java.util.StringTokenizer;

public class TestDriver {
  static public void main(String args[]) throws Exception {
    ServletCertification servletcertification = new ServletCertification();
    servletcertification.searchCertification("ActiveX", "BrainBench");
    servletcertification.getProvider("ActiveX", "BrainBench");
    servletcertification.checkName("ActiveX", "BrainBench");
  }
}
```

Figura 6. Arquivo TestDriver.java

Ao final, a nossa ferramenta cria um processo Java para executar a JaBUTi, a qual é responsável por calcular e apresentar as informações de cobertura para o caso de teste definido. Na Figura 8 é apresentada a interface da JaBUTi com as informações dos percentuais de cobertura dos métodos testados, com base no critério estrutural `Todos os Nodos Primários` (*All-Nodes-ei*). De acordo com os resultados obtidos, os métodos `searchCertification`, `checkName` e `getProvider` foram, respectivamente, cobertos com percentuais de 52%, 100% e 38%. Como pode-se observar, os métodos `searchCertification` e `getProvider` não foram totalmente cobertos. Para aumentar o percentual de cobertura deste métodos diferentes entradas devem ser geradas.

```
<JABUTI>
    <PROJECT name="C:\Jabuti\JabutiProjects\Skills\Skills.jbt" type="research" mobility=
        "N" CFGOption="1">
        <BASE_CLASS name="TestDriver"/>
        <CLASSPATH path=". C?\Jabuti\bin C?\Temp\Workspace\CmTool_SkillsTest\web\WEB-INF
            \classes C?\Jabuti\lib\bcel-5.2.jar C?\Jabuti\lib\capi.jar C?\Jabuti\lib\
            junit-4.4.jar C?\Jabuti\lib\mucode.jar"/>
        <JUNIT_SRC_DIR dir=""/>
        <JUNIT_BIN_DIR dir=""/>
        <JUNIT_TEST_SET name=""/>
        <JUNIT_JAR name=""/>
        <AVOIDED_PACKAGES>
        </AVOIDED_PACKAGES>
        <CLASS name="TestDriver" size="0000" checksum="00000000">
            <EXTEND name="java.lang.Object" level="1"/>
        </CLASS>
        <INST_CLASS name="servlets.ServletCertification" size="9250" checksum="50-0">
            <EXTEND name="javax.servlet.http.HttpServlet" level="1"/>
            <SOURCE name="ServletCertification.java"/>
            <METHOD id="1" name="searchCertification(Ljava/lang/String;Ljava/lang/...)">
            ...
    </PROJECT>
</JABUTI>
```

Figura 7. Arquivo de projeto (`.jbt`)

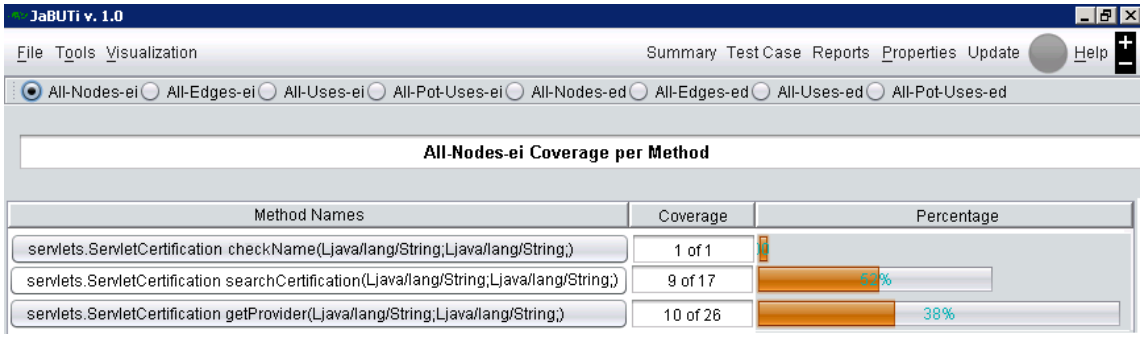

Figura 8. Interface da JaBUTi com as informações de cobertura dos métodos testados

5. Conclusão e Trabalhos Futuros

Este trabalho apresentou uma abordagem para automatizar o processo de geração e execução de casos de teste estrutural para diversas tecnologias a partir de diagramas de sequência UML. Com base nesta abordagem foi gerada uma ferramenta a partir de uma linha de produto de *software* que automatiza este processo de teste utilizando a JaBUTi.

Uma das vantagens em utilizar esta abordagem refere-se à capacidade de reuso das informações do teste descritas nos diagramas de sequência. Desta forma, uma empresa de TI pode facilmente migrar de uma tecnologia de teste para outra e aproveitar os casos de teste previamente definidos. A tecnologia de teste utilizada para exemplificar nossa abordagem foi a JaBUTi, porém tecnologias utilizadas na indústria como, por exemplo, Semantic Designs Test Coverage, IBM Rational PurifyPlus e também tecnologias acadêmicas como a Poke-Tool poderiam ser utilizadas para este fim. A Poke-Tool é capaz de prover análise estrutural de programas implementados na linguagem C++. Esta abordagem não seria aplicada para a análise de programas escritos essencialmente em C, pois ela se baseia na extração de informações de teste a partir de diagramas de sequência,

necessitando que os programas a serem testados tenham sido desenvolvidos utilizando o paradigma de programação orientada a objetos.

Outra vantagem de nossa abordagem refere-se à capacidade de integração de diferentes métodos para a geração de dados de teste. Desta forma, é possível realizar uma análise estrutural mais eficaz e que satisfaça os critérios estruturais mais relevantes a serem avaliados para o conjunto de classes e métodos que se deseja testar.

Apesar das vantagens adquiridas com a utilização de nossa abordagem, existem algumas questões em aberto e pontos que poderiam ser aprimorados. Ainda que nossa abordagem possibilite automatizar a geração e execução de casos de teste para diversas tecnologias de teste estrutural, é necessário um conhecimento aprofundado relacionado à tecnologia de automatização de teste que se deseja utilizar. É necessário conhecer os arquivos de configuração de teste da ferramenta e como as informações para o teste estão estruturadas nestes arquivos. Além disso, nossa abordagem poderia implementar outras técnicas de geração de dados de teste, *e.g.* geração com execução simbólica e geração com execução dinâmica, pois são mais eficazes e garantem a seleção de dados com maior probabilidade de revelar a existência de falhas.

Outra questão em aberto diz respeito à falta de informações de análise de resultados do teste. A abordagem apresentada permite gerar e executar casos de teste de forma automatizada, mas não dispõe de informações que podem contribuir na automatização da análise dos resultados. Por exemplo, esta abordagem poderia descrever quais as informações do teste são mais relevantes de serem consideradas para análise e qual relação entre esta ou aquela informação poderia contribuir em uma análise mais criteriosa do teste. Neste contexto, seria interessante que nossa ferramenta implementasse funções que desempenhassem o papel de oráculos, apresentando informações do teste relativas ao cumprimento ou não dos requisitos de teste especificados.

6. Agradecimentos

Avelino F. Zorzo possui bolsa de produtividade CNPq/Brasil. Elder M. Rodrigues possui bolsa CAPES/INCT-SEC. Agradecemos a DELL pelo apoio no desenvolvimento deste trabalho. Agradecemos também aos revisores do artigo.

Referências

Avizienis, A., Laprie, J. C., Randell, B., and Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transaction on Dependable Secure Computing*, 1:11–33.

Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., and Pretschner, A. (2005). *Model-Based Testing of Reactive Systems: Advanced Lectures*. Springer.

Chaim, M. L. (1991). POKE-TOOL: A Tool to Support Data Flow Based Structural Test of Programs. Master's thesis, DCA/FEEC/UNICAMP, Campinas, SP, Brasil.

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

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. (1999). Model-Based Testing in Practice. In *Proceedings of the International Conference on Software Engineering*, pages 285–294.

Dara, R., Li, S., Liu, W., Smith-Ghorbani, A., and Tahvildari, L. (2009). Using Dynamic Execution Data to Generate Test Cases. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 433–436.

Delamaro, M. E., Maldonado, J. C., and Jino, M. (2007). *Introdução ao Teste de Software*. Elsevier Editora.

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*. Wiley.

Eler, M. M., Endo, A. T., Masiero, P. C., Delamaro, M. E., Maldonado, J. C., Vincenzi, A. M. R., Chaim, M. L., and Beder, D. M. (2009). JaBUTiService: A Web Service for Structural Testing of Java Programs. In *Proceedings of the 33rd IEEE International on Software Engineering Workshop*, pages 69–76.

Halili, E. (2008). *Apache JMeter*. Packt Publishing.

Hamlet, D. and Taylor, R. (1990). Partition Testing does not Inspire Confidence (Program Testing). *IEEE Transactions on Software Engineering*, 16:1402–1411.

Krishnan, P. (2004). Uniform Descriptions for Model Based Testing. In *Proceedings of the Australian Software Engineering Conference*, pages 96–105.

Lin, M., Chen, Y., Yu, K., and Wu, G. (2011). Lazy Symbolic Execution for Test Data Generation. *IET Software*, pages 132–141.

Pezzè, M. and Young, M. (2008). *Teste e Análise de Software - Processos, Princípios e Técnicas*. John Wiley & Sons.

Rodrigues, E. M., Viccari, L. D., Zorzo, A. F., and Gimenes, I. M. (2010). PLeTs Tool - 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.

Sarma, M., Murthy, P. V. R., Jell, S., and Ulrich, A. (2010). Model-based testing in industry: a case study with two MBT tools. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 87–90.

Silveira, M. B., Rodrigues, E. M., Zorzo, A. F., Costa, L. T., Vieira, H. V., and de Oliveira, F. M. (2011). Generation of Scripts for Performance Testing Based on UML Models. In *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, pages 258–563.

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

Yang, Q., Li, J. J., and Weiss, D. (2006). A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 99–103.