Teste baseado em modelos para serviços RESTful usando máquinas de estados de protocolos UML

Pedro Victor Pontes Pinheiro

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura:________________________

Teste baseado em modelos para serviços RESTful usando máquinas de estados de protocolos UML

Pedro Victor Pontes Pinheiro

Orientador: **Prof. Dr. Adenilso da Silva Simão**

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

USP – São Carlos
Junho de 2014

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

P654t

Pinheiro, Pedro Victor Pontes
Teste baseado em modelos para serviços RESTful usando máquinas de estados de protocolos UML / Pedro Victor Pontes Pinheiro; orientador Adenilso da Silva Simão. -- São Carlos, 2014.
95 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2014.

1. teste baseado em modelos. 2. serviços RESTful. 3. arquitetura orientada a recursos. 4. máquina de estados de protocolos UML. 5. geração de casos de teste. I. Simão, Adenilso da Silva, orient. II. Título.

Aos meus pais, Pedro Góes e Consuelo
e minha irmã Paola.

Agradecimentos

- A toda minha família que sempre cuidou de mim, me deu amor, fez de tudo para que nada me faltasse e me ensinou a ser um bom cidadão deste mundo.
- A Nádila Karime que me acompanhou nesta jornada, me fez companhia, me deu forças e me ajudou quando eu estava pra baixo. Obrigado pelos momentos perfeitos e pelo carinho!
- Ao Prof. Dr. Adenilso da Silva Simão, pelos conselhos, compreensão, ensinamentos e paciência na orientação deste trabalho. Obrigado de coração por tudo! Admiro o sr não só como pessoa, mas como um grande profissional.
- Ao grande André Takeshi Endo, por toda a ajuda dada ao longo da elaboração e conclusão deste trabalho. Espero um dia me tornar tão inteligente e competente como você.
- Às minhas amizades por todos os momentos de descontração, carinho, felicidade e/ou ajuda no trabalho: Alan, Ariel, Davi, Sérgio, Vinicius, Rafael, Dário, Victor, Carlos, Aline, Lívia e a todos os outros integrantes do Labes/ICMC!
- A todos os professores do ICMC que contribuiram para a minha formação e conhecimentos essenciais deste trabalho, em especial os professores: Masiero, Ellen, Simone, Adilson, Graça Nunes, Fabiano e Regina Helena.
- À CAPES, pelo apoio financeiro.

Resumo

A Arquitetura Orientada a Serviços (SOA) é um estilo arquitetural formado por um conjunto de restrições que visa promover a escalabilidade e a flexibilidade de um sistema, provendo suas funcionalidades como serviços. Nos últimos anos, um estilo alternativo foi proposto e amplamente adotado, que projeta as funcionalidades de um sistema como recursos. Este estilo arquitetural orientado a recursos é chamado de REST. O teste de serviços *web* em geral apresenta vários desafios devido a sua natureza distribuída, canal de comunicação pouco confiável, baixo acoplamento e a falta de uma interface de usuário. O teste de serviços RESTful (serviços que utilizam o REST) compartilham estes mesmos desafios e ainda necessitam que suas restrições sejam obedecidas. Estes desafios demandam testes mais sistemáticos e formais. Neste contexto, o teste baseado em modelos (TBM) se apresenta como um processo viável para abordar essas necessidades. O modelo que representa o sistema deve ser simples e ao mesmo tempo preciso para que sejam gerados casos de teste com qualidade. Com base nesse contexto, este projeto de mestrado propõe uma abordagem baseada em modelos para testar serviços RESTful. O modelo comportamental adotado foi a máquina de estados de protocolos UML, capaz de formalizar a interface do serviço enquanto esconde o seu funcionamento interno. Uma ferramenta foi desenvolvida para gerar automaticamente os casos de teste usando critérios de cobertura de estados e transições para percorrer o modelo.

Palavras-chave: teste baseado em modelos, serviços RESTful, arquitetura orientada a recursos, máquina de estados de protocolos UML, geração de casos de teste.

Abstract

Service Oriented Architecture (SOA) is an architectural style consisting of a set of restrictions aimed at promoting the scalability and flexibility of a system, providing its functionalities as services. In recent years, an alternative style was proposed and widely adopted, which designs the system's functionalities as resources. This resource oriented architectural style is called REST. In general, the test of web services has several challenges due to its distributed nature, unreliable communication channel, low coupling and the lack of a user interface. Testing RESTful web services (services that use REST) share these same challenges and also need to obey the REST constraints. These challenges require a more systematic and formal testing approach. In this context, model based testing presents itself as a viable process for addressing those needs. The model that represents the system should be simple and precise enough to generate quality test cases. Based on this context, this work proposes a model based approach to test RESTful web services. The behavioral model used was the UML protocol state machine, which is capable to provide a formalization of the service interface, while hiding its internal behaviour. A tool was developed to automatically generate test cases using the state and transition coverage criteria to traverse the model.

Keywords: model based testing, resource oriented architecture, RESTful web services, UML protocol state machine, test case generation.

Sumário

1 Introdução **1**
1.1 Contextualização 1
1.2 Motivação 3
1.3 Objetivo 4
1.4 Organização do Trabalho 5

2 Serviços RESTful **7**
2.1 Considerações Iniciais 7
2.2 Origem do REST 8
2.3 O Estilo de Arquitetura REST 10
2.3.1 Recursos 12
2.3.2 Representação de Recurso 13
2.3.3 URI 13
2.3.4 Interface Uniforme 14
2.3.5 *Estado não persistente* 15
2.3.6 Conectividade 16
2.4 Descrição de serviços RESTful 17
2.5 Classificação de serviços RESTful 18
2.6 Segurança 19
2.7 Desempenho 21
2.8 Uso do REST 22
2.9 Ferramentas 23
2.10 Considerações Finais 25

3 Teste de Software **27**
3.1 Considerações Iniciais 27
3.2 Conceitos Básicos 27
3.3 Fases de Teste 30
3.4 Técnicas e Critérios de Teste 31
3.4.1 Técnica de Teste Funcional 32
3.4.2 Técnica de Teste Estrutural 34
3.4.3 Técnica de Teste Baseado em Defeitos 38
3.4.4 Técnica de Teste Baseado em Modelos 39

3.4.4.1 Máquinas de Estados Finitos 43
3.4.4.2 Máquinas de Estados Finitos Estendidas 46
3.4.4.3 Statecharts 47
3.4.4.4 Redes de Petri 49
3.4.4.5 Máquina de Estados UML 49
3.5 Ferramentas de Teste de Software 51
3.6 Considerações Finais 52

4 Modelagem e Teste de Serviços RESTful 53
4.1 Considerações Inicias 53
4.2 Modelagem de Serviços RESTful 54
4.3 Teste de Serviços RESTful 55
4.3.1 Ferramentas de Teste 60
4.4 Considerações Finais 61

5 Abordagem para Teste de Serviços RESTful 63
5.1 Considerações Iniciais 63
5.2 Processo de Teste Baseado em Modelos para Serviços RESTful 64
5.2.1 Criação do modelo comportamental 65
5.2.2 Processamento do modelo 70
5.2.3 Análise das expressões 72
5.2.4 Seleção e execução do critério de cobertura, e dados de entrada . . 73
5.2.5 Geração e execução do conjunto de teste 74
5.3 Estudo de caso 77
5.4 Considerações Finais 80

6 Conclusão 81
6.1 Contribuições 81
6.2 Dificuldades e Limitações 82
6.3 Trabalhos Futuros 83

Referências 95

Lista de Figuras

2.1 Processo de derivação do REST (Fielding e Taylor (2002)). 9
2.2 Arquitetura de um serviço RESTful (Adaptado de Feng et al. (2009)). . . . 10
2.3 Tipos de serviços e suas conexões (Adaptado de Richardson e Ruby (2007)). 17
2.4 Exemplo de requisição feita a um serviço RESTful. 23
2.5 Exemplo de resposta recebida de um serviço RESTful. 23

3.1 Processo de *debugging* (Adaptado de Sommerville (2007)). 29
3.2 *Driver* e *stubs* aplicados no teste de unidade. 31
3.3 Processo de teste (Sommerville (2007)). 31
3.4 Código do *Identifier* (Barbosa et al. (2007)). 35
3.5 GFC da função *main* do programa *Identifier* (Adaptado de Barbosa et al. (2007)). 36
3.6 GFCs de estruturas sequenciais, condicionais e de repetição (Barbosa et al. (2007)). 36
3.7 Grafo Def-Uso da função *main* (Barbosa et al. (2007)). 38
3.8 Visão geral do TBM (Adaptado de Utting e Legeard (2007)). 41
3.9 MEF para um extrator de comentários (Chow (1978)). 44
3.10 Exemplo de MEFE (Petrenko et al. (2004)). 47
3.11 Exemplo de *statechart* (Adaptado de Harel (1987)). 48
3.12 Exemplo de rede de Petri (Peterson (1977)). 49
3.13 Exemplo máquina de estados comportamental UML. 50
3.14 Exemplo máquina de estados de protocolos UML. 51

4.1 Exemplo de POST class graph (Chakrabarti e Rodriquez (2010)). 57
4.2 POST object graph baseado na PCG (Chakrabarti e Rodriquez (2010)). . . 57
4.3 Diagrama de classes de um Serviço de Cursos (Correa et al. (2012)). 58
4.4 Tabela de decisão da operação *createCourse()* (Correa et al. (2012)). 59
4.5 Diagrama de classe representando o serviço RESTful (Borges (2009)). . . . 60
4.6 Diagrama de sequência representando os casos de teste (Borges (2009)). . . 60

5.1 Abordagem de teste baseada em modelos. 64
5.2 Modelo comportamental de um serviço RESTful para agendamento de quartos de um hotel (Adaptado de Porres e Rauf (2011)). 66

5.3 Exemplo do modelo sendo criado na ferramenta ArgoUML. 68
5.4 A DAG gerada que corresponde ao modelo HRB. 71
5.5 Exemplo de uma árvore sintática abstrata. 72
5.6 Exemplo de caso de teste gerado. 74
5.7 Máquina de estados de protocolos do Google Task. 78
5.8 DAG gerada a partir do modelo. 78

Lista de Tabelas

2.1 Equivalência entre operações sobre dados e métodos HTTP. 14

3.1 Classes de equivalência da especificação Cadeia de Caracteres (Fabbri et al. (2007)). 33

Lista de Abreviaturas e Siglas

API	–	*Application Programming Interface*
GFC	–	Grafo de Fluxo de Controle
HTML	–	*HyperText Markup Language*
HTTP	–	*HyperText Transfer Protocol*
HTTPS	–	*HyperText Transfer Protocol Secure*
IBM	–	*International Business Machines*
IDE	–	*Integrated Development Environment*
IEEE	–	*Institute of Electrical and Electronics Engineers*
IETF	–	*Internet Engineering Task Force*
JSON	–	*JavaScript Object Notation*
MEF	–	Máquina de Estados Finitos
MEFE	–	Máquinas de Estados Finitos Estendidas
OASIS	–	*Organization for the Advancement of Structured Information Standards*
OCL	–	*Object Constraint Language*
OMG	–	*Object Management Group*
POX	–	*Plain Old XML*
REST	–	*Representational State Transfer*
ROA	–	*Resource-Oriented Architecture*
RoR	–	*Ruby on Rails*
RPC	–	*Remote Procedure Call*
SOA	–	*Service-Oriented Architecture*
SOAP	–	*Simple Object Access Protocol*
SQA	–	*Software Quality Assurance*
SUT	–	*System Under Test*
TBM	–	Teste Baseado em Modelos
U2TP	–	*UML 2.0 Testing Profile*
UDDI	–	*Universal, Discovery, Description and Integration*
UML	–	*Unified Modeling Language*
URI	–	*Uniform Resource Identifier*
V&V	–	Verificação e Validação
W3C	–	*World Wide Web Consortium*
WS	–	*Web Services*

WSDL – *Web Services Description Language*
XMI – *XML Metadata Interchange*
XML – *eXtensible Markup Language*

Capítulo

1

Introdução

1.1 Contextualização

Ao longo da história da Engenharia de Software, diversas tecnologias, técnicas e paradigmas foram criados para suprir necessidades e resolver problemas da área. A partir dos anos 2000, houve uma forte tendência para o desenvolvimento rápido de aplicações, e se percebe um ritmo de mudança acelerado com relação à tecnologia da informação, organizações e no próprio ambiente (globalização) (Boehm, 2006). O desenvolvimento tradicional de software se baseia na premissa de que o ambiente sofre poucas mudanças e de que o software consegue se manter estável durante longos períodos de tempo. Logo se percebe que essa premissa passa a não se encaixar no cenário atual, marcado por mudanças na infraestrutura e no domínio do software, tais como: distribuição em massa, computação móvel e computação pervasiva. Buscaram-se, então, soluções que passam a desenvolver o sistema de uma forma mais dinâmica, modular e distribuída. Dentre estas soluções está a Arquitetura Orientada a Serviços (Baresi et al., 2006).

A Arquitetura Orientada a Serviços (SOA – *Service-Oriented Architecture*) é um estilo de arquitetura que promove a escalabilidade e a flexibilidade de um sistema. É recomendada para sistemas distribuídos complexos e heterogêneos (Josuttis, 2007). Na SOA, serviços consistem em componentes bem definidos e autocontidos, capazes de fornecer uma funcionalidade específica sem depender de outros serviços. Serviços são descritos por meio de uma linguagem padrão de definição e se comunicam por uma interface. Por meio

dessa comunicação, é possível executar uma tarefa de negócio ou um processo (Papazoglou e Heuvel, 2007).

Várias implementações da SOA foram propostas, sendo os *Web Services* a mais utilizada. *Web Services* consistem em um conjunto de padrões que permitem que programas acessem informações por meio da definição e publicação da interface de um serviço. Essa interface define como os dados estão disponíveis e como podem ser acessados. A descrição da interface dos serviços é feita utilizando a *Web Service Description Language* (WSDL), para o descobrimento do serviço é utilizado o *Universal Description, Discovery and Integration* (UDDI) e para a comunicação com o serviço é utilizado o *Simple Object Access Protocol* (SOAP). A comunicação com os serviços é feita independentemente de plataformas ou linguagens de programação (Josuttis, 2007; Sommerville, 2007).

Nos últimos anos, outro estilo arquitetural foi proposto e amplamente adotado para guiar a estruturação de um sistema de forma que suas funcionalidades sejam expostas como recursos (informação útil) e não como serviços (processamento). Este estilo é chamado de REST (Transferência de Estado Representacional ou *Representational State Transfer*). REST é tido como uma alternativa que dispensa as tecnologias WS-* usadas nos *Web Services*, tais como: SOAP, WSDL e UDDI (Richardson e Ruby, 2007). As restrições impostas pelo REST objetivam maximizar a escalabilidade dos componentes e sua independência. REST é mais comumente utilizado em sistemas distribuídos de hipermídia como a *World Wide Web*. Um serviço definido como RESTful é um serviço que respeita as restrições do REST. Um cliente interage com um serviço RESTful por meio de uma interface uniforme (requisições HTTP), e trocam representações de recursos (entidades) do sistema (Feng et al., 2009).

O teste é uma etapa muito importante em qualquer tipo de sistema, pois ele visa melhorar a confiabilidade e a qualidade do sistema por meio da identificação dos defeitos. É importante que cada processo de desenvolvimento tenha um processo de teste, o qual define, dentre outras coisas, quais os passos, técnicas e métodos são utilizados (Pressman, 2011). Testar serviços em geral apresentam alguns desafios em razão de sua natureza distribuída e de baixo acoplamento, além da pouca confiabilidade que seu canal de comunicação possui (Chakrabarti e Kumar, 2009). Nesse contexto, é necessário que abordagens de teste mais sistemáticas e formais sejam aplicadas. Dentro da área de teste de software, o teste baseado em modelos apresenta-se como uma técnica promissora para lidar com esses desafios. Além disso, a técnica por se tratar de um teste funcional e caixa-preta permite que testes sejam realizados baseados na descrição da interface do serviço e seu comportamento, ambos expressos em um modelo.

O Teste Baseado em Modelos (TBM) é um tipo de teste funcional que se baseia em um modelo criado a partir da especificação de um “sistema em teste” para gerar os casos

de teste de uma maneira automatizada (Utting e Legeard, 2007). O TBM é dependente de três elementos fundamentais: a linguagem de modelagem usada para representar o modelo, o algoritmo de geração de testes e a ferramenta que gera a infraestrutura de apoio para os testes (Dalal et al., 1999). Este trabalho de mestrado está inserido no contexto de teste funcional para serviços RESTful, mais especificamente utilizando o TBM como abordagem de teste.

É importante que os modelos sejam precisos e escritos em uma notação de modelagem formal com semânticas precisas para gerar corretamente os casos de teste. Porém, eles devem ser mais simples que o sistema, ou ao menos mais fáceis de verificar, modificar e manter (Utting et al., 2011). Um dos modelos que podem ser utilizados para a realização do TBM são as máquinas de estados UML. Consideradas como uma notação baseada em transição, essas máquinas são mais utilizadas em sistemas orientados a controle, ou seja, sistemas em que as operações disponíveis variam de acordo com o estado do sistema (Utting et al., 2011).

1.2 Motivação

O REST está cada vez mais presente na indústria e esse fato é comprovado por grandes empresas que fazem seu uso, tais como: Google[1], Yahoo[2], Facebook[3], Twitter[4], e LinkedIn[5].

Embora a interface de um serviço RESTful permita um número limitado de operações, seu comportamento pode se tornar complexo. Além disso, poucos estudos podem ser encontrados que envolvem o teste de serviços RESTful (AlShahwan e Moessner, 2010; Borges, 2009; Chakrabarti e Kumar, 2009; Chakrabarti e Rodriquez, 2010; Correa et al., 2012; Feller, 2010; Meng et al., 2009; Reza e Van Gilst, 2010; Seijas et al., 2013; Silva, 2011) quando comparados com a quantidade de estudos encontrados em se tratando de teste de *Web Services*. Mais contribuições na área são necessárias, pois além de compartilhar os desafios do teste de *Web Services* (natureza distribuída, baixo acoplamento, baixa confiabilidade no canal de comunicação), o teste de serviços RESTful impõe a necessidade de respeitar as propriedades exclusivas do REST, tais como: sua interface uniforme, endereçabilidade, estado não persistente e representações de recursos utilizadas.

Neste contexto técnicas mais sistemáticas e formais são adequadas para abordar esses desafios. O teste baseado em modelos (TBM) se apresenta como um processo viável

[1] `http://developers.google.com/custom-search/v1/usingrest`
[2] `http://developer.yahoo.com/search/rest.html`
[3] `http://developers.facebook.com/docs/reference/api`
[4] `http://dev.twitter.com/docs/api`
[5] `http://developer.linkedin.com/rest`

para abordar essas necessidades. O modelo que representa o sistema deve ser simples e ao mesmo tempo preciso para que sejam gerados casos de teste com qualidade. O modelo escolhido para representar os serviços RESTful neste trabalho foram as máquinas de estados de protocolos UML (OMG, 2011) as quais enfatizam mais nas transições (na forma de precondições, eventos e poscondições) entre os estados do que nas ações que ocorrem em cada estado. Desta forma, é possível omitir detalhes de implementação e descrever a interface e o comportamento do serviço RESTful correspondendo ao nível de abstração que o TBM necessita (Xu et al., 2007). Esse modelo também é adequado para representar o comportamento do serviço, pois fornece especificações de interface com informações sobre as entradas, condições em que os eventos podem ser invocados e a saída esperada (Porres e Rauf, 2011). Neste contexto as máquinas de estados de protocolos UML já foram utilizadas em trabalhos para projetar serviços RESTful (Porres e Rauf, 2011; Rauf e Porres, 2011; Rauf et al., 2010). Embora este tipo de modelo possa ser usado para gerar automaticamente casos de teste (Xu et al., 2007), nenhuma iniciativa foi encontrada que trata de sua aplicação no teste de serviços RESTful, mais especificamente na geração automática de casos de teste.

1.3 Objetivo

O objetivo deste trabalho é propor uma abordagem de teste, com base nos passos do TBM, para a geração automática de casos de teste para serviços RESTful. A geração automática dos testes tem o objetivo de facilitar o teste funcional destes serviços respeitando as restrições REST e fornecendo um teste mais sistemático e formal. O modelo UML utilizado na abordagem permite que o testador expresse comportamentos do serviço sem interferir em propriedades como interface uniforme, endereçabilidade e estado não persistente do REST.

Este trabalho apresenta as seguintes contribuições:

1. Um processo de teste baseado em modelos para serviços RESTful. Por meio deste processo, a tarefa do testador passa ser de criar um modelo que representa cada comportamento da especificação que deseja ser testado, para depois executar os casos de teste gerados automaticamente e analisar os resultados;

2. Uma ferramenta para automatizar o processo de teste. A ferramenta é a responsável por utilizar o modelo para gerar os casos de teste a partir do critério de cobertura por estados ou transições; e

3. Exemplos de uso da abordagem com a ferramenta. O processo e a ferramenta são aplicados em um estudo de caso.

1.4 Organização do Trabalho

O restante deste trabalho está organizado da seguinte forma. No Capítulo 2 são apresentados os principais conceitos de REST, tais como: sua origem, arquitetura, desempenho e segurança. No Capítulo 3 são apresentados os principais conceitos relacionados a teste de software, tais como: seus objetivos, fases, técnicas, critérios e ferramentas. No Capítulo 4 são apresentados os principais conceitos envolvendo as máquinas de estados de protocolos UML, assim como os trabalhos relacionados ao teste e modelagem de serviços RESTful encontrados na literatura. No Capítulo 5 será apresentada a abordagem de teste desenvolvida para o teste de serviços RESTful, assim como a ferramenta desenvolvida para automatizar o processo. Por fim, no Capítulo 6 são apresentadas as conclusões deste trabalho, destacando as contribuições, dificuldades, limitações e sugestões de trabalhos futuros.

Capítulo

2

Serviços RESTful

2.1 Considerações Iniciais

Uma arquitetura de *software* pode ser definida como uma configuração de três elementos: dados, elementos de processamento ou componentes, e elementos de conexão (Perry e Wolf, 1992). Os dados representam as informações usadas pelo sistema, elementos de processamento são aqueles que alteram as informações dos dados e os elementos de conexão unem as diferentes partes da arquitetura (Perry e Wolf, 1992). Pode-se definir estilo de arquitetura como um conjunto de restrições que, quando aplicadas a um sistema, restringem os papéis e funcionalidades dos elementos arquiteturais e o relacionamento permitido entre eles (Fielding, 2000). Os benefícios em se usar estilos de arquitetura são: eles estimulam o reúso de projetos arquiteturais e de código, melhoram o entendimento da organização de um sistema, e alguns estilos permitem análises especializadas de uma determinada característica do sistema (Kim e Garlan, 2006).

Neste capítulo são apresentados os conceitos sobre o estilo arquitetural REST. Serão apresentadas as restrições do REST e como essas restrições fazem a arquitetura de um serviço ser orientada a recurso, funcionando de maneira similar ao que ocorre com *websites* da Internet. Será apresentado também como essas restrições impactam no desempenho e segurança, e algumas ferramentas existentes que auxiliam na construção de serviços RESTful. Este capítulo faz parte do embasamento teórico necessário para o desenvolvimento deste trabalho de mestrado.

O capítulo está organizado da seguinte forma. Na Seção 2.2 é apresentada a origem do estilo de arquitetura REST junto com seu processo de derivação a partir de outros estilos. Na Seção 2.3 são apresentados os principais conceitos envolvendo a arquitetura de serviços RESTful. Na Seção 2.4 é apresentado como é realizada a descrição de serviços RESTful. Na Seção 2.5 são apresentadas algumas formas de classificar serviços RESTful. Na Seção 2.6 são apresentados conceitos de segurança. Na Seção 2.7 são apresentados conceitos de desempenho. Na Seção 2.8 são apresentados alguns exemplos de uso do REST na indústria e no meio acadêmico. Na Seção 4.3.1 são apresentadas ferramentas que auxiliam na construção de serviços RESTful.

2.2 Origem do REST

REST é um estilo arquitetural criado por Fielding (2000) em sua tese de doutorado. Durante o tempo em que trabalhou na IETF[1] (*Internet Engineering Taskforce*), Fielding participou no desenvolvimento do protocolo HTTP/1.0, foi o arquiteto principal do desenvolvimento do HTTP/1.1 e também foi o autor da sintaxe genérica do URI (Identificador Uniforme de Recursos ou *Uniform Resource Identifier*) (Feng et al., 2009).

A primeira edição do REST foi desenvolvida entre outubro de 1994 e agosto de 1995, e melhorada nos anos seguintes. O nome "Transferência de Estado Representacional" tem como objetivo dar a ideia de como uma aplicação *web* bem projetada deve se comportar. Um conjunto de páginas *web* forma uma máquina de estados virtual e o usuário a percorre por meio de *links* ou por submissão de formulários. Cada uma dessas ações resulta na transferência de uma representação do próximo estado da máquina. A representação é usada para realizar a transição de um estado para outro (Fielding e Taylor, 2002).

Fielding (2000) define REST como um estilo arquitetural híbrido, pois ele foi derivado de diversos estilos arquiteturais baseados em rede e foi combinado com restrições adicionais que definem uma interface uniforme de conexão. Este processo é ilustrado na Figura 2.1. Para desenvolver o estilo REST, Fielding definiu como ponto de partida o chamado estilo nulo (*null style*) representando um sistema no qual não existem limites entre seus componentes. A partir desse estilo foram tomadas as seguintes ações (Fielding e Taylor, 2002):

1. Adição das restrições do estilo de arquitetura cliente-servidor (CS). Ao separar a responsabilidade da interface do usuário das responsabilidades de armazenamento de dados, são melhoradas a escalabilidade do sistema por meio da simplificação de componentes e a portabilidade da interface por meio de múltiplas plataformas;

[1] `http://www.ietf.org/`

2. Adição de restrições para que a comunicação entre cliente e servidor seja sem estado (*stateless*), ou seja, deve ser possível que o servidor entenda a requisição por meio das informações contidas na requisição;

3. Adição das restrições de *cache* para que a resposta do servidor a uma requisição seja sempre rotulada como "armazenável em *cache*" ou não. Essa medida foi tomada para melhorar a eficiência da rede, pois o *cache* possui o potencial de eliminar parcial ou totalmente algumas interações. Caso a resposta esteja no *cache*, então um cliente pode reusar seus dados para requisições equivalentes mais tarde;

4. Adição das restrições que implementam a chamada Interface Uniforme (U – *Uniform Interface*) entre seus componentes. A partir disso, a arquitetura do sistema é simplificada e a visibilidade das interações é melhorada;

5. Adição de restrições para implementar o estilo de Sistema em Camadas (LS – *Layered System Style*). Isso teve como objetivo melhorar a escalabilidade do sistema. Essas restrições limitam o comportamento dos componentes do sistema de modo que eles não tenham acesso às camadas além das quais eles estão interagindo; e

6. Adição da restrição de Código sob Demanda (COD – *Code-on-Demand style*). Esta é a única restrição opcional do REST e permite que a funcionalidade do cliente seja estendida por meio da transferência e execução de código na forma de *scripts* ou *applets*.

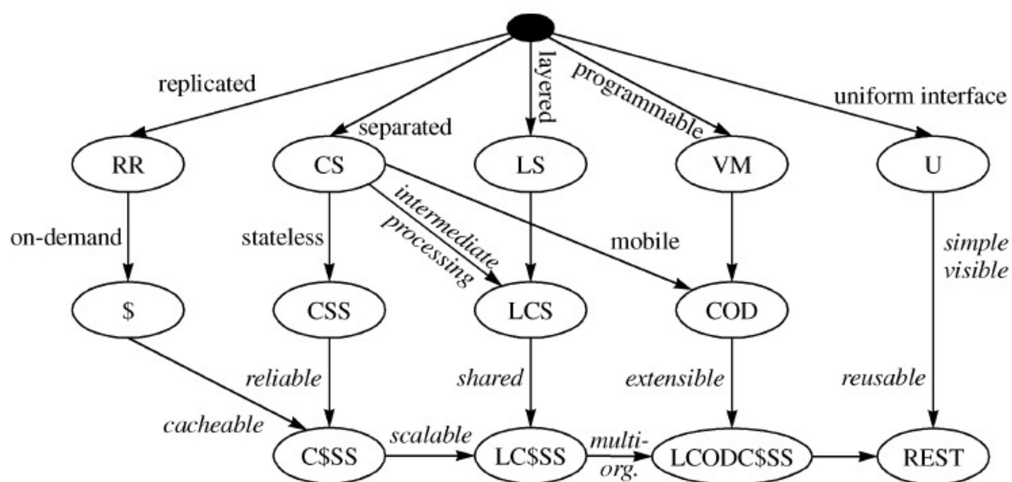

Figura 2.1: Processo de derivação do REST (Fielding e Taylor (2002)).

2.3 O Estilo de Arquitetura REST

Segundo Fielding (2000), REST é um conjunto coordenado de restrições de arquitetura que tenta maximizar a independência e a escalabilidade das implementações dos componentes. REST é um estilo de arquitetura para sistemas hipermídia distribuídos como a *World Wide Web*. Sistemas que seguem os princípios de REST de Fielding são comumente referenciados como “RESTful”.

As restrições definidas por Fielding não determinam quais tecnologias devem ser usadas, elas apenas definem como os dados são transferidos entre os componentes e quais os benefícios de se seguir essas recomendações. Considerando que as restrições não são acopladas a tecnologias, o REST pode ser implementado em qualquer arquitetura de rede disponível. Basicamente, as restrições que definem um serviço como RESTful são (Sandoval, 2009):

1. Deve adotar a arquitetura cliente-servidor;
2. Deve ser *stateless*;
3. Deve utilizar *cache*;
4. Deve ser uniformemente acessível;
5. Deve ser em camadas; e
6. Deve fornecer código sob demanda (restrição opcional).

Na Figura 2.2 é ilustrada a arquitetura de um serviço RESTful. O cliente utiliza uma interface uniforme para interagir com o serviço e trocar representações de recursos. Todo esse processo é realizado utilizando interações do tipo *stateless* (Feng et al., 2009).

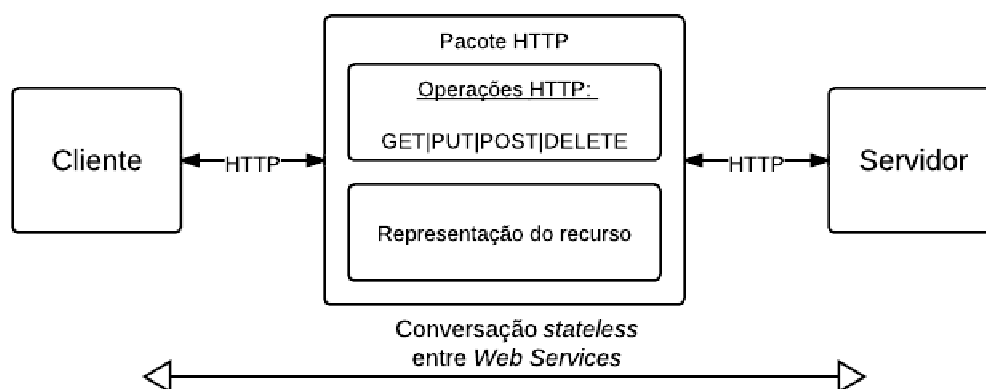

Figura 2.2: Arquitetura de um serviço RESTful (Adaptado de Feng et al. (2009)).

Pautasso et al. (2008) definem quatro princípios básicos do REST:

- Identificação de recursos por meio de um URI: Um serviço sempre expõe um conjunto de recursos os quais são acessíveis por meio de um determinado URI. Esses recursos representam os alvos de interação com o usuário. O URI é utilizada para a descoberta de serviços e para criar um espaço de endereçamento global do recurso;

- Interface uniforme: O usuário interage com os recursos por meio das operações de POST, GET, PUT e DELETE, correspondendo respectivamente a criar, ler, atualizar e remover um recurso;

- Mensagens autodescritivas: Os recursos são desacoplados de suas representações para que seu conteúdo possa ser acessado em uma variedade de formatos, tais como: HTML, XML, texto puro, PDF e JPEG; e

- Interações *stateless*: Toda a interação com um recurso é sem estado (*stateless*), ou seja, as mensagens são autocontidas, assim as interações são baseadas no conceito de transferência de estado.

Dentre as diversas técnicas existentes para a transferência de estado estão a reescrita de URI, *cookies* e campos de formulário escondidos. Os estados também podem ser embutidos nas respostas para apontar para possíveis estados futuros. Embora o uso de *cookies* seja uma técnica conhecida para a troca de estado, ela viola o conceito de REST. *Cookies* são ligados junto às futuras requisições para um conjunto de identificadores de recursos. Ao fazerem isso, geralmente eles abrangem *websites* inteiros em vez de serem associados a um estado da aplicação. Caso o histórico do *browser* seja acionado e a visão fornecida pela função anteceda a visão fornecida pelo *cookie*, o estado da aplicação não mais iguala o estado armazenado no *cookie*. Assim, a próxima requisição enviada ao servidor irá conter um *cookie* que não representa o contexto atual da aplicação. Além disso, *cookies* permitem que dados sejam passados sem uma identificação suficiente de sua semântica. Eles se tornam um problema tanto para a segurança quanto para a privacidade, por isso as aplicações *web* baseadas no uso de *cookies* não são confiáveis (Fielding e Taylor, 2002).

Rauf e Porres (2011) ainda explicam que para que um serviço esteja de acordo com o estilo arquitetural REST ele deve possuir quatro propriedades: endereçabilidade, conectividade, interface uniforme e estado não persistente (*statelessness*). A endereçabilidade indica que toda informação importante para um sistema deve ser tratada como um recurso e esse recurso deve ser sempre endereçado por um URI. Conectividade requer que um recurso sempre contenha *links* para outros recursos. A interface uniforme do sistema implica que todos os recursos sejam manipulados por meio de métodos padrões do protocolo HTTP: GET, POST, PUT e DELETE. A propriedade de estado não persistente impõe que não haja informações ocultas de uma sessão ou informações de estado.

Essas propriedades quando aplicadas a um serviço implica em uma Arquitetura Orientada a Recurso (ROA – *Resource-Oriented Architecture*). Segundo Richardson e Ruby (2007), ROA é uma arquitetura RESTful concreta que pode transformar um problema em um serviço RESTful. Todo sistema que se enquadra na arquitetura orientada a recurso também pode ser considerada como RESTful.

O conceito de orientação a recurso foi formalizado como uma diretriz para a transição do HTTP/1.0 para o HTTP/1.1, por isso o HTTP é a aplicação dominante do estilo de ROA. No entanto, o HTTP não impõe sua adesão, por isso existem várias aplicações que violam os conceitos de ROA. A implementação dominante de *Web Services* que usam as tecnologias (WS-*) é um exemplo de incompatibilidade com a orientação a recursos, pois somente usa o HTTP como um mecanismo de transporte (Overdick, 2007).

Segundo Richardson e Ruby (2007), REST ainda é um estilo muito genérico, não é conectado com a *web* e não depende do protocolo HTTP ou de URIs estruturados. O uso dos métodos HTTP pelos serviços é decorrente da aplicação do REST ao contexto *web*. Além das quatro propriedades citadas anteriormente, na descrição da ROA também estão contidos quatro conceitos: Recursos, URIs, representações de recurso e as ligações entre os recursos (Richardson e Ruby, 2007). Nos subtópicos a seguir são abordados em maior profundidade esses conceitos e propriedades.

2.3.1 Recursos

Um recurso RESTful é qualquer elemento que pode ser endereçável na *web*, ou seja, é possível que o recurso seja sempre acessado e transferido entre clientes e servidores. Ele representa nada mais do que os objetos que o *Web Service* precisa lidar para resolver um problema de um determinado domínio. Alguns exemplos de recursos são: uma notícia, um resultado de uma busca na *web*, um estudante de uma sala de aula ou a temperatura em uma cidade (Sandoval, 2009).

Do ponto de vista do cliente, um recurso é qualquer elemento com o qual ele pode interagir enquanto progride para um objetivo. Muitos recursos do mundo real podem parecer inicialmente difíceis de serem projetados para a *web*, porém qualquer elemento pode ser transformado em recurso desde que seja possível associá-lo com uma informação que seja acessível na *web* (Webber et al., 2010).

Considerando todos os aspectos ao se projetar um serviço RESTful, tais como: a identificação de recursos, escolha dos tipos de mídia e formato e aplicação da interface uniforme; a identificação de recurso é a parte mais flexível. Não existe uma forma única de se modelar os recursos de um sistema, o importante é que seja possível usar a interface uniforme do HTTP de forma razoável para implementar o *Web Service* (Allamaraju, 2010).

2.3.2 Representação de Recurso

A representação consiste em dados, metadados descrevendo os dados e, em alguns casos, metadados descrevendo os metadados, os quais geralmente são usados com o propósito de verificar a integridade da mensagem (Fielding, 2000). REST introduz uma camada entre a representação abstrata do recurso e sua representação concreta (Erenkrantz et al., 2007). A representação abstrata dos recursos é transmitida entre os clientes e os servidores, e nada mais é do que um estado temporal do dado real contido em alguma unidade de armazenamento. Diferentes clientes podem interagir com diferentes representações de um recurso, portanto essa representação pode assumir várias formas, tais como: uma imagem, um arquivo de texto puro, um arquivo XML ou um arquivo JSON. Ao acessar um *browser*, o cliente geralmente lida com uma representação do recurso no formato HTML. Contudo, para requisições feitas por outros *Web Services* são usadas representações mais eficientes como, por exemplo, o XML (Sandoval, 2009).

O formato do dado de uma representação é comumente chamado de tipo de mídia (*media type*). Alguns tipos de mídia são voltados para o processamento automático, outros para serem renderizados para a visão do usuário, e poucos para os dois objetivos. O projeto de tipo de mídia impacta diretamente no desempenho percebido pelo usuário de um sistema. O desempenho percebido pelo usuário aumenta significativamente caso um formato de dado priorize as informações importantes para serem renderizadas imediatamente e o resto para serem renderizadas posteriormente (Fielding, 2000).

2.3.3 URI

Para utilizar um recurso é necessário que um consumidor seja capaz de identificá-lo e que tenha algum modo de manipulá-lo. Para isso é utilizado o Identificador Uniforme de Recursos (*Uniform Resource Identifier* – URI), também chamado de Identificador Universal de Recursos. O URI é usado para identificar um recurso e torná-lo endereçável. A relação entre os URIs e os recursos é uma relação *many-to-one*, ou seja, um URI identifica apenas um recurso, e um recurso pode possuir mais de um URI. Por exemplo, a página inicial do Google pode ser acessada pelo URI "http://www.google.com" ou "http://google.com" (Webber et al., 2010).

Segundo Berners-Lee et al. (2005), os URIs são caracterizadas da seguinte forma:

- Identificador – A ideia de um "identificador" referencia ao propósito de distinguir um recurso de outro, independentemente do seu propósito. Entretanto, não se deve concluir logo de imediato que um identificador define ou incorpora a identidade do que está sendo referenciado. Embora isso possa ocorrer para alguns identificadores, para outros é possível que denotem um recurso em que não possa ser possível

acessá-lo, por exemplo, quando um recurso não é singular por natureza como, por exemplo, um conjunto ou um mapeamento;

- Uniforme – O conceito de uniformidade permite que tipos diferentes de identificadores de recursos sejam usados em um mesmo contexto ainda que seus mecanismos de acesso sejam diferentes. Os identificadores podem ser reusados em diferentes contextos e a introdução de novos tipos de identificadores de recursos não interferem na maneira pela qual os atuais são usados. Passa a ser possível, também, o uso de uma interpretação semântica uniforme de convenções sintáticas comuns por meio dos diferentes identificadores;

- Recurso – O termo recurso é usada de uma forma genérica para designar o que está sendo identificado por um URI; por exemplo, um documento eletrônico, uma imagem, um serviço ou uma coleção de serviços. Conceitos abstratos também podem ser considerados um recurso, tais como: operadores de uma equação matemática, valores numéricos e relações de parentesco.

A propriedade de endereçabilidade está intrinsecamente ligada ao conceito de URI. Uma vez que os recursos são acessados por meio de URIs, a aplicação se torna endereçável, pois expõem um URI para cada pedaço de sua informação. Do ponto de vista do usuário, essa propriedade é a mais importante no contexto de qualquer *website* ou aplicação (Richardson e Ruby, 2007).

2.3.4 Interface Uniforme

O conceito de interface uniforme implica que um consumidor deve interagir com os recursos somente por meio de quatro métodos HTTP: POST, GET, PUT, e DELETE. Eles são relacionados com as operações CRUD (*Create*, *Retrieve*, *Update* e *Delete*) sobre dados da seguinte maneira (Sandoval, 2009):

Tabela 2.1: Equivalência entre operações sobre dados e métodos HTTP.

Operação sobre dados	Protocolo HTTP equivalente
CREATE	POST
RETRIEVE	GET
UPDATE	PUT
DELETE	DELETE

O protocolo HTTP (*Hypertext Transfer Protocol*) (Fielding et al., 1999) define essas operações da seguinte maneira:

- GET – Essa operação é segura, ou seja, garante que não realizará qualquer efeito no receptor da mensagem. As respostas do GET são uma descrição do estado atual de um recurso especificado e podem ser armazenadas em *cache*.

- PUT – Essa operação causa uma alteração no recurso de uma forma idempotente, ou seja, os efeitos de N requisições possuem o mesmo efeito de uma requisição. O método requisita ao servidor que a entidade da requisição seja armazenada sob um URI especificada na requisição. As respostas para esse método não são armazenadas em *cache*;

- DELETE – Essa operação requisita ao servidor que um determinado recurso, identificado por um URI, seja removido. Essa requisição também causa alterações em um recurso de forma idempotente;

- POST – Essa operação causa um efeito no receptor e não são seguras para replicação. Ela é usada para requisitar que o servidor de origem aceite uma nova entidade, especificada na requisição, como seu novo subordinado.

Richardson e Ruby (2007) ainda consideram como parte da interface uniforme mais dois métodos HTTP: HEAD e OPTIONS. A operação HEAD recupera apenas metadados de uma representação. Esse método pode ajudar a informar, por exemplo, se um recurso existe sem precisar obter a representação inteira, como aconteceria usando o GET. A operação OPTIONS retorna quais as operações permitidas a um recurso.

2.3.5 Estado não persistente

A propriedade de estado não persistente implica que qualquer requisição HTTP feita a um servidor deve ser executava isoladamente. Todas as informações necessárias para o servidor realizar seu processamento devem estar contidas na requisição. Por consequência disso, o servidor não guarda qualquer tipo de informação de uma requisição feita anteriormente (Richardson e Ruby, 2007).

Segundo Fielding (2000), essa restrição realiza quatro funções:

1. Remove a necessidade de que os conectores (cliente e servidor) guardem o estado da aplicação entre as requisições. Isso reduz o consumo de recursos e melhora a escalabilidade;

2. Permite que as interações sejam processadas em paralelo e sem a necessidade de que o mecanismo de processamento entenda a semântica da interação;

3. Permite que um agente intermediário veja e entenda a requisição isoladamente; e

4. Força que toda a informação que possa interferir na reusabilidade de uma resposta em *cache* esteja contida em cada requisição.

A natureza *stateless* do REST permite a independência das interações entre os elementos. Os elementos que interagem apenas precisam saber da existência do outro elemento durante sua comunicação, assim não sendo necessário o conhecimento da topologia inteira dos componentes na rede. O alvo de cada requisição permite determinar dinamicamente se um componente é o destino ou o intermediário. O uso de *cookies* e de "*frames*", presentes no HTML, são as principais formas de violamento da restrição de *stateless* (Fielding, 2000).

Segundo Richardson e Ruby (2007), em um sistema RESTful existem dois tipos de estados: o estado da aplicação e o estado do recurso. O estado da aplicação fica a cargo do cliente e este é o responsável por enviar para o servidor todos os dados necessários da aplicação. A requisição é o único momento em que o servidor se preocupa em lidar com esse estado, porém após isso, o servidor esquece totalmente deste cliente. O estado do recurso é aquele que permanece o mesmo para todos os clientes. Esse tipo de estado fica armazenado no servidor e nada mais é do que o estado em que o recurso se encontra. Ele é mantido no servidor até que uma operação DELETE seja executada.

2.3.6 Conectividade

Nos serviços RESTful as representações possuem *links* para outros recursos. Um exemplo disso é uma página de mecanismo de busca na Internet. A cada procura feita por um termo, o resultado consiste em resultados da busca mais uma série de *links* para outras páginas da Internet (Richardson e Ruby, 2007).

A conectividade é a responsável por tornar a *web* um conjunto interconectado de recursos pelos quais os usuários podem acessar. Se o serviço adota essa propriedade, então os consumidores podem construir um caminho pela aplicação utilizando os *links* e os formulários (Feng et al., 2009). A Internet, por exemplo, é fácil de ser utilizada em razão de sua conectividade. O usuário é capaz de digitar um URI e depois modificá-la para acessar outros locais usando os *links* que lhe é apresentado (Richardson e Ruby, 2007).

Na Figura 2.3 é ilustrado diferentes serviços com uma mesma funcionalidade, porém sua usabilidade aumenta em direção à direita. O serviço A não possui as propriedades de endereçabilidade nem de conectividade. O serviço B é endereçável, mas não conectado, pois não existe uma indicação de relacionamentos entre seus recursos. O serviço C é endereçável e conectado, pois os recursos possuem *links* para os outros de uma maneira que faça sentido para o usuário.

O serviço A pode ser considerado como um típico serviço do tipo RPC (Chamada Remota de Procedimento ou *Remote Procedure Call*), o serviço B como um serviço híbrido entre RPC e REST, e o serviço C pode ser considerado como um totalmente RESTful (Richardson e Ruby, 2007). O RPC consiste em uma transferência de controle e dados por meio de uma rede de comunicação. Quando um procedimento remoto é invocado, o ambiente que invocou é suspenso, parâmetros são fornecidos para o ambiente no qual o procedimento está localizado, e o procedimento é executado neste local. Após o término do procedimento, os resultados retornam para o ambiente de origem, no qual o programa continua seu processamento (Birrell e Nelson, 1984).

A arquitetura RPC expõe os algoritmos internos por meio de uma interface parecida com a utilizada em linguagens de programação, a qual varia dependendo do serviço. Quando utilizado em serviços na *web*, o RPC pode usar o protocolo HTTP, porém não utiliza todas as vantagens que o HTTP possui, como seus métodos. O RPC é mais comum em serviços que utilizam o protocolo SOAP como os *Web Services* (Richardson e Ruby, 2007).

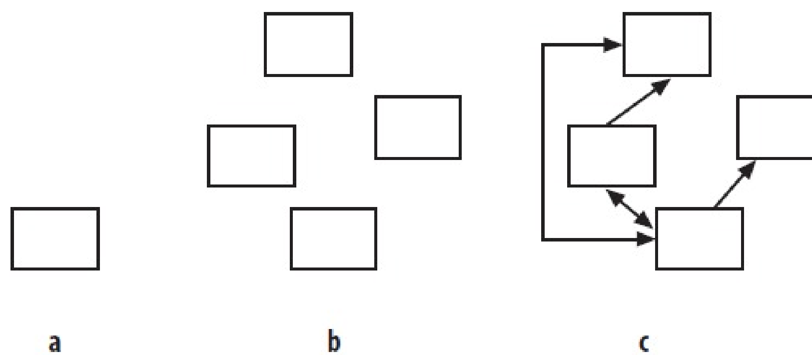

Figura 2.3: Tipos de serviços e suas conexões (Adaptado de Richardson e Ruby (2007)).

2.4 Descrição de serviços RESTful

Muitos serviços RESTful são descritos de forma textual presentes em um conjunto de páginas HTML. Essas descrições são usadas para fornecer informações importantes para os desenvolvedores da funcionalidade fornecida pelos métodos de um determinado serviço RESTful. Entretanto, essas descrições textuais não podem receber anotações semânticas, o que permitiria uma maior precisão no processo de descobrimento e procura destes serviços (Fensel et al., 2007).

Segundo Fensel et al. (2007), embora o número de serviços RESTful tenha aumentado, ainda não existe um padrão definido utilizado para descrevê-los, e as abordagens para descrever estes serviços ainda estão em estágios iniciais. Diferentemente dos servi-

ços WS-*, os quais possuem a WSDL como descrição padrão de linguagem, os serviços RESTful são descritos das mais variáveis formas. Segundo Rauf e Porres (2011), cada vez mais empresas têm construído suas aplicações adotando a linguagem de descrição WADL (*Web Application Description Language*).

A WADL é uma linguagem baseada em XML e busca fornecer um protocolo de descrição passível de ser processado para o uso com aplicações *web* baseadas no HTTP (Fensel et al., 2007). A WADL é mais simples do que a WSDL e permite a descrição das características importantes de um sistema RESTful, tais como: seus recursos, entrada, saída e até mesmo uma descrição textual do serviço (Thies e Vossen, 2009). Rauf e Porres (2011) ainda ressaltam que as informações das interfaces apresentadas utilizando a WADL são apenas sintáticas, ou seja, não declaram nada de sua semântica.

Um documento WADL é definido usando os seguintes elementos (Fensel et al., 2007):

- Aplicação - Denota um elemento raiz que contém a descrição do resto do serviço, ou seja, a descrição dos outros elementos;

- Gramáticas - Age como um recipiente armazenando as definições de qualquer estrutura XML trocada durante a execução do protocolo descrito pelo documento WADL;

- Recursos - Age como um recipiente armazenando todos os recursos que são fornecidos pela aplicação *web*;

- Recurso - Descreve um único recurso fornecido pela aplicação *web*;

- Método - Descreve a entrada e saída de um método HTTP aplicado a um determinado recurso;

- Representação - Descreve a representação do estado de um recurso; e

- Defeito - Denota uma condição de erro.

2.5 Classificação de serviços RESTful

É possível classificar os serviços RESTful de acordo com a utilização que cada serviço faz das restrições REST em sua arquitetura. Uma possível classificação é a de Hi-REST e Lo-REST. Serviços Hi-REST são os que mais se aproximam da definição de REST definida por Fielding em sua tese de doutorado (Richardson e Ruby, 2007). Estes serviços tendem a usar as quatro operações básicas HTTP (GET, POST, PUT e DELETE), URIs

"previsíveis" ao usuário e o chamado POX[2] (*Plain Old XML*) para formatar o conteúdo das mensagens. Richardson e Ruby (2007) recomendam que um URI deva ser intuitivo e apresentar um comportamento previsível para o consumidor. Por outro lado, serviços Lo-REST são os que possuem apenas o uso das duas operações básicas do HTTP (GET e POST). Além disso, eles não restringem as representações a um único formato (Pautasso et al., 2008).

Outra forma de classificação é por meio do Modelo de Maturidade de Richardson (*Richardson Maturity Model*) proposto por Leonard Richardson[3]. Seu modelo possui quatro níveis, os quais indicam a maturidade de serviços baseados no uso de URIs, HTTP e formatos de hipermídia (Webber et al., 2010). Os quatro níveis são descritos a seguir (Rauf e Porres, 2011):

- Nível 0 – Serviços usam um único URI e um único método HTTP;
- Nível 1 – Serviços empregam recursos endereçáveis, mas apenas um único método HTTP;
- Nível 2 – Serviços usam vários URIs que identificam recursos e suportam vários métodos HTTP sob esses recursos; e
- Nível 3 – Serviços possuem as mesmas características do nível 2, porém com o acréscimo da utilização de *links* para outros recursos que podem ser de interesse do consumidor. Os *links* estão contidos nas representações dos recursos.

2.6 Segurança

Ao se projetar um serviço em geral é necessário levar em consideração diversos aspectos de segurança. As principais questões de segurança envolvem (Allamaraju, 2010):

- Assegurar que apenas usuários autenticados acessem os recursos;
- Assegurar a confidencialidade e a integridade da informação do momento em que é coletado ao momento em que é armazenada;
- Prevenir clientes não autorizados ou com intenções maliciosas abusem dos recursos ou dados; e

[2]Segundo Pautasso et al. (2008), há um certo debate de qual formato de XML, ou formato de serialização como o JSON garante a melhor interoperabilidade.

[3]Richardson apresentou seu modelo durante a palestra "*Justice Will Take Us Millions Of Intricate Moves*" na QCon San Francisco 2008. Disponível em `http://www.crummy.com/writing/speaking/2008-QCon/`

- Manter a privacidade e seguir as leis que governam os diversos aspectos de segurança.

Além destas questões há também protocolos de segurança que podem ser utilizados em algumas situações, por exemplo, quando um cliente está acessando um recurso protegido em seu próprio nome ou em nome de um usuário. Os esquemas de autenticação básico e *digest*, por exemplo, adicionam uma camada de segurança ao serviço, sendo que um cliente ao tentar acessar um recurso protegido pelo servidor, este retorna o código de mensagem 401 que indica que a requisição não esta autorizada para este acesso.

A autenticação básica de HTTP consiste em definir no cabeçalho *Authentication* da requisição o usuário e senha. Embora a senha seja codificada, não são empregados processos de criptografia. Esses dados são enviados para cada requisição feita pelo usuário. Já a autenticação *digest* utiliza criptografia para enviar a senha pela rede (Allamaraju, 2010).

Outro método de autenticação é a de *token* personalizado. Este processo consiste em duas etapas:

1. Gerar um *token* único para cada usuário registrado; e

2. Enviar este *token* ao servidor a cada requisição feita pelo usuário registrado.

Embora esta técnica seja a maneira mais fácil de implementar autenticação, seu processo não é seguro. É possível copiar um *token* válido e usá-lo sem autorização, além de não existir uma forma de comprovar se a requisição é legítima (Sandoval, 2009).

Existe ainda o protocolo OAuth[4]. Este protocolo de autorização permite que serviços de terceiros possam acessar informações de usuários, mas apenas sob o seu consentimento e evitando também a passagem de usuário e senha na requisição. Basicamente esse tipo de autenticação consiste em três passos:

1. O consumidor obtém um *token* de requisição não autorizado;

2. O usuário autoriza esse *token*;

3. O consumidor troca o *token* de requisição por um *token* de acesso; e

4. O consumidor acessa o recurso protegido em nome do usuário utilizando este *token* de acesso dentro dos parâmetros da requisição.

[4] `http://oauth.net/`

2.7 Desempenho

As restrições RESTful causam impactos positivos e negativos ao desempenho do serviço. A restrição *stateless* induz as propriedades de visibilidade, confiabilidade e escalabilidade. A visibilidade é melhorada, pois um sistema de monitoramento não precisa procurar além de uma única requisição para entender o pedido. A confiabilidade aumenta porque fica mais fácil para o sistema se recuperar de falhas. A escalabilidade aumenta em virtude do servidor rapidamente liberar recursos, já que o estado entre as requisições não precisa ser armazenado, além de não ser necessário o gerenciamento dos recursos entre as requisições. Contudo, a restrição *stateless* pode reduzir o desempenho da rede. Pelo fato de que os dados não podem ser armazenados no servidor, ocorre um aumento no número de dados repetidos enviados em uma série de requisições. Além disso, colocar o estado da aplicação no lado do cliente reduz o controle do servidor sobre o comportamento consistente da aplicação (Fielding e Taylor, 2002).

O uso de *cache* aumenta a eficiência, a escalabilidade e o desempenho percebido pelo usuário. Quando é feita uma requisição por um recurso, primeiramente é verificado se o recurso já está armazenado no *cache*; caso esteja ocorre a situação chamada de *cache hit* e então a requisição não é realizada. Desta forma é possível evitar requisições feitas ao servidor. No entanto, essa restrição pode reduzir a confiabilidade caso dados obsoletos no *cache* se diferenciarem significativamente do dado contido no servidor (Fielding e Taylor, 2002).

A interface uniforme e os sistemas em camadas também possuem desvantagens. A interface uniforme degrada a eficiência já que a informação é transferida em uma forma padrão em vez de uma forma específica à necessidade da aplicação. A interface uniforme acaba não sendo ótima para outras formas de interação de arquitetura. Os sistemas em camadas têm como desvantagem a adição de cabeçalho na requisição e latência no processamento, assim reduzindo o desempenho percebido pelo usuário. Contudo, isso pode ser contornado pelo uso das restrições de *cache* (Fielding e Taylor, 2002).

Sandoval (2009) também faz algumas recomendações de desempenho para sistemas RESTful:

- Separar o tráfego da rede do tráfego da API (*Application Programming Interface*). Serviços públicos geralmente atraem muito tráfego decorrente do seu uso por aplicações de terceiros. É recomendável também que as APIs operem em servidores dedicados;

- Garantir sempre a propriedade *stateless* do serviço. Isso permite o crescimento horizontal da arquitetura (adição de mais servidores ao centro de processamento de dados);

- Escolher corretamente a representação do recurso para o problema que se deseja resolver. Algumas vezes as representações utilizadas são grandes demais para um problema que poderia ser resolvido por uma representação menor; e

- Permitir a múltipla manipulação de recursos em um mesmp URI. Permitir remover vários recursos com uma única requisição, por exemplo, evita o acesso desnecessário ao servidor.

2.8 Uso do REST

Várias empresas têm feito uso do REST para oferecer seus serviços, tais como: Google[5] e Yahoo[6] (serviços de busca), Amazon S3[7] (serviços de armazenamento de arquivos) Facebook[8] (serviços de marketing e campanhas publicitárias) e Twitter[9] (serviços de microblog). O comportamento RESTful desses serviços é fornecido como uma API (*Application Programming Interface*) pública e permite também que terceiros possam combinar e reusar os dados vindos desses serviços, assim criando uma composição das funcionalidades em um novo serviço (*mashup*) (Maleshkova et al., 2010).

Percebe-se também a presença do REST no meio acadêmico, podendo ser encontrado em diversos estudos, tais como: sistemas descentralizados (Khare e Taylor, 2004), simulação distribuída (Al-Zoubi e Wainer, 2009), aplicações na área de computação em nuvem (Christensen, 2009), aplicações na área de banco de dados relacional (Marinos et al., 2010) e uso na área geoespacial (Yang et al., 2011). Além disso, diversos eventos têm sido realizados cujas pesquisas envolvem o REST. Merecem destaque as conferências *International Workshop on RESTful Design* (WS-REST), *IEEE International Conference on Web Services* (ICWS), *IEEE International Conference on Services Computing* (SCC), *International Conference on Service Oriented Computing* (ICSOC), *European Conference on Web Services* (ECOWS) e *IEEE International Conference on Service-Oriented Computing and Applications* (SOCA). Revistas sobre o assunto são também publicadas, tais como: *IEEE Transactions on Services Computing*[10], *Service Oriented Computing and Applications Journal*[11] e a *International Journal of Web Services Research*[12].

Para que um cliente possa consumir um recurso de um serviço RESTful, ou seja, realizar requisições a esse serviço, são necessários três passos (Richardson e Ruby, 2007).

[5] `http://code.google.com/intl/pt-BR/apis/customsearch/v1/using_rest.html`
[6] `http://developer.yahoo.com/search/rest.html`
[7] `http://docs.amazonwebservices.com/AmazonS3/latest/API/`
[8] `http://developers.facebook.com/docs/reference/rest/`
[9] `http://dev.twitter.com/doc`
[10] `http://www.computer.org/tsc`
[11] `http://www.springer.com/computer/communication+networks/journal/11761`
[12] `http://www.servicescomputing.org/jwsr/`

Primeiramente, definir os dados que irão compor a requisição. Uma requisição HTTP requer a definição do método utilizado, dados do cabeçalho necessários, o URI, e a representação de recurso que irá fazer parte do corpo da requisição (*entity-body*). Este último passo se faz necessário para os métodos PUT e POST. O próximo passo é formatar os dados em uma requisição HTTP e enviar para o servidor. Grande parte das bibliotecas de clientes HTTP das linguagens de programação permite realizar essa tarefa. Na Figura 2.4 é ilustrada uma possível requisição feita a um serviço RESTful fictício. O último passo é analisar a resposta. Na Figura 2.5 é ilustrada um exemplo de resposta recebida. Ela contém o código de resposta em seu cabeçalho, o corpo da resposta e outros dados adicionais no cabeçalho.

```
POST https://www.recursos.com/recursos
Content-Type: text/xml
Authorization: Basic FHxrXkKTPMKF8iSbtIGsk8V_Q

<?xml version='1.0' standalone='yes'?>
<recurso>
        <titulo>novo_recurso</titulo>
</recurso>
```

Figura 2.4: Exemplo de requisição feita a um serviço RESTful.

```
200 OK
Content-Type: text/xml
Date: Sat, 08 Feb 2014 23:34:44 GMT
Expires: Fri, 01 Jan 1990 00:00:00 GMT
Connection: close

<?xml version='1.0' standalone='yes'?>
<recurso>
        <id> 001 </id>
        <titulo>novo_recurso</titulo>
</recurso>
```

Figura 2.5: Exemplo de resposta recebida de um serviço RESTful.

2.9 Ferramentas

A construção de serviços RESTful tem sido facilitada em consequência do surgimento de diversos *frameworks*. Na linguagem Java existe o Jersey[13] que consiste em uma implementação da especificação JAX-RS[14]. Jersey permite a implementação dos diversos

[13] `https://jersey.java.net/`
[14] `https://jax-rs-spec.java.net/`

conceitos de REST por meio das anotações (*annotations*) do Java. Por exemplo, para definir um URI que aponta para uma representação de uma lista de usuários, o código ficaria assim (Sandoval, 2009):

```
@Path("/usuarios")
public class Usuarios {
}
```

Para adicionar a restrição de comportamento interface uniforme podemos adicionar os métodos com as *annotations* de GET, POST, PUT e DELETE da seguinte maneira:

```
@Path("/usuarios")
public class Usuarios {
    @GET
    public String requisicaoGET() {
    }

    @POST
    public String requisicaoPOST(String payload) {
    }

    @PUT
    public String requisicaoPUT(String payload) {
    }

    @DELETE
    public void requisicaoDELETE() {
    }
}
```

Outro *framework* para a linguagem Java é o Restlet[15]. O mapeamento das URIs é feita utilizando uma classe que herda da classe *org.restlet.Application*. Na classe a seguir podemos ver que ela mapeia duas URIs para duas classes do tipo recurso correspondentes: /usuarios para Usuarios.class e /usuarios/nomeUsuario para Usuarios.class.

```
public class RESTWebService extends Application {
    public RESTfulWebService(Context parentContext) {
        super(parentContext);
    }
```

[15] http://www.restlet.org/downloads

```
    @Override
    public synchronized Restlet createRoot() {
        Router roteador = new Router(getContext());
        roteador.attach("/usuarios", Usuarios.class);
        roteador.attach("/usuarios/{nomeUsuario}", Usuarios.class);
        }
}
```

Para o uso dos métodos HTTP primeiramente é necessário sobrescrever os métodos *allowGet*, *allowPost*, *allowPut* e *allowDelete* acrescentando a linha *return true;* dentro método. Isso indica que essas operações passam a serem permitidas para um determinado recurso. Após isso, o comportamento de cada um desses métodos deve ser preenchido por meio da sobrescrita dos seguintes métodos: *Representation represent (Variant variant)* para o GET, *void acceptRepresentation (Representation entity)* para o POST, *void storeRepresentation (Representation entity)* para o PUT e *void removeRepresentation (Representation entity)* para o DELETE.

Outros *frameworks* que merecem destaque são o *Ruby on Rails*[16] (RoR) e o Django[17]. O sucesso do RoR se deve à grande quantidade de ferramentas que facilitam a execução de operações complexas. Versões mais atuais do RoR focam na construção de aplicações seguindo os padrões REST. O Django facilita o desenvolvimento de aplicações *web* e *Web Services* em Python. Ele é similar ao RoR, porém com menos recursos. É possível utilizá-lo para aplicar um projeto genérico de ROA na aplicação e tornar um conjunto de dados em um conjunto de recursos do tipo RESTful.

2.10 Considerações Finais

Neste capítulo foi abordado o surgimento do estilo de arquitetura REST assim como a sua arquitetura. Os conceitos e propriedades que tornam a arquitetura orientada a recurso foram descritos. Em seguida, apresentaram-se como a descrição desses serviços é feita e como eles podem ser classificados. Questões como segurança e desempenho também foram abordadas. Foram apresentados também exemplos de uso do REST e ferramentas que auxiliam em sua construção.

Como qualquer tipo de *software*, serviços RESTful precisam passar por uma rotina de teste para se garantir um certo nível de qualidade. Assim, no próximo capítulo é apresentada uma revisão bibliográfica sobre Teste de Software.

[16] http://rubyonrails.org/
[17] http://www.djangoproject.com/

Capítulo

3

Teste de Software

3.1 Considerações Iniciais

O teste de *software* é um processo vital para obter uma maior confiabilidade no sistema e, assim, garantir sua qualidade à medida que os defeitos são descobertos e corrigidos. Neste capítulo são apresentados os principais conceitos referentes ao teste de *software*. São consideradas as fases que ocorrem ao longo do processo e as principais técnicas existentes na área. Também são apresentadas ferramentas que auxiliam no teste. Neste capítulo é fornecido o embasamento teórico necessário para os trabalhos relacionados apresentados no Capítulo 4 e para o trabalho de mestrado desenvolvido.

O capítulo está organizado da seguinte forma. Na Seção 3.2 são apresentados os conceitos básicos de teste de *software*, tais como: sua definição, nomenclaturas e vantagens. Na Seção 3.3 é apresentada as diferentes fases do teste. Na Seção 3.4 é apresentada as técnicas mais conhecidas na área e seus critérios. Na Seção 3.5 são apresentadas algumas ferramentas de auxílio ao teste de *software*.

3.2 Conceitos Básicos

A maioria das atividades para a construção de um *software* dependem, de alguma forma, de pessoas para serem realizadas. Mesmo com o auxilio de ferramentas e técnicas no desenvolvimento, a presença de erros é inevitável. Em virtude disso, os testes são

fundamentais para atingir a confiabilidade do *software*. No entanto, testes não são as únicas atividades com esse objetivo, eles fazem parte de uma área maior, chamada de validação e verificação (V&V) (Pressman, 2011).

As técnicas de V&V abrangem diversas atividades para verificar se o produto, e seu modo de construção, estão de acordo com a especificação. Verificação e validação estão dentre as atividades de SQA (Garantia da Qualidade de *Software* ou *Software Quality Assurance*). As atividades de SQA buscam garantir um certo nível de qualidade ao processo de desenvolvimento e ao produto. Dentre estas atividades estão: revisões técnicas, estudo de viabilidade, simulação, revisão de documentação e monitoramento de desempenho (Pressman, 2011).

O termo "validação" está relacionado com um processo para determinar se um sistema satisfaz os requisitos. Esse processo pode ser executado durante ou após o processo de desenvolvimento. A "verificação" consiste em um processo para avaliar se a fase de desenvolvimento em que se encontra um sistema está de acordo com as condições impostas para aquela fase (IEEE, 1990). Em outras palavras, a verificação significa perguntar: "Estamos construindo o produto corretamente?", já a validação significa perguntar: "Estamos construindo o produto certo?" (Boehm, 1984).

As técnicas de V&V são divididas em estáticas e dinâmicas. As técnicas dinâmicas consistem em testes de *software*, e dependem da execução de um código ou de um modelo (Delamaro et al., 2007b). Na técnica dinâmica, ao executar o código com dados de teste, as saídas são analisadas para verificar se o comportamento do *software* está de acordo com o esperado (Sommerville, 2007). Já as técnicas estáticas, chamadas de inspeções de *software*, não requerem a execução de um programa (código) ou um modelo para serem realizadas. Elas envolvem a análise de representações do sistema, tais como: documentos de requisitos, diagramas de projeto e código-fonte do programa (Sommerville, 2007).

Teste é uma atividade para a identificação de defeitos e problemas, buscando o melhoramento da qualidade de um produto. De uma forma geral, o teste de *software* consiste em uma comparação dos resultados obtidos do *software* com o resultado esperado. Com o tempo, as práticas de teste evoluíram até um estado em que o teste não acontece somente ao final da codificação, mas é integrado na rotina de desenvolvimento e manutenção (Abran et al., 2004). O teste não melhora diretamente a qualidade do *software*, e sim indiretamente, pois ele aponta as principais fraquezas do sistema e ajuda na organização por meio de conselhos de como prosseguir, da melhor forma, durante o desenvolvimento (Broekman e Notenboom, 2002).

O teste de *software* é um elemento essencial no desenvolvimento de sistemas. É preciso ter em mente que é impossível achar todos os defeitos de um *software* e não existe tempo suficiente para testar todas as suas propriedades. Por causa disso, é necessário que seja

gasto, de uma maneira planejada, os recursos disponíveis (Broekman e Notenboom, 2002). Os testes devem ser auxiliados por um processo de teste, ou seja, um roteiro que deixe claro os passos a serem tomados para a execução do teste e quais técnicas e métodos são empregados. As estratégias de teste sempre devem incorporar as seguintes etapas: planejamento dos testes, projeto de casos de teste, execução dos testes, coleta e avaliação dos dados resultantes (Pressman, 2011).

Para criar um bom teste, o testador deve conhecer adequadamente o *software* e, assim, conseguir criar um modelo mental de como um erro pode ocorrer (Pressman, 2011). O que se espera com os testes é que seus resultados convençam desenvolvedores e clientes de que o sistema está pronto para o uso (Sommerville, 2007).

A IEEE, no padrão 610.12-1990 (IEEE, 1990), estabeleceu um glossário de termos da Engenharia de Software. Neste padrão são diferenciados os seguintes termos da área de testes: defeito, engano, erro e falha. Defeito (*fault*) significa um passo, processo ou definição incorreta de dados. O engano (*mistake*) referencia a ação humana que ocasionou o defeito. O erro (*error*) é caracterizado por um estado inconsistente ou inesperado de um programa. A falha (*failure*) é um resultado de uma execução o qual não esta de acordo com o resultado esperado. Esses termos estão relacionados da seguinte forma: um engano introduz o defeito no *software*, o defeito pode produzir um erro, e o erro pode ocasionar em uma falha. Por meio das definições se pode verificar que o defeito e o engano são conceitos estáticos, pois não estão associados com a execução do programa (Delamaro et al., 2007b).

Existe ainda uma outra atividade realizada pelos programadores chamada de *debugging*. Essa atividade tem como objetivo localizar e corrigir os defeitos que o teste revelou (Sommerville, 2007). A Figura 3.1 ilustra o processo de *debugging*.

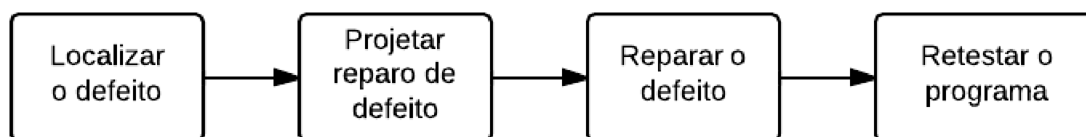

Figura 3.1: Processo de *debugging* (Adaptado de Sommerville (2007)).

O defeito primeiramente precisa ser localizado. Para isso, podem ser criados novos testes ou usar ferramentas próprias para o *debug*. Tais ferramentas mostram informações que auxiliam na localização do defeito como, por exemplo, valores de variáveis ao longo da execução do programa. Após a localização, o defeito precisa ser resolvido até que a funcionalidade esteja de acordo com o especificado. O teste, então, precisa ser reexecutado para garantir que o erro foi resolvido. Essa tarefa de reexecutar os testes é chamada de teste de regressão (Sommerville, 2007).

Outros conceitos importantes das atividades de teste são descritos abaixo (IEEE, 1990):

- Sistema em teste (SUT – *System Under Test*): O sistema em teste representa o sistema a ser testado. Pode referenciar também um subsistema ou componente;

- Caso de teste (*Test Case*): O caso de teste corresponde ao conjunto de dados de entrada e resultados esperados de um programa. Um caso de teste objetiva verificar a conformidade da funcionalidade para com um requisito;

- Conjunto de teste (*Test Set* ou *Test Suite*): É o conjunto de um ou mais casos de teste pertencentes ao sistema em teste;

- Oráculo: O oráculo, podendo ser um testador ou uma ferramenta, analisa a saída do teste com a saída esperada, para verificar se houve uma inconsistência;

- Dado de teste: É o elemento de entrada do sistema.

3.3 Fases de Teste

As atividades de teste podem ser divididas, de um modo geral, em três fases, cada uma com um objetivo específico: Teste de unidade, teste de integração e teste de sistema. O teste de unidade visa testar uma unidade do sistema. Essa unidade, componente ou módulo, consiste em uma menor parte de um *software* que pode ser testada. Cada unidade é testada separadamente de forma independente das outras. Contudo, nem sempre as unidades são independentes. Para suprir dependências de unidades são usados pseudo-controladores (*drivers*) e/ou pseudocontrolados (*stubs*) (Pressman, 2011). A Figura 3.2 ilustra como os *drivers* e os *stubs* são usados no teste de unidade. O *driver* chama o componente em teste e fornece entradas, monitora a execução e imprime resultados. Já o *stub* simula o comportamento de um módulo de que o componente em teste depende (Pressman, 2011).

Depois de testar as unidades individualmente, a próxima fase consiste em testar a comunicação entre elas; essa fase é chamada de Teste de Integração. Nessa fase são testadas as interfaces entre as unidades, ou seja, a integração entre elas. Pressman (2011) cita duas estratégias de integração: a integração descendente (*top-down*) (1), na qual a integração dos módulos é feita a partir do módulo principal (programa principal) e os subordinados são incorporados de uma maneira primeiro-em-profundidade ou primeiro-em-largura, e a integração ascendente (*bottom-up*) (2), cujos módulos mais baixos na estrutura do programa são testados por primeiro.

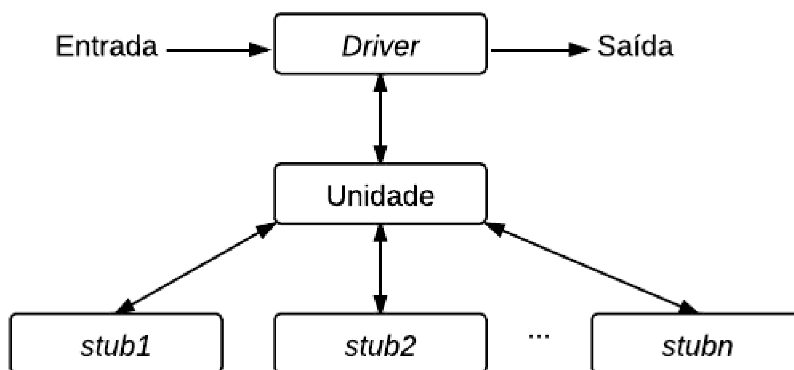

Figura 3.2: *Driver* e *stubs* aplicados no teste de unidade.

Após a integração do sistema, a próxima fase consiste em um Teste de Sistema. O objetivo desse teste é exercitar o sistema por completo. Essa fase ajuda a garantir o funcionamento correto do sistema com outros elementos como, por exemplo, o *hardware* e o banco de dados. O teste de sistema é composto de vários outros tipos de teste, tais como: teste de recuperação, teste de segurança, teste por esforço, teste de desempenho e teste de disponibilização (Pressman, 2011).

Além dessas fases, vale ressaltar outra fase importante chamada Teste de aceitação. O Teste de aceitação consiste em uma avaliação do sistema pelo cliente para a sua aceitação ou não, levando em consideração os requisitos. Nesta fase, o sistema é testado com dados fornecidos pelo cliente, e com isso é possível que erros nos requisitos do sistema sejam identificados (Sommerville, 2007).

A Figura 3.3 ilustra um processo de teste simples que consiste em três fases: teste de unidade, teste de sistema e teste de aceitação. Primeiramente as unidades do sistema são testadas individualmente, depois elas são integradas a subsistemas e sistemas, e testadas para verificar se suas interações estão corretas. Ao final, na entrega ao cliente, o sistema é testado para verificar se ele está de acordo com o que foi especificado.

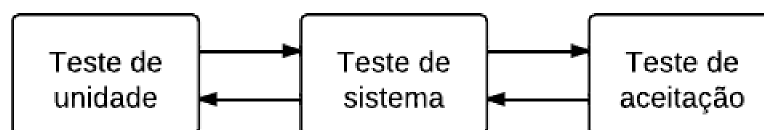

Figura 3.3: Processo de teste (Sommerville (2007)).

3.4 Técnicas e Critérios de Teste

O cenário ideal de teste seria o de testar um programa com todas as suas possíveis entradas ou executar os possíveis caminhos lógicos. Porém, esse tipo de teste, chamado de teste exaustivo, é impossível de ser executado para programas grandes, e mesmo para

pequenos programas o conjunto de caminhos lógicos possíveis pode ser muito grande, exigindo assim uma grande quantidade de tempo para se executar (Pressman, 2011).

Devido à inviabilidade dos testes exaustivos, é importante que os testes de um programa P, sejam direcionados a pontos específicos dentro do domínio de entrada $D(P)$. A ideia é que o domínio de entrada $D(P)$ seja particionado em subdomínios, sendo que os elementos (dados de teste) de cada subdomínio sejam semelhantes. Dessa forma, o conjunto de testes T será criado com o objetivo de testar um elemento de cada subdomínio, resultando em um conjunto bem menor com relação à $D(P)$ (Delamaro et al., 2007b).

Para selecionar os elementos dos subdomínios existem duas maneiras: o teste aleatório (1), cujo conjunto de teste T é formado aleatoriamente com o objetivo de se ter uma boa chance de representar os subdomínios de $D(P)$, e o teste de partição ou teste de subdomínios (2), o qual consiste em uma definição dos subdomínios e uma criação dos casos de teste para cada um desses subdomínios (Delamaro et al., 2007b).

Existem diferentes regras para identificar os subdomínios de $D(P)$; essas regras são chamadas de "critérios de teste". Cada critério possui um conjunto de "requisitos de teste". Os elementos são agrupados em um mesmo subdomínio caso satisfaçam um determinado requisito do critério. Quando o conjunto de teste T apresenta ao menos um caso de teste para cada subdomínio, é dito que esse conjunto satisfaz todos os requisitos de um critério de teste C, ou seja, ele é "C-adequado" (Delamaro et al., 2007b).

Nas subseções a seguir são apresentadas quatro técnicas de teste: funcional, estrutural, baseado em defeitos e baseado em modelos. Cada uma dessas técnicas utiliza informações diferentes para estabelecer seus subdomínios. O teste funcional leva em consideração a especificação do *software*, o estrutural considera as características internas da implementação, o baseado em erros considera os erros mais comuns encontrados durante o desenvolvimento, e o baseado em modelos considera uma especificação formal do sistema para, por exemplo, a verificação de propriedades da especificação (Delamaro et al., 2007b). As subseções a seguir apresentam mais detalhes sobre cada uma dessas técnicas.

3.4.1 Técnica de Teste Funcional

A técnica funcional, também conhecida como teste de caixa-preta, tem como base a especificação do sistema para sua definição e aplicação, e objetiva a validação dos requisitos funcionais do *software*. A natureza desse tipo de teste faz com que o foco se mantenha no domínio das informações do sistema (Pressman, 2011).

A técnica funcional pode ser realizada em qualquer fase do teste, e é independente de paradigma ou linguagem de programação. É importante que a especificação do *software* esteja bem construída e detalhada, pois a saída obtida do teste (o comportamento do *software*) é comparada com a sua especificação (Fabbri et al., 2007). Segundo Myers et

al. (2004), no teste caixa-preta, o testador está totalmente despreocupado com a estrutura e comportamento interno do programa; o que caracteriza um erro será a não conformidade do comportamento do programa com a sua especificação. Seria possível usar essa técnica para encontrar todos os erros de um programa usando todas as possíveis entradas como casos de teste (teste exaustivo). Porém, essa prática é inviável pelo fato de que o domínio de entrada, na maioria das vezes, é muito grande ou até mesmo infinito.

No intuito de diminuir o domínio de entrada, diversos critérios foram propostos. Um deles é o Critério de Particionamento em Classes de Equivalência, no qual o domínio de entrada é dividido em classes de equivalência para torná-lo finito. O sistema é testado a partir de um elemento de cada classe de equivalência definida, pois se pode assumir que todos os elementos daquela classe se comportam da mesma maneira (Fabbri et al., 2007). Para exemplificar este critério, foi extraída a especificação do programa "Cadeia de Caracteres" de (Fabbri et al., 2007): "O programa solicita do usuário um inteiro positivo no intervalo entre 1 e 20 e então solicita uma cadeia de caracteres desse comprimento. Após isso, o programa solicita um caractere e retorna a posição na cadeia em que o caractere é encontrado pela primeira vez ou uma mensagem indicando que o caractere não está presente na cadeia. O usuário tem a opção de procurar vários caracteres."

A partir dessa especificação, nota-se que o programa possui quatro variáveis de entrada: Tamanho da cadeia de caracteres (T), cadeia de caracteres (CC), caractere a ser procurado (C) e uma opção por procurar mais caracteres (O). A classe de equivalência válida da variável T consiste em valores de entrada entre 1 e 20, e a classe não válida são valores fora desse intervalo. Já as classes de equivalência de O são os valores "sim" e "não". As variáveis C e CC não possuem classes de equivalência, pois quaisquer caracteres podem ser inseridos. A Tabela 3.1 mostra as variáveis com suas respectivas classes de equivalência válidas e inválidas.

Tabela 3.1: Classes de equivalência da especificação Cadeia de Caracteres (Fabbri et al. (2007)).

Variável de entrada	Classes de equivalência válidas	Classes de equivalência inválidas
T	$1 \leq T \leq 20$	$T < 1$ e $T > 20$
O	Sim	Não
C	Caractere que pertence à cadeia	caractere que não pertence à cadeia

Após a definição das classes, o próximo passo é gerar casos de teste cujos valores de entrada façam parte das classes de equivalência. É importante que cada caso de teste cubra o maior número possível de classes válidas, e casos de teste para as classes inválidas não

devem incluir valores dentro das válidas, pois o erro encontrado por uma classe inválida pode disfarçar um erro de uma classe válida.

Outro critério complementar é o de Análise de Valor Limite, que leva em consideração os valores situados nos limites das classes de equivalência, e os valores subsequentes das classes vizinhas inválidas. O motivo disso é que ao explorar os limites das classes é mais provável encontrar defeitos. Um exemplo de casos de teste para este critério seria o de testar os valores 0, 1, 20 e 21 da variável T do programa "Cadeia de Caracteres" (Fabbri et al., 2007).

Há também o critério de Teste Funcional Sistemático, o qual consiste em uma combinação dos dois critérios anteriores: particionamento em classes de equivalência e análise de valor limite. Quando ocorre o particionamento, o teste funcional sistemático exige no mínimo dois casos de teste de cada partição. Para a construção dos casos de teste, este critério fornece diretrizes relacionadas a arranjos, números reais, valores ilegais, dentre outros (Fabbri et al., 2007).

Por último, o critério de Grafo Causa-Efeito explora as combinações dos dados de entrada do programa. Essa combinação pode ser considerada uma limitação nos critérios anteriores. A partir da especificação são identificadas as chamadas "causas" e "efeitos" do programa. As causas são as entradas ou estímulos do sistema e os efeitos são as respostas ou saídas do sistema. Após isso, é criado o grafo que liga as causas e os efeitos, chamado de "Grafo Causa-Efeito". O grafo, então, é convertido em uma tabela de decisão, usada para derivar os casos de teste (Fabbri et al., 2007).

3.4.2 Técnica de Teste Estrutural

A técnica estrutural, também chamada de teste caixa-branca, consiste em avaliar a estrutura interna, ou seja, a implementação do código a fim de gerar os casos de teste para satisfazer um determinado critério. A maioria dos critérios de teste estrutural se baseia no Grafo de Fluxo de Controle (GFC) ou Grafo de Programa, o qual é uma representação, em forma de grafo, dos comandos internos e estruturas do código (Barbosa et al., 2007).

Os casos de teste são derivados baseados na lógica do programa; testar essa lógica exaustivamente significaria testar todos os possíveis caminhos a partir dos fluxos de controle do programa, o que na prática torna-se inviável em decorrência do grande número de possíveis caminhos lógicos (Myers et al., 2004).

Um GFC $G = (N, E, s)$ é um grafo que representa um programa P, onde N representa um conjunto de nós, E um conjunto de arcos e $s \in N$ o nó de entrada. Um nó representa uma ou mais instruções no programa (blocos), tal que não existe qualquer desvio condicional para outras instruções, ou seja, as instruções são executadas todas de modo sequencial. Existe apenas um nó de entrada $n \in N$ e um nó de saída $o \in N$. Os arcos,

ou arestas, são as ligações entre os nós, e representam uma possível continuação da execução das instruções de um nó para outro. Os caminhos são uma sequência finita de nós $(n_1, n_2, ..., n_k)$, $k \geq 2$, tal que existe um arco de n_i para $n_i + 1$ para $i = 1, 2, ..., k - 1$. Um caminho é dito simples, se todos os nós deste caminho, exceto possivelmente o primeiro e o último, são distintos. Caso todos os nós sejam distintos, esse caminho é chamado de caminho livre de laço. Um caminho completo é um caminho que começa com o nó de entrada e termina com o nó de saída. Há ainda os chamados caminhos não executáveis, que são caminhos no qual a execução de pelo menos um nó é impossível de ser realizada (Barbosa et al., 2007).

Para exemplificar o critério estrutural, a Figura 3.4 mostra o código do método principal (*main*) do programa *Identifier* extraído de (Barbosa et al., 2007). Este programa determina se um identificador é válido ou não. Um identificador válido deve conter apenas letras ou dígitos e deve começar com uma letra. Além disso, o identificador deve ter o comprimento no mínimo de um caractere e no máximo de seis caracteres.

```
 1 /*01*/    main ()
 2 /*01*/    {
 3 /*01*/        char achar;
 4 /*01*/        int length, valid_id;
 5 /*01*/        length = 0;
 6 /*01*/        valid_id = 1;
 7 /*01*/        printf ("Identificador: ");
 8 /*01*/        achar = fgetc (stdin);
 9 /*01*/        valid_id = valid_s(achar);
10 /*01*/        if(valid_id)
11 /*02*/        {
12 /*02*/            length = 1;
13 /*02*/        }
14 /*03*/        achar = fgetc (stdin);
15 /*04*/        while(achar != '\n')
16 /*05*/        {
17 /*05*/            if(!(valid_f(achar)))
18 /*06*/            {
19 /*06*/                valid_id = 0;
20 /*06*/            }
21 /*07*/            length++;
22 /*07*/            achar = fgetc (stdin);
23 /*07*/        }
24 /*08*/        if(valid_id && (length >= 1) && (length < 6))
25 /*09*/        {
26 /*09*/            printf ("Valido\n");
27 /*09*/        }
28 /*10*/        else
29 /*10*/        {
30 /*10*/            printf ("Invalido\n");
31 /*10*/        }
32 /*11*/    }
```

Figura 3.4: Código do *Identifier* (Barbosa et al. (2007)).

Neste código os números presentes nos comentários representam o nó que a instrução pertence. A Figura 3.5 apresenta o GFC do programa. Esse GFC contém no total 11 nós e 14 arestas. O caminho (2,3,4,5,6,7) é um caminho simples e livre de laços. O caminho (1,2,3,4,5,7,4,8,9,11) é um caminho completo. O caminho (1,3,4,8,9) é um caminho não executável. O nó 1 e o nó 2 consistem respectivamente em nó inicial e final. A Figura

3.6 mostra os diferentes tipos de GFC que podem ser formados dependendo da estrutura sequencial, condicional ou de repetição do código.

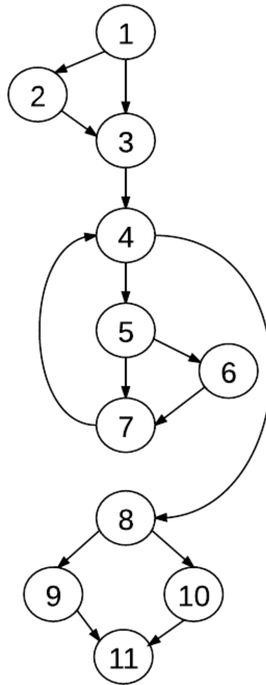

Figura 3.5: GFC da função *main* do programa *Identifier* (Adaptado de Barbosa et al. (2007)).

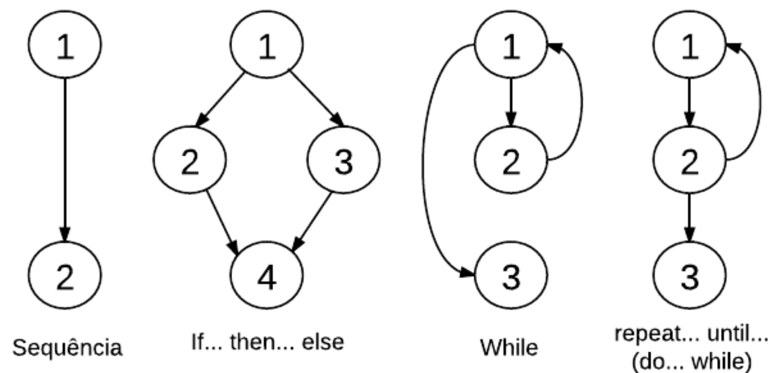

Figura 3.6: GFCs de estruturas sequenciais, condicionais e de repetição (Barbosa et al. (2007)).

O GFC é a base para a maioria dos critérios estruturais. Uma das famílias de critérios estruturais existentes, é chamada de critérios baseados em fluxo de controle. Estes critérios levam em consideração as características do controle da execução fazendo uso do GFC. Os principais critérios desta família são os (Barbosa et al., 2007):

- Todos-Nós – Requer que os casos de teste executem ao menos uma vez cada vértice do GFC;

- Todos-Arcos – Requer que os casos de teste executem ao menos uma vez cada aresta do GFC; e
- Todos-Caminhos – Requer que todos os caminhos possíveis do GFC sejam executados.

Outra família de critérios é chamada de critérios baseados na complexidade, nos quais é levada em consideração a complexidade do programa para a derivação dos requisitos de teste. Um dos critérios que se destaca é o critério de McCabe (McCabe, 1976) o qual faz uso da chamada "complexidade ciclomática" do GFC. A complexidade ciclomática consiste em uma métrica para a complexidade lógica do programa; ela oferece um limite máximo para o conjunto de casos de teste a ser derivados para que garanta a execução de todas as instruções do programa ao menos uma vez. Para calcular a complexidade ciclomática $V(G)$ de um programa, pode-se adotar uma das três estratégias abaixo (Barbosa et al., 2007):

1. Número de regiões contidas no grafo GFC. Essas regiões são todas áreas delimitadas mais a área não delimitada fora do grafo;
2. $V(G) = E - N + 2$, onde E é o número de arcos e N o número de nós do grafo; ou
3. $V(G) = P + 1$, onde P é o número de nós predicativos contido no grafo.

A última família de critérios são os critérios baseados em fluxo de dados. Estes critérios buscaram propor uma hierarquia entre os critérios todos-arcos e todos-caminhos, pois mesmo em programas pequenos os critérios baseados em fluxo de controle não eram capazes de identificar defeitos simples e triviais. Os critérios baseados em fluxo de dados podem ser divididos na família de critérios proposta por Rapps e Weyuker (1985) (Todas-Definições, Todos-Usos e Todos-Du-Caminhos) e a família de critérios potenciais-usos proposta por Maldonado (1991) (Barbosa et al., 2007). Os critérios da família de fluxo de dados fazem uso de duas informações adicionais que não estão presentes nas outras famílias. Essas informações são as definições e usos de variáveis, as quais são incorporadas no GFC gerando um "Grafo Def-Uso".

A Figura 3.7 mostra o Grafo Def-Uso da função *main* do programa *Identifier*. As definições de variável ocorrem quando o seu valor é armazenado na memória. Isso geralmente acontece quando é atribuído valor à variável. Quando a variável estiver sendo referenciada no código, mas não estiver ocorrendo uma definição, então é dito que está ocorrendo o uso desta variável. Existem dois tipos de uso: "c-uso" (uso computacional) e "p-uso" (uso predicativo). O uso computacional significa que ocorreu uma computação daquela variável como, por exemplo, uma alteração de valor. Já o uso predicativo está

relacionado ao controle do fluxo do programa como, por exemplo, o uso em um laço de repetição ou condicional (Barbosa et al., 2007).

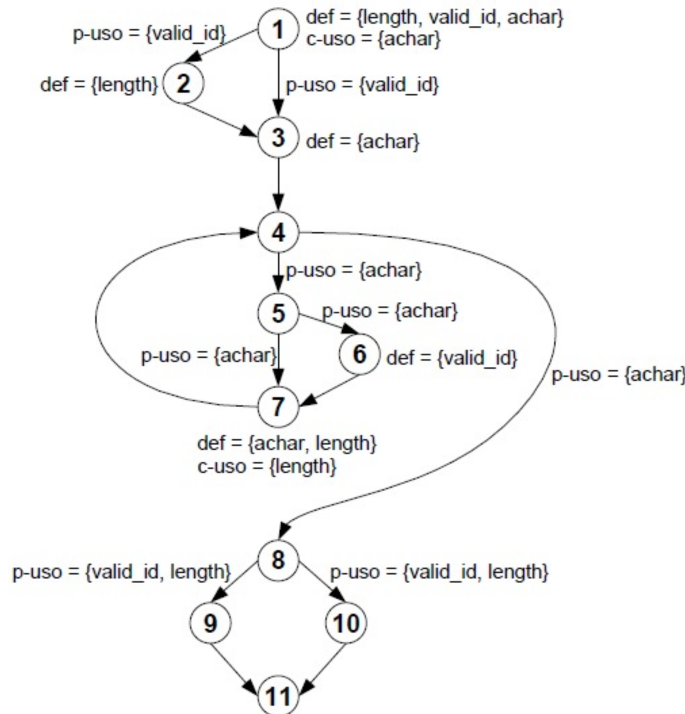

Figura 3.7: Grafo Def-Uso da função *main* (Barbosa et al. (2007)).

3.4.3 Técnica de Teste Baseado em Defeitos

Nessa técnica são utilizados os defeitos mais comuns encontrados no desenvolvimento de *software* para derivar os requisitos de teste. Um dos critérios mais conhecidos é a análise de mutantes ou teste de mutação (*Mutation Analysis*) (DeMillo et al., 1978). A análise de mutantes consiste em gerar um conjunto de programas "mutantes" por meio de pequenas alterações feitas no código, ou seja, defeitos inseridos propositalmente no programa. O nível de confiança dos casos de teste é medido baseado em uma relação entre o número de mutantes mortos (os que apresentaram comportamento diferente do programa original) e o número de mutantes gerados. O teste de mutação geralmente é mais utilizado no teste de unidade, porém ele pode ser adaptado para outras fases do teste. (Delamaro et al., 2007a).

A ideia do teste de mutantes é que cada mutante representa um subdomínio do domínio de entrada, e, diferentemente das outras técnicas, este subdomínio está ligado diretamente a um defeito específico. O testador deve selecionar casos de teste que demonstrem que o programa mutante possui um comportamento diferente do programa original. Para a aplicação da técnica são necessários os seguintes passos: geração dos mutantes, execução

do programa em teste, execução dos mutantes e análise dos mutantes. Esses passos, no entanto, são difíceis de serem realizados sem o auxílio de ferramentas. Essas ferramentas auxiliam na criação de casos de teste, execução dos mutantes e análise do resultado da execução, tarefas das quais exigiriam muito esforço e tempo do testador caso fossem feitas manualmente. Vale ressaltar que mesmo com o uso de ferramentas, programas podem gerar um grande número de mutantes, exigindo assim um alto tempo de execução (Delamaro et al., 2007a).

No geral, a técnica gera mutantes $P_1, P_2, ..., P_n$ baseados um determinado programa P. Os mutantes são gerados por meio de operadores de mutação (*mutation operators*), eles determinam as mudanças que irão ocorrer no código. Alguns operadores de mutação na linguagem C, por exemplo, eliminam comandos, trocam de operador relacional e trocam de variáveis escalares. Após isso, são executados os mutantes contra um conjunto de casos de teste T. Se o mutante apresentar um comportamento diferente do programa original, então esse mutante é “morto”. Os mutantes “vivos” devem permanecer armazenados até serem eventualmente mortos por novos casos de teste. Ao final, a avaliação da adequação do conjunto de casos de teste T para um programa P, é realizada por meio de um “escore de mutação” (*mutation score*). Um escore de mutação $ms(P,T)$ é calculado usando a seguinte fórmula (Delamaro et al., 2007a):

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)}$$

, sendo:

- $DM(P,T)$: número de mutantes mortos pelo conjunto de casos de teste T;
- $M(P)$: número total de mutantes gerados a partir do programa P; e
- $EM(P)$: número de mutantes gerados que são equivalentes a P.

O escore é calculado a partir da razão entre o número de mutantes mortos pelo conjunto de casos de teste T e o número de mutantes não equivalentes gerados. O resultado do cálculo varia no intervalo entre 0 e 1, quanto maior o escore mais adequado é o conjunto de casos de teste para o programa. Com isso, torna-se possível também mensurar a confiança no programa testado (Delamaro et al., 2007a).

3.4.4 Técnica de Teste Baseado em Modelos

Uma prática comum quando se testa um sistema é de usar primeiramente as técnicas de caixa-preta para projetar os casos de teste, e depois as técnicas de teste caixa-branca para detectar partes do sistema que ainda não foram propriamente testadas para melhorar ainda mais os casos de teste (Utting e Legeard, 2007). A crescente pressão sobre

os desenvolvedores e testadores por ciclos de lançamentos mais curtos, as exigências de clientes por *software* de maior confiabilidade e a concorrência no mercado exigindo redução de custos, foram alguns fatores que influenciaram a pesquisa por melhores formas de geração de casos de teste do que a abordagem tradicional (construção manual). O teste baseado em modelos é uma das formas de fornecer essa geração automática de testes. Por meio dele, os testadores se concentram principalmente no modelo do sistema e em uma infraestrutura de geração dos casos de teste (Dalal et al., 1999).

A técnica de Teste Baseado em Modelos (TBM) é um tipo de teste caixa-preta, pois ela gera os casos de teste por meio de um modelo construído a partir da especificação do *software*. Este tipo de teste é capaz de automatizar a geração de oráculos de teste e também de casos de teste (Utting e Legeard, 2007). Segundo Sinha e Smidts (2006), o teste funcional (caixa-preta) pode ser automatizado por meio da geração de casos de teste com base em um modelo estrutural ou comportamental do sistema, chamado de modelo de teste. Essas abordagens são, portanto, coletivamente conhecidas como Teste Baseado em Modelos.

A Figura 3.8 ilustra algumas das características do TBM. Pode-se perceber que ele engloba o teste funcional, e pode ser utilizado em qualquer escala que o sistema se encontra, desde o teste de unidade até o teste de sistema. Na primeira fase, teste de unidade, um modelo pode ser construído de um módulo descrevendo os parâmetros de entrada. A partir disso, um conjunto de testes pode ser gerado rapidamente. Na fase intermediária, teste de integração, é possível testar o comportamento fornecido pela integração das unidades. A geração de testes, nessa fase, pode utilizar um modelo dos dados de entrada, e isso permite o cálculo das saídas esperadas, evitando a necessidade de criação de um oráculo. Na última fase, teste de sistema, é possível usar um modelo comportamental do sistema para a geração dos testes (Dalal et al., 1999)

O teste baseado em modelos consiste nos seguintes passos (Pressman, 2011; Utting e Legeard, 2007):

1. Criar um modelo do *software* ou de seu ambiente;

2. Percorrer o modelo e especificar as entradas (casos de teste) que forçarão o *software* a uma transição de estado;

3. Rever o modelo observando as saídas e mudanças de estado do *software*, e compará-las com as saídas esperadas de acordo com o modelo;

4. Executar os casos de teste; e

5. Comparar os resultados reais com os esperados e tomar as ações necessárias para corrigir defeitos encontrados.

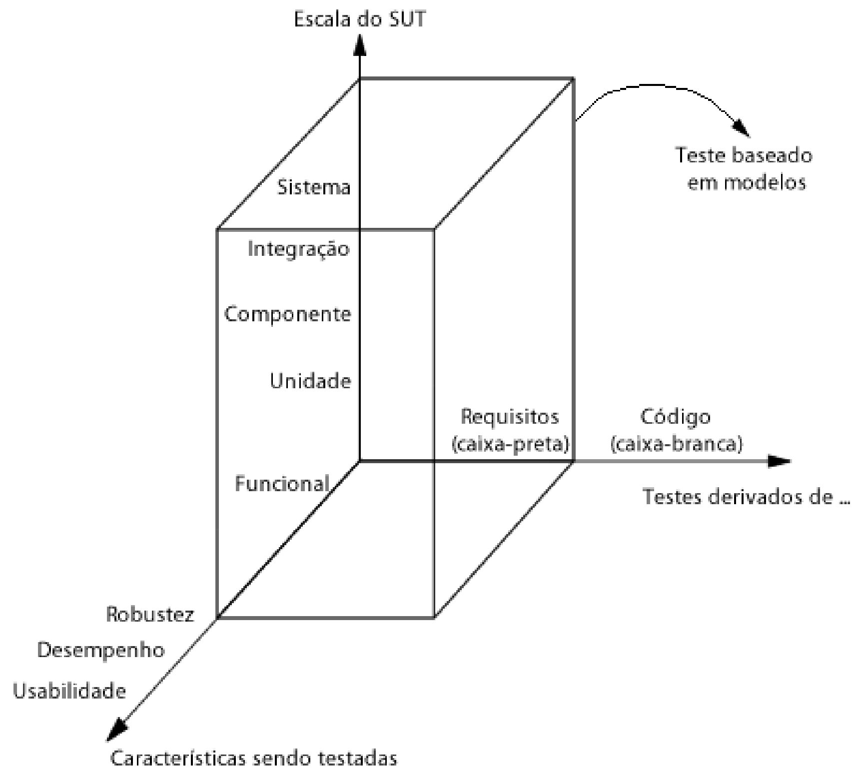

Figura 3.8: Visão geral do TBM (Adaptado de Utting e Legeard (2007)).

Testes tradicionais são criados baseados em um documento de especificação do sistema. No entanto, esse tipo de documento possui a tendência de apresentar especificações incompletas, ambíguas e contraditórias, principalmente pelo fato de serem escritos em linguagem natural. Gerar testes baseados nesse tipo de documento pode apresentar problemas ao longo do processo de desenvolvimento. Ao se usar o TBM, o comportamento do sistema precisa ser modelado de uma forma precisa para poder gerar os casos de teste. Muitas vezes esses modelos são tão precisos que passam a ser executáveis e servem como um protótipo ao sistema (Pretschner et al., 2005). Os modelos também podem ser usados para codificar requisitos não funcionais (relacionados às propriedades do sistema), tais como o desempenho e segurança (Utting et al., 2011).

O TBM depende de três tecnologias importantes: a notação usada para o modelo, o algoritmo de geração de testes e ferramentas que geram a infraestrutura de apoio para os testes. O modelo pode ser desenvolvido nas primeiras fases do ciclo de desenvolvimento do *software* a partir da especificação; isso também ajuda na detecção de defeitos em fases iniciais do processo de desenvolvimento (Dalal et al., 1999). Embora os modelos necessitem de precisão, eles devem ser mais abstratos que o SUT. O modelo deve ser mais simples que o sistema, ou ao menos mais fácil de verificar, modificar e manter (Utting et al., 2011). A abstração pode ser realizada por meio de encapsulamento ou por omissão de detalhes, isso ocasiona a eventual perda de certas informações do sistema (Pretschner et al., 2005).

Os casos de teste são gerados a partir da seleção de rastros (*traces*) do modelo, os quais possuem informações das entradas e saídas esperadas (Pretschner et al., 2005). Os pares entrada/saída do modelo são interpretados como casos de teste, cuja saída do modelo corresponde à saída esperada do SUT. Os testes produzidos pelos modelos são testes abstratos e precisam ser transformados em testes executáveis. Essa tarefa requer a intervenção do testador, porém a maioria das ferramentas para testes baseado em modelos auxiliam nesse processo (Utting e Legeard, 2007). Vale ressaltar, que é essencial que o modelo do SUT tenha sido validado, ou seja, verificar que o modelo correto do SUT tenha sido construído. A validação também implica em uma análise dos requisitos por consistência e completude (Utting et al., 2011).

As principais vantagens do TBM são (Utting e Legeard, 2007):

- Redução de custo e tempo no processo de teste;
- Teste sistemático – Provê uma sistematização no processo de teste, evitando, assim, processos *ad-hoc*;
- Detecção precoce de falhas e ambiguidades em especificações;
- Resposta rápida a requisitos voláteis (requisitos com grande chance de sofrer mudanças durante o processo de desenvolvimento): Requer apenas uma mudança no modelo para depois gerar os testes novamente;
- Rastreabilidade automatizada: ajuda a explicar os casos de teste, relacionando-o com o modelo ou com o critério de seleção de teste; e
- Melhoria na qualidade dos testes.

Já as limitações ou desvantagens da técnica do TBM são:

- Alta curvatura de aprendizado;
- Documentos de requisitos ultrapassados geram modelos ultrapassados do SUT;
- Dificuldades na modelagem de partes específicas de um sistema podem causar o uso inapropriado do TBM;
- Necessidade de tempo para analisar testes que falharam; e
- Métricas relacionadas a quantidade de casos de teste criados, para medir como os testes estão progredindo, tornam-se inúteis. Isso ocorre graças à facilidade de gerar vários casos de teste rapidamente.

As ferramentas de teste baseado em modelos são muito importantes no processo de teste. Elas fazem uso de diversos algoritmos de geração de testes e estratégias para gerar os testes a partir do modelo comportamental. As ferramentas podem automaticamente gerar casos de teste, dados de teste e oráculos a partir do modelo. O testador também pode controlar o número de casos de teste que são gerados (Utting e Legeard, 2007). Existem casos de uso do TBM na indústria em empresas conhecidas como a IBM – usando para a redução de custos nos testes de seus sistemas – e a Microsoft – usando para a construção de ferramentas e no uso interno (Utting et al., 2011).

É importante que o modelo do sistema em teste seja formal o suficiente para permitir a derivação de casos de teste. Existem diversas notações para modelar sistemas, tais como: notações de pre/pos (ou baseadas em estado, ou baseadas em modelo), notações baseadas em transições, em estado, em história, notações funcionais, notações operacionais, dentre outras (Utting e Legeard, 2007). Por este trabalho de mestrado utilizar máquinas de estados de UML, a seguir são apresentados alguns dos modelos mais utilizados e conhecidos no TBM que possui uma notação baseada em transições.

3.4.4.1 Máquinas de Estados Finitos

Máquinas de Estados Finitos (MEF) ou *Finite State Machines* (FSM) são máquinas hipotéticas formadas por estados e transições. Cada transição estabelece uma conexão entre um estado a e um estado b, sendo que ambos podem ser o mesmo estado. As transições são percorridas na máquina por meio de uma entrada, ao ler essa entrada ocorre um consumo de símbolo de entrada e uma produção de símbolo de saída. Essa relação entre os símbolos de entrada e saída é expressa por um rótulo contido nas transições (Simão, 2007). Uma MEF pode ser definida formalmente como uma tupla $M = (X, Z, S, s_0, f_z, f_s)$, sendo:

- X é um conjunto finito não vazio de símbolos de entrada;
- Z é conjunto finito não vazio de símbolo de saída;
- S é um conjunto finito não vazio de estados;
- s_0 é o estado inicial, sendo que $s_0 \in S$;
- f_z é a função de saída, sendo que $f_z : (S \times X) \to Z$;
- f_s é a função do próximo estado, sendo que $f_s : (S \times X) \to S$.

Existem dois tipos de MEFs, as máquinas de Mealy e as de Moore. Máquinas de Mealy, ao ocorrer um evento de entrada, têm sua saída associada à transição de um

estado para outro. Já nas máquinas de Moore, sua saída está associada com o estado, mais especificamente, ao estado de destino após a transição. Embora os dois tipos de máquinas sejam similares, as máquinas de Mealy são mais adequadas para modelar sistemas de estados finitos e são mais genéricas que as máquinas de Moore (Lee e Yannakakis, 1996).

As MEFs possuem dois tipos de representação: por diagramas de transição de estados e por tabelas de transição. Os diagramas de transição de estados são grafos cujos círculos representam os estados e os arcos direcionados representam as transições. As transições apresentam rótulos do tipo *entrada:saída*, *entrada* sendo o símbolo necessário para ocorrer a transição, e *saída* o símbolo gerado como saída. Nas tabelas de transição, os estados são as linhas da tabela e as entradas são as colunas (Simão, 2007). Na Figura 3.9 é apresentado um diagrama de estados de uma MEF extraída de Chow (1978), com quatro estados e nove transições que extrai comentários de uma sequência de caracteres. Nesse diagrama, os símbolos de entrada são compostos por '*', '/' e 'v', sendo 'v' qualquer caractere diferente de '*' e '/'. Os comentários são quaisquer cadeias de caracteres entre '/*' e '*/' (Simão, 2007).

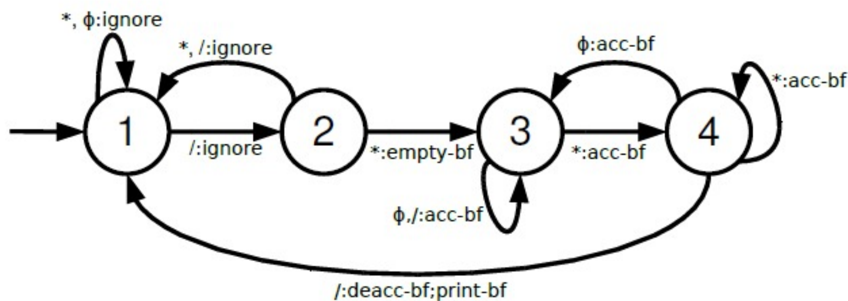

Figura 3.9: MEF para um extrator de comentários (Chow (1978)).

Cada MEF possui algumas propriedades relacionadas à sua estrutura, tais como (Simão, 2007):

- Completude: Uma MEF é dita "completamente especificada" se todos os estados tratam todas as entradas, caso contrário ela é dita "parcialmente especificada";

- Conectividade: Uma MEF é "fortemente conexa" caso seja possível estabelecer um caminho por transições para cada par de estados da máquina. Caso qualquer estado possa ser atingido a partir do estado inicial, é dita que a MEF é "inicialmente conectada";

- Minimalidade: Uma MEF é "minimal" (ou reduzida) caso não existam quaisquer dois estados equivalentes. A equivalência de estados ocorre quando dois estados produzem as mesmas saídas para quaisquer sequências de entrada; e

- Determinismo: Uma MEF é "determinística" quando, em qualquer estado, existe apenas uma única transição para cada entrada específica.

O teste que se baseia em máquinas de estados finitos deve confrontar uma MEF M reduzida com uma MEF I reduzida, que representa a MEF implementada. Ao se comparar uma MEF minimal M, que representa a versão correta da MEF especificada, com a MEF I, é possível acontecer os seguintes erros (Chow, 1978):

- Erros de saída: Quando I não é equivalente a M, e I pode ser modificada para ser equivalente a M, por meio da mudança da função de saída de I;
- Erros de transferência: Quando I não é equivalente a M, e I pode ser modificada para ser equivalente a M, por meio da mudança de transição de I;
- Erro de transição: Quando há um erro de saída, e/ou de transferência;
- Erro de estado inicial: Quando I não é equivalente a M, e I pode ser modificada para ser equivalente a M, por meio da mudança do estado inicial de I;
- Estados extras: Quando I não é equivalente a M, e I pode ser modificada para ser equivalente a M, por meio da diminuição de estados de I; e
- Estados ausentes: Quando I não é equivalente a M, e I pode ser modificada para ser equivalente a M, por meio do incremento de estados de I.

Quando um sistema é modelado por uma MEF, os casos de teste passam a ser descritos como uma sequência de ações necessárias que passam por uma sequência de estados. Por sua vez, as saídas do caso de teste passam a ser as saídas esperadas das transições realizadas pela sequência de ações. Essa sequência recebe o nome de "sequência de entrada". Existem diferentes métodos de geração automática das sequências de entrada, cada um com suas próprias características, custo de geração das sequências de teste, custo de execução dos testes gerados e garantias de cobertura de defeitos. Os métodos de geração têm como objetivo testar se a MEF está de acordo com a especificação.

Existem outros tipos de sequências que geram resultados parciais importantes para os métodos de geração, tais como: as sequências de sincronização (SS), as de distinção (DS), as sequências únicas de estrada e saída (UIO) e os conjuntos de caracterização. As SS servem como uma função de "*reset*" para a MEF, pois por meio dessa sequência é possível voltar para o estado inicial não importando em qual estado a MEF se encontra. As DS são sequências que produzem saídas únicas para cada um dos estados da MEF. Com isso é possível determinar em qual estado a MEF estava originalmente. As UIO são uma forma de fornecer a distinção entre os estados quando a MEF não apresenta uma

DS. Cada estado apresenta uma UIO distinta que ajuda a verificar em qual estado a MEF está situada. Os conjuntos de caracterização consistem em um conjunto de sequências de entrada que, em conjunto, servem para identificar qual era o estado original da MEF.

Alguns exemplos de métodos de geração das sequências de teste são (Simão, 2007):

- Método TT (*Transition Tour*) – O método TT é um método de geração simples, o qual gera um conjunto de sequências de teste que parte do estado inicial, percorre toda a MEF e retorna ao estado inicial. Com isso, é possível garantir que todas as transições da MEF foram percorridas. Esse método permite a detecção de erros de saída, mas não os de transferência.

- Método DS (Sequência de distinção) – O método DS faz uso das sequências de distinção e de sincronização para gerar as sequências de teste da MEF, logo sua aplicação depende da existência dessas sequências na MEF. O método utiliza uma sequência de verificação composta por uma sequência-α e uma sequência-β, responsáveis respectivamente pela verificação de todos os estados da MEF e de todas as suas transições.

- Método UIO – O método UIO utiliza uma sequência de identificação de estado (UIO) para a geração do conjunto de sequências de entrada. Ele não garante a cobertura total de erros e necessita que as MEFs possuam sequências UIO.

- Método W – O método W é um dos métodos mais conhecidos. Ele faz uso do conjunto de caracterização para verificar se os estados e transições estão corretamente implementados. Este método garante a detecção dos erros estruturais quando a MEF for completa, mínima e fortemente conexa.

3.4.4.2 Máquinas de Estados Finitos Estendidas

Máquinas de Estados Finitos Estendidas (MEFE) ou *Extended Finite State Machines* (EFSM) acrescentam às MEFs conceitos de parâmetros de entrada e saída, variáveis de contexto, operações e predicados definidos sobre as variáveis de contexto e parâmetros de entrada. As MEFEs permitem a criação de transições dependentes de parâmetros. É dito que a MEF subjacente à MEFE modela o fluxo de controle de um sistema, enquanto que os parâmetros, variáveis, predicados e as operações refletem seu fluxo de dados. Os parâmetros consistem das entradas da MEFE e os predicados indicam as possíveis transições (Petrenko et al., 2004). Tanto as MEFs quando as MEFEs podem ser usadas para o teste de sistema ou para o teste de unidade, e os testes gerados podem ser executados manualmente ou automaticamente (Utting e Legeard, 2007).

Ao receber as entradas, a MEFE calcula qual transição deve ser habilitada e executada. Cada ação de transição gera uma atualização nas variáveis para então ocorrer a movimentação para o próximo estado. Todas essas características tornam as MEFEs modelos poderosos usados para a derivação de testes (Petrenko et al., 2004).

A Figura 3.10 mostra o exemplo de uma MEFE com quatro estados e dez transições. As transições são rotuladas com duas entradas a e b, três saídas x, y, e z, quatro predicados, e atualizações de variáveis. A máquina possui dois contextos de variáveis, u e w, e um parâmetro de saída, associado com o símbolo de saída z. A notação $z(u)$ significa que o valor atual da variável de contexto u é atribuído ao parâmetro de saída (Petrenko et al., 2004).

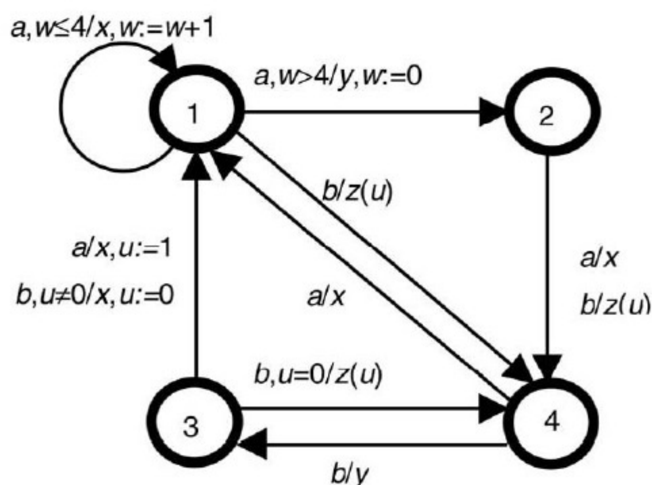

Figura 3.10: Exemplo de MEFE (Petrenko et al. (2004)).

As MEFs e MEFEs podem ser vistas como casos especiais de uma outra notação de modelagem chamada *Labeled Transition Systems* (LTS), ou Sistemas de Transição Rotulada. A diferença principal entre LTS e MEF é que o LTS permite um conjunto infinito de estados e/ou um conjunto infinito de rótulos (Utting e Legeard, 2007). O conceito de LTS foi definido por Keller (1976) e é muito utilizado na modelagem de sistemas reativos não determinísticos, no qual os testes gerados devem estar preparados para eventos vindos do SUT espontaneamente em vez de fornecer uma resposta direta para as entradas de teste. Podem ser usados, também, em programas sequenciais e concorrentes, e circuitos de *hardware* (Utting e Legeard, 2007).

3.4.4.3 Statecharts

Os *statecharts* foram propostos por Harel (1987) para a especificação de sistemas reativos complexos. *Statecharts* são também extensões de MEFs, com adição de alguns recursos, tais como: ortogonalidade, decomposição e *broadcasting* (Souza et al., 2000).

Harel mostrou que essas extensões permitem expressar comportamentos complexos como diagramas pequenos de uma maneira modularizada, uma vez que agrega independência aos subestados e permite modelar paralelismo. A característica de decomposição permite modularizar e criar uma hierarquia de estados. Já a ortogonalidade e *broadcasting* permitem representar características de paralelismo (Utting e Legeard, 2007).

A Figura 3.11 apresenta um exemplo de um *statechart*. Neste diagrama existem os estados A, B e C e, por exemplo, cada evento y ocorridos no estado A transfere o sistema para o estado C, porém somente se a condição P for válida. As transições de um *statechart* podem conter um *trigger* (gatilho), um *guard* (condição) e uma *action* (ação) usando a sintaxe[1]:

$$Trigger[Guard]/Action$$

O *Trigger* é o nome do evento da transição. *Guard* é uma expressão booleana que deve ser verdadeira para que a transição seja válida. A *Action* descreve o efeito da transição. Todos os três itens são opcionais (Utting e Legeard, 2007).

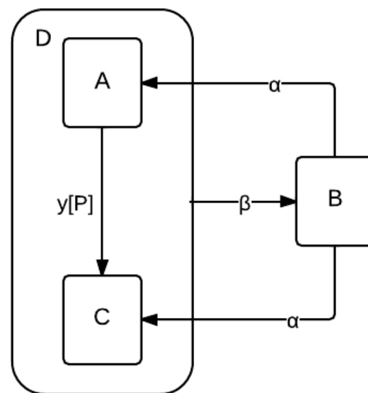

Figura 3.11: Exemplo de *statechart* (Adaptado de Harel (1987)).

Pode ser identificado um agrupamento do estado A com o estado C para um superestado D. O conceito de superestados e subestados tem o objetivo de fornecer um mecanismo hierárquico para controlar complexidades visuais de um diagrama. Por exemplo, uma ferramenta pode esconder os subestados de D para apenas mostrar o seu comportamento. Além disso, superestados e subestados ajudam em abstrair transições do modelo, sendo que uma única transição que sai de um superestado implica em um conjunto de transições que sai de cada um dos seus subestados (Utting e Legeard, 2007).

[1] Harel em seu trabalho original usou a sintaxe *Trigger*(*Guard*)/*Action*, porém em trabalhos posteriores a sintaxe passou a ser *Trigger*[*Guard*]/*Action*.

3.4.4.4 Redes de Petri

O conceito de rede de Petri (*Petri nets*) foi originado como o trabalho de doutorado de Carl Adam Petri (Petri, 1962) e consiste em uma modelagem gráfica aplicada em sistemas de processamento de informações, tais como: processos concorrentes, assíncronos, distribuídos, não determinísticos e estocásticos (Peterson, 1977).

Muitas teorias de sistemas concorrentes foram derivadas a partir das redes de Petri. O grafo é composto por dois tipos de vértices: os lugares e as transições, cujos dois são ligados por arcos direcionados. Outro elemento importante é o marcador (*token*) o qual simula uma execução da rede. Há a possibilidade de definir pesos nos arcos, para indicar uma quantidade específica de marcadores necessários para ativar uma transição (Peterson, 1977).

Na Figura 3.12 é apresentada um exemplo de uma Rede de Petri. A transição $t2$ está ativada, pois existe um *token* em $p1$. Caso a transição $t2$ ocorra, então o *token* de $p1$ é removido e *tokens* são colocados nos estados $p2$ e $p3$, os quais correspondem à saída da transição $t2$.

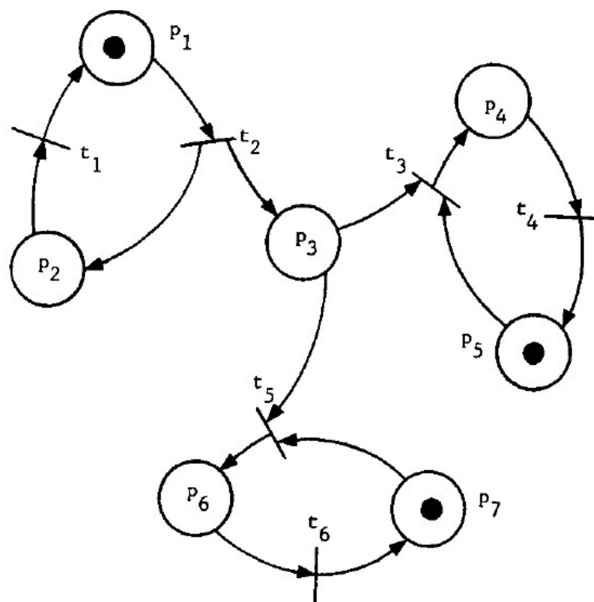

Figura 3.12: Exemplo de rede de Petri (Peterson (1977)).

As redes de Petri também podem ser usadas como uma ferramenta matemática para definir equações algébricas, equações de estado, e outros modelos matemáticos que regem o comportamento de sistemas (Murata, 1989).

3.4.4.5 Máquina de Estados UML

A máquina de estados UML (OMG, 2011) é uma modificação dos *statecharts* de Harel. É um tipo de diagrama que modela os diferentes estados de um objeto durante a execução

de um processo. Normalmente é utilizada para monitorar os diferentes estados pelos quais a instância de uma classe passa durante o seu ciclo de vida em resposta a eventos (Guedes, 2008).

O papel principal da máquina de estados UML é ajudar a definir o comportamento em estados a partir de uma classe. Essa classe normalmente é definida utilizando diagramas de classe UML, os quais contém os atributos, métodos e associações com outras classes. As transições na máquina podem, por exemplo, ler e atualizar os atributos, e chamar métodos (Utting e Legeard, 2007).

Na UML existem dois tipos de máquinas de estados: máquinas de estados comportamentais e máquinas de estados de protocolos. Máquinas de estados comportamentais podem ser empregadas na modelagem do comportamento de entidades individuais, tais como instâncias de classes (OMG, 2011). Um estado pode receber cláusulas adicionais indicando atividades na entrada (*entry*), permanência (*do*) e saída do estado (*exit*). Essas atividades normalmente consistem da execução de métodos. Na Figura 3.13 é apresentado um exemplo de máquina de estado comportamental UML. Os conceitos como superestados (ou estados compostos) e subestados, presentes nos *statecharts*, também são utilizados nas máquinas de estados UML, porém o rótulo da transição difere dependendo do tipo de máquina. Nas comportamentais a transição possui a notação *Trigger*[*Guard*]/*Action*, encontrada também nos *statecharts*. Já nas máquinas de protocolo, exemplificada na Figura 3.14, é utilizada a notação [*Precondition*] *Event* / [*Postcondition*]. As transições de uma máquina de estados de protocolos são interpretadas em termos de precondições (*Precondition*), também chamadas de *guards*, e poscondições (*Postcondition*) sobre evento (*Event*) associado (OMG, 2011).

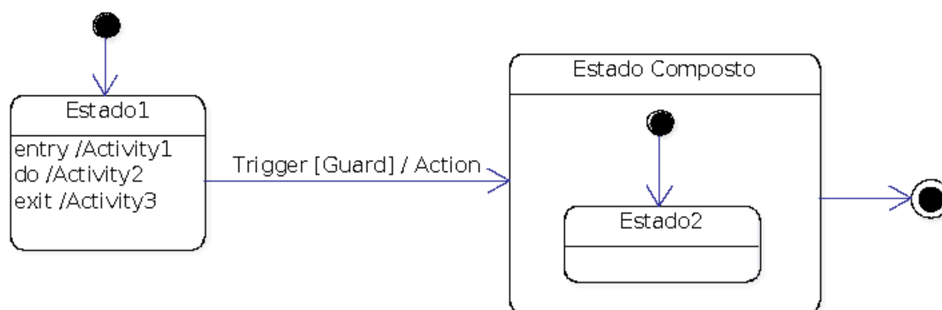

Figura 3.13: Exemplo máquina de estados comportamental UML.

Outros estados também são utilizados para adicionar comportamentos mais específicos ao diagrama. Os estados iniciais (Símbolo •) são usados para indicar o início de um processo, e os estados finais (Símbolo ⊙) representam o fim. Mais detalhes sobre as máquinas de estados de protocolos UML são abordados no Capítulo 4.

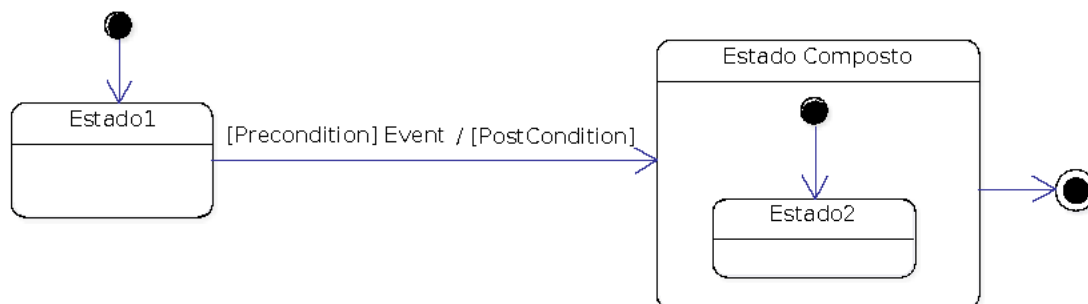

Figura 3.14: Exemplo máquina de estados de protocolos UML.

3.5 Ferramentas de Teste de Software

No final da década de 1970 houve grande progresso na área de ferramentas de auxílio ao teste, com o desenvolvimento de analisadores de cobertura de código, ferramentas de teste unitário e ferramentas de análise de dados de testes (Boehm, 2006). As atividades de testes tendem a lidar com grandes quantidades de dados, além disso, diversas técnicas demandam muito tempo para serem realizadas manualmente. As ferramentas melhoram a produtividade do testador e diminuem a probabilidade de erros humanos (Pressman, 2011). A seguir são descritas algumas ferramentas de teste encontradas na literatura.

- POKE-TOOL – A POKE-TOOL (*Potencial Uses Criteria Tool for Program Testing*) é uma ferramenta desenvolvida na FEEC/UNICAMP em colaboração com o ICMC/USP que apoia a aplicação dos critérios estruturais baseados em fluxo de controle: todos-nós e todos-arcos; assim como os critérios baseados em fluxos de dados: potenciais-usos. A ferramenta pode ser configurada para trabalhar com as linguagens C, Cobol, Pascal, Fortran e Clipper. A motivação de sua criação foi criar uma ferramenta de teste que auxilie no uso dos critérios e no auxilio à experimentação em estudos comparativos desses critérios (Vilela et al., 1995);

- JaBUTi – A JaBUTi (*Java Bytecode Understanding and Testing*) foi desenvolvida no ICMC/USP e consiste em um conjunto de ferramentas para auxiliar no entendimento e no teste de programas Java. A aplicação suporta o uso dos critérios estruturais todos-nós, todos-arcos, todos-potenciais-usos, dentre outros. Ela fornece informações sobre os elementos requeridos para cada critério e também disponibiliza o grafo de programa na visualização da cobertura dos casos de teste (Vincenzi et al., 2003). Extensões da ferramenta podem ser encontradas como, por exemplo: a JaBUTi/AJ (Lemos, 2005), ferramenta usada para aplicar o teste estrutural em programas orientados a aspectos; e a JaBUTiService (Eler et al., 2009), usada para aplicar teste estrutural em aplicações em Java (Vincenzi et al., 2010).

- Proteum – A Proteum é uma ferramenta que auxilia no teste baseado na análise de mutantes para programas na linguagem C (Delamaro, 1993). A partir dela foram desenvolvidas várias outras extensões para atuar em outros domínios. A Proteum/IM (Delamaro e Maldonado, 1996), da apoio ao critério de Mutação de Interface. A Proteum-RS/FSM (Fabbri, 1996) apoia à validação de especificações em MEFs, a partir dela foi desenvolvida a extensão Proteum-RS/ST (Sugeta, 1999) a qual valida especificações em Statecharts e a Proteum-RS/PN (Simão et al., 2000) que valida especificações em redes de Petri;

- Plavis/FSM – A Plavis/FSM é um ambiente que reúne três ferramentas: a Proteum/FSM (Fabbri et al., 1999), a ConData (Martins et al., 1999) e a MGASet (Candolo et al., 2001). Esse conjunto fornece, respectivamente, operações para a validação de especificações baseadas em MEFs, *statecharts* e redes de Petri. Todas possuem como foco o critério de análise de mutantes. A Plavis/FSM foi um dos resultados do projeto PLAVIS[2] (*PLAtform for Software Validation & Integration on Space Systems*), o qual tinha como objetivo auxiliar as atividades de VV & T para *softwares* espaciais (Simão et al., 2005);

- Ferramenta TAG – A ferramenta TAG (Test Automatic Generation) é usada para a geração automática de casos de teste para especificações em máquinas de estados. A ferramenta auxilia o usuário na geração de casos de teste que testam as transições da MEF (Tan et al., 1996).

3.6 Considerações Finais

Neste capítulo foram apresentados os principais conceitos envolvendo o teste de *software*, tais como: processo de teste, objetivos e suas fases. Foram apresentados também diferentes técnicas e critérios de teste que visam tratar o domínio de entrada dos casos de teste de uma maneira diferente. Por fim, foram mostradas algumas ferramentas para o auxílio das atividades de teste de *software*.

Percebe-se que a atividade de teste possui diferentes técnicas que abordam perspectivas distintas, portanto é necessário definir um processo de teste que faça uso das técnicas mais adequadas dependendo das necessidades do teste e do tipo de sistema a ser testado. Nesse contexto, será apresentado no próximo capítulo os principais trabalhos relacionados a este trabalho que envolvem a modelagem e o teste de serviços RESTful, além de algumas ferramentas da indústria que auxiliam o teste destes serviços.

[2]`http://www.labes.icmc.usp.br/plavis/index.html`

Capítulo

4

Modelagem e Teste de Serviços RESTful

4.1 Considerações Inicias

A adoção de um estilo de arquitetura, como o SOA ou REST, aplica muitas mudanças não só na arquitetura de um sistema, como também no seu processo de construção e utilização. São muitos os impactos que as aplicações baseadas em serviços causam na tarefa de teste. Sistemas baseados em serviço são intrinsicamente distribuídos, o que requer que a qualidade do serviço (QoS) seja garantida para diferentes configurações de implementações. Serviços dentro de um sistema podem ser modificados independentemente, assim causando impacto em testes de regressão. Há, também, uma confiança limitada por parte dos usuários na descrição dos serviços dificultando o projeto de casos de teste (Canfora e Di Penta, 2009).

Além disso, caso os serviços sejam oferecidos por provedores externos, novas dificuldades são adicionadas ao teste. As inspeções de *software* não podem ser realizadas, em decorrência da especificação e do código-fonte não estarem disponíveis. Além disso, o provedor de serviços pode retirar o serviço de operação quando desejar alterá-lo, o que invalida experiências anteriores de teste (Sommerville, 2007). Muitas vezes o uso de um serviço também implica em cotas restringindo a quantidade de requisições no dia ou no

tempo entre elas. Além disso, uma quantidade muito grande de testes pode ocasionar uma situação de negação de serviço (*denial-of-service*) (Canfora e Di Penta, 2009).

Neste capítulo são apresentadas as principais pesquisas relacionadas na área de teste de serviços RESTful. Serviços RESTful compartilham muitos dos desafios encontrados no teste de SOA e também precisam ser testados de acordo com suas características exclusivas, tais como: estado não persistente, interface uniforme, endereçabilidade e conectividade.

O capítulo está organizado da seguinte forma. Na Seção 4.2 são apresentados trabalhos que tratam da modelagem de serviços RESTful, o que auxilia na escolha de um modelo apropriado para realizar o TBM. Na Seção 4.3 são apresentados trabalhos que tratam do teste de serviços RESTful e algumas ferramentas encontradas na indústria que auxiliam este tipo de teste.

4.2 Modelagem de Serviços RESTful

Porres e Rauf (2011) apresentam uma abordagem de modelagem para serviços RESTful utilizando diagramas de classes como diagramas conceituais e máquinas de estados de protocolos como diagramas comportamentais. O objetivo principal da abordagem é de fornecer uma maneira de descrever corretamente as interfaces dos serviços, chamado de contrato. Um contrato determina, por exemplo, as operações que podem ser executadas, e os parâmetros de entrada e saída dessas operações. Posteriormente, Rauf e Porres (2011) adicionam outras características ao modelo para que os serviços modelados estejam compatíveis com o nível 3 no modelo de maturidade de Richardson. Tais características incluem a adição de parâmetros nos métodos HTTP e a definição de participantes (atores), os quais podem invocar diferentes métodos para diferentes recursos, assim adicionando restrições de autorização ao modelo. A modelagem comportamental definida por Porres e Rauf (2011) foi utilizada como base para este trabalho de mestrado e será explicada em mais detalhes no próximo capítulo.

No trabalho de Allam (2012) é apresentado um modelo com a proposta de unificar os estilos de arquitetura REST e SOA para verificar e assegurar propriedades de segurança. O modelo parte da premissa de que os serviços são compostos de maneira distribuída por meio de suas interfaces. O modelo apenas tem como foco o controle de segurança no nível de troca de mensagens.

No trabalho de Schreier (2011) é apresentado um metamodelo para a modelagem de serviços RESTful usando diagramas similares aos diagramas de classe e diagrama de estados. O metamodelo permite a modelagem estrutural e comportamental. A estrutural descreve os tipos de recursos, seus atributos, relações, interface e representações. A com-

portamental oferece a possibilidade de descrever o comportamento por meio de máquinas de estado de recursos. No entanto, o metamodelo é focado apenas na modelagem dos recursos e não apresenta uma contribuição de fato para problemas relacionados à teste.

Os trabalhos apresentados podem ser divididos em modelagem para a geração de contratos, verificação de segurança e arquitetura orientada a modelos para serviços RESTful. Outros trabalhos abordam alguma forma de modelagem de serviços RESTful aliada com testes. Estes são apresentados na seção a seguir.

4.3 Teste de Serviços RESTful

O trabalho de AlShahwan e Moessner (2010) consiste em uma comparação entre dois *frameworks* implementados para dispositivos móveis. Um *framework* foi implementado utilizando tecnologias de *Web Services* e o outro foi implementado utilizando REST. Esses *frameworks* são chamados de *Mobile Web Service Framework* (MWSF). Um MWSF é uma tecnologia que permite a implantação, publicação, descobrimento e execução de *Web Services* em um ambiente de comunicação portátil usando protocolos padrões. Seus resultados demonstram que o MWSF do tipo REST é mais próprio para esse tipo de ambiente. No entanto o estudo teve como objetivo testar apenas critérios de desempenho, confiabilidade e escalabilidade dos *frameworks*.

Na mesma área de teste de desempenho, Meng et al. (2009) realizam uma análise sobre os *Web Services* tradicionais e os RESTful. É projetado um esquema de testes para analisar o desempenho de *Web Services* RESTful para demonstrar que esses tipos de sistemas são mais adequados para a integração de dados distribuídos na escala da Internet. São desenvolvidos três *Web Services*, sendo dois tradicionais e um RESTful. Para simular o acesso concorrente aos serviços são utilizadas *multi-threads*. O desempenho dos serviços é medido em dois aspectos: desempenho percebido pelo usuário, o qual é medido pela latência, e desempenho na rede, o qual é medido pela carga. Seus resultados demonstram que os serviços RESTful possuem um alto desempenho e estabilidade em ambientes com alta taxa de acesso.

Reza e Van Gilst (2010) apresentam um *framework* o qual fornece uma variedade de entradas de testes para *softwares* que dependem de serviços RESTful. O *framework* faz uso de casos de teste predeterminados ou a geração de casos de teste aleatórios para fornecer as entradas de testes. O objetivo do *framework* consiste em simular um serviço RESTful o qual ainda está indisponível para o desenvolvedor. Esse tipo de abordagem tem a capacidade de reduzir o custo de testes por evitar as chamadas aos serviços durante a etapa de teste. No entanto, a abordagem não auxilia no teste dos serviços RESTful, apenas na simulação do mesmo.

O trabalho de Seijas et al. (2013) descreve um mecanismo e um modelo para o teste baseado em propriedades de serviços RESTful sem efeitos secundários. O teste baseado em propriedades utiliza especificações de propriedades importantes para produzir critérios de teste e procedimentos que focam nessas propriedades de uma maneira sistemática (Fink e Levitt, 1994). O modelo utiliza funções na linguagem Erlang[1] para declarar as operações do serviço e a ferramenta QuickCheck para gerar os dados de entrada. QuickCheck (Claessen e Hughes, 2000) utiliza dados gerados aleatoriamente como entrada para o teste de funções. A abordagem também pode utilizar MEFs abstraindo algumas declarações de pre e poscondições. O mecanismo é adaptado para testar dois serviços: Storage Room[2] e Google Tasks[3]. Esta abordagem exige bastante trabalho manual por parte do testador para definir todas as propriedades do serviço, ou seja, não auxilia na geração automática dos testes.

Chakrabarti e Rodriquez (2010) apresentam um método para testar automaticamente, utilizando métodos formais, o aspecto de conectividade de *Web Services* do tipo RESTful. A conectividade referencia a característica REST de que cada recurso é alcançável por meio de um recurso base usando requisições GET do HTTP. Para verificar a conectividade são utilizados dois grafos: POST class graph (PCG) e POST object graph (POG). O PCG é um grafo direcionado cujos nós representam classes de recursos e as arestas representam as operações POST. Na Figura 4.1 é apresentado um PCG de um serviço RESTful *eBlog.com*. O recurso *eBlog.com* é a origem do POST para o recurso *User*. A cardinalidade nas arestas indicam a possibilidade de criação de zero ou mais recursos. O grafo também precisa ter uma notação textual associado a ele, no qual estará presenta as informações sobre a requisição (id, URI) e resposta (corpo da requisição, valores de *header*). A partir do PCG e das notações textuais é gerado o POG da Figura 4.2. O POG é uma árvore que organiza todos os recursos criados, e é então usada para criar uma lista das URIs desses recursos. Paralelamente uma outra lista é criada baseada nas URIs encontradas no PCG e nas respostas. Ao final do processo as duas listas são comparadas, caso elas sejam iguais significa que o serviço esteve de acordo com a característica de conectividade.

No trabalho de Chakrabarti e Kumar (2009), é apresentada uma abordagem de teste de serviços RESTful que utiliza uma ferramenta chamada TTR (*Test-the-REST*). Chakrabarti e Kumar (2009) listam em duas categorias (funcionais e não funcionais) algumas funcionalidades desejáveis às ferramentas que testam serviços RESTful. Os requisitos funcionais englobam compatibilidade de versões, *caching*, GET parcial e condicional, código de status HTTP, teste de conectividade, dentre outras. Os requisitos não funcionais

[1] `http://www.erlang.org/`
[2] `http://storageroomapp.com`
[3] `https://mail.google.com/tasks`

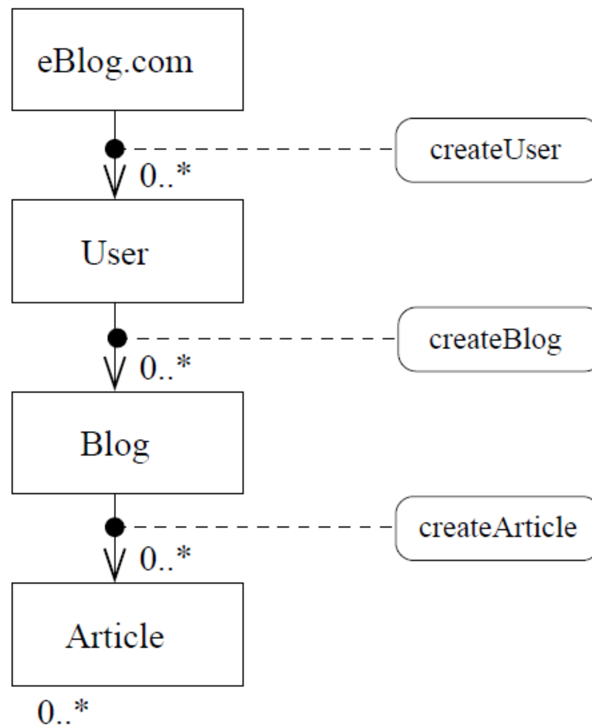

Figura 4.1: Exemplo de POST class graph (Chakrabarti e Rodriquez (2010)).

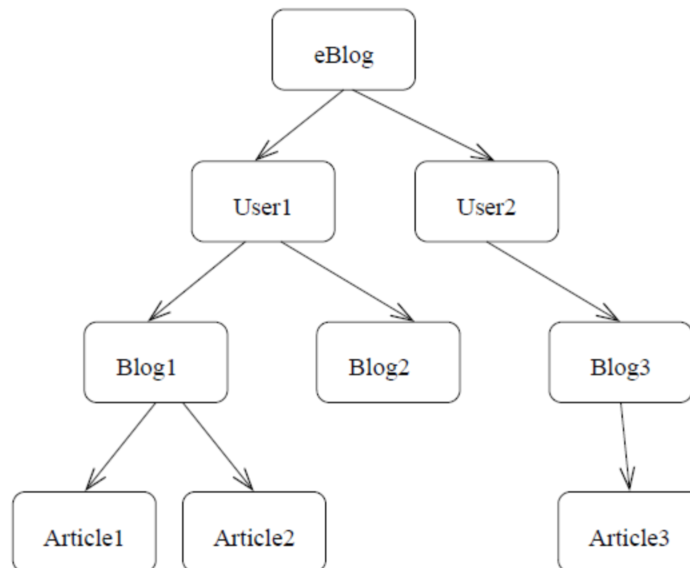

Figura 4.2: POST object graph baseado na PCG (Chakrabarti e Rodriquez (2010)).

englobam teste de desempenho, teste de escalabilidade/carga e testes de disponibilidade e confiabilidade. As ferramentas disponíveis no mercado analisadas pelos autores auxiliavam na execução de testes simples e genéricos, mas não forneciam apoio para o teste de requisitos específicos do REST.

Para executar os testes, a TTR faz uso de uma linguagem de especificação para testes baseada em XML. Os casos de teste são escritos em arquivos externos XML e contém o URI

do recurso, o método HTTP utilizado, dados de entrada e saídas esperadas. Além disso, casos de teste podem compor outros casos de teste. TTR é restrito a testes caixa-preta e para utilizar todas as suas capacidades é necessário que o SUT esteja hospedado em um ambiente controlado. A abordagem é utilizada em um serviço RESTful real e foram executadas rotinas de teste com duração de 5 minutos para a execução de 300 casos de teste. Diariamente, dos 300 casos de teste, de 5 a 10 falhavam. Os autores também geram um conjunto de 42,767 casos de teste a partir de uma combinação da lista com todos os possíveis parâmetros de entrada. Houve 38,016 casos de teste que falharam inicialmente, e na segunda tentativa, após corrigir os defeitos, 1781 casos de teste ainda falharam. Embora a ferramenta forneça uma forma robusta de declarar e executar casos de teste, estes não são automaticamente gerados.

No trabalho de Correa et al. (2012), é apresentada uma abordagem sistemática para a geração automática de casos de teste CRUD de serviços RESTful. A abordagem utiliza modelos de classe UML enriquecidos com a *Object Constraint Language* (OCL). A OCL é uma noção textual similar a expressões encontradas em linguagens de programação orientadas a objeto. Ela pode ser usada para definir restrições dentro de diagramas de classe, e especificar precondições, poscondições e invariâncias (Utting e Legeard, 2007). Na Figura 4.3 é apresentado o diagrama de classes de um serviço que gerencia cursos. O diagrama é específico da plataforma REST e contém informações sobre as operações da interface, localização e entidades utilizadas. As expressões OCL mais o diagrama são usados para gerar tabelas de decisão que representam os casos de teste. Na Figura 4.4 são apresentados os casos de teste para a operação *createCourse()*. A requisição são os métodos HTTP encontrados no modelo e os resultados esperados são definidos a partir das expressões OCL. Embora a abordagem forneça a geração de casos de teste, estes não são executáveis.

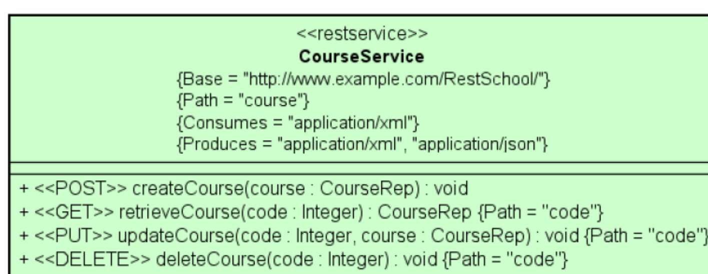

Figura 4.3: Diagrama de classes de um Serviço de Cursos (Correa et al. (2012)).

No trabalho de Borges (2009), é proposta a ferramenta REST-Unit, com o propósito de fornecer a geração automática de *drivers* de teste. Os *drivers* são gerados a partir de modelos especificados em U2TP (*UML 2.0 Testing Profile*) (Object Management Group (OMG), 2005). A linguagem U2TP estende a UML com conceitos específicos de teste, tais

Request Sequence	Expected Result in the Response
GET http://www.example.com/RestSchool/course/1 Accept: application/xml, application/json	*Status Code* = 404
POST http://www.example.com/RestSchool/course Accept: application/xml, application/json Content-Type: application/xml <course> <code>1</code><name>a</name> </course>	*Status Code* = 201 *Location* <> null *Content-Type* = *Content-Type* of request
GET http://www.example.com/RestSchool/course/1 Accept: application/xml, application/json	*Status Code* = 200 *Content-Type* ⊂ *Accept* of request
DELETE http://www.example.com/RestSchool/course/1 Accept: application/xml, application/json	*Status Code* = 200

Figura 4.4: Tabela de decisão da operação *createCourse()* (Correa et al. (2012)).

como: arquitetura de teste, dados de teste e comportamento. É utilizado um diagrama de classes como modelo estrutural e um diagrama de sequência como modelo comportamental. Na Figura 4.5 é apresentado o diagrama de classes que representa a interface de um serviço RESTful que cria e responde com *websites* favoritos *bookmarks*. No diagrama são especificados o recurso, endereço do servidor, possíveis códigos de resposta e as operações sob o recurso. Na Figura 4.6 são apresentados os diagramas de sequência que representa respectivamente o caso de teste para a criação do recurso (*sd test_create_bookmark*) e o recebimento do recurso (*sd test_get_bookmarks*). A ferramenta é responsável por transformar as informações desses diagramas

A ferramenta exporta os diagramas para o formato XMI (*XML Metadata Interchange*) e a partir dessas informações um algoritmo gera o código dos *drivers* de teste dos casos de teste. O código tem o propósito de ser executado no *framework* de teste unitário *Test::Unit* da linguagem *Ruby*. Com base no trabalho de Borges (2009), Feller (2010) estendeu a ferramenta para a REST-Unit+, com o objetivo de relacionar a geração automática dos *drivers* com os seus respectivos dados de teste. Assim, os testes gerados são mais completos e com sua execução facilitada, em consequência dos testes já estarem previamente especificados e documentados no modelo de classes. Em um trabalho relacionado, Silva (2011) apresenta a ferramenta PyRester, escrita em *Python*, a qual faz uso de modelos U2TP do sistema para a geração de código de teste unitário para serviços RESTful. Segundo Cartaxo et al. (2007), existem problemas quando se usam diagramas de sequência para a extração de casos de teste, já que eles não possuem uma boa representação de sequências condicionais, recursivas e repetitivas, suprimindo muitos detalhes de interface e controle.

Pode-se perceber, baseado nos trabalhos apresentados, que as principais pesquisas que envolvem o teste de serviços RESTful estão concentradas na área de teste de desempenho,

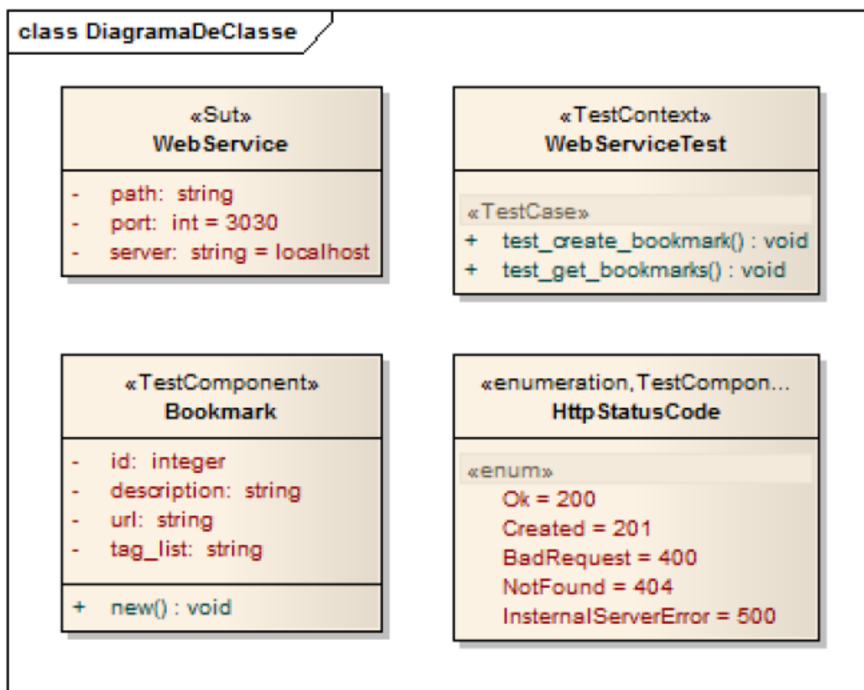

Figura 4.5: Diagrama de classe representando o serviço RESTful (Borges (2009)).

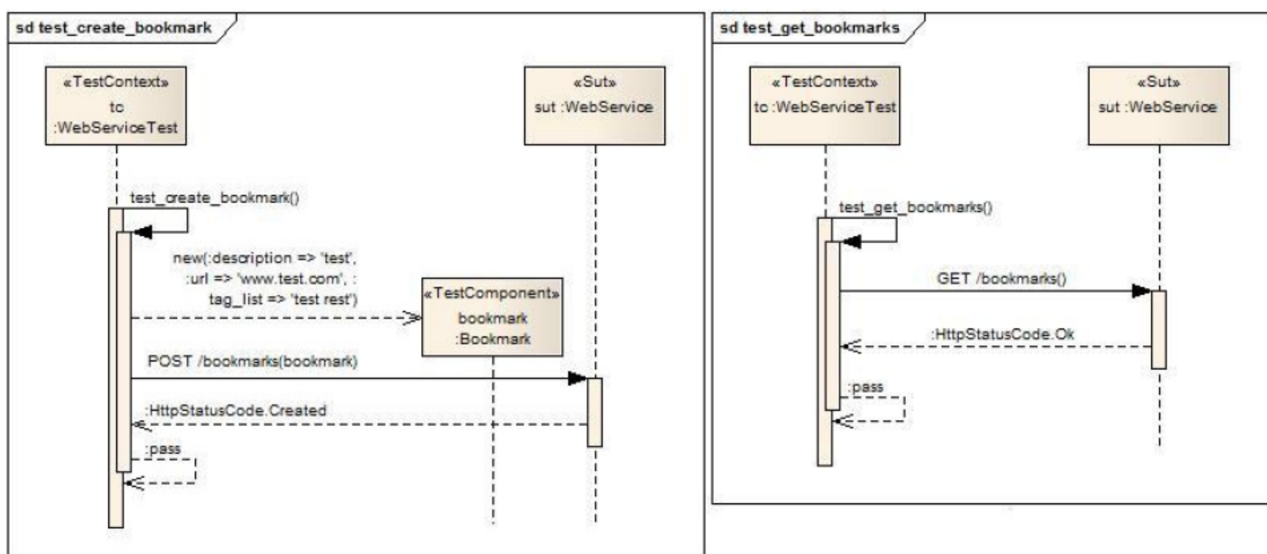

Figura 4.6: Diagrama de sequência representando os casos de teste (Borges (2009)).

simulação de serviços, teste de propriedades, teste de conectividade REST, teste funcional e geração de casos de teste.

4.3.1 Ferramentas de Teste

Além das pesquisas encontradas, são apresentadas a seguir ferramentas que apoiam a atividade de teste de serviços RESTful de forma manual:

- RESTClient. RESTClient[4], é uma ferramenta construída na linguagem Java que fornece apoio ao teste da interface dos serviços RESTful. É possível definir os dados como o URI do recurso, realizar as requisições e então utilizar os métodos HTTP para obter a resposta. Todos os dados dos *headers* da resposta são disponibilizados para o testador.

- cURL. O cURL[5] é uma ferramenta de linha de comando capaz de transferir dados por meio de diversos protocolos, dentre eles o HTTP. Além disso, ele suporta diversas outras funcionalidades, tais como: autenticação de usuário, *cookies* e conexões SSL. Para executar operações HTTP o cURL permite o envio por comando todos os parâmetros necessários.

- NetBeans IDE. O NetBeans IDE[6] é um Ambiente de Desenvolvimento Integrado (IDE) gratuito e de código aberto podendo ser utilizado para o desenvolvimento de *softwares* em diversas linguagens de programação. Uma das funcionalidades do NetBeans é seu apoio ao desenvolvimento e teste de serviços RESTful. Ao criar um novo projeto de aplicação *web* o NetBeans permite ao testador interagir com a aplicação por meio dos métodos HTTP.

- Eclipse HTTP Client. O Eclipse HTTP Client[7] (HTTP4e) é um *plugin* para o IDE Eclipse[8] que facilita o desenvolvimento e o teste de serviços RESTful. O *plugin* permite que sejam feitas chamadas HTTP para serviços RESTful e *Web Services.*

- SoapUI. A ferramenta SoapUI [9] é uma ferramenta de código aberto que auxilia ao teste funcional de *Web Services* que seguem os padrões SOAP e WSDL, e serviços RESTful. SoapUI possui uma interface gráfica amigável e permite que rapidamente sejam criados e executados testes funcionais, de regressão, de conformidade e de carga.

4.4 Considerações Finais

Os trabalhos apresentados podem ser divididos em modelagem para a geração de contratos, verificação de segurança e arquitetura orientada a modelos para serviços RESTful. Outros trabalhos abordam alguma forma de modelagem de serviços RESTful aliada com testes. Estes são apresentados na seção a seguir.

[4]`http://code.google.com/p/rest-client/`
[5]`http://curl.haxx.se/`
[6]`http://netbeans.org`
[7]`http://www.ywebb.com/eclipse-restful-http-client-plugin-http4e/`
[8]`http://www.eclipse.org`
[9]`http://www.soapui.org`

Neste capítulo foram apresentados alguns dos principais trabalhos relacionados a modelagem e teste de serviços RESTful. Os trabalhos que envolvem exclusivamente a modelagem podem ser classificados em modelagem para a geração de contratos, verificação de segurança e arquitetura orientada a modelos para serviços RESTful. Já os trabalhos que lidam com o teste de serviços RESTful podem ser classificados em teste de desempenho, simulação de serviços, teste de propriedades, teste de conectividade REST, teste funcional e geração de casos de teste. Foram também apresentadas algumas ferramentas que auxiliam no teste manual.

Este capítulo também ressaltou a falta de pesquisas na área do teste serviços RESTful. Neste contexto mais contribuições são necessárias para que serviços sejam produzidos com maior qualidade e maior produtividade. No próximo capítulo é apresentada uma abordagem de teste, na qual procura-se explorar potencialidades da aplicação do teste baseado em modelos no contexto de serviços RESTful.

Capítulo

5

Abordagem para Teste de Serviços RESTful

5.1 Considerações Iniciais

O uso de uma abordagem automatizada de geração de casos de teste para serviços RESTful pode fornecer uma agilidade e produtividade na etapa de testes auxiliando assim o processo de teste. Embora os serviços RESTful utilizem tecnologias fáceis de serem utilizadas, como o HTTP, eles ainda podem ser usados para projetar serviços sofisticados, e podem representar soluções viáveis para problemas empresariais (Vinoski, 2008).

Neste capítulo é apresentada uma abordagem de teste baseado em modelos para serviços RESTful utilizando a máquina de estados de protocolos UML. Este modelo já foi investigado para apoiar o projeto de serviços RESTful (Porres e Rauf, 2011; Rauf e Porres, 2011). Porém nenhuma iniciativa foi encontrada que o utiliza na geração automática de casos de teste para estes serviços. São apresentados também em detalhes os passos da abordagem, os critérios de cobertura utilizados para a geração automática dos casos de teste e a estrutura dos testes gerados. A ferramenta de apoio à abordagem também é apresentada, bem como os resultados de um estudo de caso realizado.

O capítulo está organizado da seguinte forma. Na Seção 5.2 são apresentados os passos da abordagem, além da ferramenta desenvolvida para auxiliar o processo. A descrição é

baseada em um exemplo de modelo. Na Seção 5.3 é apresentada a abordagem sendo aplicada em um estudo de caso.

5.2 Processo de Teste Baseado em Modelos para Serviços RESTful

A abordagem busca dar apoio aos conceitos de recurso, representação de recursos e URIs, assim como as propriedades de interface uniforme, estado não persistente e endereçabilidade. A partir dos testes busca-se detectar erros no comportamento da interface uniforme do serviço e nos dados presentes na representação do recurso. No estado atual as representações de recursos utilizadas são o XML e JSON.

A abordagem consiste dos passos descritos na Figura 5.1 e será tomada como base para a explicação da abordagem. A ferramenta desenvolvida para auxiliar o processo também será descrita ao longo da explicação do processo. A abordagem consiste em sete passos, três deles executados manualmente pelo testador e os outros quatro executados automaticamente pela ferramenta. A partir desse capítulo a ferramenta irá ser referida pelo nome RSTT (*RESTful Services Testing Tool*).

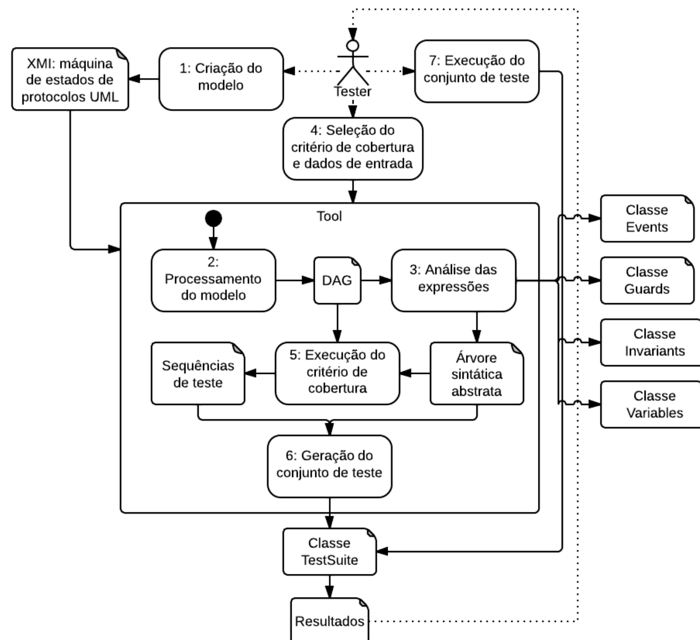

Figura 5.1: Abordagem de teste baseada em modelos.

5.2.1 Criação do modelo comportamental

O início do processo (Passo 1) consiste da criação do modelo comportamental do SUT. A criação do modelo é realizada pelo testador a partir de uma especificação informal em linguagem natural do sistema. O objetivo do modelo é representar um comportamento ou funcionalidade que se deseja testar por meio dos diferentes estados em que o sistema pode apresentar e as transições existentes entre os estados. Para representar o modelo comportamental é utilizado a máquina de estados de protocolos UML.

A notação da UML contém conceitos da orientação a objetos que podem entrar em conflito com as restrições REST (Schreier, 2011). No entanto, seu uso ainda é possível fazendo adequações ao modelo, como demonstrado no trabalho de Porres e Rauf (2011). Porres e Rauf (2011) apresentam um estudo de como projetar serviços RESTful utilizando modelos UML, mais especificamente usando diagramas de classe e máquinas de estados de protocolos UML.

A máquina de estados de protocolos proposta utiliza o conceito de invariâncias em estados e recursos endereçáveis para estar de acordo com a propriedade de estado não persistente do REST, e os eventos nas transições são restringidos aos métodos padrões HTTP para estar de acordo com a propriedade da interface uniforme. O trabalho tem como objetivo fornecer uma maneira de descrever corretamente as interfaces dos serviços, chamado de contrato. O contrato determina, por exemplo, as operações que podem ser executadas, e os parâmetros de entrada e saída dessas operações. O contrato eventualmente pode ser transformado para o formato WADL e pode ser usado como documentação, geração de *stubs*, testes e monitoramento.

Para a criação dessa máquina de estados, Porres e Rauf (2011) se basearam nas formalizações definidas em Porres e Rauf (2009), no qual havia sido utilizado as máquinas de estados de protocolos para a geração de contratos. Posteriormente, Rauf e Porres (2011) adicionaram outras características ao modelo para que os serviços modelados estejam compatíveis com o nível 3 no modelo de maturidade de Richardson. Tais características incluem a adição de parâmetros nos métodos HTTP e a definição de participantes (atores), os quais podem invocar diferentes métodos para diferentes recursos, assim adicionando restrições de autorização ao modelo.

Como exemplo inicial do uso da abordagem será utilizado o modelo da Figura 5.2 na qual é apresentada um serviço RESTful que gerencia agendamentos de quartos de hotel. O modelo descreve uma rotina do serviço ocorrendo um agendamento, pagamento, verificação por confirmação de pagamento, cancelamento do agendamento e por fim sua remoção.

A formalização definida a seguir é baseada nos trabalhos de Porres e Rauf (2009, 2011); Rauf e Porres (2011). Uma máquina de protocolos UML é definida como o con-

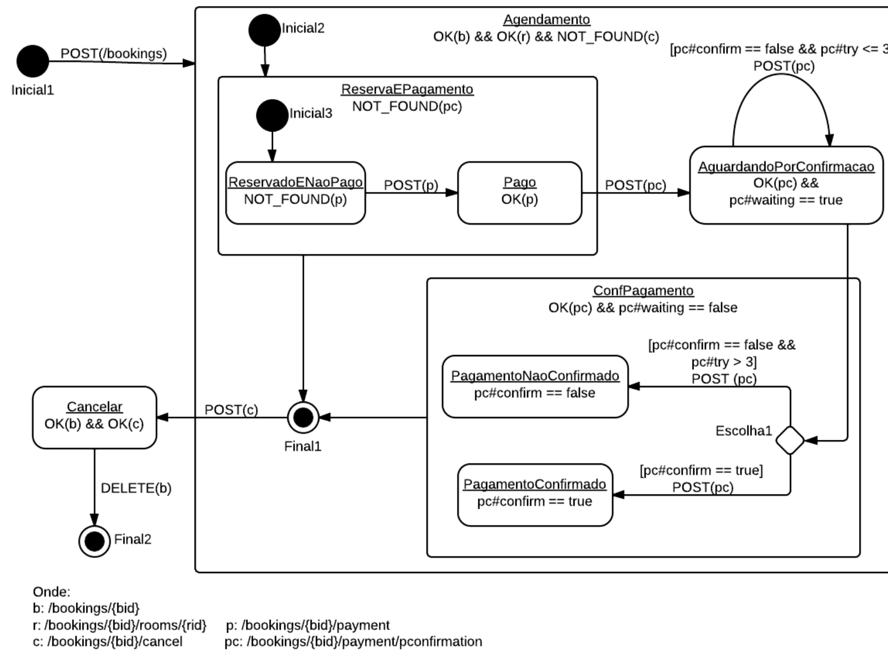

Figura 5.2: Modelo comportamental de um serviço RESTful para agendamento de quartos de um hotel (Adaptado de Porres e Rauf (2011)).

junto $\{S_S, S_C, S_I, S_F, P, T, E, I, G\}$, onde S_S é um conjunto de estados simples, S_C é um conjunto de estados compostos (ou superestados), S_I é um conjunto de estados iniciais, S_F é um conjunto de estados finais, P é um conjunto de pseudoestados, T é um conjunto de transições, E é um conjunto de eventos, I é um conjunto de invariâncias e G é um conjunto de precondições (*guards*).

A união dos diferentes tipos de estados do modelo formam o conjunto de estados da máquina de estados de protocolo $S = S_S \cup S_C \cup S_I \cup S_F \cup P$, sendo que S_S e S_C são conjuntos disjuntivos, assim como P e S_F. No modelo utilizado neste trabalho de mestrado são apenas usados os pseudoestados iniciais e de escolha (*choice*), mais os estados finais. Na UML os estados iniciais são considerados parte do conjunto de pseudoestados. Um estado inicial (Símbolo $\bullet$) é usado para indicar um estado padrão de início para um estado composto. Um pseudoestado do tipo escolha (Símbolo $\diamond$) representa um ponto de ramificação dinâmica. Por exemplo, o estado *EsperandoPorConfirmacao* possui uma transição para um pseudoestado do tipo escolha, no qual as diferentes precondições são analisadas para decidir qual transição será ativada. Um estado final (Símbolo $\odot$), por sua vez, não é considerado como um pseudoestado, mas um estado especial que indica o término do estado composto em que está situado (Liu et al., 2013).

Uma transição $t \in T$ representa um evento $e \in E$ a ser realizado conectando um estado de origem $s_i \in S$ e um estado de destino $s_j \in S$, sendo que a transição pode utilizar uma precondição $g \in G$ para ser considerada ativa, assim $t = (s_i, e[g], s_j)$. Se uma precondição não for definida no modelo, então ela é assumida como verdadeira. Elementos

do conjunto de estados finais jamais podem ser um estado de origem de uma transição. Já os estados iniciais não podem ser um estado de destino de uma transição. Não podem existir, também, transições entre superestados com seus próprios subestados e vice-versa.

Um evento pode apenas ser definido como operações POST, PUT ou DELETE, já que estas operações são as únicas que podem alterar o estado de um serviço, portanto $E = \{post, put, delete\}$. As operações $post(uri, p_1, p_2, ..., p_n, entity)$ e $put(uri, p_1, p_2, ..., p_n, entity)$ necessitam do endereço uri do recurso mais os possíveis dados (parâmetros) $p_1, p_2, ..., p_n$ a serem inseridos na representação de recurso *entity*. Os parâmetros mais a representação do recurso consistem nos dados de entrada para o evento, sendo que os parâmetros são os dados contidos dentro da representação. A operação $delete(uri)$ precisa apenas do endereço do recurso a ser removido.

Uma invariância consiste em uma ou mais expressões unidas por operadores Booleanos. Uma invariância retorna um valor Booleano utilizado para verificar certas características no serviço, como por exemplo, se um recurso existe ou se um valor de um dado em sua representação está no valor correto. A UML define invariâncias como condições adicionais para as precondições de transições que partem de um estado, e para poscondições de transições que chegam a um estado (OMG, 2011). Dentre as maneiras de compor uma expressão da invariância estão dois métodos: OK e NOT_FOUND. O método $ok(uri)$ retorna *true* se o código de resposta HTTP obtido a partir de uma requisição GET for 200 (solicitação bem sucedida); caso contrário retorna *false*. O método $not_found(uri)$ retorna *true* se o código de resposta HTTP for 404 (um recurso não foi encontrado no servidor); caso contrário retorna *false*. Ambos os métodos aceitam a URI do recurso como parâmetro.

Um estado é considerado ativo se sua invariância for avaliada como verdadeira. Uma invariância de um estado composto deve sempre persistir para os seus subestados. Por exemplo, a invariância "NOT_FOUND(pc)" do estado *ReservaEPagamento* é também aplicada ao estado *Pago*, resultando na invariância $NOT_FOUND(pc)\&\&OK(p)$[1].

Além desses métodos, uma expressão também pode ser composta utilizando atributos da representação de um recurso, aliada com operadores relacionais e constantes. Para acessar um atributo deve-se seguir a sintaxe $uri\#attr$. Por exemplo, a invariância $pc\#confirm == true$ do estado *PagamentoConfirmado* avalia se o atributo $confirm$ da representação do recurso $pconfirmation$ é verdadeiro. As precondições usam as mesmas expressões utilizadas na invariância que verificam os atributos das representações. Neste modelo as poscondições não são definidas na transição, e sim na invariância do estado de destino da transição.

[1] Os operadores lógicos *and*, *or* e *not* foram representados por seus equivalentes utilizados em algumas linguagens de programação, respectivamente &&, || e !

Para a realização da modelagem será utilizada a ferramenta ArgoUML[2]. A ArgoUML é uma ferramenta livre de código aberto que permite a criação de diversos diagramas UML, tais como: diagramas de casos de uso, classes, sequência, colaboração, estados, atividades e distribuição. A Figura 5.3 demonstra o uso da ArgoUML na criação do modelo. Os eventos das transições são chamados de "Eventos de Chamadas" (*CallEvent*) dentro do campo de "Gatilho" (*Trigger*), com a possibilidade de adicionar os parâmetros. As precondições são definidas no campo de "Guarda" (*Guard*).

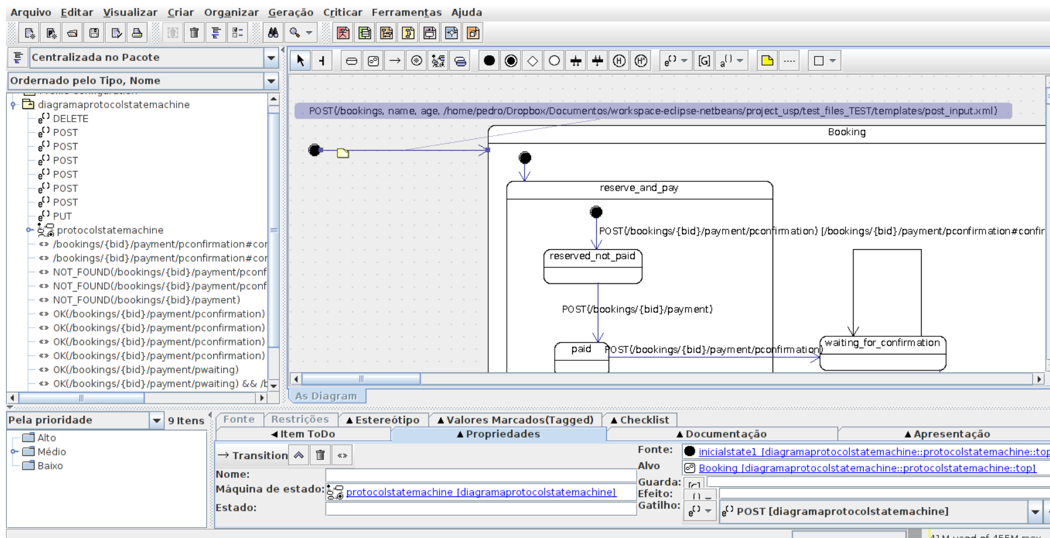

Figura 5.3: Exemplo do modelo sendo criado na ferramenta ArgoUML.

Em decorrência do uso de uma ferramenta externa para a criação do modelo, não existe um apoio nativo para todas as características de nossa abordagem. Foram detectadas quatro funcionalidades importantes as quais a ArgoUML não fornece um apoio explícito: invariâncias, definição de cabeçalho (*headers*) HTTP, criação de novas variáveis e definição de representação de recurso na requisição. As três primeiras funcionalidades foram implementadas por meio do uso de estereótipos, já a última por meio de *templates*.

Nos diagramas UML, estereótipos são utilizados para expressar alguma característica ou função diferenciada de um componente em relação aos outros (Guedes, 2008). Na abordagem proposta, os estereótipos servem para outro propósito: o de adicionar informações complementares a um estado ou transição. Em se tratando de estados, o estereótipo representa a invariância daquele estado, ou o *header* de autorização (*Authorization*) da requisição. Já nas transições, o estereótipo representa informações adicionais importantes para o evento. O estereótipo quando usado para definir informações adicionais deve seguir a sintaxe: "nome : valor". Essa sintaxe pode representar informações do cabeçalho da requisição, tais como: *Authorization* e *Content-Type*, ou de declarações de variáveis. Este último é importante caso se queira criar uma nova variável baseada em um valor

[2]`http://argouml.tigris.org/`

encontrado na resposta da requisição, ou seja, na representação de recurso de resposta do evento. Por exemplo, um estereótipo do tipo "{newVar} : response#id" indica que deve ser criada uma nova variável chamada *newVar* cujo valor será o atributo *id* da resposta do evento. O valor destas variáveis são preenchidas em tempo de execução do código gerado. Porém outras variáveis contidas no URI do recurso, tais como: "{bid}" e "{rid}"; ou em parâmetros de eventos, são preenchidas ao longo do processamento do modelo.

Existem duas formas, não excludentes, de definir dados em uma requisição ao servidor:

1. Definir os dados no próprio URI – Por exemplo:

   ```
   http://www.google.com/search?q=REST
   ```

   O valor "REST" é enviado diretamente para o servidor pelo URI da requisição. É possível também utilizar variáveis para que a RSTT pergunte ao testador pelo dado, neste caso as variáveis são definidas utilizando chaves, tal como em:

   ```
   http://www.google.com/search?q={var}
   ```

2. Definir os dados por meio de templates – A abordagem depende do testador para definir como será a representação do recurso a ser enviada ao servidor. Essa definição se da por meio de um *template* criado pelo testador. Ao utilizar uma operação do tipo $post(uri, p_1, p_2, ..., p_n, entity)$, sempre o último parâmetro ($entity$) deve ser a localização do template. Isso é necessário em decorrência da ArgoUML não permitir uma definição de representação de recurso a ser enviada na requisição. Já os outros parâmetros $p_1, p_2, ..., p_n$ são variáveis que a RSTT irá posteriormente requisitar ao testador por seus valores. Ao longo de sua execução, a RSTT irá substituir as variáveis no *template* pelos dados corretos e irá gerar o código com a representação do recurso correta da operação, como, por exemplo:

   ```
   <cancel>
       <cdate> #par1 </cdate>
       <note> #par2 </note>
   </cancel>
   ```

   O uso de *templates* foi uma solução para um problema comum de serviços RESTful. As representações de recursos variam de acordo com o serviço utilizado, e, como descrito no Capítulo 2, serviços RESTful muitas vezes não possuem uma documentação formal para a descrição da interface de comunicação.

Durante a criação do modelo também deve se estar atento ao definir estados compostos. Todo estado composto precisa ter um estado inicial, o qual indica, por meio de uma transição, qual o estado que representa o início do estado composto. Um estado composto pode ter também um estado final, o qual indica, por meio de uma transição, qual o estado que representa o fim daquele estado composto. Os estados finais podem ser evitados se o subestado apresentar uma transição específica para fora do estado composto.

5.2.2 Processamento do modelo

A partir do Passo 2 a RSTT passa a auxiliar o processo de teste. Ela foi desenvolvida na linguagem Java e atualmente possui quase 6000 linhas de código distribuídas em 36 classes. Essas classes podem ser divididas em categorias de acordo com as tarefas que desempenham, que são: construção do grafo, análises de expressões, execução de algoritmos de cobertura, e construção de código-fonte.

Inicialmente, o testador informa o *host* do serviço em teste mediante requisição da RSTT, o qual é adicionado na frente das URIs. Após isso, o testador fornece o arquivo XMI que representa o modelo criado no Passo 1. O arquivo é gerado pela ArgoUML. A estrutura de dados correpondente aos estados e as transições do modelo é construída na RSTT adicionando os possíveis eventos, precondições e invariâncias que possam existir. Essa estrutura consiste em um grafo que será utilizado nos próximos passos. A leitura do arquivo XMI é feita por meio da linguagem XPath[3], e essa sempre será utilizada ao lidar com arquivos do tipo XML.

Após a construção do grafo, este passa por um processo de simplificação de três subpassos. O resultado do processo é um grafo acíclico direcionado (*Directed Acyclic Graph* – DAG). O grafo foi a estrutura escolhida pois é mais comumente utilizado para descrever o comportamento em máquinas de estados e em redes Petri (OMG, 2011), e é acíclico para prevenir a ocorrência de ciclos infinitos. A Figura 5.4 mostra uma DAG gerada pela RSTT após o processamento do modelo da Figura 5.2. As invariâncias dos estados foram omitidos para a simplificação da ilustração do grafo. Os subpassos do processo de simplificação são:

1. Os estados compostos são eliminados. Para eliminar um estado composto primeiro deve-se incorporar aos subestados a invariância do estado composto. Uma invariância de um estado composto ao ser aplicada à invariância dos subestados é equivalente à adição de um "&&" entre as duas invariâncias. Depois são eliminadas as transições que envolvem o estado composto. Estas transições são adaptadas para os subestados. As transições que apontam para os estados compostos passam a apontar para

[3] `http://www.w3schools.com/xpath/`

o subestado inicial e as transições que saem do estado composto passam a valer também para os subestados.

2. As transições que não possuem um evento nem uma precondição são eliminadas, assim, os estados envolvidos nessa transição podem ser mesclados. Caso existam invariâncias, elas também são mescladas da mesma forma descrita no primeiro passo. Por exemplo, o estado *Inicial2+Inicial3+ReservadoENaoPago+Final1* da DAG representa a junção de quatro estados que possuem transições entre si sem um evento ou precondição;

3. Os ciclos são encontrados e eliminados. Para encontrar um ciclo é utilizado o seguinte algoritmo recursivo: *(i)* a partir do estado atual, adicione este estado à lista de estados visitados; *(ii)* para cada uma das transições desse estado percorra o estado alvo; *(iii)* caso o estado alvo já existir na lista de estados visitados então o estado alvo marca o início de um um ciclo; caso contrário aplique o passo *(i)* ao estado alvo junto com a nova lista de estados visitados. Para eliminar um ciclo é criado um estado idêntico ao estado que marca o início do ciclo, porém sem transições para outros estados, assim o estado não é expandido. Por exemplo, o ciclo do modelo encontrado no estado *AguardandoPorConfirmacao* é simplificado em uma repetição deste mesmo estado, porém sem futuras expansões. Este passo é necessário pois a ferramenta não possui uma forma de parar uma execução de um ciclo, ou saber quantas vezes deve ser repetido a execução de um ciclo.

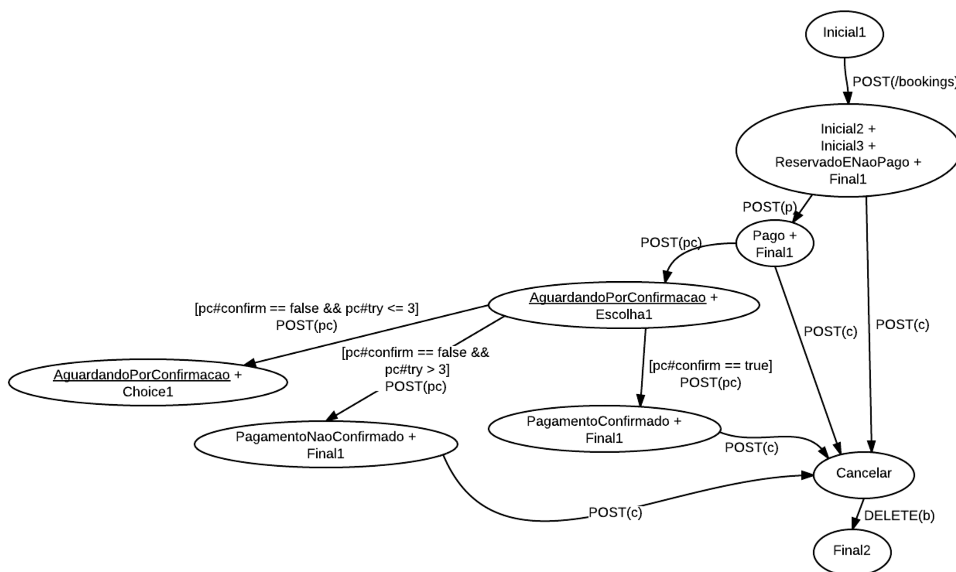

Figura 5.4: A DAG gerada que corresponde ao modelo HRB.

5.2.3 Análise das expressões

No Passo 3 todas as expressões contidas nos estados e transições da DAG são avaliadas por meio de um *parser*; são elas: invariâncias e precondições. Para a criação do *parser* foi utilizado o *framework* Javacc[4]. Para uma análise mais profunda das expressões foi utilizado o JJTree[5], assim cada expressão possui uma árvore sintática abstrata (*Abstract Syntactic Tree* – AST) como resultado do *parser*. Ao longo desse processo a RSTT identifica todas as variáveis nas expressões e as armazenam, tais como: "{bid}" e "{rid}".

Na Figura 5.5 é apresentado um exemplo de uma AST baseada na invariância "OK (/bookings/{bid}) && NOT_FOUND(/bookings/{bid}/cancel)", cuja raiz consiste da operação de conjunção lógica, possuindo como filhos os métodos OK e NOT_FOUND. A árvore deve sempre respeitar as ordens de precedência das operações, respectivamente: negação, conjunção lógica e disjunção lógica.

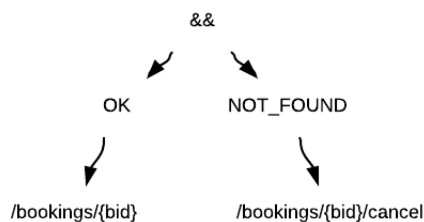

Figura 5.5: Exemplo de uma árvore sintática abstrata.

À medida que as expressões são analisadas o código responsável para executá-las é gerado e armazenado em quatro classes Java. Essas classes são utilizadas mais adiante para auxiliar a execução dos casos de teste. São elas: Classe *Invariants*, Classe *Events*, Classe *Guards* e Classe *Variables*. A Classe *Invariants* possui métodos estáticos Booleanos para executar todas as expressões utilizadas na invariância. A Classe *Events* possui métodos estáticos Booleanos que executam todas as operações POST, PUT e DELETE. Cada operação possui o seu próprio código de resposta HTTP para verificar se o evento foi executado com sucesso. Por exemplo, uma requisição bem sucedida retorna um código da faixa 2xx (200 a 204). A Classe *Guards* possui métodos estáticos Booleanos que executam todas as expressões utilizadas nas precondições. Por fim, a classe *Variables* armazena, caso existam, as variáveis definidas nos estereótipos das transições. As variáveis são públicas e também estão em um contexto estático para facilitar o acesso.

[4] `https://javacc.java.net/`
[5] `https://javacc.java.net/doc/JJTree.html`

5.2.4 Seleção e execução do critério de cobertura, e dados de entrada

No Passo 4 a RSTT solicita para que o testador insira o critério de cobertura desejado dentre duas opções: cobertura de estados e cobertura de transições. Os dois critérios são considerados básicos e comuns na literatura científica em se tratando de cobertura de testes.

Os algoritmos recursivos para cada um dos critérios foram baseados nos trabalhos de Xu et al. (2007) e Rauf e Porres (2011). Os algoritmos utilizados por Xu et al. (2007) são aplicáveis às estruturas de dados do tipo árvore, e, tendo em vista que este trabalho utiliza DAGs, algumas alterações devem ser feitas de modo que o critério funcione corretamente. Uma variável Booleana chamada de $keepTraversing$ é utilizada para certificar que o caminho que o algoritmo está percorrendo representa um novo caminho. O critério de cobertura de estados gera o código baseado na DAG da seguinte forma: *(i)* marque o nó atual como visitado e o adicione à sequência de teste; *(ii)* caso o nó atual não possua filhos, então guarde a sequência de teste e defina a $keepTraversing$ como falsa, caso contrário: *(iii)* ache as transições que possuem como origem o nó atual; *(iv)* para cada uma das transições, se o nó alvo da transição ainda não tiver sido visitado, então defina a $keepTraversing$ como verdadeiro; *(v)* se a variável $keepTraversing$ for verdadeira, então execute o passo *(i)* para o nó alvo da transição e passe a usar uma nova sequência de teste contendo a sequência anterior mais a transição.

O critério de cobertura de transição funciona da seguinte forma: *(i)* adicione o nó atual à sequência de teste; *(ii)* caso o nó atual não possua filhos, então guarde a sequência de teste e defina a $keepTraversing$ como falsa, caso contrário: *(iii)* ache as transições que possuem como origem o nó atual; *(iv)* para cada uma das transições, se a transição atual ainda não tiver sido visitada, então defina a $keepTraversing$ como verdadeiro; *(v)* se a variável $keepTraversing$ for verdadeira, então execute o passo *(i)* para o nó alvo da transição e passe a usar uma nova sequência de teste contendo a sequência anterior mais a nova transição e marque a transição como visitada.

Durante esse processo a ferramenta também solicita ao testador os valores das variáveis identificadas durante o processamento do *parser*.

Ao final da execução dos algoritmos de cobertura (Passo 5) uma lista de sequências de transições é preenchida. Uma sequência de transição pode ser definida como:

$$(s_0, e_0[p_0, q_0], s_1), (s_1, e_1[p_1, q_1], s_2), ..., (s_{n-1}, e_{n-1}[p_{n-1}, q_{n-1}], s_n)$$

a qual representa todos os estados e transições que o algoritmo percorreu e que são posteriormente traduzidas em sequências de teste.

5.2.5 Geração e execução do conjunto de teste

No Passo 5 as sequências de transições obtidas com os critérios de cobertura são utilizadas junto com as ASTs das expressões para gerar as sequências de teste. Uma sequência de teste abrange toda a sequência de estados e transições, porém também aborda as invariâncias contidas nos estados. Os casos de teste representam em Java essas sequências. O código gerado utiliza o *framework* JUnit[6] para executar os testes. A Classe *TestSuite* agrupa os métodos das outras classes já criadas de maneira que o caso de teste siga exatamente a sequência de estados e transições.

A Figura 5.6 mostra um exemplo de código de um caso de teste gerado pela ferramenta com uma única transição a partir de um estado.

```
@Test
public void testCaseExample() {
        if(Invariants.methodOK1()) {  // invariant
                if(Guards.guard1()) { // precondition
                        assertEquals(true, Events.event1()); // event
                } else {
                        fail();
                }
        } else {
                fail();
        }
}
```

Figura 5.6: Exemplo de caso de teste gerado.

Todo caso de teste segue o procedimento de verificar primeiramente a invariância do estado atual (*methodOK1*) e a precondição (*guard1*) em seguida, para finalmente executar o evento (*event1*). Qualquer situação de erro neste caso de teste é ocasionada por algum método retornando falso (eventos, precondições, e invariâncias). Neste caso o código gerado é estruturado para falhar o caso de teste. Se um caso de teste falha, um erro pode ser identificado e relacionado a uma parte específica do modelo.

A RSTT compila automaticamente as classes geradas e organiza as bibliotecas necessárias, assim gerando a infraestrutura necessária para executar os casos de teste. O testador necessita executar a classe *TestSuite* em linha de comando ou em um ambiente de desenvolvimento integrado. Utilizando o critério de cobertura por transições, no total foram gerados 5 casos de teste, um para cada caminho único percorrido na DAG. A seguir consta o primeiro caso de teste gerado, correspondendo ao que antes era um *loop* no modelo.

```
@Test
public void testCase1() {
  /* Transição do estado "Inicial1" para
  "Inicial2 + Inicial3 + ReservadoENaoPago + Final1" */
```

[6] http://junit.org

```
//EVENTO
assertEquals(true, Events.methodPOST1());

/* Invariância correspondente ao estado
"Inicial2 + Inicial3 + ReservadoENaoPago + Final1" */
if (((Invariants.methodNOT_FOUND1()) &&
    (Invariants.methodNOT_FOUND2()) &&
    (Invariants.methodOK1()) &&
    (Invariants.methodOK2()) &&
    (Invariants.methodNOT_FOUND3()))) {

    //EVENTO
    assertEquals(true, Events.methodPOST2());
} else {
    fail();
}

/* Transição do estado
"Inicial2 + Inicial3 + ReservadoENaoPago + Final1"
para o estado "Pago + Final1" */

//Invariância correspondente ao estado "Pago + Final1"
if (((Invariants.methodOK3()) &&
    (Invariants.methodNOT_FOUND4()) &&
    (Invariants.methodOK4()) &&
    (Invariants.methodOK5()) &&
    (Invariants.methodNOT_FOUND5()))) {

    //EVENTO
    assertEquals(true, Events.methodPOST3());
} else {
    fail();
}

/* Transição do estado "Pago + Final1"
para o estado "AguardandoPorConfirmacao + Escolha1" */
```

```
  /* Invariância correspondente ao estado
  "AguardandoPorConfirmacao + Escolha1" */
  if (((Invariants.methodOK9()) &&
      (Guards.methodGuard4()) &&
      (Invariants.methodOK10()) &&
      (Invariants.methodOK11()) &&
      (Invariants.methodNOT_FOUND7()))) {

      //PRECONDIÇÃO
      if (((Guards.methodGuard2()) && (Guards.methodGuard3()))) {

          //EVENTO
          assertEquals(true, Events.methodPOST4());
      } else {
          fail();
      }
  } else {
      fail();
  }

  /* Transição do estado "AguardandoPorConfirmacao + Escolha1"
  para o estado "AguardandoPorConfirmacao + Escolha1" */

  /* Invariância correspondente ao estado
  "AguardandoPorConfirmacao + Escolha1" */
  if (((Invariants.methodOK9()) &&
      (Guards.methodGuard4()) &&
      (Invariants.methodOK10()) &&
      (Invariants.methodOK11()) &&
      (Invariants.methodNOT_FOUND7()))) {
      // SEM EVENTO
  } else {
      fail();
  }
}
```

Com a execução de casos de teste espera-se detectar possíveis erros envolvendo as operações da interface uniforme (GET, POST, PUT, DELETE) e erros nas representações de recurso do serviço. Pelo fato de que o código do serviço HRB não foi disponibilizado, não houve a possibilidade da execução do código gerado pela RSTT. A seguir é descrito um estudo de caso para demonstrar a abordagem em um serviço disponível para uso.

5.3 Estudo de caso

Para demonstrar o uso da abordagem e da ferramenta RSTT, e também para testar a sua aplicabilidade, será utilizado o serviço Google Tasks[7]. Google Tasks é uma aplicação *web* utilizada no GMail[8] e no Google Calendar[9] que permite ao usuário gerenciar várias tarefas (*Tasks*) e listas de tarefas (*TaskList*). É possível definir datas de vencimento para as tarefas e marcá-las como completada. A aplicação fornece uma API que se intitula como RESTful para fornecer estas mesmas funcionalidades. As informações sobre o recurso, URIs e parâmetros são disponibilizadas pelo Google por meio de descrições textuais em páginas HTML[10]. Este serviço já foi utilizado no trabalho de Seijas et al. (2013) também com o propósito de testar serviços RESTful, como citado no Capítulo 4.

O modelo criado na Figura 5.7 mostra um comportamento de criação de uma lista de tarefas (*taskList*) de supermercado, seguido da adição da tarefa (*task*) comprar leite, conclusão da tarefa, remoção da tarefa e remoção da lista de tarefas.

Na prática, o acesso a recursos pode ser restringido por mecanismos de autenticação e controles de acesso. No caso do Google Tasks este mecanismo é realizado por meio de um *token* de autorização. Esse *token* deve ser adicionado pelo testador como o estereótipo "*Authorization : token*" em todos os estados e transições, já que no estado atual a RSTT não fornece apoio à estes mecanismos de autorização. As requisições POST e PUT do modelo necessitam do estereótipo "*Content-Type : application/json*", pois o serviço utiliza o formato JSON como representação de recurso. Ao criar um novo recurso *task* ou *taskList* o servidor fornece como resposta o recurso criado, porém o modo de acessá-los novamente depende de um atributo *id* criado pelo serviço. Caso esse *id* não seja armazenado o acesso a esses recursos ficam mais complexos, sendo necessário acessar uma lista desses recursos. Por isso, o primeiro evento POST contém um estereótipo {*taskListId*} *: response#id* que armazena o *id* do recurso *taskList* criado, já o segundo POST contém o estereótipo {*taskId*} *: response#id*.

[7] `https://mail.google.com/mail/help/tasks/`
[8] `http://www.gmail.com`
[9] `https://www.google.com/calendar`
[10] `https://developers.google.com/google-apps/tasks/v1/reference/`

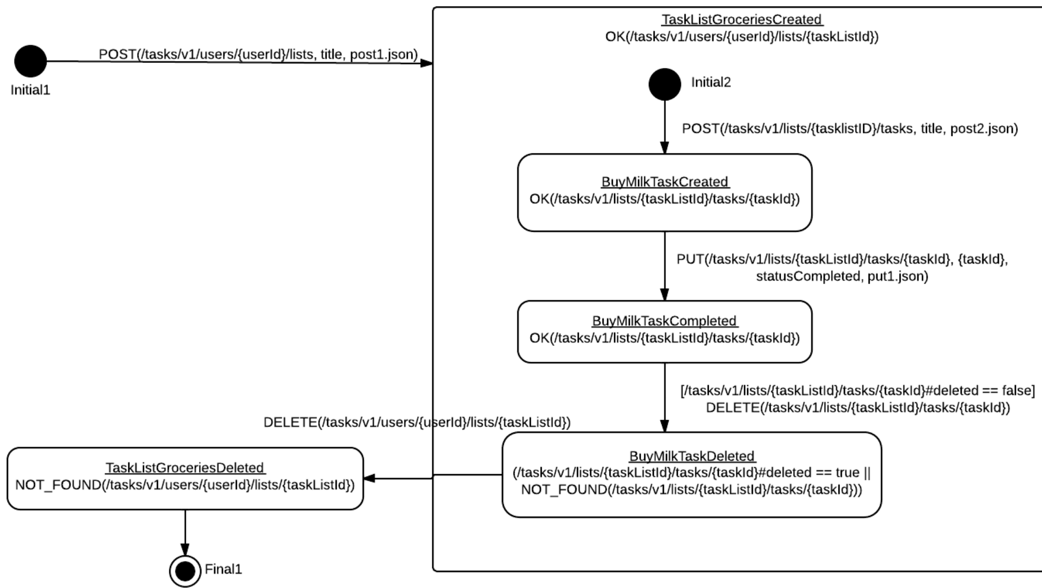

Figura 5.7: Máquina de estados de protocolos do Google Task.

Ao realizar o método DELETE, deve-se ter cuidado para não causar um falso erro no teste. Normalmente a requisição DELETE significa excluir o recurso totalmente do serviço, porém em muitos serviços o que acontece é que o recurso fica apenas marcado como excluído, assim, ficando indisponível para o acesso pelo usuário. O serviço, no entanto, ainda pode fornecer o recurso dando a falsa impressão de que o recurso não foi excluído. Para garantir que o recurso de tarefa foi excluído de fato, a invariância do estado *BuyMilkTaskDeleted* inclui o método NOT_FOUND para o recurso, mais também uma verificação pelo atributo *deleted* na representação.

A Figura 5.8 mostra a DAG gerada a partir do modelo. Neste caso o critério de cobertura de estados e transições são equivalentes.

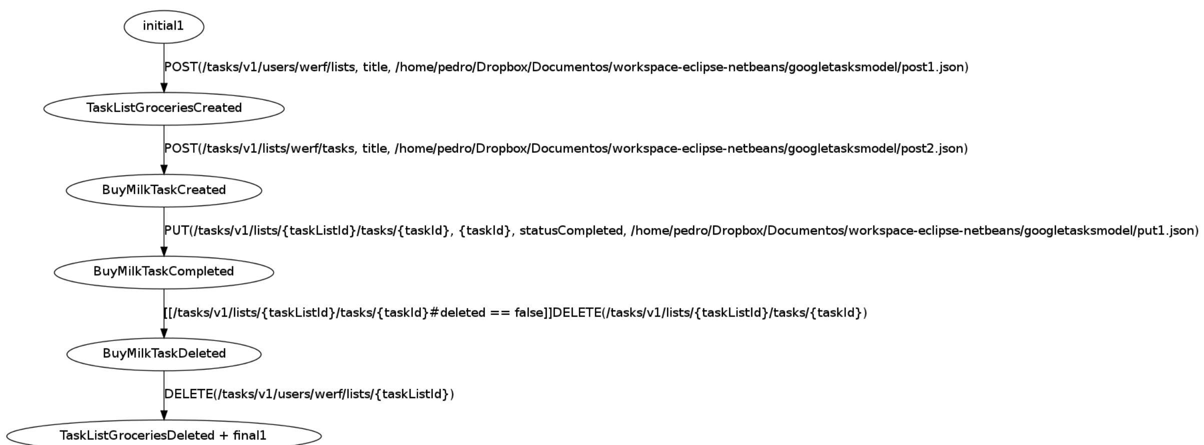

Figura 5.8: DAG gerada a partir do modelo.

Os templates utilizados (*post*1.*json*, *post*2.*json* e *put*1.*json*) representam os recursos a serem enviados para o servidor. A seguir consta os casos de teste gerados para o modelo.

```
@Test
public void testCase1() {
    assertEquals(true, Events.methodPOST1());

    if ((Invariants.methodOK1())) {
        assertEquals(true, Events.methodPOST2());
    } else {
        fail();
    }
    if (((Invariants.methodOK2()) && (Invariants.methodOK3()))) {
        assertEquals(true, Events.methodPUT1());
    } else {
        fail();
    }

    if (((Invariants.methodOK4()) && (Invariants.methodOK5()))) {
        if ((Guards.methodGuard1())) {
            assertEquals(true, Events.methodDELETE1());
        } else {
            fail();
        }
    } else {
        fail();
    }

    if ((((Guards.methodGuard2()) || (Invariants.methodNOT_FOUND1()))
           && (Invariants.methodOK6()))) {
        assertEquals(true, Events.methodDELETE2());
    } else {
        fail();
    }

    if ((Invariants.methodNOT_FOUND2())) {
    } else {
```

```
            fail();
        }
    }
```

Cada sequência de teste representa um caso de teste, assim, apenas um caso de teste foi gerado para o modelo. A execução do caso de teste não revelou a presença de erros, porém representa um passo inicial para adequar a abordagem às características fundamentais dos serviços RESTful. Tendo em vista os resultados obtidos, pode-se perceber que a aplicação da abordagem de teste é possível. A ferramenta RSTT apresentada também facilitará a realização de novos estudos com outros exemplos de serviços.

5.4 Considerações Finais

Neste capítulo foi apresentada uma abordagem para o uso do teste baseado em modelos para serviços RESTful usando as máquinas de protocolos UML como modelo comportamental. Atualmente poucos trabalhos focam em contribuir para a área de teste de serviços RESTful. Dessa forma, espera-se que essa abordagem contribua para melhorar a qualidade desses serviços. No próximo capítulo, são apresentadas as conclusões deste trabalho, destacando as contribuições, limitações, dificuldades encontradas e sugestões de trabalhos futuros.

Capítulo 6

Conclusão

Este trabalho propôs uma abordagem de teste baseado em modelos para a geração automática de casos de teste para serviços RESTful. A abordagem utiliza um modelo adaptado de máquinas de estados de protocolos UML que respeita uma arquitetura orientada a recurso. Este trabalho também apresentou um protótipo da ferramenta que auxilia o uso da abordagem. O estado atual da implementação gera casos de teste a partir do modelo baseados na cobertura de estados e transições. Foi apresentado um exemplo para ilustrar os passos da abordagem e um estudo de caso para testar a aplicação e uso da abordagem/ferramenta.

6.1 Contribuições

Atualmente, poucos trabalhos objetivam fornecer modelos ou abordagens para o teste de serviços RESTful. Portanto, este trabalho buscou com essas contribuições motivar mais pesquisas relacionadas ao teste baseado em modelos de serviços RESTful, do ponto de vista teórico e experimental, além de apresentar uma abordagem a qual pode ser utilizada na pesquisa por melhorias na qualidade de *software* e facilitar a geração de testes para os serviços RESTful.

Para que um modelo conceitual consiga retratar os serviços RESTful eles devem ser capazes de modelar os recursos e seus relacionamentos, as operações para cada recurso (utilizando os métodos do protocolo HTTP), e a definição dos formatos de dados para eles

(Laitkorpi et al., 2009). Com a abordagem proposta foi possível definir essas informações no modelo comportamental do serviço, além dos estados que o serviço está situado após cada requisição que o afete.

Pode-se destacar como principais contribuições deste trabalho:

- A definição de uma abordagem para o teste baseado em modelos de serviços RESTful. A abordagem foi publicada no *7th Brazilian Workshop on Systematic and Automated Software Testing* (SAST 2013) (Pinheiro et al., 2013).

- A implementação de uma ferramenta que fornece apoio semiautomatizado à estratégia proposta.

Essas contribuições também demonstram que em razão da simplicidade do estilo REST e das convenções que ele aplica, é possível generalizar uma rotina de teste baseado na técnica TBM. Desta forma os casos de teste são gerados apenas na especificação e descrição do serviço, sem a necessidade de conhecimento de código interno. Os casos de teste podem ser gerados em serviços por mais diferentes que sejam seus contextos de aplicação.

Embora o trabalho ainda não possua evidências empíricas suficientes para atestar para a qualidade da abordagem em encontrar os defeitos, é esperado que este processo apoie os testadores na criação e melhoria dos testes, além de fornecer uma forma de modelar o comportamento dos serviços baseado apenas em descrições textuais dos mesmos.

6.2 Dificuldades e Limitações

Embora existam esforços na busca por representações eficientes para descrever os serviços RESTful, muitos serviços ainda utilizam descrições textuais em páginas HTML. Com isso a adequação automática da ferramenta a diferentes serviços torna-se um grande desafio, cabendo ao testador definir manualmente no modelo a maneira que as requisições devem ser feitas.

Uma dificuldade encontrada ao se definir uma abordagem para o teste de serviços RESTful em geral é que diferentes serviços RESTful podem empregar diferentes representações de recurso. Isso implica que a ferramenta deve ser preparada para aceitar as principais representações utilizadas no mercado. A abordagem contorna esse problema deixando essa tarefa com o testador para definir um *template* que o serviço aceita (ao lidar com a requisição), e a ferramenta aceita apenas os formatos XML e JSON (ao lidar com a resposta).

Outra dificuldade é que a grande maioria dos serviços RESTful utilizam protocolos de autenticação e/ou autorização, os quais também podem ser diferentes dependendo de cada serviço. O desafio nesse ponto é da ferramenta não só ser capaz de lidar com todos

os protocolos, mas também de gerar código capaz de identificar quando, por exemplo, um *token* expirou, sendo necessária assim sua renovação. Alguns serviços de terceiros também podem limitar o número de requisições a serem feitas no dia, ou o tempo entre elas.

A ferramenta, no estágio atual, possui limitações ao lidar com representações de recursos. Assume-se que a representação de recurso possua atributos de nome único para poder obter os dados. Assim, ela é limitada ao lidar com representações que listam vários recursos ou representações muito complexas. Melhorias nas expressões XPath podem ser feitas para contornar esse problema, dando, assim, mais controle para o testador sob as representações. A abordagem/ferramenta também não é capaz de testar acessos concorrentes aos recursos.

Uma outra grande limitação da abordagem é o próprio *software* utilizado para criar o modelo (ArgoUML). Isso restringe a ferramenta a apenas as funcionalidades e ao XMI gerado pelo programa de modelagem. Uma solução seria a implementação de um *software* de modelagem que atenda as necessidades da máquina de estados de protocolos para os serviços RESTful. Por fim, há também uma dificuldade de definir e usar no modelo os metadados contidos no cabeçalho das respostas, o que permitiria a criação de requisições mais complexas, como por exemplo, requisições condicionais. No estágio atual a ferramenta utiliza os estereótipos nos estados ou transições para lidar com metadados mais fundamentais às requisições como o tipo de conteúdo (*Content-Type*) e autorização (*Authorization*).

6.3 Trabalhos Futuros

Como trabalhos futuros pode-se destacar:

- Estender a ferramenta RSTT para incluir a geração automática de dados de teste;
- Implementar outros critérios de cobertura, como os critérios *basic* e *extended round-trip*, descritos no trabalho de Xu et al. (2007);
- Especificar as informações conceituais (invariâncias, precondições, representação de recurso), contidas no modelo comportamental, em expressões formais como, por exemplo, expressões OCL (Hamann et al., 2012).
- Implementar apoio para as tecnologias mais comuns na maioria dos serviços, tais como autenticação, autorização e diversas representações de recursos;
- Adequar a abordagem para permitir o uso na composição de composição de serviços RESTful, levando em consideração avanços recentes sobre o tópico como em Rauf et al. (2010).

- Analisar o desempenho da abordagem com mais exemplos do mundo real de serviços industriais RESTful, além de mais avaliações experimentais para demonstrar a eficácia em se tratando de localização de erros.

Referências

Abran, A.; Bourque, P.; Dupuis, R.; Moore, J. W.; Tripp, L. L. *Guide to the software engineering body of knowledge - SWEBOK*. 2004 ed. Piscataway, NJ, USA: IEEE Press, 1–202 p., 2004.
Disponível em http://ocw.unican.es/ensenanzas-tecnicas/ingenieria-del-software-i/otros-recursos-1/SWEBOK_Guide_2004.pdf (Acessado em 06/03/2014)

Al-Zoubi, K.; Wainer, G. Using REST web-services architecture for distributed simulation. In: *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, Washington, DC, USA: IEEE Computer Society, 2009, p. 114–121 (*PADS '09*,).
Disponível em http://dx.doi.org/10.1109/PADS.2009.16 (Acessado em 06/03/2014)

Allam, D. A unified formal model for service oriented architecture to enforce security contracts. In: *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development Companion*, AOSD Companion '12, New York, NY, USA: ACM, 2012, p. 9–10 (*AOSD Companion '12*,).
Disponível em http://doi.acm.org/10.1145/2162110.2162120

Allamaraju, S. *RESTful web services cookbook*. 1st ed. O'Reilly Media, Inc., 2010.

AlShahwan, F.; Moessner, K. Providing SOAP web services and RESTful web services from mobile hosts. In: *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, 2010, p. 174–179.

Barbosa, E. F.; Chaim, M. L.; Vincenzi, A. M. R.; Delamaro, M. E.; Jino, M.; Maldonado., J. C. *Introdução ao teste de software*, cáp. Teste estrutural. 1st ed Elsevier, p. 47–74, 2007.

Baresi, L.; Di Nitto, E.; Ghezzi, C. Toward open-world software: Issue and challenges. *Computer*, v. 39, n. 10, p. 36–43, 2006.

Berners-Lee, T.; Fielding, R.; Masinter, L. Uniform resource identifier (URI): Generic syntax. 2005.
Disponível em http://tools.ietf.org/html/rfc3986 (Acessado em 06/03/2014)

Birrell, A. D.; Nelson, B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, v. 2, n. 1, p. 39–59, 1984.
Disponível em http://doi.acm.org/10.1145/2080.357392

Boehm, B. A view of 20th and 21st century software engineering. In: *Proceedings of the 28th international conference on Software engineering*, ICSE '06, New York, NY, USA: ACM, 2006, p. 12–29 (*ICSE '06*,).
Disponível em http://doi.acm.org/10.1145/1134285.1134288 (Acessado em 06/03/2014)

Boehm, B. W. Verifying and validating software requirements and design specifications. *IEEE Softw.*, v. 1, p. 75–88, 1984.
Disponível em http://dl.acm.org/citation.cfm?id=1308434.1308744 (Acessado em 06/03/2014)

Borges, F. Q. REST-Unit: Geração baseada em U2TP de drivers de teste para restful web services. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre, 2009.

Broekman, B.; Notenboom, E. *Testing embedded software.* 1st ed. Addison-Wesley Professional, 2002.

Candolo, M. A. P.; Simão, A. S.; Maldonado, J. C. MGASet - uma ferramenta para apoiar o teste e validação de especificações baseadas em máquinas de estado finito. In: *Anais do XV Simpósio Brasileiro de Engenharia de Software*, Rio de Janeiro, RJ, 2001, p. 386–391.

Canfora, G.; Di Penta, M. Service-oriented architectures testing: A survey. In: *Software Engineering: International Summer Schools (ISSSE)*, Berlin, Heidelberg: Springer-Verlag, 2009, p. 78–105.

Cartaxo, E.; Neto, F.; Machado, P. Test case generation by means of uml sequence diagrams and labeled transition systems. In: *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, 2007, p. 1292–1297.

Chakrabarti, S.; Kumar, P. Test-the-REST: An approach to testing RESTful web-services. In: *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World:*, 2009, p. 302–308.

Chakrabarti, S. K.; Rodriquez, R. Connectedness testing of RESTful web-services. In: *India software engineering conference (ISEC)*, New York, NY, USA: ACM, 2010, p. 143–152.

Chow, T. Testing software design modeled by finite-state machines. *Software Engineering, IEEE Transactions on*, v. SE-4, n. 3, p. 178–187, 1978.

Christensen, J. H. Using RESTful web-services and cloud computing to create next generation mobile applications. In: *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, New York, NY, USA: ACM, 2009, p. 627–634 (*OOPSLA '09*,).
Disponível em `http://doi.acm.org/10.1145/1639950.1639958` (Acessado em 06/03/2014)

Claessen, K.; Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, New York, NY, USA: ACM, 2000, p. 268–279 (*ICFP '00*,).
Disponível em `http://doi.acm.org/10.1145/351240.351266`

Correa, A. L.; de Souza, T. S.; Schmitz, E. A.; Alencar, A. J. Defining restful web services test cases from uml models. In: *SEKE*, Knowledge Systems Institute Graduate School, 2012, p. 319–323.
Disponível em `http://dblp.uni-trier.de/db/conf/seke/seke2012.html#CorreaSSA12`

Dalal, S. R.; Jain, A.; Karunanithi, N.; Leaton, J. M.; Lott, C. M.; Patton, G. C.; Horowitz, B. M. Model-based testing in practice. In: *International conference on Software engineering (ICSE)*, Los Angeles, USA: ACM, 1999, p. 285–294.

Delamaro, M. E. *Proteum: Um ambiente de teste baseado na análise de mutantes*. Dissertação de Mestrado, ICMC-USP, São Carlos, SP, 1993.

Delamaro, M. E.; Barbosa, E. F.; Vincenzi, A. M. R.; Maldonado, J. C. *Introdução ao teste de software*, cáp. Teste de Mutação. 1st ed Elsevier, p. 77–117, 2007a.

Delamaro, M. E.; Maldonado, J. C. *Proteum/im, version 1.1 c – user's guide.* 1996.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. *Introdução ao teste de software*, cáp. Conceitos Básicos. 1st ed Elsevier, p. 1–7, 2007b.

DeMillo, R.; Lipton, R.; Sayward, F. Hints on test data selection: Help for the practicing programmer. *Computer*, v. 11, n. 4, p. 34–41, 1978.

Eler, M.; Endo, A.; Masiero, P.; Delamaro, M.; Maldonado, J.; Vincenzi, A.; Chaim, M.; Beder, D. JaBUTiService: A web service for structural testing of java programs. In: *Software Engineering Workshop (SEW), 2009 33rd Annual IEEE*, 2009, p. 69–76.

Erenkrantz, J. R.; Gorlick, M.; Suryanarayana, G.; Taylor, R. N. From representations to computations: the evolution of web architectures. In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, New York, NY, USA: ACM, 2007, p. 255–264 (*ESEC-FSE '07*,).
Disponível em `http://doi.acm.org/10.1145/1287624.1287660` (Acessado em 06/03/2014)

Fabbri, S. C. P. F. *A análise de mutantes no contexto de sistemas reativos: Uma contribuição para o estabelecimento de estratégias de teste e validação.* Tese de Doutoramento, IFSC/USP, São Carlos, SP, 1996.

Fabbri, S. C. P. F.; Maldonado, J. C.; Delamaro, M. E.; Masiero, P. C. Proteum/FSM: A tool to support finite state machine validation based on mutation testing. In: *In XIX SCCC - International Conference of the Chilean Computer Science Society*, Talca, Chile, 1999, p. 96–104.

Fabbri, S. C. P. F.; Vincenzi, A. M. R.; Maldonado, J. C. *Introdução ao teste de software*, cáp. Teste Funcional. 1st ed Elsevier, p. 9–24, 2007.

Feller, N. J. Estendendo rest-unit: geração baseada em U2TP de drivers e dados de teste para RESTful web services. Projeto de Diplomação (Bacharelado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre, 2010.

Feng, X.; Shen, J.; Fan, Y. REST: An alternative to RPC for web services architecture. In: *Future Information Networks, 2009. ICFIN 2009. First International Conference on*, 2009, p. 7–10.

Fensel, D.; Facca, F. M.; Simperl, E.; Toma, I. *Semantic web services: Concepts, technologies, and applications.* Springer, 406 p., 2007.

Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P.; Berners-Lee, T. Hypertext transfer protocol – HTTP/1.1. 1999.
Disponível em `http://www.w3.org/Protocols/rfc2616/rfc2616.html` (Acessado em 06/03/2014)

Fielding, R. T. *Architectural styles and the design of network-based software architectures.* Doctoral dissertation, University of California, Irvine, 2000.

Fielding, R. T.; Taylor, R. N. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, v. 2, p. 115–150, 2002.
Disponível em `http://doi.acm.org/10.1145/514183.514185` (Acessado em 06/03/2014)

Fink, G.; Levitt, K. Property-based testing of privileged programs. In: *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*, 1994, p. 154–163.

Guedes, G. T. A. *Uml : uma abordagem prática.* 3 ed. ed. Novatec Editora, 2008.

Hamann, L.; Hofrichter, O.; Gogolla, M. On integrating structure and behavior modeling with ocl. In: France, R.; Kazmeier, J.; Breu, R.; Atkinson, C., eds. *Model Driven Engineering Languages and Systems*, v. 7590 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 235–251, 2012.
Disponível em `http://dx.doi.org/10.1007/978-3-642-33666-9_16`

Harel, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, v. 8, p. 231–274, 1987.
Disponível em `http://dl.acm.org/citation.cfm?id=34884.34886` (Acessado em 06/03/2014)

IEEE IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, p. 1, 1990.

Josuttis, N. *SOA in practice: The art of distributed system design.* O'Reilly Media, Inc., 2007.

Keller, R. M. Formal verification of parallel programs. *Commun. ACM*, v. 19, n. 7, p. 371–384, 1976.
Disponível em `http://doi.acm.org/10.1145/360248.360251`

Khare, R.; Taylor, R. N. Extending the representational state transfer (REST) architectural style for decentralized systems. In: *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, Washington, DC, USA: IEEE Computer Society, 2004, p. 428–437 (*ICSE '04*,).
Disponível em http://dl.acm.org/citation.cfm?id=998675.999447 (Acessado em 06/03/2014)

Kim, J. S.; Garlan, D. Analyzing architectural styles with alloy. In: *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, ROSATEA '06, New York, NY, USA: ACM, 2006, p. 70–80 (*ROSATEA '06*,).
Disponível em http://doi.acm.org/10.1145/1147249.1147259 (Acessado em 06/03/2014)

Laitkorpi, M.; Selonen, P.; Systä, T. Towards a model-driven process for designing RESTful web services. In: *2009 IEEE International Conference on Web Services*, IEEE, 2009, p. 173–180.

Lee, D.; Yannakakis, M. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, v. 84, n. 8, p. 1090–1123, 1996.

Lemos, O. A. L. *Teste de programas orientados a aspectos: uma abordagem estrutural para AspectJ*. Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 2005.

Liu, S.; Liu, Y.; André, É.; Choppy, C.; Sun, J.; Wadhwa, B.; Dong, J. A formal semantics for complete UML state machines with communications. In: Johnsen, E.; Petre, L., eds. *Integrated Formal Methods*, v. 7940 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, p. 331–346, 2013.
Disponível em http://dx.doi.org/10.1007/978-3-642-38613-8_23

Maldonado, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software*. Tese de Doutoramento, DCA/FEE/UNICAMP, Campinas, SP, 1991.

Maleshkova, M.; Pedrinaci, C.; Domingue, J. Investigating web APIs on the world wide web. In: *Proceedings of the 2010 Eighth IEEE European Conference on Web Services*, ECOWS '10, Washington, DC, USA: IEEE Computer Society, 2010, p. 107–114 (*ECOWS '10*,).
Disponível em http://dx.doi.org/10.1109/ECOWS.2010.9 (Acessado em 06/03/2014)

Marinos, A.; Wilde, E.; Lu, J. HTTP database connector (HDBC): RESTful access to relational databases. In: *Proceedings of the 19th international conference on World*

wide web, WWW '10, New York, NY, USA: ACM, 2010, p. 1157–1158 (*WWW '10*,). Disponível em `http://doi.acm.org/10.1145/1772690.1772852` (Acessado em 06/03/2014)

Martins, E.; Sabião, S. B.; Ambrosio, A. M. ConData: A tool for automating specification-based test case generation for communication systems. *Software Quality Control*, v. 8, p. 303–320, 1999.
Disponível em `http://dl.acm.org/citation.cfm?id=599121.599190` (Acessado em 06/03/2014)

McCabe, T. A complexity measure. *Software Engineering, IEEE Transactions on*, v. SE-2, n. 4, p. 308–320, 1976.

Meng, J.; Mei, S.; Yan, Z. RESTful Web Services: A solution for distributed data integration. In: *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, 2009, p. 1–4.

Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, v. 77, n. 4, p. 541–580, 1989.

Myers, G. J.; Sandler, C.; Badgett, T.; Thomas, T. M. *The art of software testing*. 2nd ed. John Wiley & Sons, Ltd., 2004.

Object Management Group (OMG) Uml 2.0 testing profile version 1.0. 2005.
Disponível em `http://www.omg.org/spec/UTP/1.0/` (Acessado em 06/03/2014)

OMG OMG unified modeling language, superstructure version 2.4.1. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF, 2011.
Disponível em `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF` (Acessado em 06/03/2014)

Overdick, H. The resource-oriented architecture. In: *Services, 2007 IEEE Congress on*, 2007, p. 340–347.

Papazoglou, M. P.; Heuvel, W.-J. Service-oriented architectures: approaches, technologies and research issues. *The International Journal on Very Large Databases (VLDB)*, v. 16, n. 3, p. 389–415, 2007.

Pautasso, C.; Zimmermann, O.; Leymann, F. RESTful web services vs. "big" web services: making the right architectural decision. In: *Proceeding of the 17th international conference on World Wide Web*, WWW '08, New York, NY, USA: ACM, 2008, p. 805–814 (*WWW '08*,).

Disponível em http://doi.acm.org/10.1145/1367497.1367606 (Acessado em 06/03/2014)

Perry, D. E.; Wolf, A. L. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, v. 17, p. 40–52, 1992.
Disponível em http://doi.acm.org/10.1145/141874.141884 (Acessado em 06/03/2014)

Peterson, J. L. Petri nets. *ACM Comput. Surv.*, v. 9, p. 223–252, 1977.
Disponível em http://doi.acm.org/10.1145/356698.356702 (Acessado em 06/03/2014)

Petrenko, A.; Boroday, S.; Groz, R. Confirming configurations in EFSM testing. *Software Engineering, IEEE Transactions on*, v. 30, n. 1, p. 29–42, 2004.

Petri, C. A. *Kommunikation mit automaten.* Tese de Doutoramento, Universität Hamburg, 1962.

Pinheiro, P.; Endo, A.; Simao, A. Model-based testing of RESTful web services using UML protocol state machines. In: *SAST 2013*, 2013.

Porres, I.; Rauf, I. Generating class contracts from uml protocol statemachines. In: *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '09, New York, NY, USA: ACM, 2009, p. 8:1–8:10 (*MoDeVVa '09*,).
Disponível em http://doi.acm.org/10.1145/1656485.1656493

Porres, I.; Rauf, I. Modeling behavioral RESTful web service interfaces in UML. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, 2011, p. 1598–1605 (*SAC '11*,).
Disponível em http://doi.acm.org/10.1145/1982185.1982521

Pressman, R. S. *Engenharia de software: Uma abordagem profissional.* 7th ed. AMGH, 2011.

Pretschner, A.; Prenninger, W.; Wagner, S.; Kühnel, C.; Baumgartner, M.; Sostawa, B.; Zölch, R.; Stauner, T. One evaluation of model-based testing and its automation. In: *Proceedings of the 27th international conference on Software engineering*, ICSE '05, New York, NY, USA: ACM, 2005, p. 392–401 (*ICSE '05*,).
Disponível em http://doi.acm.org/10.1145/1062455.1062529 (Acessado em 06/03/2014)

Rapps, S.; Weyuker, E. J. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, v. 11, p. 367–375, 1985.
Disponível em http://dx.doi.org/10.1109/TSE.1985.232226 (Acessado em 06/03/2014)

Rauf, I.; Porres, I. Designing level 3 behavioral RESTful web service interfaces. *SIGAPP Appl. Comput. Rev.*, v. 11, p. 19–31, 2011.
Disponível em http://doi.acm.org/10.1145/2034594.2034596 (Acessado em 06/03/2014)

Rauf, I.; Ruokonen, A.; Systa, T.; Porres, I. Modeling a composite RESTful web service with UML. In: *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, 2010, p. 253–260 (*ECSA '10*,).
Disponível em http://doi.acm.org/10.1145/1842752.1842801

Reza, H.; Van Gilst, D. A framework for testing RESTful web services. In: *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations*, ITNG '10, Washington, DC, USA: IEEE Computer Society, 2010, p. 216–221 (*ITNG '10*,).
Disponível em http://dx.doi.org/10.1109/ITNG.2010.175 (Acessado em 06/03/2014)

Richardson, L.; Ruby, S. *RESTful web services*. 1st ed. O'Reilly Media, Inc., 2007.

Sandoval, J. *RESTful java web services*. 1st ed. Packt Publishing Ltd, 2009.

Schreier, S. Modeling RESTful applications. In: *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11, New York, NY, USA: ACM, 2011, p. 15–21 (*WS-REST '11*,).
Disponível em http://doi.acm.org/10.1145/1967428.1967434 (Acessado em 06/03/2014)

Seijas, P. L.; Li, H.; Thompson, S. Towards property-based testing of restful web services. In: *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, New York, NY, USA: ACM, 2013, p. 77–78 (*Erlang '13*,).
Disponível em http://doi.acm.org/10.1145/2505305.2505317

Silva, T. R. PyRester: uma abordagem baseada em modelos U2TP para geração de código de teste unitário para restful web services. Projeto de Diplomação (Bacharelado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre, 2011.

Simão, A. S. *Introdução ao teste de software*, cáp. Teste Baseado em Modelos. 1st ed Elsevier, p. 27–45, 2007.

Simão, A. S.; Ambrosio, A. M.; Fabbri, S. C. P. F.; Amaral, A. S.; Martins, E.; Maldonado, J. C. Plavis/FSM: an environment to integrate fsm-based testing tools. In: *Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software*, Uberlândia, MG, 2005, p. 1–6.

Simão, A. S.; Maldonado, J. C.; Fabbri, S. C. P. F. Proteum-RS/PN: A tool to support edition, simulation and validation of petri nets based on mutation testing. In: *Anais do XIV Simpósio Brasileiro de Engenharia de Software*, João Pessoa, PB, 2000, p. 227–242.

Sinha, A.; Smidts, C. HOTTest: A model-based test design technique for enhanced testing of domain-specific applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 15, n. 3, p. 242–278, 2006.

Sommerville, I. *Engenharia de software.* 8th ed. Pearson Addison-Wesley, 2007.

Souza, S. R. S.; Maldonado, J. C.; Fabbri, S. C. P. F.; Masiero, P. C. Statecharts specifications: A family of coverage testing criteria. In: *XXVI Conferência Latinoamericana de Informática*, CLEI 2000, Monterrey, México, 2000 (*CLEI 2000*,).

Sugeta, T. *Proteum-RS/ST: Uma ferramenta para apoiar a validação de especificações statecharts baseada na análise de mutantes 1999.* Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 1999.

Tan, Q. M.; Petrenko, A.; Bochmann, G. V. A test generation tool for specifications in the form of state machines. In: *Proceedings of the International Communications Conference*, 1996, p. 225–229.

Thies, G.; Vossen, G. Modelling web-oriented architectures. In: *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96*, APCCM '09, Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2009, p. 97–106 (*APCCM '09*,).
Disponível em `http://dl.acm.org/citation.cfm?id=1862739.1862753` (Acessado em 06/03/2014)

Utting, M.; Legeard, B. *Practical model-based testing: A tools approach.* Elsevier, 2007.

Utting, M.; Pretschner, A.; Legeard, B. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 2011.
Disponível em `http://dx.doi.org/10.1002/stvr.456` (Acessado em 06/03/2014)

Vilela, P. R.; Vergilio, S. R.; Maldonado, J. C.; Jino, M. *Introdução aos critérios potenciais usos e à poke-tool*. Campinas, SP, Brasil, 1995.
Disponível em http://www.inf.ufpr.br/silvia/topicos/pokemanual.ps (Acessado em 06/03/2014)

Vincenzi, A.; Delamaro, M.; Höhn, E.; Maldonado, J. Functional, control and data flow, and mutation testing: Theory and practice. In: Borba, P.; Cavalcanti, A.; Sampaio, A.; Woodcook, J., eds. *Testing Techniques in Software Engineering*, v. 6153 de *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, p. 18–58, 2010.
Disponível em http://dx.doi.org/10.1007/978-3-642-14335-9_2 (Acessado em 06/03/2014)

Vincenzi, A. M. R.; Delamaro, M. E.; Maldonado, J. C. *JaBUTi - java bytecode understanding and testing – user's guide*. 2003.

Vinoski, S. Restful web services development checklist. *Internet Computing, IEEE*, v. 12, n. 6, p. 96–95, 2008.

Webber, J.; Parastatidis, S.; Robinson, I. *REST in practice: Hypermedia and systems architecture*. O'Reilly Media, Inc., 2010.

Xu, D.; Xu, W.; Wong, W. E. Automated test code generation from UML protocol state machines. In: *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE'07)*, 2007.

Yang, C.; Chen, N.; Di, L. RESTful based heterogeneous geoprocessing workflow interoperation for sensor web service. *Computers & Geosciences*, 2011.
Disponível em http://dx.doi.org/10.1016/j.cageo.2011.11.010 (Acessado em 06/03/2014)