Geração automática de casos de testes para web services

Paulo Silveira

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Área de Concentração: Ciência da Computação
Orientador: Profa. Dra. Ana Cristina Vieira de Melo

São Paulo, junho de 2008

Geração automática de casos de testes para web services

Este exemplar corresponde à redação
final da dissertação/tese devidamente corrigida
e defendida por Paulo Silveira
e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Profa. Dra. Ana Cristina Vieira de Melo (orientadora) - IME-USP
- Prof. Dr. Flavio Soares Correa da Silva - IME-USP
- Profa.Dra. Silvia Regina Vergilio - UFPR

Agradecimentos

É imenso o reflexo que pequenos contatos tem na vida das pessoas. Desde que comecei a pensar em cursar um programa de mestrado são inúmeras as pessoas que contribuíram de alguma forma, o que dificulta muito a agradecer.

Primeiramente agradeço aos meus pais que sempre deram apoio e suporte a minhas idéias e deixaram sempre meu lado espontâneo fluir. Aos meus grandes amigos e companheiros de jornada, Renato (meu verdadeiro irmão) e Bruno (irmão de coração).

Agradeço à minha orientadora, Ana Cristina Vieira de Melo, principalmente por ter aceitado o desafio de realizar um trabalho à distância. Aos demais funcionários e professores do IME, em especial ao Professor Siang Wun Song, qual me deu a oportunidade de cursar sua disciplina quando ainda era aluno especial.

Aos amigos do Banco Real pelo apoio, em particular a Evandro, Marcelo, Hideyuki, Maurício, Inês, Edson (pelo bom conselho no momento certo) e a todos os demais da equipe. Foi nesta empresa onde encontrei grandes amigos, sendo vinte e quatro deles estagiários alucinados e três *jolie filles*.

A todos os meus familiares, entre eles: Guiomar, Divina, Marfisa, Hilda, Mariana, Gustavo, etc.

Às minhas amigas Andrea, Janaina, Bárbara e Helenice. E a meus novos amigos no Canadá, em especial: Marli, Ricardo, Paula, Robin (o único aqui citado duas vezes), Elisa e André. Ao Ivo, que me ensinou os primeiros conceitos de programação.

Um imenso obrigado a minha querida e amada esposa, Leila, que foi sempre companheira e ajudante.

Agradeço também a Deus que criou, é e faz parte desta imensa energia que é o universo, onde podemos senti-lo e observá-lo.

Resumo

Atualmente, na grande maioria das organizações os sistemas de informação possuem uma importância fundamental no suporte aos fluxos de negócios. A necessidade de troca de informações entre instituições demanda cada vez mais a capacidade de flexibilidade e integração. É neste contexto que surgiram os *Web Services*, fornecendo suporte a este tipo de aplicação. Os níveis de qualidade requeridos por estes serviços são altos e a necessidade por ferramentas para automatizar os testes são eminentes.

Este trabalho apresenta melhorias em uma técnica de teste chamada Perturbação de Dados (*Data Perturbation*). Ela é utilizada para gerar casos de testes de forma automática baseada na especificação do serviço. Uma ferramenta chamada GenAutoWS foi construída implementando estas melhorias e tem como enfoque auxiliar dois dos papéis da arquitetura SOA: o consumidor e o provedor de serviços, embora que esta também ofereça integração com o servidor UDDI. Estudos de casos foram realizados com o objetivo de auxiliar a verificar a eficiência das propostas de melhoria desta técnica.

Palavras-chave: Web Services, geração de casos de testes, perturbação de dados.

Abstract

Most organizations rely on information systems as part of their business process. The need to exchange data between different applications require more flexibility and interoperability. As a result, Web Services emerged to support such requirements. Services can communicate with each other by passing data from one service to another, or by coordinating an activity between two or more services. The quality level of such systems must be very high and automated test tools can help to improve their quality.

This work presents modification and enhancement for a technique called Data Perturbation. This technique is used with Web Services specification to generate test cases. A tool named GenAutoWS was built implementing this technique. The goal of this tool is to help service providers and subscribers, although it offers UDDI integration as well. A case study was done to verify the efficiency of our proposed method.

Keywords: Web Services, test case generation, data perturbation.

Sumário

Lista de Abreviaturas

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
EJB	Enterprise Java Beans
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
MIME	Multipurpose Internet Mail Extensions
NASA	National Aeronautics and Space Administration
NIST	National Institute of Standards and Technology
OASIS	Organization for the Advancement of Structured Information Standards
OWL-S	Web Service Ontology Language
RMI	Java Remote Method Invocation
RPC	Remote Procedure Call
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TCP/IP	Transmission Control Protocol / Internet Protocol
UDDI	Universal Description, Discovery and Integration
WADL	Web Application Description Language
W3C	The World Wide Web Consortium
WSDL	Web Services Description Language
WSDL-S	Web Services Semantics
XML	Extensible Markup Language

Lista de Figuras

Lista de Tabelas

Lista de Trechos de Código

Capítulo 1

Introdução

O software tornou-se parte intrínseca dos negócios nas últimas décadas. Nos dias atuais é difícil encontrar alguma área de estudo ou trabalho que não necessite da ajuda de computadores. Embora o avanço tecnológico seja grande, os gastos ocasionados devido à inadequada infraestrutura para testes chegam a 59.5 bilhões de dólares por ano, segundo a pesquisa [55] feita pelo NIST (*National Institute of Standards and Technology*) [46].

Teste de software é uma atividade pertencente à engenharia de software, e é usada para validar e aumentar a qualidade de sistemas. Qualquer sistema traz consigo erros, alguns são detectados e removidos ainda em fase de desenvolvimento, outros são solucionados em uma fase posterior dedicada a testes. Contudo, todo software manufaturado possui erros remanescentes que deverão ser resolvidos no futuro [36].

Um erro pode ocasionar uma falha, que é por definição, quando o software não faz aquilo que foi definido em seus requisitos. O propósito de realizar testes em software é descobrir defeitos que ocorreriam em situação de uso, e não provar a correção do sistema. Esta, não pode ser demonstrada através de testes [17], devido ao amplo domínio das possíveis entradas e o grande número de caminhos através do programa.

Como não é possível provar a correção através de testes (especialmente para grandes sistemas), na prática, estes são limitados a uma série de experimentos para mostrar o funcionamento correto do sistema em certas situações. Cada escolha de uma entrada de teste ao sistema é chamada de dado de teste (*test data*).

Pesquisas sobre testes são importantes para aumentar a sua qualidade e eficiência, e diminuir o

alto custo causado por defeitos. Para ilustrar o custos envolvidos, os autores Kanglin e Mengqi [36] citam a falha ocorrida no satélite militar que fora lançado do Cabo Canaveral em abril de 1999, estimado em 1.2 bilhões de dólares. Segundo eles, esta provavelmente tenha sido a falha mais cara de software (desconsiderando casos onde estão envolvidas vidas humanas). Em outubro do mesmo ano, ocorreu a perda da *NASA Mars Climate Orbiter* no espaço em razão da incompatibilidade entre dois componentes de softwares, um trabalhava com sistema inglês de medidas e o outro com sistema métrico. O relatório apresentado pela NASA (*National Aeronautics and Space Administration*) [44] indica que os testes aplicados haviam sido inadequados.

Realizar testes é um pré-requisito para o sucesso na implementação de qualquer sistema, mas com as tecnologias disponíveis, essa atividade é freqüentemente vista como tediosa, complicada, demorada e inadequada [36]. Na tentativa de diminuir custos e aumentar a eficiência dos testes, a maioria dos profissionais envolvidos no processo de testes de software acredita que a automação destes não é apenas desejável, mas é um fato necessário, devido à grande demanda do mercado [42].

No cenário atual de desenvolvimento de software, muito tem se discutido sobre *Web Services* e orientação a serviços. O desenvolvimento orientado a serviços é uma arquitetura emergente de grande importância, que demanda um alto número de pesquisas acadêmicas e de empresas interessadas em prover e se beneficiar de suas características de integração entre sistemas [29]. O desenvolvimento orientado a serviços está fortemente baseado no uso de *Web services*, embora seja possível o uso de outra tecnologia.

Os *Web services* representam uma tecnologia para a implementação de aplicações baseadas em componentes distribuídos. Eles são softwares executados, em sua maioria, na Internet, provendo serviços individuais ou agregando fluxos de negócios. Eles podem ser considerados como a última tendência na implementação de sistemas baseados em componentes [25].

Realizar testes para *Web Services* não é uma atividade simples e difere-se em diversos aspectos dos testes realizados em aplicações tradicionais. Segundo Tsai [66] testes para *Web services* envolve entre outras as seguintes dificuldades:

- Arquitetura de baixo acoplamento exige alta qualidade;
- Comportamento de tempo de execução: descobrimento dinâmico e *binding* com múltiplos elementos, *middlewares* e outros *Web services*;
- Invocação por partes desconhecidas;

- Processamento concorrente e acesso compartilhado a objetos;
- Requisitos de desempenho;
- Requisitos de segurança.

Em [15] , Davidson mostra que testar *Web Services* oferece um desafio único que só pode ser vencido com o uso de ferramentas automáticas de testes. Muitas empresas têm utilizado ferramentas na tentativa de automatizar seus testes. Estas ferramentas gravam macros que são guardadas para futura execução simulando a interação do usuário com o sistema. Esta abordagem não funciona com *Web Services*, pois estes não têm interface com o usuário e utilizam protocolos padrão para a troca de informações. Isto causa igual dificuldade no cenário onde os testes são feitos manualmente [40].

Algumas pesquisas investigam a geração automática de testes para *Web Services*, outras se dedicam à verificação de modelo para eles.

Em [29], Huang descreve a verificação de um modelo para *Web Services* chamado OWL-S (*Web Service Ontology Language*). Em [66] e [64], Tsai propõem um framework para testes, sendo que em [66] é feita uma proposta para incluir novas informações na interface atual dos serviços procurando facilitar a geração de testes, e em [64], embora sem oferecer detalhes, é descrito um processo para geração automática de casos de testes. Os autores Offutt e Xu [49] apresentam uma abordagem baseada na modificação de mensagens capturadas para a geração de casos de testes.

Bloomberg [7] faz um panorama do estado atual de testes para *Web Services*, e quais são as perspectivas para o futuro. Devido à grande perspectiva de evolução da tecnologia, o autor cita que a geração automática de casos de testes será uma necessidade, e deve ser baseada na interface do serviço.

1.1 Objetivos da Pesquisa

Esta pesquisa teve como objetivo o desenvolvimento de uma ferramenta que possibilitasse a geração automática de casos de testes para *Web Services* baseado na especificação da interface (WSDL - *Web Services Description Language*) e em mensagens previamente capturadas. Testes que se baseiam na interface de um componente são categorizados como testes "caixa preta" [25].

Esse tipo de teste foi escolhido devido ao fato de os consumidores de *Web Services* possuírem apenas a referência ao WSDL (que é freqüentemente publicada em um repositório UDDI - *Universal*

Description, Discovery and Integration).

A presente pesquisa teve os seguintes objetivos:

- Apresentar técnicas que podem ser utilizadas na geração de testes automatizados;
- Propor um método para geração automática de dados de testes para *Web Services*;
- Implementar uma ferramenta capaz de aplicar este método;
- Realizar um estudo de caso com *Web Services* de uso comercial aplicando o método proposto, a fim de verificar sua eficácia.

1.2 Organização do trabalho

Abaixo segue uma breve descrição de cada capítulo deste trabalho:

- Capítulo 2: são apresentados os principais conceitos e tecnologias envolvendo *Web Services*;
- Capítulo 3: fornece uma descrição sobre testes, os possíveis tipos de testes e quais serão aplicados neste trabalho;
- Capítulo 4: descreve pesquisas envolvendo testes para *Web Services*. Apresenta também um estudo sobre algumas ferramentas para testes de *Web Services* e um exemplo da geração manual de teste.
- Capítulo 5: contém as modificações e melhorias das abordagens para a geração automática de casos de teste para *Web Services* introduzidas neste trabalho.
- Capítulo 6: apresenta a ferramenta GenAutoWS para testes de *Web Services* que implementa as melhorias apresentadas no Capítulo 5.
- Capítulo 7: apresenta os estudos de casos realizados com a ferramenta e os resultados obtidos.
- Capítulo 8: conclusão e trabalhos futuros.

Capítulo 2

Web Services

Existem atualmente várias definições para *Web Services*, por exemplo, o *World Wide Web Consortium* (W3C) - organismo que regulamenta os padrões de *Web Services* - o define como: "um sistema de software projetado para dar suporte à interoperabilidade entre máquinas sobre a rede. Possui uma interface descrita em um formato processado por computadores (especificamente WSDL). Sistemas interagem com *Web Services* de maneira estipulada pela sua descrição usando mensagens SOAP (*Simple Object Access Protocol*), tipicamente enviando-as sobre HTTP (*HyperText Transfer Protocol*) com uma serialização baseada em XML (*Extensible Markup Language*) em conjunto com outros padrões *Web* relacionados [79]".

Uma outra definição mais simples (e talvez mais útil) é dada por [33]: "*Web Services* é uma tecnologia emergente para a exposição de serviços através da Internet, possibilitando a interação entre aplicações dinamicamente. As funcionalidades que podem ser realizadas por *Web Services* são desde simples trocas de informações a complicados processos de negócios".

Empresas vendedoras de serviços podem encapsular seus processos de negócios em aplicações *Web Services* e expor a seus clientes através da Internet [33]. Web Services são facilmente aplicados como uma tecnologia de embrulho [1] a sistemas já existentes, possibilitando que estes sejam recompostos como novas oportunidades de negócio [24].

Web Services representam uma oportunidade para que empresas possam expandir suas ofertas de negócios, aumentando a eficiência e melhorando o relacionamento com os consumidores dos serviços. Incluindo oferta de seus serviços a parceiros, a empresa e os próprios parceiros expandem suas poten-

[1] *wrapping*

cialidades e sua base de negócio [57].

Web Services ganharam popularidade devido aos benefícios que sua utilização proporciona. Abaixo estão diversos deles, descritos em [57]:

1. **Interoperabilidade em ambientes heterogêneos**: Provavelmente, o principal benefício do modelo de *Web Services*, permitindo que diferentes serviços executem e interajam em uma variedade de plataformas de software e que sejam escritos em linguagens diferentes.

2. **Serviços de negócios através da Internet**: Empresas podem usar o poder e vantagens da *World Wide Web* para suas operações. Exemplo: uma empresa pode disponibilizar o seu catálogo de produtos e inventário através de *Web Services* para melhorar a gerência de seus fornecimentos e vendas.

3. **Integração com sistemas existentes**: A maioria das empresas tem grandes quantidades de informações guardadas em sistemas existentes, acarretando em grandes custos com integração de sistemas legados. *Web Services* permitem que desenvolvedores reusem e revitalizem as informações existentes.

4. **Liberdade de escolha**: Os padrões *Web Services* têm aberto um grande mercado de ferramentas, produtos e tecnologias. Isto aumenta a possibilidade de escolha de um software realmente adequado às necessidades das empresas.

5. **Produtividade de programação**: Como *Web Services* são baseados inteiramente em padrões, isto evita que desenvolvedores tenham que lidar com diferentes conjuntos de tecnologias a cada nova integração com outros sistemas; também permite que uma única implementação seja integrada com sistemas em diferentes plataformas e linguagens de programação.

2.1 Arquitetura Orientada a Serviço e Web Services

SOA (*Service Oriented Architecture*) é uma arquitetura cujo objetivo é desenvolver a interação entre softwares com baixo acoplamento [28]. Em SOA, o desafio do baixo acoplamento é abordado através da noção de serviços [1], que são comumente implementados com *Web Services*. A interação pode envolver a troca de simples mensagens de dados ou a coordenação de alguma atividade [5].

SOA é, essencialmente, uma coleção de serviços que se comunicam entre si [5], e que são capazes de interagir de três formas, comumente referidas como: publicar, procurar e invocar [1]. As interações

são realizadas pelos três papéis existentes na arquitetura SOA, que são [14]:

- **Provedor de serviço**: pessoa ou organização que fornece um agente apropriado implementando um determinado serviço.
- **Consumidor de serviço**: pessoa ou organização que deseja fazer uso de um determinado serviço.
- **Serviço de registro**: armazena informações sobre os serviços, semelhante a um catálogo. Ele é a localização central onde o provedor pode relacionar seus serviços, e no qual o consumidor pode realizar pesquisas.

O fluxo é formado com a entidade provedora colocando à disposição seus serviços para que algum consumidor possa localizá-los e, posteriormente, utilizá-los. A Figura 2.1 ilustra o fluxo de SOA e algumas tecnologias envolvidas, que serão detalhadas posteriormente.

Figura 2.1: Fluxo de SOA e as entidades participantes

A Figura 2.2 demonstra em nível conceitual as diferenças entre uma aplicação orientada a serviço (aplicação composta) e uma outra aplicação.

Web Services são considerados atualmente a tecnologia mais apropriada para a implementação de SOA [1]. Eles forão construídos sobre o conhecimento obtido de ambientes maduros de computação distribuída, como CORBA (*Common Object Request Broker Architecture*) e RMI (*Java Remote Method Invocation*), permitindo a comunicação entre aplicações e interoperabilidade. Eles proporcionam independência de linguagem, neutralidade de ambiente e um modelo de programação que permite o rápido desenvolvimento para integração de aplicações [57].

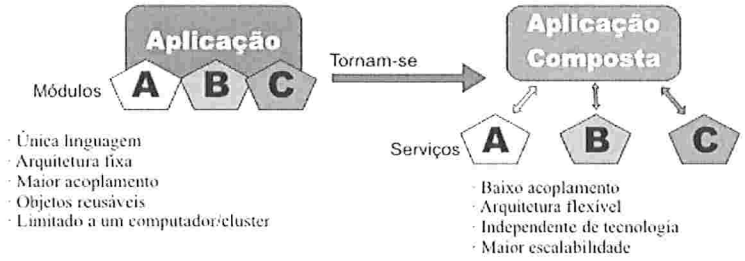

Figura 2.2: Comparativo entre aplicações

A arquitetura de *Web Services* é formada por uma camada de mensagens, uma linguagem para descrever a interface do serviço e protocolos bem difundidos para a camada de transporte e de rede. A Figura 2.3 apresenta, de uma maneira simples, esta arquitetura.

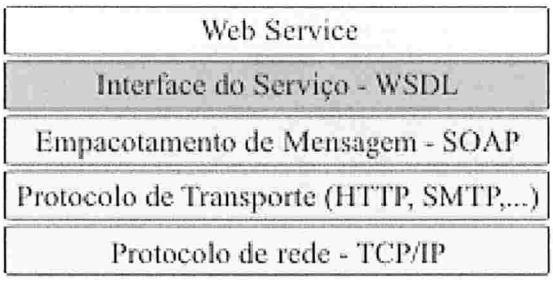

Figura 2.3: Arquitetura Web Service

A grande maioria dos *Web Services* utilizam como camada de mensagem o protocolo SOAP. Sendo este um protocolo de comunicação (cujas mensagens são no formato XML - o que facilita a publicação), busca e invocação de operações [24]. Esta camada é responsável por estruturar e transcrever as informações (como dados de entrada e saída dos *Web Services*) no formato em que será transmitido, assim como fazer uso do protocolo de transporte para enviar e receber as informações.

Os *Web Services* são representados usando uma maneira padronizada chamada *service description*. Este provê todos os detalhes necessários para a interação com o serviço, incluindo formato das mensagens, protocolo de transporte a ser utilizado e localização. *Service Description* é expresso utilizando *Web Services Description Language* (WSDL) [24].

Nas seções que se seguem serão descritas as tecnologias envolvidas na arquitetura de *Web Services* e o repositório de serviço, com exceção dos protocolos de transporte e de rede, que são bem difundidos em todos os tipos de aplicações.

2.2 SOAP

SOAP é a especificação de um protocolo leve que tem como objetivo a troca de informações estruturadas de forma descentralizada e em ambiente distribuído [78]. Originalmente projetado pela Microsoft, mas atualmente mantido pelo grupo *XML Protocol Working Group* [71] do W3C, SOAP era o acrônimo para *Simple Object Access Protocol*, mas a partir da especificação 1.2 [78], a atual, o acrônimo não faz mais parte da nomeação da especificação.

Os requisitos da especificação do SOAP estão definidos em [76] e são: simplicidade e extensibilidade.

O protocolo SOAP é independente de protocolo de transporte, mas usualmente HTTP(S) sobre TCP/IP é utilizado para o transporte das mensagens SOAP [58].

SOAP é fortemente baseado em padrões XML, que são utilizados na definição das mensagens e na funcionalidade do protocolo, entre os padrões podemos citar: *XML Schema* e *XML Namespaces* [58].

Como XML é independente de aplicação, sistema operacional ou linguagem de programação, as mensagens SOAP podem ser utilizadas em todos os ambientes. A idéia fundamental é que duas aplicações, independentes do sistema operacional no qual estão sendo executadas, da linguagem de programação na qual foram programadas ou de qualquer outro detalhe técnico de implementação, possam trocar informações usando mensagens simples codificadas em um formato compartilhado por ambas as aplicações [58].

2.2.1 Estrutura da mensagem SOAP

O elemento raiz da mensagem XML do SOAP é chamado de `Envelope`. O Envelope é formado de cabeçalhos (opcional) e corpo da mensagem (obrigatório). Os dados da aplicação são carregados no corpo da mensagem e dados adicionais do protocolo são levados nos cabeçalhos [9]. A Figura 2.4 exibe a estrutura da mensagem SOAP.

O Envelope é uma estrutura `container` para a mensagem SOAP e associada com o *namespace*: `http://www.w3.org/2002/06/soap-envelope`. A Listagem 2.1 apresenta um exemplo de mensagem

Figura 2.4: Estrutura da mensagem SOAP

SOAP de requisição de uma operação de soma.

Listagem 2.1: Mensagem de requisição SOAP

```
<soapenv:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Header>
      <tns:transaction-id>
          UGFyYWLpbnMsIFZvY+ogZXN04SByZWFsbWVudGUgbGVuZG8gbyB0
          cmFiYWxoby4gU2Ugdm9j6iBhY2hvdSBlc3NhIG1lbnNhZ2VtIGVu
          dmllIHVtIGVtYWlsIHBhcmEgcHNpbHZlaXJhcEBnbWFpbC5jb20=
      </tns:transaction-id>
   </soapenv:Header>
   <soapenv:Body>
     <somaDoisInteiros xmlns="http://www.ime.usp.br/~silveira/ws">
       <parcela1>98765</parcela1>
       <parcela2>12345</parcela2>
     </somaDoisInteiros>
   </soapenv:Body>
 </soapenv:Envelope>
```

O conteúdo do corpo da mensagem (trecho contido dentro do elemento `Body`) é um simples

repositório para os dados da aplicação. No exemplo da Listagem 2.1, os dados da aplicação são as parcelas de uma soma. A especificação SOAP não define nenhuma estrutura particular nem faz a interpretação destes elementos [9].

O cabeçalho da mensagem é definido pelo elemento `Header`. Ele provê um mecanismo para estender as funcionalidades de SOAP, inserindo informações como: autenticação, segurança, controle de transação e rota da mensagem [9]. Cada informação é colocada em um bloco de cabeçalho e possui seu formato de acordo com definições da própria funcionalidade, mas é possível incluir dois atributos ao elemento delimitador, indicando funcionalidades pré-determinadas, são elas:

- `mustUnderstand`: Atributo do tipo booleano, indicando se o receptor da mensagem pode ignorar este cabeçalho ou se seu entendimento é obrigatório.
- `Role`: Permite especificar para quem é direcionada a mensagem, pode ser utilizada quando a mensagem trafega entre diversos serviços até seu destino final.

Na Listagem 2.1 é mostrado um exemplo de cabeçalho para controle de transação.

2.2.2 Estilo de codificação SOAP

XML é uma linguagem expressiva e não restringe a forma de um documento. É possível definir diferentes documentos semanticamente equivalentes. A Listagem 2.2 mostra uma forma alternativa para o corpo da mensagem da Listagem 2.1:

Listagem 2.2: Mensagem XML equivalente ao código 2.1

```
<somaDoisInteiros xmlns="http://www.ime.usp.br/~silveira/ws"
parcela1="98765" parcela2="12345"/>
```

Codificação refere-se à forma como os dados serão representados para serem transmitidos entre aplicações [67]. SOAP define mais de um método de codificação para converter os dados entre sistemas e XML.

Os dados codificados da mensagem SOAP são colocados no corpo da mensagem, dentro do elemento `body`, e são enviados para o outro `host`. Este irá decodificá-los e disponibilizá-los para o consumidor ou provedor do serviço [11]. Desta forma, justifica-se a importância de utilizar uma codificação comum entre o consumidor e o provedor do serviço.

As partes envolvidas na construção de um *Web Service* podem entrar em acordo sobre uma codificação prédefinida, ou utilizar *XML Schema* diretamente para definir os tipos dos dados.

Encoded

Um corpo de mensagem *encoded* sinaliza que as regras para a codificação foram pré-definidas em uma URL especificada pelo atributo `encodingStyle`, ou seja, é utilizada uma codificação predefinida, que informa como fazer o mapeamento dos tipos de dados.

O SOAP define um formato de codificação que especifica como *string*, tipos primitivos e matrizes são representados [67]. Essa representação é conhecida como *SOAP Data Model*. A Listagem 2.3 mostra um exemplo de mensagem encoded utilizando *SOAP Data Model*.

Listagem 2.3: Exemplo utilizando SOAP Data Model

```
<soapenv:Body>
   <ns1:somaDoisInteiros
   soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <ns1:arg0 xsi:type="soapenc:int">4</ns1:arg0>
      <ns1:arg1 xsi:type="soapenc:int">4</ns1:arg1>
   </ns1:somaDoisInteiros>
</soapenv:Body>
```

Literal

Uma mensagem chamada de literal indica que as regras para a codificação do corpo da mensagem foram definidas em um *XML Schema*. O seu uso possibilita o reaproveitamento de documentos *XML Schema* pré-existentes, assim como códigos de sistemas que tratam essas mensagens.

O corpo da mensagem SOAP mostrada na Listagem 2.1 é uma mensagem literal, definida pelo trecho do *XML Schema* da Listagem 2.4.

Listagem 2.4: XML Schema utilizado para validar a mensagem da Listagem 2.1

```
<xsd:element name="somaDoisInteiros">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="parcela1">
```

```
                <xsd:simpleType>
                    <xsd:restriction base="xsd:integer"/>
                </xsd:simpleType>
            </xsd:element>
            <xsd:element name="parcela2">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:integer"/>
                </xsd:simpleType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

2.2.3 Tipos de mensagens SOAP

O tipo da mensagem define a forma como as informações serão colocadas no corpo da mensagem e como fazer o mapeamento entre a mensagem e o código do serviço. Atualmente são utilizados dois tipos de mensagens: *SOAP RPC* e *SOAP Document*.[2]

SOAP RPC

SOAP RPC é uma forma de utilizar SOAP que provê uma convenção para empacotar e mapear os métodos de programas em mensagens, permitindo que estes sejam invocados remotamente.

As regras utilizadas para a construção das mensagens são descritas na especificação SOAP 1.1 [74] e SOAP 1.2 [77]. De acordo com essas regras, o corpo da mensagem pode conter apenas um elemento cujo nome é o mesmo da operação, e todos os parâmetros devem ser representados como subelementos.

É provável que SOAP RPC não será o paradigma dominante para SOAP a longo prazo, mas com ele é fácil de alcançar resultados rapidamente, devido ao abrangente suporte por parte das ferramentas comerciais [9].

A Figura 2.5 representa a interação entre o consumidor e o *Web Services* utilizando mensagem *SOAP RPC*. Nesse caso, cada método da aplicação é mapeado em uma operação do *Web Services*.

[2] em algumas literaturas *tipo de mensagem* é referido como *estilo de mensagem*

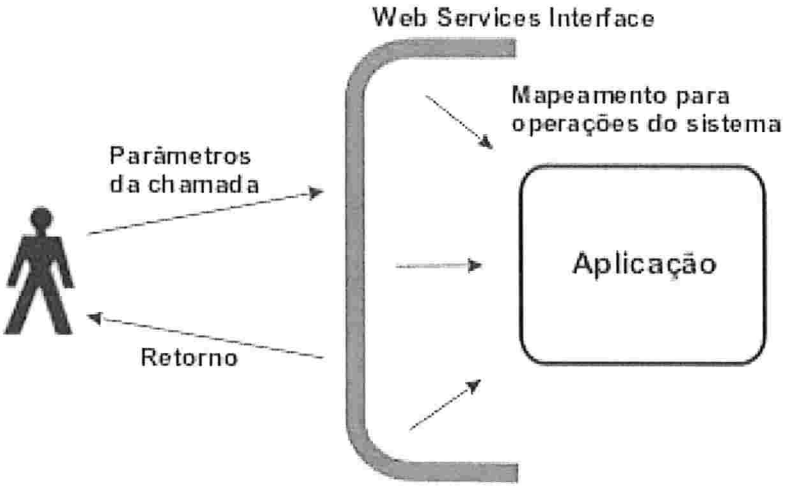

Figura 2.5: Ilustração da interação utilizando *SOAP RPC*

SOAP Document

O *SOAP Document* é uma outra forma de utilizar SOAP, e seu nome refere-se na forma em que os dados da aplicação são colocados no corpo da mensagem. Em um *Web Service* que utiliza mensagens *SOAP Document*, o corpo do documento XML contendo as informações da aplicação é inserido diretamente na seqüência do elemento `Body` da mensagem [9]. A Listagem 2.1, mostrada anteriormente, ilustra uma mensagem *SOAP Document*.

Em *SOAP Document*, a interação entre o consumidor e o serviço é feita utilizando o documento XML, o qual representa uma completa unidade de negócio. Deste modo, existe alta probabilidade de o documento ser completamente autodescritivo [67].

A Figura 2.6 mostra a interação entre o consumidor e o *Web Service* utilizando mensagens *SOAP Document*. O consumidor visualiza o serviço como uma completa unidade do negócio diferenciando assim das mensagens RPC (que representam um outro paradigma na realização do negócio).

A Figura 2.7 resume os tipos de mensagens SOAP, assim como os estilos de codificações.

2.2.4 Tendências de Estilos de Codificação e Mensagens

A combinação do uso ou não de mensagens codificadas (`encoded` ou `literal`) e os tipos de mensagens (`rpc` ou `document`) resulta em quatro opções de mensagens SOAP, contudo, segundo Chatter-

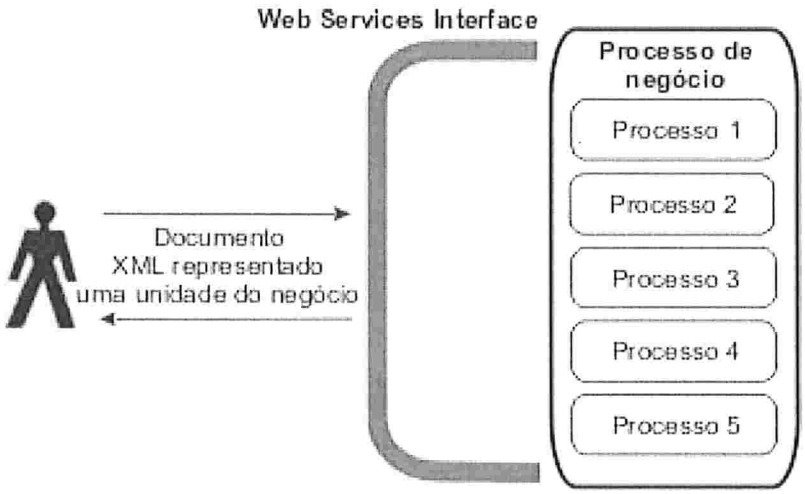

Figura 2.6: Ilustração da interação utilizando *SOAP Document*

	Document	RPC

Literal	`<soap:Body>`
	`  <carro xmlns="http://...>`
	`  <modelo...>`
	`  <ano...>`
	`  </carro>`
	`</soap:Body>`
		`<soap:Body>`
		`  <alugar>`
		`  <carro xmlns="http://...>`
		`  <modelo...>`
		`  <ano...>`
		`  </carro>`
		`  </alugar>`
		`</soap:Body>`

Encoded	`<soapenv:Body`
	`  soap:encodingStyle="http://schemas...>`
	`  <ns1:carro>`
	`  <ns1:modelo xsi:type="soapenc:string"...>`
	`  <ns1:ano xsi:type="soapenc:int"...>`
	`  </ns1:carro>`
	`</soap:Body>`
		`<soapenv:Body`
		`  soap:encodingStyle="http://schemas...>`
		`  <alugar>`
		`  <ns1:carro>`
		`  <ns1:modelo xsi:type="soapenc:string"...>`
		`  <ns1:ano xsi:type="soapenc:int"...>`
		`  </ns1:carro>`
		`  </alugar>`
		`</soap:Body>`

Figura 2.7: Comparação entre os tipos de mensagem e os estilos de codificações

jee [9] a tendência de uso é apenas de mensagens SOAP `encoded-rpc` e SOAP `document-literal`. Ferramentas para a construção de *Web Services* geralmente possuem suporte apenas a estas duas opções de mensagens.

Ainda segundo Chatterjee [9], `document-literal` é a preferida nas trocas de mensagens SOAP,

pois é apenas um empacotamento de documentos XML de nível de aplicação dentro do corpo da mensagem SOAP.

2.2.5 Mensagem de falha SOAP

Uma mensagem SOAP pode conter um elemento chamado `Fault`, a existência desse elemento em uma mensagem possui a função de indicar que alguma falha ocorreu [78]. Este elemento aparece como sendo um elemento filho do elemento `body` e pode aparecer apenas uma vez em uma mensagem. O elemento `Fault` contém os seguintes sub elementos:

- code: Um código identificando a falha.
- reason: Um texto explicativo para humanos sobre a falha.
- detail: Elemento opcional. Possui detalhes do erro específicos a aplicação, exemplo: linha de código onde o erro ocorreu.
- node: Elemento opcional que contém informações sobre o elemento da mensagem origem que causou a falha.
- role: Elemento opcional que contém uma *uri* para identificar quem estava executando quando a falha ocorreu.

2.3 WSDL

Web Services Description Language (WSDL) provê o mecanismo através do qual as definições dos serviços são expostas, contendo informações que devem ser seguidas durante a troca de mensagens [45]. WSDL é descrito em um arquivo XML, utilizando padrões *schema* e *namespace* [75]. Em resumo, o documento WSDL responde às perguntas: O que o Web Service faz? Onde ele reside? Como invocá-lo?

WSDL descreve as estruturas e os tipos de dados utilizados pelo serviço, explica como mapeá-los nas mensagens e inclui informações para amarrar estas mensagens à implementação [45].

Ambas as partes de uma comunicação *Web Service* compartilham o arquivo WSDL. O consumidor cria a mensagem no formato esperado e utiliza o protocolo indicado, enquanto o provedor utiliza-o para entender, destrinchar a mensagem e mapear para o programa apropriado [45].

WSDL pode ser usado para especificar interfaces de Web Services sobre diversos protocolos, incluindo HTTP GET/POST, MIME, mas o protocolo mais difundido é o SOAP [9].

Como o restante do framework de *Web Services*, WSDL é projetado para usar ambos os tipos de interação: procedural (RPC) e baseada em documentos. Como as demais tecnologias XML, WSDL é bastante extensível e possui uma gama muito grande de opções [45].

Em uma visão lógica, o documento WSDL é dividido em duas partes: uma abstrata (que descreve as operações que o serviço suporta e os tipos de mensagens destas operações); e a outra concreta (que descreve como estas operações são amarradas a uma rede física, e como as mensagens são empacotadas para serem transferidas dentro do referido protocolo) [9].

Nos parágrafos que se seguem serão explicados os principais elementos da interface WSDL. Estes elementos são: `type, part, message, operation, portType, binding e service` [8, 9, 75].

O fundamento da interface WSDL é o conjunto de mensagens que o serviço espera receber e enviar. O elemento `message` explica cada uma destas mensagens e contém zero ou mais elementos `part` (que se refere aos parâmetros e aos valores de retorno da mensagem).

O elemento `part` contém informações sobre os tipos de dados, estes são descritos dentro do elemento types. O WSDL não está amarrado exclusivamente a um sistema de definição de tipos de dados, mas faz uso do *W3C XML Schema* como escolha padrão. A Listagem 2.5 ilustra um exemplo de definição dos elementos `message, part e type` para o exemplo da soma de dois inteiros apresentado no tópico sobre SOAP:

Listagem 2.5: Definição das mensagens no documento WSDL.

```
<message name=''RequisicaoDeSoma''>
  <part name=''parcela1'' type=''xsd:int''/>
  <part name=''parcela2'' type=''xsd:int''/>
</message>
<message name=''ResultadoDeSoma''>
  <part name=''resultado'' type=''xsd:int''/>
</message>
```

As mensagens são agrupadas no elemento WSDL chamado `operation` que possui uma semântica semelhante às assinaturas de funções em uma linguagem de programação imperativa. Cada mensagem que forma a operação é sinalizada como sendo de entrada, saída ou exceção.

O elemento `portType` contém uma coleção de operações que são consideradas para formar os *Web Services*. Contudo, neste elemento as operações ainda são descritas em termos abstratos, simplesmente agrupando um conjunto de operações. Na Listagem 2.6 podemos observar a definição de uma operação e um `portType`.

Listagem 2.6: Definição do conjunto de operações.

```
<portType name=''SomaPortType''>
   <operation name=''somaDoisInteiros''>
      <input message=''tns:RequisicaoDeSoma''/>
      <output message=''tns:ResultadoDeSoma''/>
   </operation>
</portType>
```

A seção `binding` da interface WSDL descreve como mapear cada uma das operações em um protocolo físico que irá carregar as mensagens no momento da comunicação. Cada subelemento `operation` do elemento `portType` será mapeado a um protocolo específico. A Listagem 2.7 traz a definição do elemento `binding`. Observe que o protocolo de transporte é SOAP sobre HTTP. A mensagem será do tipo RPC, como definido no atributo `style` e os atributos serão codificados, conforme indicação do atributo `use`.

Listagem 2.7: Definição do elemento `binding`.

```
<binding name=''SomaBinding'' type=''tns:SomaPortType''>
  <soap:binding style=''rpc''
      transport=''http://schemas.xmlsoap.org/soap/http''/>
   <operation name=''somaDoisInteiros''>
      <soap:operation soapAction=''somaDoisInteiros''/>
      <input>
         <soap:body
            encodingStyle=''http://schemas.xmlsoap.org/soap/encoding/''
            namespace=''http://DefaultNamespace''
            use=''encoded''/>
      </input>
      <output>
         <soap:body
```

```
            encodingStyle=''http://schemas.xmlsoap.org/soap/encoding/''
            namespace=''http://DefaultNamespace''
            use=''encoded''/>
      </output>
   </operation>
</binding>
```

O último elemento é chamado de `service` e contém a definição do endereço físico para a chamada do serviço que está sendo especificado. O mais comum é incluir a URL do serviço. A Listagem 2.8 mostra o elemento `service`.

Listagem 2.8: Elemento `service` do WSDL

```
<service name=''Soma_Service''>
   <documentation>Arquivo WSDL para Soma Dois Inteiros</documentation>
   <port binding=''tns:SomaBinding'' name=''SomaPort''>
      <soap:address
         location=''http://www.ime.usp.br/~silveira/ws''/>
   </port>
</service>
```

O diagrama de classes da Figura 2.8 demonstra a estrutura do documento WSDL, assim como a relação entre os seus elementos.

Existem diversas ferramentas no mercado que a partir da definição da interface em uma linguagem de programação, como: Java, C++ ou C#, por exemplo, gera o documento WSDL de forma automática, não sendo requerido ao desenvolvedor do serviço o conhecimento de detalhes dos elementos dentro do WSDL.

2.3.1 Stub

O código de software que escreve e lê as informações do protocolo de transporte é sempre gerado e interpretado automaticamente, baseado nas informações da interface do documento WSDL. No momento da invocação do serviço, o programa consumidor delega a uma parte do software chamado *stub*, este irá escrever a requisição ao provedor do serviço e codificar a mensagem no formato apropriado. De forma análoga o provedor do serviço Web também possui o seu *stub* que faz a leitura e a

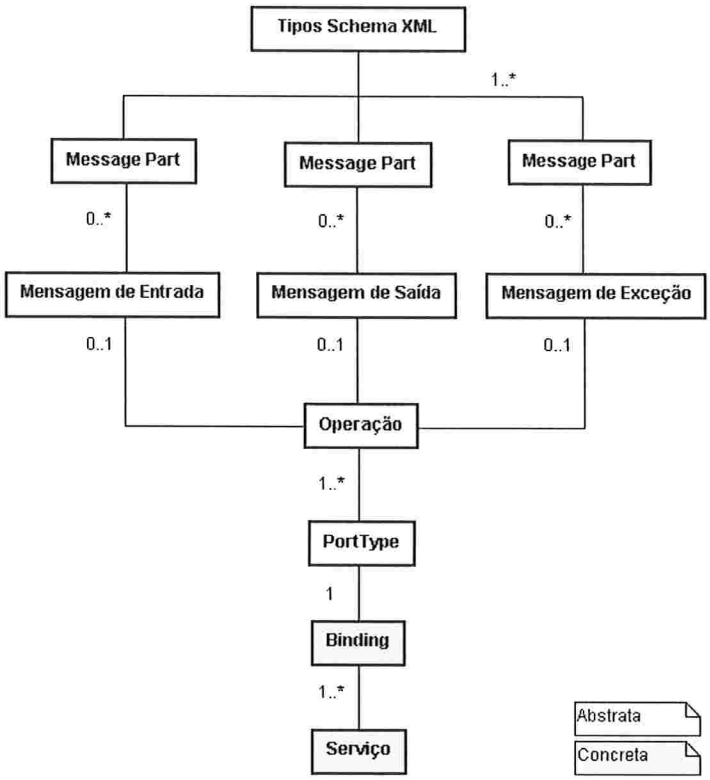

Figura 2.8: Estrutura do documento WSDL.

interpretação dos dados trafegados, assim como a codificação da mensagem de resposta [9].

O diagrama de seqüência da Figura 2.9 demonstra o fluxo de chamada de serviço. O *stub* do provedor de serviço em algumas literaturas é chamado de *skeleton*.

2.4 UDDI

UDDI (*Universal Description, Discovery, and Integration*) [4] é uma especificação técnica mantida pelo grupo de padrões OASIS (*Organization for the Advancement of Structured Information Standards*), para a criação de um serviço de registro que cataloga e organiza *Web Services*. A implementação desta especificação é chamada de UDDI *registry*. Um UDDI *registry* contém um banco de dados que dá suporte um conjunto de estruturas padrões definidas na especificação do UDDI.

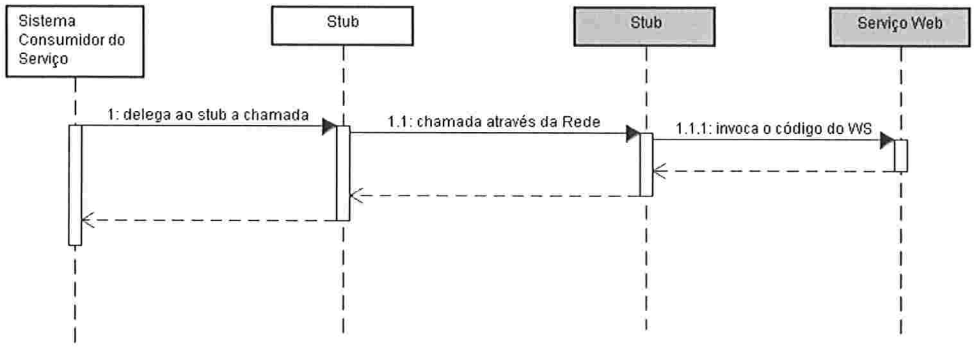

Figura 2.9: Fluxo de chamada de serviço.

UDDI foi especificado com o objetivo de permitir a descrição, descoberta e integração de *Web Services* [8]. A funcionalidade do servidor de registro é de gerenciar meta informações sobre os *Web Services*, permitindo que sejam efetuadas inclusões e buscas de informações com o objetivo de integrar soluções de serviços. O UDDI é considerado pelo OASIS como um elemento central na arquitetura de *Web Services* [47].

O funcionamento do servidor de registro UDDI é semelhante ao de uma lista telefônica, os provedores de serviço adicionam e removem informações sobre o serviço, os consumidores realizam consultas e obtém o WSDL para poder utilizá-lo [9]. O serviço UDDI inclui páginas brancas (*white pages*) que disponibilizam informações sobre a empresa provedora do serviço; páginas amarelas (*yellow pages*) que são classificadas por categorias de negócios; e páginas verdes (*green pages*) que provêem detalhes técnicos sobre os serviços.

A ferramenta implementada neste trabalho oferece suporte ao servidor de registros UDDI. Este suporte permite consultar serviços utilizando palavras chaves. O serviço selecionado pode ser incluído na ferramenta possibilitando a criação de casos de testes automaticamente.

2.5 Transferência de Estado Representacional

Transferência de Estado Representacional, REST, sigla em inglês para *Representational State Transfer*), é uma arquitetura para a construção de sistemas hipermídia. O maior exemplo desse tipo de sistema é a *World Wide Web*. Esta arquitetura foi proposta por Roy Fielding [20], que definiu em sua tese de doutorado os princípios dessa arquitetura, que são mostrados abaixo:

- Todas as informações são vistas como recurso e identificadas de maneira única;
- Ausência de estado da aplicação, ou seja, a mensagem trocada contém toda a informação para ser entendida;
- Conjunto fixo de operações que se aplica aos recursos;
- A utilização de hipermídia, possibilitando a navegação de um recurso a outro.

Web Services que seguem estes princípios são chamados RESTful web services [54]. Um serviço RESTful utiliza o protocolo http e suas primitivas GET, POST, PUT e DELETE como conjunto de operações, e os recursos são identificados através de uma URI. Alguns exemplos de serviços RESTful são:

- Amazon Simple Storage Service (S3) (http://aws.amazon.com/s3)
- Serviços Ebay (http://developer.ebay.com/rest/)
- Yahoo web services (http://developer.yahoo.com)

Os recursos de um serviço RESTful que representa um cadastro de clientes poderia ser identificado com as URLs e utilizando as operações mostradas na Tabela 2.1.

Tabela 2.1: Exemplos de recursos RESTful para um cadastro de clientes

Recurso	Operação	Descrição
http://hostname/cadastro/clientes	GET	obtém a lista de clientes
http://hostname/cadastro/cliente/{codigo}	GET	informações de um cliente
http://hostname/cadastro/cliente/{codigo}	DELETE	exclui um cliente
http://hostname/cadastro/cliente/{codigo}/telefone/{novo numero}	POST	atualiza o número de telefone

Em serviços REST a reposta de um serviço é tipicamente XML ou JSON (JavaScript Object Notation). Um serviço REST pode ser especificado com o uso de WADL (Web Application Description Language) [39]. Serviços REST vêm se tornando mais populares com a adoção de AJAX (Asynchronous JavaScript and XML).

2.6 Exemplos de Web Services

Foi realizada uma pesquisa na Internet para coletar alguns exemplos de Web Services que são utilizados no mercado brasileiro. Empresas clientes (consumidores) os utilizam, mas não possuem

o programa fonte para efetuar testes, deste modo devem efetuar testes caixa preta. Abaixo são apresentados alguns exemplos:

- Receita federal disponibiliza para os bancos um serviço de consulta de CPF/CNPJ cuja interface é Web Services.
- O Banco Central fornece aos bancos informações de crédito de pessoas física e jurídica através do sistema SCR.
- O ministério da saúde oferece a farmácias credenciadas a possibilidade de reembolso do valor do medicamento (farmácia popular), a interface é Web Services.
- A empresa Check Express vende consulta e proteção ao crédito com interface *Web Services*.

2.6.1 Uma Aplicação Exemplo

Com o objetivo de ilustrar o uso de Web Services, esta seção mostrará uma aplicação exemplo utilizando *Web Services*. O sistema será denominado *Auto-Aluguel* e tem o propósito de agregar ofertas de locação de diversas locadoras de veículos diferentes. Em uma única aplicação o cliente que está procurando um automóvel para alugar consulta o preço e as condições de diversas locadoras de veículos, permitindo a escolha daquela que, segundo seus critérios, oferece melhores benefícios. A aplicação oferece também informações meteorológicas e serviços que agreguem valor ao sistema que podem ser implementados futuramente.

O sistema possui interface *Web*, em que o cliente poderá consultar e comparar valores. Caso ele deseje, após um cadastro, será possível realizar reservas de locação. Internamente, o sistema agrega serviços de aplicações de outros fornecedores (as locadoras de carro e as empresas que fornecem informações meteorológicas). Os dados de clientes e os registros de suas transações são armazenados em uma base de dados. A cada reserva o usuário receberá um e-mail de confirmação contendo os detalhes da mesma.

Web Services são utilizados entre o servidor *Web* e o servidor de aplicação, as operações de: consulta, reserva, login e verificação meteorológica são serviços. A integração entre o *Auto-Aluguel* e as locadoras de carros, será efetuada por serviços disponibilizados por cada locadora de veículos pertencente à rede de parceiros. Para efeitos ilustrativos uma locadora de carro utilizará um *Web Service* para encapsular um sistema legado previamente existente, situação muito comum entre empresas. O

sistema responsável pela gestão de e-mail também é um sistema existente e possui uma nova interface *Web Service*.

No total, o sistema faz uso de doze *Web Services*, sendo quatro serviços baseados em mensagens `Document` e oito `RPC`.

O sistema *Auto-Aluguel* possui características e problemas comuns às aplicações que utilizam *Web Services*. O reuso de sistemas legados (desenvolvidos com outra tecnologia), a troca de informações com empresas parceiras e a necessidade de interoperabilidade entre diferentes soluções são alguns deles.

A Figura 2.10 apresenta um diagrama lógico da aplicação *Auto-Aluguel*.

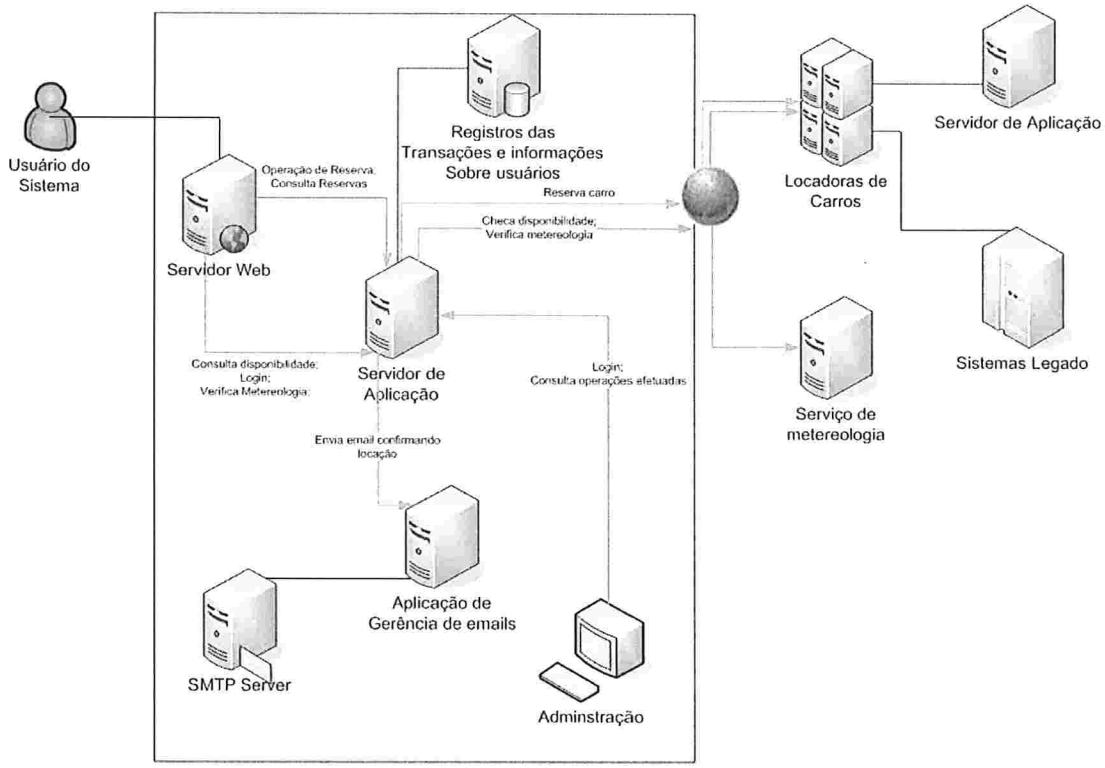

Figura 2.10: Diagrama da aplicação *Auto-Aluguel*.

2.7 Conclusão

O capítulo mostrou os benefícios que a utilização de *Web Services* pode trazer, justificando assim porque eles se tornaram populares nos últimos anos. Foram apresentados conceitos e tecnologias envolvidas com *Web Services*, tais como: SOA, WSDL, SOAP e UDDI.

No final deste capítulo encontra-se ainda uma aplicação de exemplo para ilustrar o uso de *Web Services*.

Capítulo 3

Teste de software

Atualmente, a sociedade vem se tornando cada vez mais dependente de softwares e máquinas. Sistemas são construídos e utilizados em operações críticas. Dentro da engenharia de software uma das atividades mais importante é a de promover a qualidade do sistema que está sendo construído, e essa atividade envolve a aplicação de testes. Neste capítulo serão apresentados os conceitos envolvendo testes de software e as principais técnicas utilizadas.

3.1 Verificação e Validação

Teste de software é um dos elementos do amplo campo que é freqüentemente chamado de Verificação e Validação (V&V). A *Verificação* é formada por uma série de atividades que procuram assegurar a correta implementação no software das funções especificadas, ou seja, que o produto construído está correto. A *Validação* tem o objetivo de assegurar que o software em desenvolvimento atende aos requisitos do usuário, ou seja, que o produto é aquele que o usuário espera [52].

As atividades de verificação e validação incluem: revisão técnica formal, audição da configuração, monitoração de desempenho, simulação, estudo de viabilidade, revisão da documentação, revisão da base de dados, análise de algoritmo, teste de desenvolvimento, teste de usabilidade, teste de qualificação e teste de instalação. Testes possuem um papel extremamente importante dentro da V&V [52].

3.2 Objetivos de Testes

Teste de software é definido em [23] como o processo de executar um programa com o objetivo de encontrar defeitos. Assim, é considerado um teste de sucesso aquele capaz de demonstrá-los. Dentro

das etapas de construção de software a probabilidade da ocorrência de falhas humana é enorme [52].

Existe uma considerável ambigüidade na definição de termos centrais, como falha (*failure*), defeito (*fault*) e erro. O termo defeito é usualmente utilizado para nomear um problema no nível mais baixo de abstração, como por exemplo: uma célula de memória que contém um bit grudado em zero (*stuck-at-zero*) ou um algoritmo desenvolvido incorretamente [26].

Um defeito talvez cause um erro, o qual é um estado do sistema. Erros são associados ao universo da informação, portanto o bit grudado em zero pode provocar uma interpretação errada da informação de uma estrutura de dados, ou passar desapercebido se essa região de memória não for utilizada, o que não caracterizaria um erro.

Um erro pode ter o efeito de uma falha, significando que o sistema desviou-se da especificação correta. A interpretação errada da estrutura de dados pode passar ao usuário uma informação inconsistente, ou não, no caso de haver redundância de informações.

Defeitos são inevitáveis. Componentes físicos envelhecem e sofrem interferências externas, sejam ambientais ou humanas. Projetos de software são vítima de sua alta complexidade e da fragilidade humana em trabalhar com grande volume de detalhes ou com deficiência de especificação [47]. A aplicação de teste sobre sistema é uma forma de melhorar a qualidade e antecipar as falhas que ocorreriam na aplicação em uso futuro.

O custo para resolver uma falha é diferente dependendo do estágio do software e em última instância é equivalente ao próprio valor do sistema, podendo ainda ser de valor imensurável (como quando há vidas humanas envolvidas). Normalmente, o custo de uma falha em produção é muito superior a uma em fase de desenvolvimento.

A presente pesquisa visa testar *Web Services* no intuito de encontrar defeitos no domínio de implementação, bem como demonstrar defeitos de especificação.

3.2.1 Casos de teste

Casos de teste são definidos como um conjunto de entradas, condições de execução e resultados esperados para um particular objetivo (como o de exercitar um determinado caminho do programa, ou verificar a especificação contra seus requisitos) [30].

É considerado um bom caso de teste aquele capaz de revelar um defeito ainda desconhecido. A ausência de erros na execução dos casos de testes não prova a correção do sistema, é possível que o

conjunto dos casos de teste não seja adequado para a realização do teste, ou ainda que estes sejam adequados, a correção de um sistema não pode ser demonstrada através de testes [17], devido ao grande domínio de suas entradas e ao elevado número de caminhos do programa.

Casos de teste representam um investimento e não devem ser descartados após a sua execução, pelo contrário eles devem ser guardados para a reaplicação no futuro. Uma nova implementação, ou modificação de alguma funcionalidade do sistema (mesmo que pequena), pode ocasionar falhas, inclusive em funcionalidades que não apresentavam defeitos anteriormente ou que não são diretamente relacionadas à modificação. A reexecução de casos de teste com objetivo de verificar se as modificações acrescentadas em uma nova versão do sistema não causaram defeitos em funcionalidades já existentes é conhecido como teste de regressão. Este tipo de teste salienta a importância de preservar os casos de teste.

3.2.2 Planejamento

Teste de software é uma atividade que pode ser sistematicamente planejada e especificada. A estratégia e a técnica de teste precisam ser definidas, casos de teste devem ser construídos, e os resultados avaliados em relação ao que era esperado [52].

A atividade de especificar os testes inicia-se com o levantamento dos objetivos e da estratégia de teste que deve ser utilizada. Cria-se então o plano de teste, que é um documento dinâmico, pois pode sofrer modificações durante a aplicação dos testes. Este deve incluir a previsão dos recursos (tanto humanos quanto financeiros) e tempo disponíveis, os processos a serem seguidos, as técnicas a serem aplicadas e as ferramentas que podem ser utilizadas, entre outras informações.

O projeto de casos de teste transforma os requisitos do sistema e comportamentos esperados em casos de teste documentados. O desenvolvimento do teste avalia a forma como ele será executado. Devem ser criados *scripts*, que são roteiros descrevendo a execução dos casos de teste. O projeto dos casos de teste e a preparação de seus dados constituem atividades fundamentais do planejamento de testes [56].

A execução do teste pode ser feita de forma manual, com a execução de cada passo descrito no roteiro, ou automaticamente. No último, é criado um programa para execução do teste ou utilizada alguma ferramenta onde é inserida a descrição do roteiro (normalmente utilizando alguma linguagem proprietária da ferramenta).

O resultado obtido com a execução do teste é comparado com o esperado. Caso sejam diferentes,

tem-se uma falha. Na maioria das vezes, para identificar e corrigir o defeito associado é necessário o processo de depuração, o qual é realizado devido à execução com sucesso do teste, isto é, quando houver a ocorrência de uma falha. A depuração tem como resultado a remoção de um erro [52].

3.3 Processo de Teste

Ao longo da construção de um sistema espera-se que este seja submetido a inúmeros testes, evitando assim a realização de uma fase específica para este fim. O software deve ser testado durante todo seu desenvolvimento, desde a sua concepção (validando os seus requisitos) até sua entrega [3,80].

Pressman [52] cita que a inclusão da atividade de testes diariamente durante a construção do sistema é muito eficaz, mas que a maioria das equipes de desenvolvimento optam pela visão incremental desta atividade, iniciando com testes de porções individuais dos programas, mudando para testes de integração e culminando com testes que exercitam o sistema como um todo. A Figura 3.1 demonstra a atividade incremental da execução do teste em comparação com o processo de construção de software [52]. Este processo tem início com a engenharia do sistema (em que é definido o seu papel), passando à análise de requisitos (em que as informações de domínio, função, comportamento, desempenho, restrições e critérios para validação são definidos), passando para o projeto e, finalmente, para a codificação.

Figura 3.1: Comparação da atividade de teste e o processo para construção de software

As subseções que seguem mostram as etapas de testes descritas por Pressman [52] e Ki [36]. Vale ressaltar a possibilidade de submeter o sistema a outras etapas de testes (como testes de requisito, mostrada por Allmann [3]).

3.3.1 Testes unitários

Testes unitários são aqueles cujo objetivo é exercitar a menor unidade do sistema. Estas unidades são testadas individualmente, procurando certificar que elas operam corretamente. Tal teste pode ser conduzido paralelamente em múltiplos componentes.

Neste tipo de teste, estruturas de dados locais são examinadas (para assegurar que os dados armazenados mantenham a integridade durante toda a execução do algoritmo), caminhos independentes são percorridos (para garantir que todas as declarações foram executadas pelo menos uma vez) e condições de fronteira são testadas (pois muitos erros acontecem quando é atingido o valor máximo - ou mínimo - de uma variável, quando um loop alcança a última iteração e ao se processar a última posição de um vetor ou array). Enfim, todos os tratamentos de erro devem ser testados.

O teste unitário é normalmente considerado um adjunto da etapa de codificação. O projeto dele pode ser realizado antes da etapa da codificação ou logo depois que o código fonte tenha sido gerado.

A aplicação de teste unitário em um sistema orientado a objeto diferencia-se pelo fato de que os testes não podem ser focados em uma função específica vista isoladamente, mas como parte de uma classe. Os testes unitários devem ser dirigidos para operações encapsuladas pela classe e o comportamento de estado da mesma.

3.3.2 Testes de integração

Testes de integração visam testar à integração entre as unidades do sistema, descobrindo defeitos associadas às suas interfaces. Uma vez que as unidades do sistema foram adequadamente testadas durante o teste unitário, intuitivamente pode-se pensar que elas também funcionem quando colocadas juntas, mas isto não é verdade. Entre os possíveis problemas pode-se verificar que:

- Dados podem ser perdidos;
- Um módulo pode ter um funcionamento adverso sobre outro;
- Dois módulos quando combinados não executam determinada função da forma desejada;
- Problemas não perceptíveis quando considerada a unidade, ganham magnitude;
- Estruturas de dados de uso global podem apresentar erros.

No processo de integração entre os componentes unitários testados pode ser adotada a abordagem de combinar todas as unidades e depois testar o programa como um todo. Essa abordagem não é aconselhável, pois uma grande quantidade de erros irá ocorrer e será muito difícil buscar as suas causas. A integração incremental é uma alternativa para este problema, uma vez que nela, o programa é construído e testado em pequenos incrementos, onde os erros são facilmente isolados e corrigidos. Duas estratégias são comumente utilizadas:

Top-down: os módulos são integrados caminhando de cima para baixo na hierarquia de controle, começando do módulo principal para os módulos subordinados.

Bottom-up: começa-se a integração com os módulos atômicos, componentes mais baixos na hierarquia, seguindo-se para os de nível mais alto.

A modelagem orientada a objetos não possui uma estrutura hierárquica de controle óbvia. Assim, os termos *top-down* ou *bottom-up* têm pouco significado na estratégia de integração. Normalmente é utilizada uma de outras duas estratégias:

Thread-based: o conjunto de classes que irá responder à determinada entrada ou a um determinado evento do sistema é integrado e testado.

Use-based: começa-se o teste pelas classes com menor (ou nenhum) nível de dependência das demais classes. Na seqüência, é testado o nível imediatamente acima.

3.3.3 Testes de validação

Após a disponibilização de uma versão do software é aplicado o teste de validação. Este possui o objetivo de validar o sistema quanto aos seus requisitos funcionais, definidos na fase de análise (ou incluídos/modificados durante a construção do sistema). O usuário deve verificar se o software atende a todas as suas expectativas.

É muito comum verificar sistemas que estão em versões alfa ou beta, as quais são utilizadas durante os testes de validação. A versão alfa refere-se ao uso do software pelos usuários finais, mas em um ambiente assistido; enquanto que a versão beta também é usada pelos usuários, mas em um ambiente que não é controlado pelo desenvolvedor. Nos dois casos todos os problemas de uso deveriam ser relatados à equipe desenvolvedora do sistema, mesmo nos testes em ambientes fora do controle do desenvolvedor (versão beta).

3.3.4 Testes de sistema

Testes de sistema verificam o sistema com relação a outros elementos, como hardware, usuário, banco de dados, entre outros. São propostos para esta etapa os seguintes tipos de teste: de recuperação ou tolerância à falhas, de segurança ou invasão, de estresse e de desempenho.

O processo de teste para aplicações Web diferencia-se um pouco do visto acima, mas as técnicas aplicadas são semelhantes. Ele inicia-se com a validação do conteúdo e da interface, em seguida são testados aspectos da arquitetura e da navegação, por fim, o foco do teste muda para exercitar a capacidade tecnológica, a infraestrutura e a instalação [52].

As funcionalidades da aplicação Web são implementadas por componentes de software. Estes podem ser implementados com o uso de Web Services. Testes de componentes Web devem ser realizados com a estratégia de caixa preta [1].

3.4 Técnicas e critérios para teste de software

As técnicas utilizadas para selecionar bons casos de testes podem ser classificadas em duas categorias: testes "caixa branca" ou "caixa preta". Cada uma destas categorias estabelece critérios para a seleção dos casos de teste. A seleção de todos eles é inviável. Pressman [52] cita que para um programa em linguagem C com dois loops aninhados executando de 1 a 20 vezes, dependendo das condições de entrada e contendo quatro construções de seleção (*if-else*), seriam necessários aproximadamente 10^{14} caminhos possíveis.

As técnicas de testes tentam selecionar boas entradas e as que são mais favoráveis a descobrir defeitos. Critérios ditam requisitos a serem satisfeitos, orientando na seleção dos dados para teste, na avaliação da sua qualidade e na definição de requisitos de suficiência a serem alcançados [50].

Critérios também devem ser estabelecidos pela equipe de desenvolvimento do software para determinar quando finalizar os testes e deixar que futuros erros sejam relatados por usuários. Não existe uma resposta definitiva para que se estabeleça esse critério. Pressman [52] aponta duas abordagens: a primeira no sentido que os testes fiquem a cargo do usuário (a cada execução do sistema o usuário está efetuando um teste) e a outra de que testes devem ser executados até o término do prazo ou o fim do dinheiro que financia o projeto.

[1] estratégia de teste que será apresentada neste capítulo

3.4.1 Teste caixa branca

Teste caixa branca (ou teste estrutural) é uma categoria de teste com o propósito de examinar o sistema logicamente. Os testes são realizados analisando a implementação e a estrutura interna dos componentes sem se preocupar com seus requisitos [35].

Utilizando testes caixa branca o engenheiro de software cria casos de teste que:

- Executam caminhos diferentes dentro código fonte do sistema;
- Exercitam mais de uma possibilidade das decisões lógicas;
- Executam as estruturas internas de laços e verificam suas fronteiras;
- Exercitam as estruturas de dados internas.

Neste trabalho não são gerados casos de teste desta categoria e, por isso, não haverá maior detalhamento destes.

3.4.2 Teste caixa preta

Teste caixa preta, também conhecido como teste de comportamento ou funcionais, é focado nos requisitos funcionais do software, isto é, permitem que sejam criadas entradas que irão exercitar os requisitos do sistema [52]. As funcionalidades são testadas contra a especificação e o sistema é visto como uma caixa preta, não considerando detalhes das estruturas internas ou de sua implementação.

O objetivo do teste caixa preta é efetuar operações sobre as funcionalidades e verificar se o resultado gerado é o esperado. Em [32], Whittaker e Thomason definem teste caixa preta de maneira formal, da seguinte maneira: Dado um programa P contendo a função f e com o domínio em d, o objetivo do teste é selecionar uma seqüência de entradas a partir de d, e aplicá-las em P, comparando o resultado obtido com a resposta esperada de f. Qualquer desvio do resultado esperado é caracterizado como uma falha.

Nesse tipo de teste procuram-se descobrir erros relacionados a cinco categorias [52]:

- funções incorretas ou omitidas;
- erros de interface;

- erros de estrutura de dados ou de acesso a dados externos;
- erros de comportamento ou desempenho;
- erros de iniciação e término.

Tipos de testes que se encaixam nesta categoria são: tabela de decisão, particionamento em classes de equivalência, testes de fronteiras, testes de integridade de banco de dados e grafo de causa e efeito.

3.5 Estratégias de Testes de Software

Nesta seção são apresentados os conceitos de algumas estratégias populares de testes. No presente trabalho foi utilizada uma adaptação dos conceitos aqui apresentados devido ao objetivo voltado para a geração automática de casos de teste para *Web Services*. Outras estratégias de testes não serão mostradas, pois não foram aplicadas neste trabalho.

3.5.1 Teste de Fronteira

Teste de fronteira (ou teste de domínio) é uma técnica que utiliza a fronteira dos dados da aplicação para a criação de casos de teste. Esse tipo de teste parte da premissa que grande parte dos erros tende a ocorrer nas fronteiras do domínio de entrada da aplicação [43, 51].

Abaixo são apresentadas as diretrizes para a aplicação da técnica de teste de fronteira:

1. Se uma condição de entrada especifica limites de valores, como a e b, devem ser gerados casos de teste para os valores a e b, assim como para os valores logo inferior e superior a estes.
2. Se uma condição de entrada especifica valores numéricos, devem ser desenvolvidos casos de teste para exercitar os valores mínimo e máximo. Valores exatamente abaixo e acima também devem ser testados.
3. Aplicar as diretrizes 1 e 2 também para a saída do sistema, ou seja, procurar que a saída atinja os valores limites.
4. Se existe algum limite em alguma estrutura do sistema (exemplo: um vetor que comporta 100 elementos), devem ser criados casos de teste para exercitar esses limites.

Um exemplo da possível aplicação desta técnica é o episódio ocorrido com o foguete Ariane 5 vôo 501. Ele explodiu em 1996 durante a sua decolagem devido a um defeito no software envolvendo a conversão de um ponto flutuante de 64-bit para um valor inteiro de 16-bit [37]. A aplicação de teste de fronteira poderia ter identificado o problema e, possivelmente, evitado o acidente.

3.5.2 Teste de Regressão

Teste de regressão é uma técnica aplicada após o sistema ter sofrido alguma melhoria ou reparação. Seu propósito é de verificar se mudanças afetaram algo no sistema que funcionava anteriormente, provocando desta forma, uma *regressão*. O uso desta técnica baseia-se na execução de casos de teste já executados anteriormente e na comparação entre os resultados [23]. Sua importância reside no fato de que mudanças e correção de erros tendem a ser muito mais causadoras de defeitos do que a concepção inicial do sistema [23].

Segundo Dustin [18], automatizar testes de regressão é importante devido ao grande número de casos de teste requeridos normalmente pelos sistemas. A execução manual desta tarefa é um esforço demorado e tedioso, o que acarreta na sua aplicação incompleta. Ainda segundo o autor, é comum que o executor sinta-se confortável e não realize todos os casos de teste por eles já terem funcionado anteriormente.

3.5.3 Teste de Mutação

Teste de mutação é uma técnica que mede o quanto um conjunto de casos de teste é adequado para identificar defeitos no sistema [17]. Esta técnica foi introduzida em [53] e supõe que o programa estará bem testado se todos os defeitos simples forem encontrados e removidos [34]. Ela se apóia na hipótese do efeito de acoplamento, definido em [48], que estipula que "defeitos complexos estão sempre acoplados a defeitos simples, e um caso de teste capaz de revelar um defeito simples irá defeitos mais complexos também".

Teste de mutação baseia-se na criação de diversas versões do mesmo programa, cada um contendo um defeito. Os defeitos são definidos por operadores de mutação, e cada mudança ou mutação criada por um destes operadores é codificada em um programa mutante [34].

Os operadores de mutação são construídos com um dos objetivos a seguir [2]:

- Induzir mudanças sintáticas, inserindo erros que os programadores cometem normalmente (como

trocar um nome de variável);

- Forçar objetivos comuns de testes (como executar um trecho de código);

O objetivo do engenheiro de teste é de encontrar para cada programa mutante um caso de teste que diferencie a sua saída do programa original. Quando encontrado um caso de teste com essa propriedade o programa mutante é considerado como *morto*, e os casos de teste subseqüentes não são executados. Um conjunto de casos de teste que *mata* todos os programas mutantes é considerado adequado para aqueles programas mutantes.

Um programa mutante *vivo* ao fim da execução dos casos de teste implica, segundo DeMillo [27], em uma das alternativas a seguir:

- O conjunto de casos de teste precisa ser aumentado, pois não houve uma entrada capaz de diferenciar os dois programas;
- O programa mutante é equivalente ao programa original.

Offutt [17] argumenta que geralmente é impossível *matar* todos os programas mutantes. Algumas mudanças não possuem efeito no comportamento funcional do programa e cabe a um ser humano realizar a verificação destes casos.

3.6 Testes automáticos

Testes automáticos são testes em que ao menos parte de sua execução é realizada por um computador. Os métodos atuais de desenvolvimento de software exigem que planos de testes sejam executados a cada entrega do sistema. Testes automáticos podem oferecer um grande auxílio, já que quando desenvolvidos de forma adequada, são facilmente executados [56].

Segundo Staknis [61] grandes testes só podem ser realizados por um processo automático. Os benefícios do uso de testes automáticos incluem redução de tempo, esforço, trabalho e custo na realização dos testes [63].

Testes automáticos se enquadram dentro do processo de software com a proposta de diminuir o tempo e o custo de testar o sistema. Pelo fato de eles facilitarem a execução dos testes, possibilitam que estes sejam executados a cada manutenção, implicando em uma melhor qualidade do software.

Existem muitas ferramentas de teste no mercado para softwares de propósito geral. Contudo, elas são consideradas primitivas tendo em vista os requisitos de qualidade [36]. Algumas ferramentas oferecem alguns níveis de automação, mas a automação é limitada à simples engenharia reversa, gravação e execução de scripts [36]. O objetivo da automação da execução dos testes é minimizar a quantidade de trabalho manual e ganhar maior cobertura permitindo que uma grande quantidade de casos de testes seja executada [10].

As ferramentas de gravação e execução de scripts trabalham apenas com sistema de interface gráfica [56]. Todos os objetos visualizados no sistema a ser testado são registrados e todas as seqüências de interações realizadas com estes objetos são gravadas em um arquivo. Este contém todos os movimentos do mouse, entradas inseridas, opções escolhidas e o resultado obtido. A ferramenta possui uma função de repetição ou execução, que a partir desse arquivo realiza a seqüência de comandos descrita e compara o resultado com o esperado [56].

Algumas outras ferramentas possuem como alvo problemas e sistemas de alguma área específica, tendo seu escopo bem definido, podendo ser utilizadas apenas para um determinado propósito [36]. A pesquisa apresentada por Buy, Orso e Pezze [68] cujo o objetivo é a criação de um método para geração automática de testes para classes, é um exemplo de estudo para uma função especifica. Os autores exploram a dependência entre o estado de um objeto e o seu comportamento. Eles defendem que um dos principais problemas no teste de classes surge com variáveis membro e seu efeito sobre os métodos definidos na classe. O artigo utiliza os resultados da *data flow analysis* (análise de fluxo de dados) definida por Harrold e Rothermel [38] como parte do método para a geração de casos de teste para classes.

Em [13], Beyer, Chlipala e Majumdar apresentam uma abordagem para a automatização de testes em um domínio específico. A pesquisa tem o objetivo de encontrar *deadcode* em programas feitos em linguagem C. Para isto, os autores estenderam o *Model Checking - BLAST* para geração automática de casos de teste. Através destes são verificados trechos de código que não são atingidos por nenhuma entrada. Embora a abordagem seja muito interessante, vale ressaltar que para diversos casos não é possível aplicar essa estratégia, como programas em que o fluxo dependa de base de dados, arquivos ou execução de outros programas.

A tarefa de geração dos casos de teste consome quase 30% da atividade de testes [10]. Essa é uma das atividades que é executada manualmente e uma primeira candidata a economia através da automação. Contudo, a tecnologia necessária não avança rápido como esperado [10].

A Tabela 3.1 traz uma relação e uma breve descrição das funcionalidades das ferramentas de testes automáticos encontrada no mercado.

Tabela 3.1: Ferramentas de mercado para testes.

Nome da Ferramenta	**Breve descrição**
Mercury TestDirector	Criação e execução de script de teste; gerenciamento de testes e erros; cadastro de plano de testes.
Mercury LoadRunner	Teste de estresse, agendamento e execução. Abrange uma grande gama de protocolos, mas não SOAP [41].
Compuware QACenter	Execução de script de testes, análise de código fonte, cadastro e gestão de defeitos, gerenciamento de testes e ferramenta de requisitos.
IBM Rational Test Tools (possui vários produtos para teste)	Teste de *estresse* e desempenho, teste de regressão, metodologia de testes, gestão de defeitos, testes e monitoração de aplicações Web.
Borland Silk & Gauntlet	Execução de teste automática integrada ao controlador de versão, teste de perfomance para Web, gestão de defeitos, gerenciamento de testes e requisitos, teste de cenário, monitoração de aplicações Web.

Ferramentas de testes unitários, como *JUnit*, *PerlUnit* e *csUnit* são muito populares. Elas são *frameworks* de testes em que o desenvolvedor escreve trechos de códigos que invocam os métodos a serem testados. Os resultados obtidos são validados, informando-se ao *framework* se a execução foi com sucesso ou se houve uma falha. Essas ferramentas possibilitam a automação da execução dos testes, mas não a sua geração.

3.7 Conclusão

Neste capítulo foram apresentados os conceitos básicos e algumas técnicas existentes para testes de software. Foi discutida a automação da atividade de testes.

Uma pesquisa sobre as maiores ferramentas de mercado para testes foi mostrada, sendo que nenhuma das ferramentas faz geração de casos de testes para Web Services.

Capítulo 4

Testes para Web Services

Com o aumento da adoção de *Web Services*, testá-los começou a receber maior atenção [66], mas ainda é um assunto pouco explorado, sendo necessárias pesquisas acadêmicas e de empresas [29].

Tsai [64] cita que realizar testes de *Web Services* é importante para ambos os papéis no ciclo de SOA (provedor e consumidor de serviços). Para o consumidor do serviço, detalhes de *design* e implementação freqüentemente não estão disponíveis. Ele possui apenas a especificação, possibilitando apenas testes do tipo caixa preta. Em grandes aplicações, múltiplos *Web Services* estão envolvidos, sendo selecionados e invocados em tempo de execução. Estas características fazem com que testes para *Web Services* sejam um desafio [64]. As técnicas atuais de testes de software precisam ser adaptadas para atender às diferentes características e padrões tecnológicos envolvendo *Web Services*.

Bloomberg [7] oferece um panorama do estado atual de testes para *Web Services* e o que é esperado para o futuro. O autor levanta uma série de questões sobre a necessidade de testes para *Web Services*, ele cita que novos cenários de testes irão surgir e que consumidores e produtores de *Web Services* terão que se adaptar para esse novo tipo de computação.

Em [69] é listada uma série de características que devem ser consideradas ao testar *Web Services*:

- Falta de acesso ao código fonte: Teste de *Web Services* é equivalente a teste caixa preta, pois a especificação está disponível (WSDL), mas não o projeto (diagramas) ou o código fonte, e este pode ser escrito em uma variedade de linguagens de programação [70].
- Dinâmico e de baixo acoplamento: Devido ao fato de *Web Services* ser baseado em padrões abertos e utilizar interfaces padronizadas, isto torna possível o desenvolvimento de serviços com baixo acoplamento e integração dinâmica.

- Ambiente heterogêneo: *Web Services* podem ser executados em ambientes completamente heterogêneos. Os serviços podem ser executados entre empresas com linguagens, APIs e protocolos de transporte diferentes (parte dos serviços pode utilizar http e outra, MIME).
- Plataformas independentes: *Web Services* podem ser executados em plataformas com diferentes hardwares e em diversos sistemas operacionais. Mesmo desenvolvidos em diversas plataformas, os *Web Services* devem interagir um com os outros.
- Ausência de interface com o usuário: Aplicações tradicionais e Web sempre possuem interface com usuário, e os testadores podem utilizá-la para inserir os dados de entrada. Este não é o caso dos *Web Services*.

Web Services podem ser considerados mais heterogêneos que aplicações Web. Estas, usam múltiplas linguagens de programação, mas *Web Services*, além de possuirem esta característica, usam diferentes sistemas operacionais e são executados em *containers* de aplicação diferentes [81].

As abordagens das ferramentas atuais de testes citadas por [36] (engenharia reversa, gravação e execução de scripts), não podem ser aplicadas a *Web Services*, já que na maioria das vezes, apenas a interface do serviço está disponível e a gravação de scripts não é possível, pois os mesmos não possuem interface com o usuário.

Existem diversos fatores envolvidos no processo de testar *Web Services*, como:

- Funcionalidade: assegura que o serviço fornece respostas apropriadas para as requisições, evitando que uma falha aconteça com ele em ambiente de produção.
- Interoperabilidade: Teste cujo objetivo é verificar a aderência do serviço a padrões WS-I [12]
- Desempenho: mede a capacidade do serviço a atender requisições. Para efetuar esse tipo de teste é necessário ter informações de volume e carga que o serviço deve suportar.
- Segurança: verifica os aspectos de autorização para a execução do serviço, a criptografia das mensagens trocadas (ou de parte delas); evita ataques que se aproveitam de alguma fraqueza das mensagens, entre outros.

A abordagem desse trabalho consiste em testes de funcionalidade, embora a ferramenta possa ser facilmente adaptada para executar teste de desempenho e coletar dados simples de estatística,

tais como: número total de requisições, respostas a requisições por segundo e chamadas simultâneas. Alguns fatores de segurança também são tratados como a geração de casos de testes que utilizam técnicas de *SQL Injection* [60].

4.1 Exemplo de criação manual de testes para Web Services

Abaixo será ilustrado como seria o processo manual para testar os *Web Services* do sistema exemplo *Auto-Aluguel*, apresentado no Capítulo 2. Vale ressaltar que o objetivo não é testar a interface Web do sistema, mas sim a camada de serviços da qual a aplicação possui dependência.

Cada locadora de carro parceira construirá o seu próprio *Web Service* para realização de negócio com o sistema *Auto-Aluguel*, mas estes *Web Services* devem ser testados por este último para garantir a qualidade do serviço prestado pelo site. Para isto, é necessário criar casos de teste manualmente e aplicá-los a cada uma das empresas parceiras, repetindo os mesmos testes diversas vezes.

A Listagem 4.1 apresenta o *XML Schema* utilizado no documento de inteface (WSDL) do serviço de confirmação do aluguel (mensagem de requisição). O trecho contém os dados do veículo, assim como o preço do aluguel. Todas as locadoras parceiras implementam essa interface em uma tecnologia e plataforma independentes. A Listagem 4.2 mostra um trecho exemplo de mensagem para o serviço da Listagem 4.1.

Listagem 4.1: Trecho do XML Schema utilizado no WSDL da mensagem de requisição da confirmação de aluguel.

```
<xs:element name="modelo" type="xs:string"/>

<xs:element name="ano">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:totalDigits value="4"/>
            <xs:maxInclusive value="2008"/>
            <xs:minInclusive value="1900"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
```

```
<xs:element name="valorAluguel">
    <xs:simpleType>
        <xs:restriction base="xs:decimal">
            <xs:maxInclusive value='99999.99'/>
            <xs:minInclusive value='1'/>
            <xs:fractionDigits value="2"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

<xs:element name="numPortas">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:enumeration value="2"/>
            <xs:enumeration value="3"/>
            <xs:enumeration value="4"/>
            <xs:enumeration value="5"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

<xs:element name="placa">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:pattern value="[A-Z][A-Z][A-Z]-[0-9][0-9][0-9][0-9]"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

<xs:element name="siglaEstado">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:length value="2"/>
```

```
        </xs:restriction>
    </xs:simpleType>
</xs:element>
```

Listagem 4.2: Trecho de chamada de um serviço que segue o WSDL da Listagem 4.1.

```
<veiculo>
    <modelo>Gol</modelo>
    <ano>2006</ano>
    <valorAluguel>55.00</valorAluguel>
    <numPortas>2</numPortas>
    <placa>XYZ-1234</placa>
    <siglaEstado>SP</siglaEstado>
</veiculo>
```

Como o código fonte das aplicações parceiras não estão disponíveis, resta a alternativa do teste caixa preta. Para cada operação é necessário escolher casos de teste que procurem validar o serviço. A Tabela 4.1 traz alguns dados para os casos de testes da operação da Listagem 4.1, cada linha da tabela irá resultar em uma mensagem. Para cada serviço da aplicação *Auto-Aluguel* se faz necessária a criação de casos de teste.

Wallace [80] cita que não deve ser aguardado o fim do projeto para que os testes sejam iniciados. A fim de exercitar os testes nos *Web Services* antes do fim do projeto, é necessário criar uma aplicação cliente para testes, esta irá invocar o serviço e fornecer as informações dos casos de teste.

Pelo fato de cada serviço do *Auto-Aluguel* poder ser construído por uma equipe de desenvolvimento diferente, em etapas distintas do projeto e utilizando diversas tecnologias (como o caso do sistema de gerência de e-mail que está sendo reutilizado), é essencial submeter cada serviço a testes durante todas as fases do desenvolvimento.

Normalmente, quando é criada uma aplicação cliente para testar um serviço, após a execução do caso de teste, este é substituído pelo caso de teste seguinte, o que ocasiona um retrabalho em uma futura manutenção do serviço (em que será necessário reaplicar os testes). O mesmo se aplica aos resultados das execuções dos testes, que são verificados e depois descartados, não possibilitando um teste de regressão em uma futura manutenção.

É freqüente a criação de casos de teste sem metodologia e sem conhecimento de técnicas de

Tabela 4.1: Casos de teste para teste manual.

Exemplo de Mensagem	Objetivo do teste
modelo=""	Obter mensagem de erro, como "veículo inválido".
ano=0	Obter mensagem de erro, como "ano inválido".
numPortas=2,3,4,5	Cada valor da enumeração precisa ser testado.
placa="aaa-aaaa"	Obter mensagem de erro "placa inválida".
siglaEstado="aaa"	Obter mensagem de erro "sigla de estado inválida".
modelo="aaaaaaaaaaaaaaaaaaaa"	Testar nome de carro inválido.
modelo="Celta"	Testar nome de carro válido.
ano=2111	Testar ano inválido.
placa="abc-1234"	Placa de automóvel válida e com letras minúsculas.
modelo="CELTA"	Testar modelo do carro com letras maiúsculas.
modelo="celta"	Testar modelo do carro com letras minúsculas.
placa="AAA-1234"	Placa de automóvel com letras maiúsculas.
valor=0	Obter erro "valor inválido de aluguel".
valor=99999.99	Valor máximo permitido.
valor=1234	Testar valor sem parte decimal.
valor=0,99	Testar valor do aluguel.
siglaEstado="SP"	Testar sigla do estado em maiúsculas.
siglaEstado="sp"	Testar sigla do estado em minúsculas.
siglaEstado="Sp"	Testar sigla do com a primeira letra em maiúscula.
Todos os dados em branco	Verificar mensagem de erro.

testes. Algumas vezes as entradas de todos casos de testes exploram as mesmas características do sistema. Testes de valores de fronteiras ou validação de precondições normalmente não são executados manualmente, deixando uma parte importante do sistema sem ser validada.

4.1.1 Ferramentas de Testes para Web Services

Algumas ferramentas para testes de Web Servics estão disponíveis na Internet. Elas se propõem a realizar testes unitários, simular o provedor ou consumidor do serviço, executar testes de desempenho e auxiliar na organização dos roteiros de testes.

As ferramentas de código livre: WSUnit, SoapUI e JXWeb cujo o objetivo é de contribuir com testes de Web Services foram analisadas. Tentou-se também a obtenção de uma cópia e da licença da ferramenta comercial Parasoft - SOATest, mas após preenchimento de pesquisa e algumas trocas de mensagens, uma resposta negativa sobre a obtenção da licença, devido a não intenção de compra, foi

recebida.

WSUnit

O software livre WSUnit [82] permite a criação de mensagens de requisição e de resposta de um serviço em um arquivo XML. A ferramenta pode ser utilizada pelo consumidor, quando o serviço ainda não está disponível, ou permitindo com que este escolha a resposta do serviço para testar um determinado comportamento. O provedor do serviço pode utilizar a ferramenta para simular o consumidor, criando um arquivo com a mensagem de requisição.

A ferramenta é facilmente configurável, criando os arquivos de requisição e resposta em uma estrutura de diretório particular. Os casos de teste são construídos de maneira semelhante ao JUnit, projeto do grupo Apache para a criação de testes unitários, o qual a ferramenta é baseada. A Listagem 4.3 traz um exemplo de código Java utilizando a ferramenta WSUnit.

Listagem 4.3: Exemplo de código Java utilizando a ferramenta WSUnit

```
package com.jpeople.wsunit.test;

public class AutoAluguelTest extends WSUnitTestCase {

        public AutoAluguelTest(String arg) {
                super(arg);
        }

    public void testAuto() throws Exception{
        String content = sendXML("/axis2/services/AutoAluguel",
            "autoaluguel.xml");
        int index = content.indexOf("gol");
        assertEquals("Dado de resposta encontrado " +
                content, false, index == -1);
    }

}
```

O primeiro parâmetro da função *sendXML* é a *URI* do serviço, o segundo parâmetro é o arquivo

que contém os dados utilizados na chamada de requisição. WSUnit não oferece interface gráfica para o usuário, mas pode ser beneficiado do suporte que algumas ferramentas e editores possuem ao JUnit.

SoapUI

SoapUI [59] é um software livre para teste de Web Services utilizado para a realização de teste de carga, inspeção, desenvolvimento e invocação de serviços.

Para auxiliar o desenvolvimento e a inspeção dos serviços a ferramenta permite a visualização do arquivo WSDL de forma hierárquica, facilitando a inspeção de suas operações, assim como o formato das mensagens de chamada e resposta. Para cada operação a ferramenta pode criar uma chamada de requisição modelo, permitindo que o usuário possa preencher os parâmetros e utiliza-la no futuro para execução de testes. A ferramenta facilita a integração com diversas APIs que trabalham com Web Services (JBossWS, JWSDP, Axis 1, Axis 2, XFire, .NET e Gsoap), auxiliando na criação do código para o consumidor e o provedor do serviço.

A ferramenta oferece suporte para geração manual de caso de testes, possibilitando a criação do documento XML de chamadas, assim como o XML da mensagem de resposta. O usuário pode fazer uso de linguagem de script, Groovy e valores de propriedades podem ser carregados de arquivos de configuração.

Para testes de desempenho o soapUI oferece a possibilidade da execução de teste de carga. O teste pode ser monitorado através de gráficos, contadores de thread e logs.

JXWeb

JXWeb [21] é uma extensão da ferramenta JXUnit, cujo propósito é a separação da construção dos casos de testes do código fonte do programa, os casos de testes são descritos em uma linguagem de script em um arquivo XML. O JXWeb permite a criação de scripts para a invocação de Web Services. A ferramenta está em um estágio inicial de desenvolvimento, ela apresentou dificuldades para a realização de testes simples de Web Services.

4.2 Geração automática de testes para Web Services

As poucas abordagens para verificação e validação de *Web Services* são baseadas em testes automatizados ou verificação de modelo [29]. Abaixo são apresentados os trabalhos relevantes envolvendo estas duas abordagens e um comentário sobre cada uma das pesquisas.

Tsai [66] propõe um *framework* para testes de Web Services que utiliza as seguintes etapas: especificação de cenário, geração automática de *scripts* de testes, monitoração, execução automática de testes distribuídos e gerenciamento de mudanças. Neste mesmo artigo são propostas extensões no WSDL (linguagem utilizada para especificar a interface do serviço), pois os autores defendem que as informações do WSDL não são suficientes para a realização de testes. Estas extensões são: dependência de entrada e saída, descrição funcional hierárquica e seqüência de invocação de serviços.

Os autores apresentam de forma muito geral as técnicas que estão sendo aplicadas e não abordam como é feita a geração automática de scripts de testes. A suposição de uma extensão na linguagem WSDL é muito forte e não aplicável aos serviços atuais.

Em [65] propõem uma técnica para teste de Web Services baseada na utilização de testes de grupo progressivo. Esta técnica é aplicada para testar um grande conjunto de Web Services. No nível de unidade, Web Services com a mesma funcionalidade são testados usando um número progressivamente crescente de casos de testes. Após a aplicação destes testes, aqueles serviços que obtiveram melhores resultados serão integrados em um ambiente real para testes operacionais. No nível de integração, várias composições diferentes serão testadas utilizando os serviços que foram selecionados na etapa anterior. Os autores não dizem se existe implementação deste trabalho.

A pesquisa apresentada por Huang [29] mostra a verificação de *Web Services* baseada na checagem do modelo de OWL-S (*Web Ontology Language for Web Services*). A pesquisa utiliza o BLAST [6] (software que faz a checagem de programas escritos em linguagem C) para validar o modelo de OWL-S. A validação do modelo é feita utilizando os casos de teste gerados automaticamente durante a execução do BLAST. Foi proposta uma extensão no BLAST para tratar concorrência (que é permitida no OWL-S) e melhorias no OWL-S para facilitar a geração automática dos casos de teste.

Esta abordagem possui a premissa da existência de um modelo OWL-S. Isto não é verdade para a grande maioria dos *Web Services* e também não permite que o consumidor de serviço possa validá-lo, não sendo adequado ao modelo de desenvolvimento orientado a serviço.

Em [64], Tsai descreve um *framework* baseado em XML para testar Web Services. O *framework* é chamado Coyote, e é composto de duas partes: *Test Master* e *Test Engine*. O *Test Master* extrai informações da interface WSDL e faz o mapeamento para cenários de testes. Os casos de teste são gerados a partir dos cenários, que estão no formato XML e são interpretados pelo *Test Engine* para a execução, invocando o *Web Service* alvo.

Tsai [64] não oferecem ao público detalhes de como é feita a geração dos casos de teste a partir dos cenários. Aparentemente, o *framework* proposto possui a função de automatizar a execução e não de gerar casos de teste automáticos.

Offutt e Xu [49] mostram uma nova abordagem para testar *Web Services* baseada em *data perturbation*. O processo funciona modificando mensagens de requisição, reenviando e analisando a mensagem de resposta para observar se o comportamento está correto.

No referido artigo, *data perturbation* é dividido em dois métodos: ***data value perturbation***, que modifica os valores na mensagem SOAP levando em consideração o tipo de dados, e ***interaction perturbation***, que modifica as mensagens levando em consideração o seu tipo (RPC ou *document*).

Data value perturbation é fundamentada nas idéias de teste de valor de fronteira. As mensagens são modificadas de acordo com as regras estabelecidas para os tipos de dados. No artigo os autores apresentam as regras para cinco tipos de dados, mas informam que a pesquisa também abrange os demais. Os casos de teste são criados substituindo cada valor pelo seu valor de fronteira correspondente. A Tabela 4.2 mostra as regras para os tipos de dados apresentada no artigo.

Tabela 4.2: Regras para os tipos de dados do artigo [49].

Tipo de Dados	**Valor de Fronteira**
String	Tamanho máximo, tamanho mínimo, String vazia, todos caracteres maiúsculos e minúsculos.
decimal, float e double	Valor máximo, valor mínimo e zero
Boolean	true, false

Interation perturbation é subdividida em outras duas abordagens, uma para mensagens RPC, chamada de *RPC Communication Perturbation (RCP)* e outra para mensagens Document, chamada de *Data Communication Perturbations (DCP)*.

O RCP é focado em testar o uso que o serviço faz dos dados, aplicando funções sobre os mesmos. Estas funções são baseadas nas idéias de operadores mutantes. Em um trabalho anterior [34], Offutt havia apresentado uma técnica de teste utilizando análise mutante para testar componentes cuja interação é baseada em XML. A interação entre os componentes é especificada utilizando *Interation Specification Model (ISM)* que contém a definição da mensagem, assim como o documento WSDL para *Web Services*. Formalmente um operador mutante é:

Definição 1. *Dado um conjunto de todas as instâncias dos elementos N, um operador mutante é* $r = f(n_1,..., n_i)$*, onde f é uma função,* $i \geq 1$*, cada* $n_1,..., n_i \in N$ *e possuem o mesmo tipo de dados, e r possui o mesmo tipo de dados que* $n_1,..., n_i$*.*

A Tabela 4.3 traz uma lista de operações apresentadas no artigo.

Tabela 4.3: Relação de operações apresentadas no artigo [49].

Função	**Descrição**
divide(n)	Trocar o valor de n por 1/n, sendo n um double.
Multiply(n)	Trocar o valor de n pelo quadrado de n.
Negative(n)	Trocar o valor de n por -n.
Absolute(n)	Trocar o valor de n pelo seu módulo.
Exchange(n1, n2)	Substitui o valor de n1 por n2, e vice-versa.
Unauthorized(str)	Modifica o valor da string (str) por: "**str' OR '1' = '1**" para simular um SQL Injection.

Enquanto isso, o DCP consiste em modificar os valores de acordo com o tipo de dado focado no teste de relacionamentos e restrições. As mensagens representando o XML Schema são definidas utilizando o modelo formal RTG (*Regular Tree Grammar*). Uma *Regular Tree Grammar* é uma 6-tupla $< E, D, N, A, P, n_s >$, onde:

1. E é um conjunto finito dos elementos
2. D é um conjunto finito dos tipos de dados
3. N é um conjunto finito de não terminais
4. A é um conjunto finito de atributos
5. P é um conjunto finito de regras de produção com duas formas:
   - $n \rightarrow a < d >$, onde n é um não terminal em N; a pode ser um atributo em A ou um elemento em E, e d é um tipo de dado em D;
   - $n \rightarrow e < r >$, onde n é um não terminal em N; e é um elemento em E, e r é uma expressão regular composta de não terminais.

6. n_s é o elemento inicial não terminal, $n_s \in N$

Baseado no atributo *maxOccurs* presente no XML Schema que especifica a mensagem, os autores especificaram relacionamentos entre os elementos pais e filhos. Para estes relacionamentos os autores definiram as seguintes estratégias de testes:

- Dado um relacionamento $n \rightarrow e < r >$, se existe uma expressão $\alpha?$ em r, existirão dois casos de testes. Um contendo uma instância α e outro contendo um conjunto vazio de instâncias.
- Dado um relacionamento $n \rightarrow e < r >$, se existe uma expressão $\alpha+$ em r, existirão dois casos de testes. Um contendo uma instância α e um outro contendo um número permitido de instâncias α.
- Dado um relacionamento $n \rightarrow e < r >$, se existe uma expressão $\alpha^*\alpha$ em r, existirão dois casos de testes. Um contendo $\alpha^*\alpha$ e o outro contendo α^{*-1}, onde $\alpha^*\alpha$ duplica uma instância do elemento e α^{*-1} apaga uma instância do elemento.

As restrições são consideradas, mas os autores apenas demonstraram a restrição *totalDigitsValues*. O método apresentado neste trabalho é restrito a interações ponto a ponto.

Em [83] os autores descrevem um método para gerar testes para comunicação baseada em XML. O método baseia-se em modificar o XML Schema que especifica as mensagens. O XML Schema é representado como uma árvore com nós e arestas, na qual cada nó pode representar um tipo de dados. A Figura 4.1 mostra uma árvore exemplo.

Os esquemas são modificados utilizando operadores de perturbação. O objetivo é perturbar XML Schema para criar mensagens XML inválidas. Esta abordagem assume que XML Schema é usado pela aplicação e isso nem sempre é verdade para Web Services.

Os autores propõem sete operadores de perturbação divididos em dois tipos, um aplica-se para nós e o outro para sub-árvores. Os operadores que se aplicam sobre os nós são: inserir e apagar um nó entre dois outros nós (as arestas são ajustadas para manter a árvore), inserir e apagar um nó que contém um tipo de dados. Os operadores que se aplicam em sub-árvores são: inserir e apagar uma sub-árvore abaixo um nó e modificar a aresta entre dois nós usando uma aresta com diferentes restrições (*constraints*).

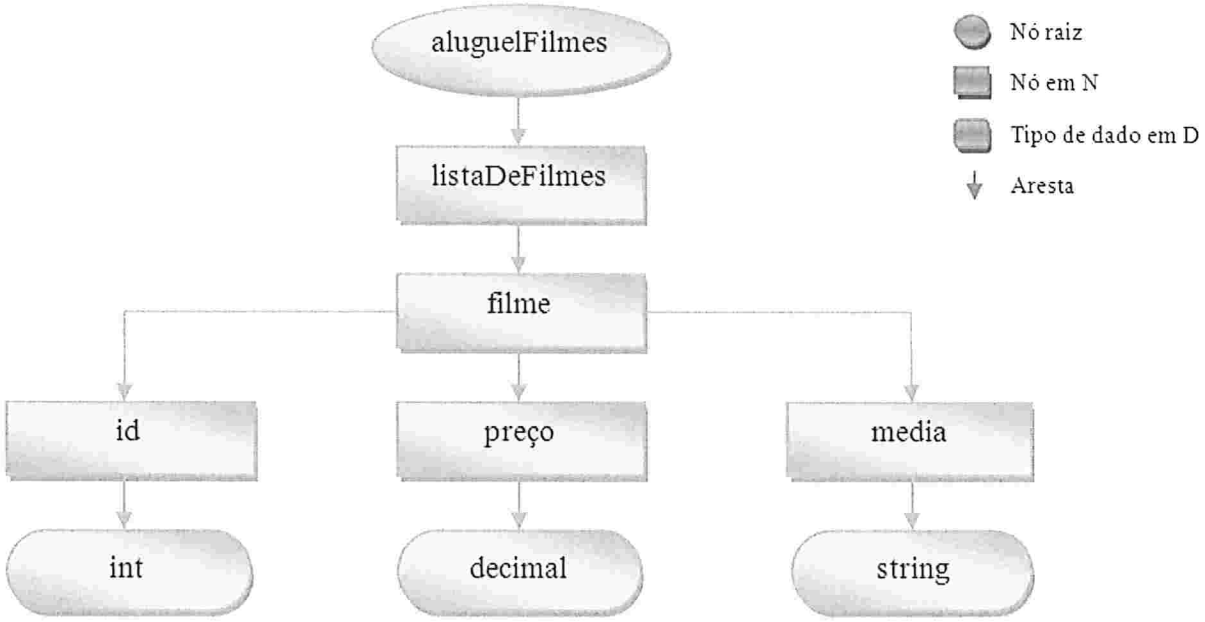

Figura 4.1: Árvore utilizada para representar o XML Schema

Vergilio e Almeida [16] mostraram seis novos operadores mutantes para mensagens SOAP e descreveram uma ferramenta de suporte para gerar mensagens modificadas. Eles consideram um mensagem SOAP como uma árvore regular formada por um conjunto de nós N e $n_1, ..., n_i \in N$, onde $i \geq 1$ e n_c é o número e nós de um dado n. A Tabela 4.4 contém os operados descritos neste trabalho.

4.3 Conclusão

No início desse capítulo foram discutidas as características de testes para Web Services, e os elementos que devem ser considerados ao testá-los. Foi mostrado um estudo sobre ferramentas gratuitas onde não foi identificada nenhuma ferramenta que gerasse casos de testes automatizados.

Foi apresentado um estudo sobre as pesquisas atuais envolvendo testes automatizados de Web Services. Foi dada ênfase ao artigo de Offutt e Xu [49], cuja abordagem é utilizada como base para este trabalho. Neste artigo o autor não cria mensagens com valores fora do domínio válido, estas devem causar erro quando executado. Não utiliza todos os valores de restrições contidas na especificação de

Tabela 4.4: Operadores mutantes para mensagens SOAP

Nome do Operador	Breve descrição
Null(n)	Modifica para *null* o valor assinalado de n na mensagem SOAP.
Incomplete(n)	Apaga o nó n e seus nós filhos da mensagen SOAP.
Inversion (n)	Inverte a ordem dos nós filhos de n de uma mensagem SOAP.
ValueInversion (n)	Inverte a ordem dos valores assinalados aos filhos do nó n de uma data mensagem SOAP.
Mod_Len (n)	Modifica o tamanho dos valores assinalados ao nó n.
Space (n)	Modifica para ' ' o valor assinalado ao nó n.

cada elemento, tais como: padrão e enumeração. E aplica perturbação apenas utilizando a restrição *maxOccurs*.

Capítulo 5

Melhorias na geração automática de casos de teste para Web Services

O capítulo anterior apresentou trabalhos na área da geração automática de casos de testes para Web Services. Este capítulo apresenta modificações e melhorias na técnica de perturbação de dados. Foi introduzido um novo conjunto de perturbação de valores de fronteira, quatro novos operadores de mutação e melhorias na perturbação baseada no relacionamento entre elementos contidos na especificação do serviço.

5.1 Modificação das técnicas de testes

As técnicas de testes utilizadas neste trabalho são baseadas nas propostas dos trabalhos [49] [83] e [16]. Esta seção apresenta as novas idéias e melhorias das técnicas de perturbação de dados.

5.1.1 Indicador de erro esperado

Cada mensagem gerada neste trabalho contém um indicador verdadeiro ou falso informando se a resposta esperada de sua execução deve ser um defeito (indicador com o valor verdadeiro). Esse indicador permite que as ferramentas que implementam as técnicas demonstradas neste capítulo decidam de forma automática, ou seja, sem o auxílio de um oráculo, se a execução de um caso de teste ocorreu com sucesso. As futuras mensagens descritas nas próximas seções e classificadas como mensagens inválidas possuem esse indicador na posição verdadeiro.

A execução de um caso de teste que contém o indicador de resultado esperado deve ser validada pela ferramenta que decidirá utilizando algum critério se a resposta obtida é válida ou não. Algum

dos critérios sugeridos abaixo pode ser utilizado:

- Verificação se a resposta da execução do caso de teste é uma mensagem SOAP Fault, como apresentada na seção 2.2.5;
- Utilização de XPath para verificar a existência de algum elemento previamente configurado pelo utilizador do sistema que indique a ocorrência de um erro. Exemplo: O usuário pode configurar que mensagens contendo o elemento *códigoDeErro* sinaliza um erro;
- Verificação da ocorrência ou não de algum conteúdo previamente configurado pelo utilizador do sistema na mensagem de resposta, ex: a não ocorrência do conteúdo "sucesso" indica execução com erro.

Um exemplo da utilização desse indicador é a geração de um caso de teste que possui um elemento com o valor fora do domínio válido. A execução deste caso de teste deve ocasionar uma mensagem de resposta contendo um erro, caso contrário, a ferramenta deve apontar um defeito.

5.1.2 Perturbação dos valores dos dados e Facetas de Restrição

A primeira aplicação da perturbação de dados é a criação de casos de teste de acordo com regras definidas para cada tipo de dados. Ela é baseada na aplicação da técnica de teste de fronteira [19] apresentada na seção 3.5.1. Casos de testes são criados substituindo os valores da mensagem por seu correspondente valor de fronteira de acordo com o seu tipo de dados. De acordo com a especificação W3C do *XML Schema* [73] existem 19 tipos de dados primitivos que podem ser usados em XML. A Tabela 5.1 descreve cada um dos 19 tipos de dados e a Tabela 5.2 a perturbação aplicada a cada um. Alguns outros tipos derivados (como é o caso do `integer` que é derivado do tipo de dados `decimal`) também são mostrados.

A abordagem aqui apresentada diferencia-se do artigo [49] em diversos aspectos. Nos parágrafos seguintes serão mostradas as diferenças.

Testes de fronteira devem exercitar os valores limites assim como valores logo inferior e superior a estes [43, 51]. A abordagem desta pesquisa também gera casos de testes para estes valores. As mensagens geradas fora do domínio válido são marcadas com o indicador de erro ligado 5.1.1.

Web Services usando mensagens literais podem ser definidos por um *XML Schema* e os valores

válidos para cada tipo de dados podem ser especificados utilizando facetas de restrição [73]. Existem doze diferentes facetas de restrição.

Offutt e Xu [49] demonstram as facetas que restringem o valor máximo, mínimo e número total de dígitos. Esta pesquisa adiciona casos de testes inválidos a estas três facetas e abrange também as demais. Os itens abaixo descrevem todas facetas e as mensagens geradas ao encontrar tais restrições.

- **Pattern**: Define o conteúdo válido para um tipo de dado utilizando uma expressão regular. A faceta *pattern* foi utilizada para gerar mensagens válidas e inválidas, por exemplo: para o tipo *placa* especificado no WSDL da Listagem 4.1 é gerado as mensagens da Tabela 5.3.

  Uma expressão regular pode conter quantificadores para determinar quantas vezes um caractere ou um grupo de caracteres precedentes ao quantificador é permitido de ocorrer. Um número predefinido deve ser usado para criar casos de testes que contenham quantificadores infinitos.

- **Enumeration**: Restringe em um conjunto específico os valores válidos de um tipo de dado. Uma mensagem é gerada para cada valor contido na enumeração. É gerada também uma mensagem inválida com um valor fora do conjunto. A Listagem 4.1 especifica o tipo *numPortas*, as mensagens com os elementos descritos na Tabela 5.4 são geradas.

- **FractionDigits**: Determina o número de dígitos da parte decimal de um número. O valor deve ser igual ou maior que zero. Três mensagens são criadas: a primeira contendo o número máximo de dígitos decimais especificado, a segunda com apenas um dígito decimal e a última utilizando mais dígitos que o máximo permitido.

- **Length**: Determina o número exato de caracteres ou itens permitidos em uma lista. Uma mensagem válida e uma inválida são geradas. A mensagem inválida contém um caractere a mais que o especificado pelo *length*.

- **TotalDigits**: Define o número máximo de valores permitido restringindo aos números que são expressos como $i \times 10^{-n}$ onde i e n são inteiros tais que $|i| < 10^{totalDigits}$ e $0 \leq n \leq totalDigits$. Exemplo: Seja $totalDigits$ definido com o valor 4, o valor 55.51 é válido, pois pode ser expressado como 5551×10^{-2}, $i = 5551$ e $n = 2$. Uma mensagem é gerada com o valor máximo permitido e uma outra inválida ultrapassando o valor máximo. Uma terceira mensagem extra é criada utilizando dígitos fracionários, caso o elemento possui também a restrição *FractionDigits*.

- **WhiteSpace**: Especifica como espaços, tabulação e quebra de linha irão ser tratados. Dependendo do valor do *whiteSpace* (*preserve, replace, collapse*) mensagens são geradas contendo quebra de linha e tabulador.
- **MaxInclusive, minInclusive, maxExclusive, minExclusive**: Aplicam-se aos tipos de dados que possuem ordem matemática, elas definem as fronteiras dos valores numérico e são considerados para a criação de casos de testes. Mensagens inválidas são geradas com valores fora das fronteiras especificadas.
- **MaxLength, minLength**: Determina o número máximo e mínimo de unidades do tipo de dado string ou um dos seus derivados. O valor não pode ser negativo. Casos de testes são criados para o tamanho mínimo e máximo, assim como valores fora do domínio.

Para elemento que especifica *nillable="true"*, é gerado uma mensagem com o elemento sem conteúdo e possui o atributo *nil="true"*.

A geração de casos de testes baseado em uma restrição definida pela faceta gera mensagens de melhor qualidade do ponto de vista de testes do que uma mensagem que segue apenas restrições baseadas no tipo de dados do elemento, pois cria mensagens exatamente na fronteira do domínio válido. Um exemplo desta vantagem são as mensagens geradas na Tabela 5.3. Se não fosse utilizado a expressão contida na faceta *Pattern* seria apenas gerada uma mensagem contendo um número predeterminado de caracteres como descrito em [49].

A Tabela 5.5 apresenta mensagens geradas aplicando esta abordagem em uma mensagem definada pelo XML Schema apresentado na Listagem 4.1. As linhas da tabela com o fundo cinza claro são casos de testes que não seriam gerados aplicando-se somente a técnica como proposta pelos autores de [49]. As linhas com o fundo cinza escuro são exemplos de casos de teste que se diferenciam dos que seriam gerados por [49].

5.1.3 Perturbando mensagens do tipo *Document* utilizando relacionamento e restrições

O consumidor e o provedor de Web services que utilizam mensagens do tipo *Document*, interagem usando documentos completos. Estes documentos são tipicamente XML, o qual foi definido por esquema comum e acordado.

Esta abordagem busca testar os relacionamentos e restrições existentes no esquema que define a mensagem. Foram utilizadas as mesmas definições de Offutt e Xu [49], ou seja, mensagens são

definidas em um RTG (*Regular Tree Grammar*) como demonstrado previamente. Os relacionamentos e restrições são incluídos em um conjunto finito de regras de produção de uma RTG. A definição formal de um relacionamento é dada como a seguir:

Definição 2. *Dado um XML Schema* $< E, D, N, A, P, n_s >$, *um relacionamento é um regra de produção em* $P : n \rightarrow e < r >$, *onde* n *é um não terminal em* N, e *é um elemento em* E, *e* r *é uma expressão regular formada por não terminais.*

Offutt e Xu usaram a ocorrência do indicador *maxOccurs* para especificar os relacionamento entre elementos pais e filhos. Este trabalho introduz uma nova abordagem que utiliza o outro indicador de ocorrência *minOccurs* e os indicadores de ordem *all* e *choice*, e o elemento *any*. A Tabela 5.6 descreve cada indicador do XML Schema utilizado e a expressão regular associada.

Além das três estratégias de testes estabelecidas por Offutt e Xu [49], foram adicionadas as seguintes estratégias:

- Dado um relacionamento $n \rightarrow e < r >$, se existe uma expressão $\alpha+$ em r, existirá um caso de teste extra contendo nenhuma instância de α[1]. Este caso de teste possui o indicador de erro esperado ligado.
- Dado um relacionamento $n \rightarrow e < r >$, se existe uma expressão $\alpha*$ em r, existirão dois casos de testes extras. Um excluindo todas as instâncias α[1], e o outro contendo n instâncias α, onde n é um número predefinido representando *unbounded*[2].
- Dado um relacionamento $n \rightarrow e < r >$, se existe uma expressão contendo '.' em r, um caso de teste será criado. Ele irá conter uma instância do elemento β, onde β representa um elemento qualquer registrado em uma base de dados interna[3].
- Dado um relacionamento $n \rightarrow e < r >$, se existe uma expressão $\alpha\{x, y\}$ em r, existirão dois casos de testes. Um contendo x instâncias do elemento α[4] e o outro contendo y instâncias de α[5]. Se y possui o valor *unbounded*, y terá o valor de n, onde n é um número predefinido[2].
- Dado um relacionamento $n \rightarrow e < r >$, se existe uma expressão $\{x_1, .., x_n\}$ em r, existirão três casos de testes. Um contendo uma permutação randômica de $\{x_1, .., x_n\}$[6], outro contendo $\{x_1, .., x_{n-1}\}$[7] e o outro contendo apenas o elemento $\{x_1\}$[8].

- Dado um relacionamento $n \rightarrow e < r >$, se existe uma expressão $x_1|..|x_n$ em r, existirão $n+1$ casos de testes diferentes. Cada um dos primeiros n casos de testes contém x_i onde i é um inteiro e $1 \leq i \leq n$.[9] O outro caso de teste irá conter todos os n elementos e ele contém o indicador de erro esperado na posição verdadeiro[6].

Listagem 5.1: XML Schema para uma locadora de filmes

```
<xs:element name=''AluguelFilmes''>
  <xs:complexType>
    <xs:sequence minOccurs=''0'' maxOccurs=''1''>
      <xs:element name=''ID''>
        <xs:complexType>
          <xs:choice>
            <xs:element name=''registroGeral''>
              <xs:simpleType>
                <xs:restriction base=''xs:string''>
                 <xs:pattern value=''[0-9]{8}-[0-9]-[A-Z]{3}''/>
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
            <xs:element name=''numeroMembro'' type=''xs:decimal''/>
          </xs:choice>
        </xs:complexType>
      </xs:element>
      <xs:element name=''listaFilmes''>
        <xs:complexType>
          <xs:sequence>
            <xs:element name=''filme'' minOccurs=''1''
                        maxOccurs=''5''>
              <xs:complexType>
                <xs:sequence>
                  <xs:element name=''id'' type=''xs:int''/>
                  <xs:element name=''media''>
                    <xs:simpleType>
                     <xs:restriction base=''xs:string''>
                       <xs:enumeration value=''DVD''/>
```

[1]Implementado na ferramenta GenAutoWS pelas mensagens nomeadas como *removed* e *incomplete*
[2]Mensagens *Max Unbounded*
[3]Mensagens *Any*
[4]Mensagens *Min*
[5]Mensagens *Max*
[6]Mensagens *All*
[7]Mensagens *All not completed*
[8]Mensagens *All Only one*
[9]Mensagens *Choice*

```
            <xs:enumeration value=''VHS''/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name=''preco''>
        <xs:simpleType>
          <xs:restriction base=''xs:decimal''>
            <xs:fractionDigits value=''2''/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
...
```

A Listagem 5.1 mostra um exemplo de um XML Schema para uma locadora de filmes. Sua RTG contém dois relacionamentos:

$n_{id} \rightarrow ID < n_{registroGeral} | n_{numeroMembro} >$

$n_{listaFilmes} \rightarrow filme < (n_{ID}, n_{media}, n_{preco})\{1, 5\} >$

Quatro casos de testes foram gerados para ambos os relacionamentos e são mostrados nos Exemplos 5.3, 5.4, 5.5, 5.6. As mensagens foram geradas a partir do XML mostrado na Listagem 5.2.

Listagem 5.2: Exemplo de mensagem respeitando o XML Schema da Listagem 5.1

```
...
  <id>
      <registroGeral>12221098-5-SSP</registroGeral>
  </id>
  <listaFilmes>
      <filme>
          <id>12</id>
          <media>DVD</media>
          <preco>3.25</preco>
      </filme>
      <filme>
          <id>130</id>
          <media>DVD</media>
          <preco>3.25</preco>
      </filme>
  </listaFilmes>
...
```

Listagem 5.3: Caso de teste para o segundo elemento do relacionamento *choice*.

```
...
  <id>
      <numeroMembro>1234</numeroMembro>
  </id>
...
```

Listagem 5.4: Caso de teste com todos os elementos do relacionamento *choice*. Um erro é esperado nesta mensagem.

```
...
  <id>
      <registroGeral>12221098-5-SSP</registroGeral>
      <numeroMembro>1234</numeroMembro>
  </id>
...
```

Listagem 5.5: Caso de teste contendo o número máximo do relacionamento *all*.

```
...
  <id>
      <numeroMembro>1234</numeroMembro>
  </id>
  <listaFilmes>
      <filme>
          <id>12</id>
          <media>DVD</media>
          <preco>3.25</preco>
      </filme>
      <filme>
          <id>130</id>
          <media>DVD</media>
          <preco>3.25</preco>
      </filme>
      <filme>
          <id>12</id>
          <media>DVD</media>
          <preco>3.25</preco>
      </filme>
      <filme>
          <id>12</id>
          <media>DVD</media>
          <preco>3.25</preco>
      </filme>
      <filme>
          <id>12</id>
```

```
            <media>DVD</media>
            <preco>3.25</preco>
        </filme>
    </listaFilmes>
...
```

Listagem 5.6: Caso de teste contendo o número mínimo permitido do relacionamento *all*

```
...
    <id>
        <numeroMembro>1234</numeroMembro>
    </id>
    <listaFilmes>
        <filme>
            <id>12</id>
            <media>DVD</media>
            <preco>3.25</preco>
        </filme>
    </listaFilmes>
...
```

5.1.4 Operadores de Perturbação

Operadores de Perturbação baseiam-se na idéia de RPC Communication Perturbation [49] e Operadores de Perturbação SOAP [16]. Em [49] os autores aplicaram as operações mutantes apenas em Web Services utilizando o estilo RPC, este trabalho propõem também a aplicação em mensagens do estilo *Document*. Os tópicos a seguir apresentam os novos operadores de mutação incluídos neste trabalho.

Injeção numérica em SQL

Em [60], Spett apresenta a técnica para explorar aplicações que utilizam consultas SQL sem tomar precauções com as informações que são fornecidas para essas consultas. O autor cita que embora seja simples de se precaver contra esses ataques, existem inúmeras aplicações com essa vulnerabilidade.

Com o objetivo de proteger-se contra injeção SQL muitos programadores removem aspas ou inserem caractere de escape a entradas que serão utilizadas em comandos SQL, mas isto não remove completamente o risco em algumas linguagens de programação. Considere a seguinte consulta SQL:

SELECT campos FROM tabela WHERE id == $id;

O conteúdo de *$id* deve possuir um valor numérico. O programa pode apresentar um defeito e expor todos os usuários se fosse dado o valor *0 OR 1=1* sem verificar se o valor é realmente um número. Abaixo é apresentado um operador mutante que explora essa característica.

SQLNumeric(n)

Ação do operador: Troca o conteúdo do elemento n pelo valor: *0 OR 1=1*. A mensagem gerada possui o indicador de erro na posição ligada.

Injeção de código

Injeção de código é a técnica de introduzir algum código dentro de um programa utilizando alguma entrada que não esteja sendo verificada contra esse tipo de ataque. Geralmente o objetivo desta técnica é modificar a funcionalidade original do sistema. Normalmente o uso baseia-se na suposição que aspas ou ponto e vírgula nunca serão utilizados na entrada, ou apenas caracteres alfanuméricos serão digitados, ou o uso da entrada para acesso em uma posição de matriz, etc. As seguintes operações mutantes foram incluídas na tentativa de expor defeitos do sistema ao uso desta técnica.

DynamicEvaluation(n) - Operador mutante que explora a situação onde parte da entrada é utilizada em uma função *eval*, exemplo 5.7

Listagem 5.7: Código descrito na linguagem PHP utilizando *eval*

```
  $var = '';
  eval('\$var=\$inputValue;');
...
```

Ação do operador: Utiliza o valor: *0;system(\"/bin/echo mensagem de erro\");* em n.

Fileinjection(n) - Este operador é utilizado para demonstrar defeitos no uso de um parâmetro de entrada como nome de arquivo.

Ação do operador: Adiciona ao valor de n o nome de um arquivo local previamente configurado.

Null

XML Schema introduziu um mecanismo para sinalizar que o conteúdo de um elemento é ausente ou nulo [72]. A Listagem 5.8 mostra um exemplo de XML Schema e de um XML utilizando esse recurso.

Listagem 5.8: Exemplo de XML utilizando o nulo

```
XML Schema:
<element name=''nomeDoMeio'' type=''string'' nullable=''true''/>

XML Document:
<nomeDoMeio xsi:null=''true''/>
```

O seguinte operador mutante foi incluído:

Null(n)

Ação do operador: Adiciona o atributo *xsi:null="true"* a n, e remove o seu conteúdo.

5.2 Conclusão

Neste capítulo foram apresentadas melhorias para a criação de casos de testes com informações contidas na interface do Web Service. Foram mostrados também, exemplos de casos de teste para as técnicas propostas.

Várias mensagens acrescentadas testam situações onde erros são esperados, tais mensagens fora do domínio válido não eram criadas na abordagem original. Foi apresentada cada uma das facetas de restrição que não eram exploradas pelos autores anteriores. O número de mensagens criadas baseadas em restrições entre elementos pais e filhos aumentou em mais de 100%, desta forma havendo uma cobertura maior dos casos de testes. Dois dos novos operadores de mutação procuram explorar defeitos de segurança que podem existir no serviço.

Diversas estratégias aqui apresentadas podem ser aplicadas para outros tipos de aplicações além de Web Service, especialmente para sistemas que utilizem XML Schema para especificar suas entradas.

Tabela 5.1: Descrição dos tipos de dados considerados para a perturbação dos valores dos dados

Tipo de Dados	Descrição
String	Qualquer conjunto de caracteres Unicode que um documento XML pode conter.
Boolean	true, false
Decimal	Um número decimal, como: 45.312345 ou -0.22.
Float	Um ponto flutuante de 4-byte IEEE-754 [31].
Double	Um ponto flutuante de 8-byte IEEE-754 [31].
Duration	Uma duração de um período de tempo, descrito em ISO8601 [62]. Possui o seguinte formato: PnYnMnDTnHnMnS. Exemplo: P3Y1M2DT1H4M2.7S, corresponde a três anos, um mês, dois dias, uma hora, quatro minutos e 2.7 segundos.
dateTime	Um particular momento de tempo, descrito em ISO8601 [62], e que possui o seguinte formato: CCYY-MM-DDThh:mm:ss. Pode possuir o sufixo Z para indicar uma compensação do UTC. Exemplo: 1906-10-23T15:28:00-04:00.
Time	Representa uma hora específica, descrita em ISO8601 [62], com o seguinte formato: hh:mm:ss.sss.
Date	Representa uma data específica, descrita em ISO8601 [62], e que possui o seguinte formato: YYYYMMDD.
gYearMonth	Representa um certo mês em um determinado ano. Exemplo: 2005-06.
gYear	Um ano do calendário gregoriano, tendo como faixa de valor de 0001 crescendo e -0001 e retrocedendo.
gMonthDay	Um dia especifico em um mês, sem citar o ano. Exemplo: –12-25.
gDay	Um dia particular, sem citar o mês ou ano. Exemplo: —14. A faixa de valor é de —01 a —31.
gMonth	Um particular mês. Exemplo: –01–. A faixa de valor é de: –01– a –12–.
hexBinary	Codificação hexadecimal de dados binários. Cada byte é substituído por dois dígitos hexadecimal.
Base64Binary	Dado binário codificado em base 64.
anyURI	URI com caminho absoluto ou relativo.
QName	Um opcional prefixado nome de XML.
NOTATION	Um nome de uma notação declarado no schema corrente.
Integer	Um valor inteiro de tamanho arbitrário, similar a classe java.math.BigInteger.
int	Um valor inteiro entre -2147483648 e 2147483647 inclusive
short	Um valor inteiro entre -32768 e 32767 inclusive
byte	Um inteiro entre -128 e 127 inclusive

Tabela 5.2: Perturbação aplicada a cada tipo de dados

Tipo de Dados	**Valores de fronteira**
String	Tamanho máximo, mínimo, caracteres maiúsculos, minúsculos e primeira letra maiúscula.
Boolean	Verdadeiro e falso.
Decimal	Valor máximo, mínimo, zero, número total de dígitos.
Float	Valor máximo, mínimo, zero.
Double	Valor máximo, mínimo, zero.
Duration	Valor máximo, mínimo (valores negativo não são permitidos), zero (P0Y0M0DT0H0M0S, P0Y, P0Y0M...)
dateTime	Valor máximo e mínimo (respeitando a faixa de valores estabelecida por cada componente deste campo).
Time	Valor máximo, mínimo e zero. (respeitando a faixa de valores estabelecida por cada componente deste campo)
Date	Valor máximo e mínimo. (respeitando a faixa de valores estabelecida por cada componente do campo)
gYearMonth	Valor máximo, mínimo (a parte relativa ao mês tem como valores mínimo e máximo 01 a 12).
gYear	Valor máximo, mínimo (zero não é permitido neste tipo)
gMonthDay	Valor máximo, mínimo (os dígitos referentes ao mês variam na faixa entre 01 e 12, e o valor do dia entre 01 e 30 ou 31 dependendo do mês).
gDay	Valor máximo, mínimo (valor do dia entre 01 e 30 ou 31 dependendo do mês).
gMonth	Valor máximo, mínimo (os dígitos referentes ao mês variam na faixa entre 01 e 12)
hexBinary	Tamanho máximo, mínimo (00).
Base64Binary	Tamanho máximo, mínimo (AA==).
anyURI	Tamanho máximo, mínimo, falta de nome do protocolo (://).
QName	Tamanho máximo, mínimo.
NOTATION	Tamanho máximo, mínimo, caracteres maiúsculos e minúsculos.
int, short, long	Valor máximo, mínimo, zero.

Tabela 5.3: Valores gerados usando a faceta *pattern* aplicado ao Exemplo 4.1

<placa>ZZZ-9999< /placa>	Válida
<placa>AAA-0000< /placa>	Válida
<placa>999-ZZZZ< /placa>	Inválida

Tabela 5.4: Valores gerados pela enumeração apresentada no WSDL da Listagem 4.1

<numPortas>2< /numPortas>	Válido
<numPortas>3< /numPortas >	Válido
<numPortas>4< /numPortas >	Válido
<numPortas>5< /numPortas >	Válido
<numPortas>99999< /numPortas>	Inválido

Tabela 5.5: Exemplo da aplicação de perturbação de dados, utilizando valores de fronteira.

Exemplo de Valor	Valor para o caso de Teste	Regra utilizada
<modelo>Gol< /modelo>	<modelo>AAAAAAAAAAAAAA< /modelo>	Tamanho máximo
	<modelo>GOL< /modelo>	Maiúsculas
	<modelo>gol< /modelo>	Minúsculas
	<modelo>< /modelo>	Tamanho mínimo
<ano>2006< /ano>	<ano>9999< /ano>	Total Dígitos
<ano>2006< /ano>	<ano>99999< /ano>	Excesso de Dígitos
<ano>2006< /ano>	<ano>2008< /ano>	Valor máximo
	<ano>1900< /ano>	Valor mínimo
	<ano>2009< /ano>	Valor logo sup. ao max
	<ano>2007< /ano>	Valor logo inf. ao max
	<ano>999< /ano>	Valor logo inf. ao min.
	<ano>1901< /ano>	Valor logo sup. ao min.
	<ano>1899< /ano>	Valor logo inf. ao min.
<valorAluguel>55.00< /preço>	<preço>99999.99< /preço>	Valor máximo
	<valorAluguel>1< /valorAluguel>	Valor mínimo
	<valorAluguel>100000< /valorAluguel>	Valor logo sup. ao max.
	<valorAluguel>99998< /valorAluguel>	Valor logo inf. ao max.
	<valorAluguel>0< /valorAluguel>	Valor zero e logo inf. ao min.
	<valorAluguel>2< /valorAluguel>	Valor logo sup. ao min.
	<valorAluguel>0.99< /valorAluguel>	Dígitos Fracionários
	<valorAluguel>0.999< /valorAluguel>	Dígitos Fracionários Excedido
<numPortas>2< /numPortas>	<numPortas>3< /numPortas>	Enumeração
	<numPortas>4< /numPortas>	Enumeração
	<numPortas>5< /numPortas>	Enumeração
	<numPortas>99999< /numPortas>	Enumeração Inválida
	<numPortas>0< /numPortas>	Valor zero
	<numPortas>2^{32}-1< /numPortas>	Valor máximo
	<numPortas>-2^{32}< /numPortas>	Valor mínimo
<placa>XYZ-1234< /placa>	<placa>ZZZ-9999< /placa>	Padrão valor máximo
	<placa>AAA-0000< /placa>	Padrão valor mínimo
	<placa>999-ZZZZ< /placa>	Padrão inválido
	<placa>AAAAAAAAAAAAAAAA< /placa>	Tamanho máximo
	<placa>< /placa>	Elemento sem conteúdo
	<placa xsi:nil="true">< /placa>	Valor nulo
<siglaEstado>SP< /siglaEstado>	<siglaEstado>ZZ< /siglaEstado>	Valor máximo
	<siglaEstado>AAA< /siglaEstado>	Valor logo sup. max.
	<siglaEstado>< /siglaEstado>	Elemento sem conteúdo
	<siglaEstado xsi:nil="true">< /siglaEstado>	Valor nulo

Tabela 5.6: Expressão regular usada para representar o relacionamento em uma RTG.

Indicador	**expressão**	**descrição**
maxOccurs	?	zero ou mais ocorrências
maxOccurs	+	no mínimo uma ocorrência
maxOccurs	*	qualquer número de ocorrências
minOccurs, maxOccurs	$\{x,y\}$	no mínimo x e não mais que y ocorrências
choice	|	um elemento filho ou outro
all	$\{x_1, .., x_n\}$	elementos filhos podem aparecer em qualquer ordem mais todos devem ocorrer apenas uma vez
any element	.	elemento não especificado no XML Schema, qualquer elemento

Capítulo 6

Ferramenta GenAutoWS

Esse capítulo apresenta a ferramenta GenAutoWS que foi construída baseada na ferramenta soapUI e implementa as técnicas apresentadas no Capítulo anterior. Três outras funcionalidades foram introduzidas na ferramenta para auxiliar a atividade de teste de Web Services, estas são: integração com servidor de registro UDDI, captura de mensagens que estão sendo trocadas entre uma aplicação cliente e o Web Service e uma base de dados interna para guardar valores de elementos que são utilizados posteriormente para criar novas mensagens.

6.1 Introdução a Ferramenta GenAutoWS

A ferramenta GenAutoWS foi construída com o enfoque de auxiliar dois dos papéis da arquitetura SOA: o consumidor e o provedor de serviços. O consumidor pode utilizar a ferramenta para validar a qualidade de um serviço que deseja utilizar gerando casos de testes de maneira automática. Ele normalmente não possui o código fonte do serviço, apenas sua interface. O provedor do serviço pode utilizar a ferramenta para gerar mensagens automáticas e simular o sistema consumidor, não havendo a necessidade da construção de um sistema cliente para testar o Web Service.

A partir do estudo das ferramentas de código livre apresentado no Capítulo 4 foi escolhida a ferramenta soapUI como programa base para a construção da ferramenta GenAutoWS. Ela apresentava o maior número de recursos e funcionalidades em comum com as que eram necessárias pela ferramenta GenAutoWS.

SoapUI não oferecia recursos para geração automática de casos de testes, captura de mensagens e integração com servidor UDDI. As seguintes funcionalidades foram acrescentadas à ferramenta:

- Geração automática de casos de testes usando perturbação sobre mensagens previamente existentes;
- Indicador de erro esperado para cada caso de teste gerado;
- Captura de mensagens SOAP e criação de casos de testes com essas mensagens;
- Integração com repositório UDDI;
- Criação automática de mensagens de testes baseada em dados previamente capturados.

A ferramenta GenAutoWS (como a ferramenta soapUI) foi desenvolvida na linguagem Java. A ferramenta utiliza diversos outros projetos de software livre tais como:

- XMLBeans - manipulação de XML, permitindo a associação deste com objetos Java.
- Axis - framework para criação de Web Services
- Jakarta Regexp - API para validação de expressão regular que foi modificada para gerar um caso válido para uma dada expressão.
- HSQLDB - banco de dados desenvolvido com a linguagem Java.
- UDDI4J - interação com servidores de registro UDDI.

6.1.1 Geração automática de casos de testes na Ferramenta GenAutoWS

A ferramenta GenAutoWS implementa as modificações e melhorias mostradas no Capítulo 5. Outras abordagens dos trabalhos [49] e [16] que não foram alteradas no Capítulo 5 também foram implementadas, como descritas no seu trabalho de origem, exemplo: foram implementados os operadores de mutação propostos por Xu e Offutt [49], mostrados na Tabela 4.3, além dos operadores de mutação apresentados por Vergilio e Almeida [16] descritos na Tabela 4.4.

A Figura 6.1 mostra a tela principal da ferramenta GenAutoWS que é semelhante a aparência da ferramenta soapUI. No lado esquerdo da figura está a estrutura de projetos. A ferramenta trabalha com o conceito de projeto, cada projeto pode conter diversas interfaces WSDL, a ferramenta mostra todas as operações contidas para cada interface. Cada operação pode conter diversas mensagens de testes, na tela apresentada existe apenas uma mensagem, chamada "Request 1".

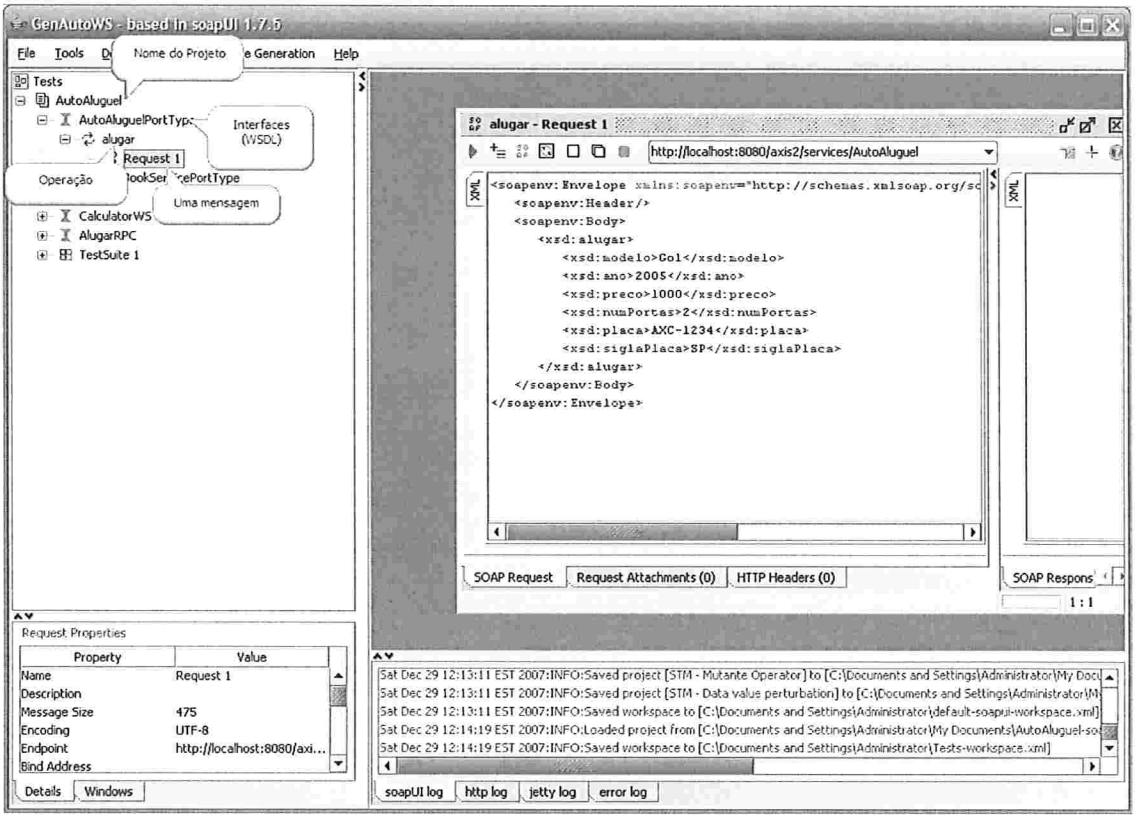

Figura 6.1: Tela principal da ferramenta GenAutoWS

A Figura 6.2 mostra o menu de contexto com as funcionalidades para a geração automática de mensagens. No lado direito podemos ver uma mensagem que foi capturada pela ferramenta utilizando a opção de captura de mensagem (menu superior). Quando esta funcionalidade é ligada, a ferramenta aguarda mensagens em uma porta predefina e redireciona toda informação para um outro endereço ip e porta previamente configurados. Durante o processo de direcionamento das informações, as mensagens de requisição são analisadas e se a ferramenta possui a operação que está sendo invocada em algum dos projetos. Uma cópia local da mensagem é incluída naquela operação.

A Figura 6.3 apresenta as mensagens que são geradas utilizando a técnica de perturbação dos valores de dados e facetas de restrição, seção 5.1.2, utilizando a mensagem de teste mostrada na Figura 6.1. Esta mensagem é especificada pelo WSDL mostrado na Listagem 4.1 do Capítulo 4.

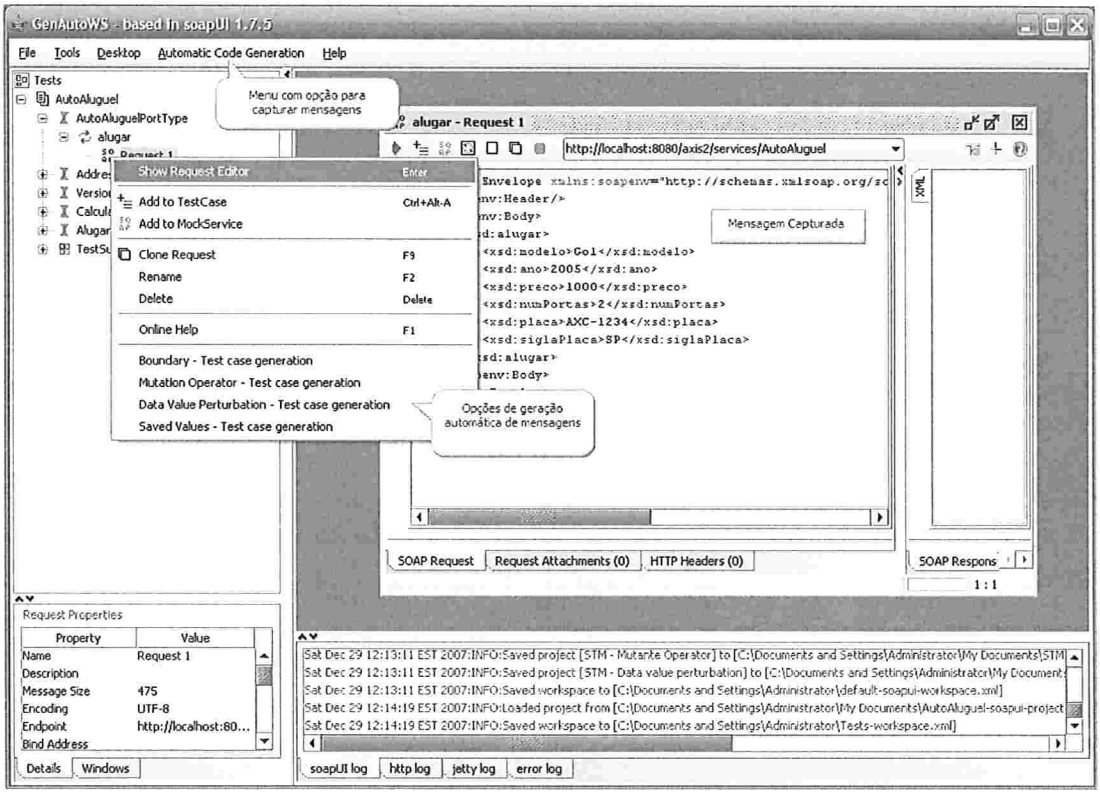

Figura 6.2: Menu de contexto para geração de mensagens automáticas

Algumas das mensagens geradas são mostradas na Figura 6.4.

Na tela mostrada na Figura 6.5 é apresentada a janela para que o usuário selecione quais operadores de mutação serão utilizados para a geração das mensagens. O usuário deve selecionar para cada elemento da mensagem semente (mensagem com que está gerando os casos de testes) o operador que deseja aplicar. Alguns operadores mutantes podem ser aplicados sobre dois elementos ao mesmo tempo, para estes casos, uma nova janela é aberta para que o usuário tenha a opção de selecionar um outro elemento.

A Figura 6.6 mostra quatro mensagens criadas com a aplicação da proposta para criação de casos de testes para mensagens *Document* utilizando relacionamento e restrições, seção 5.1.3. As mensagens foram geradas para o serviço descrito pelo XML Schema apresentado na Listagem 5.1. Observe que as mensagem apresentadas nas Listagens 5.3, 5.4, 5.5 e 5.6 foram criadas. A mensagem semente

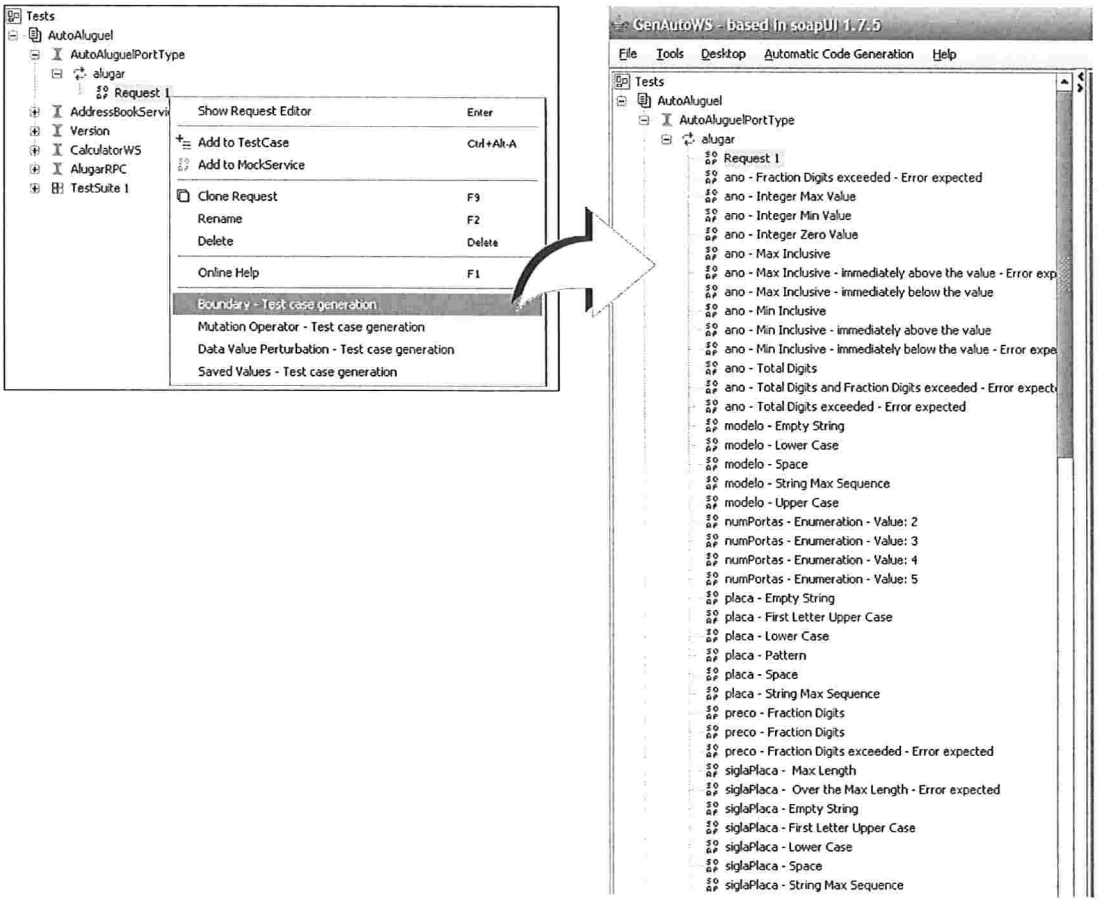

Figura 6.3: Mensagens geradas aplicando a técnica descrita em 5.1.2

utilizada é baseada no trecho de mensagem da Listagem 5.1.

As mensagens criadas podem ser incluídas em um conjunto de teste (*test suite*). A ferramenta permite que os conjuntos de casos de testes sejam executados automaticamente, ou seja, pressionando um botão cada mensagem é enviada para o seu serviço e a mensagem de resposta é acrescentada no caso de teste.

Mensagens geradas com o indicador de erro esperado ligado sinalizam à ferramenta que o caso de teste deve obter uma mensagem de exceção como retorno à sua execução, caso contrário, a ferramenta contabiliza uma falha. Falhas diagnosticadas nesta situação são classificadas como encontradas

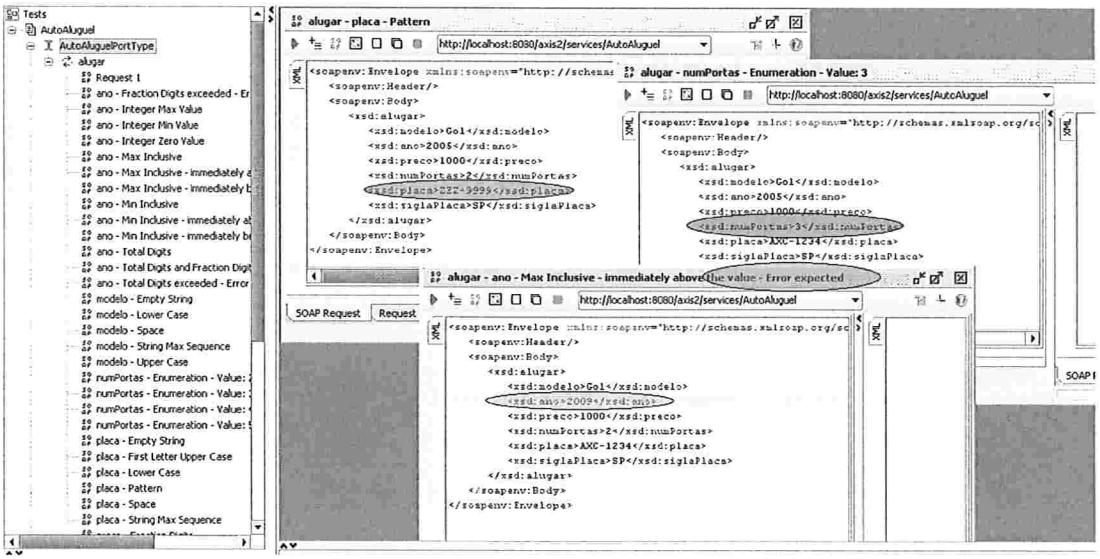

Figura 6.4: Alguma mensagens geradas pela técnica descrita em 5.1.2

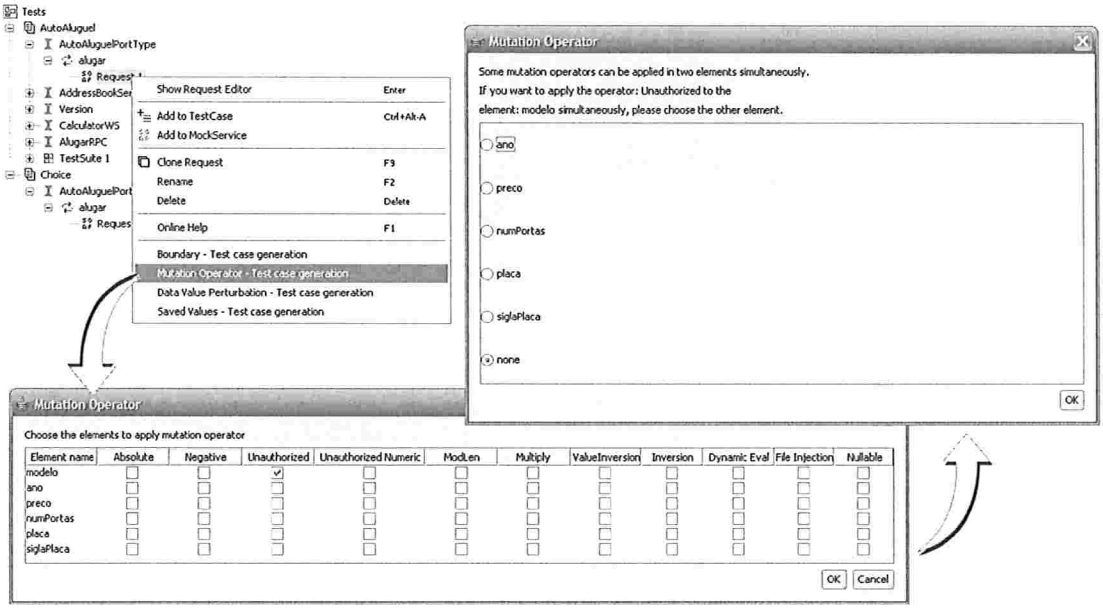

Figura 6.5: Janela para escolher operadores mutantes para aplicar sobre os elementos da mensagem.

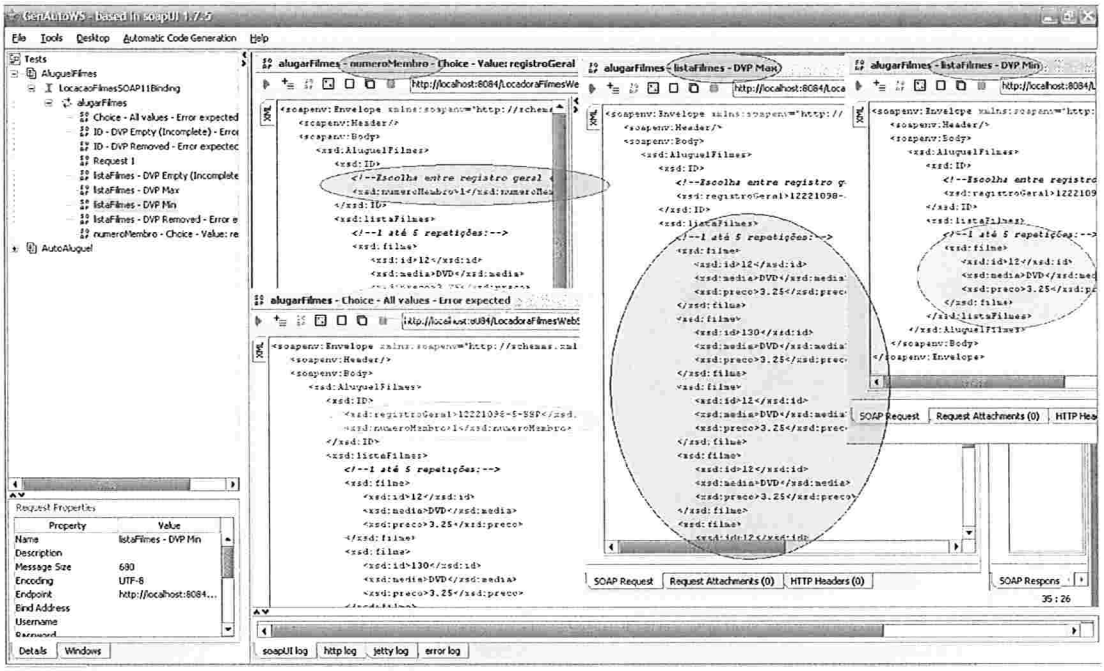

Figura 6.6: Três mensagens criadas com a técnica apresentada em 5.1.3

automaticamente pela ferramenta, pois não foi necessária a avaliação do testador. Para os demais resultados de execução é necessário que o responsável pelos testes avalie se a mensagem obtida como resposta é a esperada (execução sem falhas) ou não, fazendo o papel de oráculo.

A Figura 6.7 apresenta a janela de conjunto de casos de testes durante o processo de execução. O primeiro caso de teste mostra uma falha detectada automaticamente pela ferramenta. Esta mensagem de requisição possui o indicador de erro ligado, pois contém as duas opções de um elemento *choice*, mas o resultado de sua exceção não recebeu uma exceção como resposta, caracterizando uma falha.

O esforço do usuário da ferramenta durante todo o processode testes é de solicitar a criação dos casos de testes, ou seja, dois cliques do mouse para duas abordagens e alguns cliques para a abordagem utilizando operadores mutantes (pois deve selecionar quais operadores deseja aplicar). Depois de criadas as mensagens é necessário adicioná-las para um conjunto de casos de testes e efetuar mais um clique do mouse para solicitar a execução. Depois que a execução esteja completa algumas mensagens podem ter sido marcadas como falha, para as demais é necessário que o testador valide a resposta com o comportamento esperado do serviço.

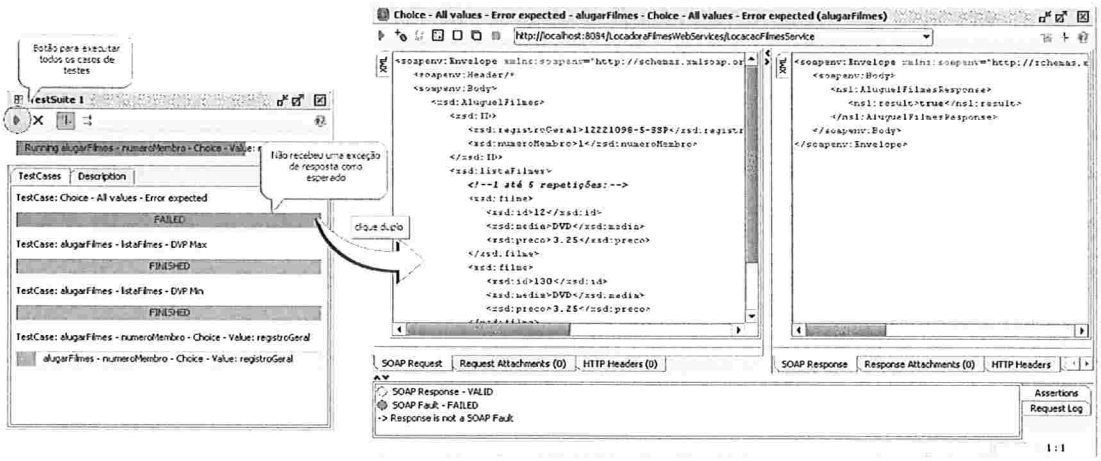

Figura 6.7: Conjunto de casos de testes

6.1.2 Outros recursos da ferramenta GenAutoWS

Mensagens utilizadas como semente para a geração de casos de testes automáticas são registradas em uma base de dados interna. GenAutoWS possui um recurso chamado *Perturbação Interna de dados*, este recurso é utilizado para criar mensagens modificando o valor de um elemento por um outro previamente gravado. Ambos os elementos devem possuir o mesmo nome considerando o *namespace*.

Bloomberg [7] mostra diversas questões de testar Web Services incluindo as característica de SOA: publicação, procura e invocação. GenAutoWS permite integração com servidor de registro UDDI disponibilizando as seguintes funcionalidades:

- integração com servidores UDDI escolhidos pelo usuário;
- permite a pesquisa ao servidor de registro UDDI utilizando palavras chave escolhidas pelo usuário;
- obtém o WSDL de especificação do serviço escolhido;
- adiciona ao projeto corrente o serviço;
- cria uma mensagem utilizando valores previamente armazenados na base interna ou utiliza valores padrão caso a base de dados não contenha valores. A partir dessa mensagem pode se criar casos de testes automatizados.

A Figura 6.8 mostra a caixa de diálogo para fazer pesquisa ao servidor de registro UDDI, assim como a inclusão do serviço na ferramenta e a mensagem criada para cada operação do serviço.

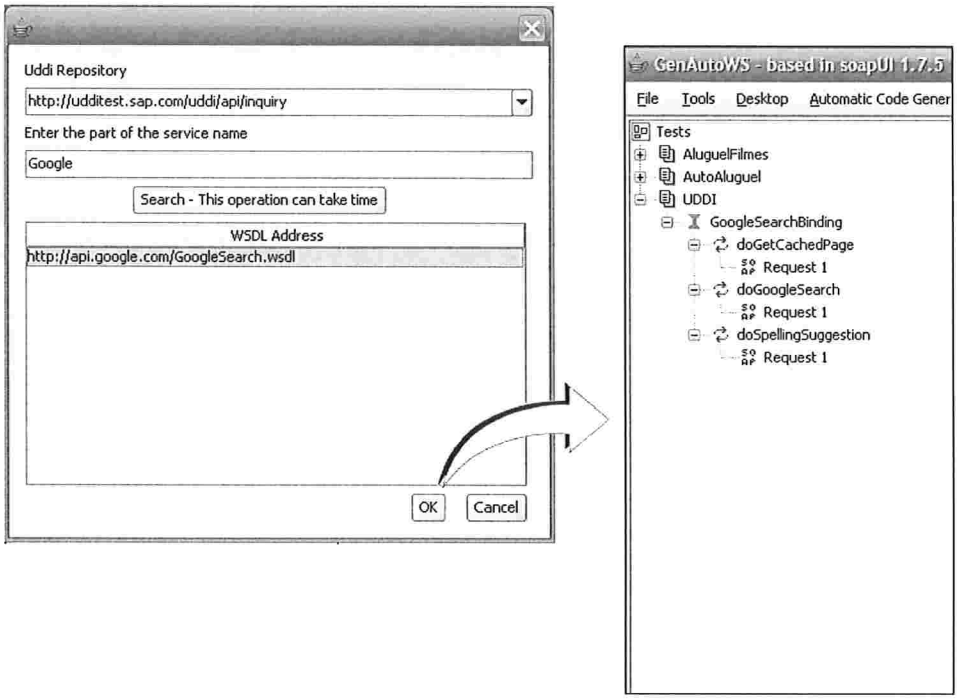

Figura 6.8: Integração com servidor de registro UDDI.

Capítulo 7

Estudos de caso

Este capítulo mostrará o estudos de casos realizados com a ferramenta GenAutoWS com o objetivo de verificar o método proposto para teste de Web Services e o grau de auxílio que a ferramenta oferece ao engenheiro de testes.

Com o intuito de melhor avaliar a eficiência da ferramenta procurou-se sistemas que são utilizados comercialmente, ao invés de desenvolver serviços experimentais e incluir defeitos para que o GenAutoWS os encontrassem. Devido ao contrato de não-divulgação de informação sobre os sistemas utilizados, detalhes técnicos, nome da empresa e código fonte não serão descritos.

Foram testados cinco serviços referentes a dois sistemas diferentes. Ambos sistemas foram desenvolvidos em Java, e são utilizados diariamente. Vale ressaltar que os dois sistemas já haviam passado pelo processo de testes da empresa e foram desenvolvidos há pelo menos dois anos.

As seguintes informações e métricas foram consideradas durante a execução desta prova de conceito:

- Número total de casos de testes gerados;
- Número de mensagens geradas para cada uma das técnicas utilizadas;
- Total de defeitos encontrados;
- Classificação dos defeitos encontrados;

7.1 Serviços avaliados

Ambos os sistemas avaliados possuem características de serviços: são aplicações modulares, possuem uma funcionalidade bem específica e são utilizados em um ambiente de aplicações distribuídas. Originalmente eles foram construídos sem uma interface Web Services, esta foi adicionada após a sua implantação em produção para facilitar a integração com outras plataformas. Antes da interface Web Services a integração com outros sistemas era feita utilizando um protocolo proprietário, baseado em XML, as aplicações clientes utilizavam um conjunto de classes Java para trocar mensagens. Atualmente ambas as interfaces são utilizadas.

O primeiro sistema é considerando crítico devido a sua importância para a empresa, trata-se de um sistema de envio de correio eletrônico, permitindo que outras aplicações desenvolvidas em outras linguagens e plataformas envie correios eletrônicos de maneira unificada.

Este sistema possui as seguintes funcionalidades: gerência do conteúdo da mensagem e arquivos anexos, linguagem de substituição com estruturas de interação e seleção para ser utilizada no corpo da mensagem e nos arquivos anexos, assinatura digital e criptografia de mensagens e anexos, e redundância de servidores SMTP. Três dos serviços testados pertencem a este sistema. Os serviços desse sistema serão chamados de WS1, WS2 e WS3. Este sistema possui requisitos de desempenho elevados, ele envia mais de cinqüenta mil correios eletrônicos diariamente.

O outro sistema contém dois serviços (WS4 e WS5) e comunica-se com um sistema do governo federal do Brasil para verificação de informações cadastrais, ele é utilizado em processamento *online* e lote.

Os cinco serviços aqui avaliados foram especificados com WSDL e utilizam o protocolo SOAP sobre HTTP. Quatro serviços são do tipo Document utilizando codificação Literal (serviço WS1, WS2, WS3 e WS4) e o outro é do tipo RPC utilizando codificação Encoded (serviço WS5).

7.2 Execução e coleta de resultados

Os casos de testes foram gerados pela ferramenta GenAutoWS a partir de uma mensagem válida de cada serviço. Utilizou a funcionalidade de captura de mensagem da ferramenta para incluí-las no sistema.

As técnicas modificadas apresentada no Capítulo 5 foram aplicadas sobre os cinco Web Services.

Para cada serviço foram criados três subprojetos na ferramenta. Cada subprojeto agrupara as mensagens geradas dentro de uma mesma abordagem. Foi criado um conjunto de teste (*test suíte*) em cada um dos subprojetos. Como descrito na seção 6.1.1, conjunto de teste possuem a funcionalidade de executar os casos de testes automaticamente e avaliar a resposta obtida para verificar se uma mensagem que contém o indicador de erro na posição ligada recebeu uma exceção como esperado. Defeitos diagnosticados nesta situação foram classificados como encontrados automaticamente pela ferramenta.

Para os casos onde não havia a detecção automática de defeitos, o testador passou a função de oráculo, e avaliou se as respostas obtidas eram adequadas as funcionalidades esperadas. Cada resultado foi anotado e serão apresentados na próxima seção. Foram utilizadas duas métricas para mensurar a eficiência da execução dos estudos de casos:

$$\text{Eficiência 1} = \frac{|\text{total de defeitos diferentes encontrados}|}{|\text{casos de testes gerados}|}$$

e

$$\text{Eficiência 2} = \frac{|\text{casos de testes com sucesso}|}{|\text{casos de testes gerados}|}$$

A segunda métrica baseia-se nos casos de testes executados com sucesso, ou seja, mostraram algum defeito, mesmo que essa já tenha sido encontrada anteriormente.

7.3 Resultados Obtidos

Os resultados obtidos com a execução dos casos de testes foram divididos pela técnica utilizada e são apresentados em tabelas separadas. Dentro de cada técnica foram contabilizados os números de casos de testes gerados e os números de defeitos descobertos. Diversos defeitos foram encontradas por mais de um caso de testes.

A Tabela 7.1 apresenta os resultados da execução dos casos de testes gerados com a técnica de perturbação para mensagens do tipo Document utilizando relacionamento e restrições, mostrada na seção 5.1.3. Cada célula da tabela mostra o número de defeitos encontros e o número total de caso de testes gerados (segundo número na célula). A coluna nomeada: "outras" contém os casos de testes que foram gerados como descrito em [49].

Tabela 7.1: Resultado da aplicação da Perturbando mensagens do tipo *Document* utilizando relacionamento e restrições

	MAX	MAX Unbounded	Removed	ELEM Value Seq	Choice	All not completed	All	All Only one	Empty (incomplete)	min	Outras
WS1	-	-	1/1	0/1	0/2	-	-	-	1/1	-	-
WS2	-	-	1/1	0/1	-	-	-	-	1/1	-	-
WS3	4/6	-	3/7	3/9	-	-	-	-	1/9	1/1	4/9
WS4	-	-	-	-	-	-	-	-	-	-	-
WS5	-	-	-	-	-	-	-	-	-	-	-
Eficiência	66.67%	-	66.67%	27.27%	0.00%	-	-	-	36.36%	100.00%	44.44%

A eficiência 1 desta técnica foi de 24.49%, ou seja, número total de defeitos diferentes encontradas, doze, pelo número total de casos de testes gerados, quarenta e nove. Vinte casos de testes executaram com sucesso (encontraram algum defeito, mesmo que esta já havia sido encontrada por algum outro caso de teste). A taxa de eficiência 2 foi de 40.82%, eficiência que considera todos os casos de testes que encontraram algum defeito.

Dois defeitos foram encontradas automaticamente pela ferramenta utilizando o indicador de erro, esses casos de testes receberam uma resposta sem erros como resposta a sua execução sendo que era esperada uma resposta inválida. Quatro dos doze defeitos encontrados foram consideradas como desvio da especificação, os demais defeitos tratam-se de mensagens de erro incompletas, vazias, ou incoerente com o caso de teste.

As inovações propostas neste trabalho geraram quarenta casos de testes, número quatro vezes superior aos nove casos de testes gerados com as técnica proposta por Offutt e Xu [49]. Apenas um *XML Schema* continha restrições com o elemento *maxOccurrs*, restrição necessária pela técnica apresentada por Offutt e Xu. A taxa de eficiência para estes casos de testes foi ligeiramente maior, 44.44% contra 40% dos testes utilizando a proposta deste trabalho. Não foi gerado nenhum caso de testes para o WS4 e o WS5. A Tabela 7.2 sintetiza os resultados obtidos por essa técnica.

Os resultados dos casos de testes gerados pela técnica Perturbação dos valores dos dados e Facetas de Restrição, apresentada na seção 5.1.2 são mostrados nas Tabelas 7.3 e 7.4.

Tabela 7.2: Resultado da aplicação da Perturbação de mensagens do tipo *Document* utilizando relacionamento e restrições

Defeitos diferentes encontrados	12
Total de Casos de Testes	49
Eficiência 1	24.49%
Defeitos encontrados pela ferramenta	2
Casos de Teste que mostraram algum defeito	20
Eficiência 2	40.82%
Defeitos de desvio da especificação	4

Tabela 7.3: Resultado da aplicação da Perturbação dos valores dos dados e Facetas de Restrição

	EMPTY STR	SPACE	FIRST LETTER	ENUM	FRACTION DIGITS	FRACT DIG EXC	MAX LEN	OVER	MAX EXCL ABOVE	MAX EXCL BELOW	MIN EXCL ABOVE
WS1	1/6	2/6	4/6	0/0	0/0	0/0	0/1	1/1	0/0	0/0	0/0
WS2	1/2	0/2	1/1	0/0	0/0	0/0	0/1	1/1	0/0	0/0	0/0
WS3	5/13	5/13	1/12	0/0	0/0	0/0	1/2	2/2	0/0	0/0	0/0
WS4	0/0	0/0	0/0	0/0	0/0	2/2	0/0	0/0	0/0	0/0	0/0
WS5	2/2	0/2	0/2	0/0	0/0	0/0	1/2	2/2	0/0	0/0	0/0
Total	9/23	7/23	6/21	0/0	0/0	2/2	2/6	6/6	0/0	0/0	0/0
Eficiência	39.13%	30.43%	28.57%	0.00%	0.00%	100.00%	33.33%	100.00%	0.00%	0.00%	0.00%

Neste grupo de casos de testes a eficiência 1 foi de 19.75% considerando o total de defeitos diferentes encontrados. A eficiência 2 que contabiliza os casos de testes executados com sucesso foi de 38.27%. A Tabela 7.5 apresenta um pequeno resumo dos resultados obtidos. Foram encontrados sete defeitos consideradas como desvio de especificação.

O último grupo de teste é apresentado na Tabela 7.6 e refere-se aos operadores de mutação apresentados na seção 5.1.4. A coluna nomeada "Outras" refere-se aos operadores mutantes de Offutt e Xu [49] e Vergilio e Almeida [16]. Foram encontrados 16 defeitos diferentes, resultando na eficiência 1 de 9.76%. O número de casos de testes executado com sucesso foi de 44, a eficiência 2 para esta métrica foi 26.83%. A Tabela 7.7 mostra um resumo da execução desta técnica de teste. Quatro dos defeitos encontrados foram consideradas desvio da especificação.

Tabela 7.4: Resultado da aplicação da Perturbação dos valores dos dados e Facetas de Restrição

	MIN EXCL BELOW	MAX INC ABOVE	MAX INC BELOW	MIN INC ABOVE	MIN INC BELOW	TOT DIG EXCE	TOT DIG AND FRAC EX	TOT DIG AND FRAC	PATTERN	WhiteSpace	Outras
WS1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	9/14
WS2	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	2/4
WS3	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	6/6	0/0	5/29
WS4	0/0	2/2	0/2	0/2	2/2	2/2	2/2	0/0	0/0	0/0	0/12
WS5	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/4
Total	0/0	2/2	0/2	0/2	2/2	2/2	2/2	0/0	6/6	0/0	16/63
Eficiência	0.00%	100.00%	0.00%	0.00%	100.00%	100.00%	100.00%	0.00%	100.00%	0.00%	25.40%

Tabela 7.5: Resultado da aplicação da Perturbação dos valores dos dados e Facetas de Restrição

Defeitos diferentes encontradas	32
Total de Casos de Testes	162
Eficiência	19.75%
Defeitos encontradas pela ferramenta	8
Casos de Teste que mostraram algum defeito	62
Eficiência	38.27%
Defeito com desvio da especificação	7

7.4 Conclusão

Este capítulo apresentou os estudos de caso realizado com a ferramenta GenAutoWS. Os resultados obtidos foram considerados bons, pois conseguiu descobrir diversos defeitos em um sistema que já havia sido submetido há um processo de testes.

Entre as três abordagens a qual se mostrou mais eficiente foi a perturbação de mensagens do tipo *Document* utilizando relacionamento e restrições, seção 5.1.3. Embora ela tenha se mostrado ligeiramente menos eficiente comparada com as mensagens geradas pela mesma abordagem apresentada por Offutt e Xu [49] para este tipo de teste (1.5% menos eficiente), o número de casos de testes assim como o número de casos de testes executado com sucesso foram quatro vezes superiores.

A abordagem perturbação dos valores dos dados e facetas de restrição, seção 5.1.2, gerou mais casos

Tabela 7.6: Resultado da aplicação da técnica Operadores de perturbção mutantes

	NUM SQL INJECT	DYNAMIC EVAL	FILE INJECT	NULL	Outras
WS1	0/0	0/6	3/6	0/0	5/30
WS2	0/0	0/2	0/2	0/0	1/10
WS3	0/0	0/13	9/13	0/0	17/61
WS4	2/2	0/0	0/0	0/0	3/8
WS5	0/0	0/2	0/2	0/0	4/7
Total	2/2	0/23	12/23	0/0	30/116
Eficiência	100.00%	0%	52.17%	0.00%	25.86%

Tabela 7.7: Resultado da aplicação da Operadores de Perturbação

Defeitos diferentes encontradas	16
Total de Casos de Testes	164
Eficiência	9.76%
Defeitos encontrados pela ferramenta	9
Casos de Teste que mostraram algum defeito	44
Eficiência	26.83%
Defeitos com desvio da especificação	4

de testes, 162, teve o maior número de casos de testes executados com sucesso (encontraram algum defeito), 62, e encontrou o maior número de defeito diferentes, 32. As melhorias da nova abordagem gerou duas vezes mais mensagens que a proposta de Offut e Xu [49], e a eficiência foi consideravelmente maior, 46.46% contra 25.40%. Apresentando-se como uma abordagem que consideramos fundamental no processo de teste destes Web Services em questão.

Os operadores de mutação acrescentados nesse trabalho geraram menos da metade das mensagens geradas pelos operados apresentados por Offut e Xu [49] e Vergilio e Almeida [16], no entanto a eficiência destes novos operados foi ligeiramente maior. De maneira geral os operadores mutantes obtiveram a menor taxa de eficiência, embora que tenham gerado o maior número de casos de testes.

O número total de casos de testes gerados com as modificações propostas neste trabalho foi o mesmo que os gerados com as técnicas propostas pelos demais autores. O número se igualou devido ao grande número de casos de testes gerados pelos operadores mutantes das abordagens de Offut e

Xu [49] e Vergilio e Almeida [16]. A eficiência total dos casos de testes[1] gerados por este trabalho superou em 60% aos demais casos de testes. Essa superação foi conclusiva para a avaliação positiva destas novas abordagens, pois testes bem sucedido é quando defeitos são encontrados.

A grande maioria dos defeitos foram consideradas simples, tais como: resposta de erro mal escrita ou diferente do esperado, sistema utilizou algum valor fora do domínio válido pela interface, elemento opcional não utilizado, etc. Este fator pode ser explicado pelo fato que os dois sistemas já haviam sido testados e são utilizados diariamente há pelo menos dois anos. Foram encontradas duas falhas de segurança, sendo que uma o sistema dava acesso a arquivos para anexar no correio eletrônico fora do diretório previamente estipulado.

[1] número total de casos de testes pelo número total de casos de testes que executaram com sucesso (encontraram defeito)

Capítulo 8

Conclusão e trabalhos futuros

A realização de teste é um elemento fundamental na implementação de qualquer sistema, mas é um processo dispendioso, e ocupa uma parte importante do custo total do projeto e dos recursos humanos. O uso de ferramentas e a automatização dos testes são uma alternativa viável na tentativa de melhorar a qualidade sem aumentar o custo e, se possível, diminuí-lo.

Esta pesquisa contribui no processo de testes aprimorando uma técnica para a geração automática de casos de testes para um domínio de aplicação que está em grande expansão e é base para implementação de um novo modelo de negócio baseado em serviços, os Web Services. Os casos de testes são criados utilizando uma nova abordagem para a perturbação de dados, técnica originalmente proposta por Offut e Xu [49].

No primeiro capítulo deste trabalho foi mostrada a importância do processo de testes no desenvolvimento de software, assim como os altos custos atrelados a esse processo. Os capítulos que seguiram, mostraram os conceitos sobre testes e a tecnologia de Web Services.

No Capítulo 5 foi apresentado os aprimoramentos na geração de casos de testes para Web Services. As modificações incluem: um novo conjunto de perturbação de valores de fronteira, novos operadores mutantes e melhoria na perturbação de mensagens *Document* utilizando informações sobre relacionamentos e restrições. Algumas mensagens criadas utilizando estes aprimoramentos podem causar uma resposta de erro quando executadas. Estas mensagens são sinalizadas com um indicador chamado "erro esperado", esse processo busca detectar, de forma automática, falhas nos serviços.

A ferramenta GenAutoWS foi implementada com as melhorias apresentadas no Capítulo 5 e inclui algumas outras funcionalidades, tais como: captura de mensagens entre sistemas, integração

com servidor UDDI e armazenamento de valores de elementos em uma base de dados para a criação de futuras mensagens. A execução de todos os casos de testes é feita clicando em um único botão (Figura 6.7). Utilizando o indicador de mensagem com erro esperado, a resposta obtida pela execução é avaliada e será sinalizada como falha, caso esta seja sucesso e a mensagem possuía o indicador com o valor verdadeiro.

Para validar as melhorias e modificações introduzidas no Capítulo 5, a implementação da ferramenta GenAutoWS foi seguida por estudos de caso utilizando dois sistemas de uso comercial. Os estudos de caso mostraram-se promissores, encontrando falhas em sistemas que já haviam sido testados e estão em uso há mais de dois anos. As melhorias e modificações propostas neste trabalho tiveram uma eficiência 60% maior comparada com a proposta original.

As melhorias e modificações propostas nesse trabalho tiveram uma eficiência maior que 60% do que a aplicação das demais técnicas como propostas pelos seus autores. A eficiência mede os casos de testes que foram capazes de apresentar alguma falha, cumprindo assim o seu objetivo.

8.1 Análise das técnicas apresentadas

Esta seção faz uma análise das técnicas apresentadas neste trabalho, mostrando algumas limitações e dificuldades que foram encontradas, além de relatar algumas comparações com os trabalhos de outros autores.

A criação de mensagens com o uso da perturbação dos valores dos dados e facetas de restrição (seção 5.1.2) gera ótimas mensagens para testes, pois como citado em [43,51] grande parte dos erros tendem a ocorrer nas fronteiras do domínio dos dados de entrada, e essa técnica utiliza todos os tipos de dados e as restrições dos elementos especificados na interface do serviço para a criação de mensagens.

A técnica proposta acima pode ser prejudicada caso o serviço não especifique bem os seus elementos, já que esta não terá informações suficientes para a geração dos casos de testes. Ela alcança melhores resultados em Web Services do tipo *Literal*, ou seja, especificado por um esquema de mensagem, tal como XML Schema. A maior vantagem que foi acrescentada nesta técnica são o uso de facetas para serviços especificados por XML Schema e a criação de mensagens inválidas.

A perturbação de mensagens do tipo *Document* utilizando relacionamento e restrições (seção 5.1.3) também depende da boa especificação do serviço, por exemplo: se um serviço especifica que uma mensagem pode conter um conjunto de elementos infinitas vezes, mas em realidade a partir de

um determinado número o serviço retorna mensagens de erro, a técnica não saberá como exercitar essa restrição, pois o serviço está mal especificado e não é possível conhecer essa informação, as técnicas aplicadas não analisam o código do serviço, apenas sua interface.

No trabalho original [49] a perturbação de mensagens utiliza apenas um indicador de ocorrência (*maxoccurrs*), a aplicação mostrada neste trabalho acrescentou o outro operador, além dos dois operadores de ordem e o elemento *any*. O número possível de mensagens que podem ser criadas aumentou em duas vezes e meia em comparação com o trabalho original, claro que as mensagens dependem da existência desses operadores no esquema que descreve as mensagens para serem geradas.

Os quatros novos operadores mutantes (seção 5.1.4) possuem suas funções bens definidas e têm poucos efeitos se aplicados fora de seus contextos, exemplo: durante os estudos de caso nenhuma falha foi encontrada com a aplicação do operador mutante *Dynamic Evaluation*, já que nenhum dos Web Services testados utilizavam valores da entrada em uma função *eval*.

Em comparação com outros trabalhos, as abordagens aqui apresentadas, assim como as pesquisas [49] e [16], podem ser aplicadas aos Web Services existentes, não dependendo da adoção de um *framework*, como exigido pelos trabalhos [66] e [64].

8.2 Análise da Ferramenta GenAutoWS

A ferramenta GenAutoWS (seção 6) é fácil e intuitiva, são necessários poucos cliques para utilizar a geração automática de casos de testes. A integração com o servidor de registro também é simples, basta escolher o servidor onde deseja fazer a consulta e utilizar palavras chaves para fazer a pesquisa dos serviços.

A ferramenta é um ótimo recurso para auxiliar no processo de teste de Web Services, pois essa gera diversas mensagens que são geradas manualmente pelo consumidor ou pelo provedor de serviço. As Tabelas 4.1 e 5.5 ilustram que a maior parte das mensagens criadas no exemplo manual estão na lista das mensagens criadas automaticamente. A ferramenta cria ainda mensagens que normalmente são desconsideradas ou esquecidas durante a criação de casos de testes.

A identificação automática de falhas utilizando o indicador de erro esperado foi implementada na ferramenta GenAutoWS utilizando o primeiro dos critérios sugeridos na seção 5.1.1. A implementação dos demais critérios ajudará na abrangência da ferramenta a uma maior gama de serviços que utilizam formas diferentes de comunicar ao consumidor que a execução terminou como erro.

Durante a implementação da ferramenta alguns pontos ofereceram maiores dificuldades, entre eles a criação de mensagens com a faceta *pattern*, onde era necessária a geração de uma mensagem válida e inválida segundo uma expressão regular, para tal, foi modificado o projeto de código livre RegExp [22] do grupo Apache, o processo de geração de mensagens foi embutido no analisador sintático da expressão.

8.3 Trabalhos futuros

No decorrer desta pesquisa algumas oportunidades de melhorias e futuros trabalhos foram deslumbradas. Os parágrafos a seguir comentam estas oportunidades.

Visto o bom resultado das duas abordagens baseadas em informações contidas no XML Schema (seção 5.1.2 e 5.1.3), é possível pensar na criação de uma extensão da ferramenta GenAutoWS para suportar serviços RESTful que sejam especificados com a linguagem WADL utilizando XML Schema. Para esta possível implementação também seria necessário um estudo de caso na tentativa de verificar sua validade.

O uso do indicador de mensagem com erro esperado facilitou o trabalho do oráculo, pois a própria ferramenta de certa forma o realizou. A expansão desse suporte seria muito interessante para o usuário do sistema GenAutoWS, informações contidas na especificação da mensagem de resposta do serviço poderia ser usada para esse fim.

Uma outra funcionalidade simples que pode ser agregada na ferramenta GenAutoWS é a validação automática de uma resposta de uma execução com um resultado obtido anteriormente, aplicando desta forma a técnica de testes de regressão.

O estudo da representação dos processos de negócio, cujo suporte a nível tecnológico é dado pelos Web Services, e são especificados por linguagens tais como: BPEL (*Business Process Execution Language*), WS-CDL (*Web Services Choreography Description Language*) e YAWL (*Yet Another Workflow Language*), apresentam-se como uma grande oportunidade para a aplicação de testes. Uma ferramenta capaz de gerar testes para especificações feitas com essas linguagens com certeza será de grande valor.

Referências Bibliográficas

[1] I. J. Taylor A. Harrison, *Wspeer - an interface to web service hosting and invocation*, IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 4 (Washington, DC, USA), IEEE Computer Society, 2005, p. 175.1.

[2] G. Rothermel R. H. Untch C. Zapf A. J. Offutt, A. Lee, *An experimental determination of sufficient mutant operators*, ACM Trans. Softw. Eng. Methodol. **5** (1996), no. 2, 99–118.

[3] C.; Olsson Allmann, C.; Denger, *Analysis of requirements-based test case creation techniques*, IESE-Report No. 046.05/E (2005).

[4] James Bach, *Heuristics of software testability*, Satisfice (2003), http://www.satisfice.com/tools/testable.pdf.

[5] D. K. Barry, *Web services and service-oriented architectures: The savvy manager's guide (the savvy manager's guides)*, Morgan Kaufmann, 2003.

[6] UC Berkeley, *BLAST (berkeley lazy abstraction software verification tool) model checker*, http://embedded.eecs.berkeley.edu/blast/.

[7] Jason Bloomberg, *Testing web services today and tomorrow*, The Rational Edge E-zine for the Rational Community (2002), http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/oct02/WebTesting_TheRationalEdge_Oct02.pdf.

[8] Ethan Cerami, *Web services essentials*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[9] Sandeep Chatterjee, *Developing enterprise web services: An architect's guide*, Prentice Hall PTR, Upper Saddle River, 2003.

[10] Ram Chillarege, *Software testing best practices*, 1999, http://www.chillarege.com/authwork/ papers1990s/TestingBestPractice.pdf.

[11] Frank Cohen, *Discover SOAP encoding's impact on web service performance*, IBM (2003), http://www-128.ibm.com/developerworks/webservices/library/ws-soapenc/.

[12] F. Cristian, *Understanding fault-tolerant distributed systems*, Commun. ACM **34** (1991), no. 2, 56–78.

[13] R. Majumdar D. Beyer, A. J. Chlipala, *Generating tests from counterexamples*, ICSE '04: Proceedings of the 26th International Conference on Software Engineering (Washington, DC, USA), IEEE Computer Society, 2004, pp. 326–335.

[14] F. McCabe E. Newcomer M. Champion C. Ferris D. Orchard D. Booth, H. Haas, *Web services architecture - W3C working group*, [Online; acessado em Janeiro-2007], 2004, http://www.w3.org/TR/ws-arch.

[15] Neil Davidson, *Web services testing*, [Online; acessado em Novembro-2006], 2002.

[16] Lourival F. Junior de Almeida and Silvia R. Vergilio, *Exploring perturbation based testing for web services*, ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06) (Washington, DC, USA), IEEE Computer Society, 2006, pp. 717–726.

[17] Richard A. DeMillo and A. Jefferson Offutt, *Constraint-based automatic test data generation*, IEEE Trans. Softw. Eng. **17** (1991), no. 9, 900–910.

[18] E. Dustin, *Effective software testing: 50 specific ways to improve your testing*, Addison-Wesley, Boston, 2002.

[19] Elfriede Dustin, Jeff Rashka, and John Paul, *Automated software testing: introduction, management, and performance*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[20] Roy T. Fielding, *Architectural styles and the design of network-based software architectures*, Ph.D. thesis, 2000.

[21] Qare framework, *JXWeb*, [Online; acessado em Fevereiro-2007], http://qare.sourceforge.net/web/2001-12/products/jxweb/index.html.

[22] Apache Software Fundation, *Jakarta regexp*, [Online; acessado em Dezembro-2007], http://jakarta.apache.org/regexp/.

[23] T. Badgett T. M. Thomas G. J. Myers, C. Sandler, *The art of software testing. 2 ed*, Wiley, New Jersey, 2004.

[24] Karl D. Gottschalk, Stephen Graham, Heather Kreger, and James Snell, *Introduction to web services architecture.*, IBM Systems Journal **41** (2002), no. 2, 170–177.

[25] Hans-Gerhard Gross, *Component-based software testing with uml*, Springer Verlag, Berlin, 2005.

[26] Felix C. Gärtner, *Fundamentals of fault-tolerant distributed computing in asynchronous environments*, ACM Comput. Surv. **31** (1999), no. 1, 1–26.

[27] R. Hathaway Wm. Hsu W. Hsu E. W. Krauser R. J. Martin A. P. Mathur H. Agrawal, R. A. DeMillo and E. H. Spafford, *Design of mutant operators for the C programming language*, Tech. Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[28] Hao He, *What is service-oriented architecture?*, O'Reilly [Online; acessado em Abril-2007], 2003, http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html.

[29] Hai Huang, Wei-Tek Tsai, Raymond Paul, and Yinong Chen, *Automated model checking and testing for composite web services*, ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05) (Washington, DC, USA), IEEE Computer Society, 2005, pp. 300–307.

[30] IEEE, *IEEE standard for software test documentation.*, IEEE Std 829-1998 (1998).

[31] IEEE-754, *Ieee 754: Standard for binary floating-point arithmetic*, http://grouper.ieee.org/groups/754/.

[32] M. G. Thomason J. A. Whittaker, *A markov chain model for statistical software testing*, IEEE Trans. Softw. Eng. **20** (1994), no. 10, 812–824.

[33] F. Zipitría L. Rodríguez, A. Vignaga, *Estudio de interoperabilidad .net/j2ee*, Universidad de la República (2004).

[34] Suet Chun Lee Lee and Jeff Offutt, *Generating test cases for XML-Based web component interactions using mutation analysis*, ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01) (Washington, DC, USA), IEEE Computer Society, 2001, p. 200.

[35] W. E. Lewis, *Software testing and continuous quality improvement, 2nd edition*, AUERBACH, 2004.

[36] Kanglin Li and Mengqi Wu, *Effective software test automation: Developing an automated software testing tool*, SYBEX Inc., Alameda, CA, USA, 2004.

[37] J. L. LIONS, *ARIANE 5 flight 501 failure*, Report by the Inquiry Board (1996).

[38] Gg Rothermel M. J. Harrold, *Performing data flow testing on classes*, SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering (New York, NY, USA), ACM Press, 1994, pp. 154–163.

[39] Sun Microsystems Inc. Marc J. Hadley, *Web application description language (WADL)*, [Online; acessado em Janeiro-2008], 2006, https://wadl.dev.java.net/#spec.

[40] James Mcgovern, *Java Web Services Architecture*, Morgan Kaufmann, San Diego, 2003.

[41] HP Mercury, *Mercury testdirector protocols*, [Online; acessado em Novembro-2006], http://www.mercury.com/us/website/pdf-viewer/?url=/us/pdf/products/loadrunner/1855-1006-loadrunner-protocols.pdf.

[42] Daniel Mosley, *Just enough software test automation*, Prentice Hall PTR, Upper Saddle River, 2002.

[43] G. J. Myers, *The art of software testing. 2 ed*, Wiley, New York, 2004.

[44] NASA, *Mars program independent assessment team summary report*, Tech. report, NASA, 2000, ftp://ftp.hq.nasa.gov/pub/pao/reports/2000/2000_mpiat_summary.pdf.

[45] Eric Newcomer, *Understanding web services: Xml, wsdl, soap, and uddi*, Addison-Wesley Professional, May 2002.

[46] NIST, *National institute of standards and technology*, [Online; acessado em Novembro-2006], http://www.nist.gov/.

[47] OASIS, *Introduction to UDDI: Important features and functional concepts, organization for the advancement of structured information standards*, [Online; acessado em Outubro-2006], 2004, http://uddi.org/pubs/uddi-tech-wp.pdf.

[48] A. Jefferson Offutt, *Investigations of the software testing coupling effect*, ACM Transactions on Software Engineering and Methodology **1** (1992), no. 1, 5–20.

[49] Jeff Offutt and Wuzhi Xu, *Generating test cases for web services using data perturbation*, SIGSOFT Softw. Eng. Notes **29** (2004), no. 5, 1–10.

[50] Mario Jino Paulo Marcos Siqueira Bueno, *Geração automática de dados e tratamento de não executabilidade no teste estrutural de software*, 1999.

[51] R. S. Pressman, *Engenharia de software. 5 ed.*, McGraw-Hill, Rio de Janeiro, 2002.

[52] Pressman S. R., *Software engineering: A practitioner's approach 6th edition*, McGraw-Hill, 2004.

[53] R. J. Lipton R. A.DeMillo and F. G. Sayward., *Hints on test data selection: Help for the practicing programmer.*, (1978).

[54] Leonard Richardson and Sam Ruby, *RESTful Web Services*, O'Reilly Media, Inc., May 2007.

[55] RTI, *The economic impacts of inadequate infrastructure for software testing*, Planning Report 02-3, National Institute of Standards and Technology, Research Triangle Park, NC, May 2002.

[56] A. R. da Silva S. A. Correia, *Técnicas para construção de testes funcionais automáticos*, in Actas da 5ª Conferência para a Qualidade nas Tecnologias da Informação e Comunicações (QUATIC'2004) (2004).

[57] Inderjeet Singh, Sean Brydon, Greg Murray, Vijay Ramachandran, Thierry Violleau, and Beth Stearns, *Designing web services with the j2ee 1.4 platform: JAX-RPC, xml services, and clients*, Pearson Education, 2004.

[58] James Snell, Doug Tidwell, and Pavel Kulchenko, *Programming web services with SOAP*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[59] SoapUI, *eviware*, [Online; acessado em Novembro-2007], http://www.soapui.org/.

[60] K. Spett, *Are your web applications vulnerable?*, White paper in SPI Dynamics Inc. (2005).

[61] M. E. Staknis, *Software quality assurance through prototyping and automated testing*, Inf. Softw. Technol. **32** (1990), no. 1, 26–33.

[62] International standard, *Iso 8601 data elements and interchange formats-information interchange-representation of dates and times*, ftp://ftp.iks-jena.de/pub/mitarb/lutz/standards/iso/iso8601

[63] H. Sthamer, *The automatic generation of software test data using genetic algorithms. PhD thesis*, (1996).

[64] W. T. Tsai, Ray Paul, Weiwei Song, and Zhibin Cao, *Coyote: An XML-Based framework for web services testing*, hase **00** (2002), 173.

[65] Wei-Tek Tsai, Yinong Chen, Zhibin Cao, Xiaoying Bai, Hai Huang, and Raymond A. Paul, *Testing web services using progressive group testing*, AWCC, 2004, pp. 314–322.

[66] Wei-Tek TSAI, Ray PAUL, Lian YU, Akihiro SAIMI, and Zhibin CAO, *Scenario-based web services testing with distributed agents(testing)(ieice/ieee joint special issue, assurance systems and networks)*, IEICE transactions on information and systems **86** (20031001), no. 10, 2130–2144.

[67] Sameer Tyagi, *Patterns and Strategies for Building Document-Based Web Services*, Tech. report, Sun Microsystem Inc, 2004, http://java.sun.com/developer/technicalArticles/xml/jaxrpcpatterns/.

[68] M. Pezze U. Buy, A. Orso, *Automated testing of classes*, ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis (New York, NY, USA), ACM Press, 2000, pp. 39–48.

[69] Xu W., *Slides: Generating test cases for web services using data perturbation*, 2004.

[70] Y. Wang C. Fan Dong Wang W. T. Tsai, R. Paul, *Extending WSDL to facilitate web services testing*, HASE '02: Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering (HASE'02) (Washington, DC, USA), IEEE Computer Society, 2002, p. 171.

[71] The World Wide Web Consortium (W3C), *Xml protocol working group*, [Online; acessado em Outubro-2006], http://www.w3.org/2000/xp/Group/.

[72] The World Wide Web Consortium (W3C), *XML Schema Part 1: Structures*, [Online; acessado em Dezembro-2007], http://www.w3.org/TR/2000/CR-xmlschema-1-20001024/.

[73] The World Wide Web Consortium (W3C), *XML schema part 2: Datatypes second edition*, [Online; acessado em Dezembro-2007], http://www.w3.org/TR/xmlschema-2/.

[74] ———, *Simple object access protocol (SOAP) 1.1*, [Online; acessado em Novembro-2007], 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383532.

[75] ———, *Web services description language (WSDL) 1.1*, 2001, http://www.w3.org/TR/wsdl.

[76] ———, *XML protocol (XMLP) requirements*, [Online; acessado em Julho-2007], 2002, http://www.w3.org/TR/2002/WD-xmlp-reqs-20020626.

[77] ———, *Soap version 1.2 part 0: Primer*, [Online; acessado em Julho-2007], 2003, http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.

[78] ———, *SOAP version 1.2 part 1: Messaging framework*, [Online; acessado em Fevereiro-2007], 2003, http://www.w3.org/TR/soap12-part1/.

[79] ———, *Web services glossary*, [Online; acessado em Dezembro-2007], 2004, http://www.w3.org/TR/ws-gloss/.

[80] Doug Wallace, Isobel Raggett, and Joel Aufgang, *Extreme programming for web projects*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[81] J. Williams, *The web services debate: J2EE vs. .NET*, Commun. ACM **46** (2003), no. 6, 58–63.

[82] WSUnit, *WSUnit - the web services testing tool*, [Online; acessado em Abril-2007], https://wsunit.dev.java.net/.

[83] Wuzhi Xu, Jeff Offutt, and Juan Luo, *Testing web services by XML perturbation*, ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (Washington, DC, USA), IEEE Computer Society, 2005, pp. 257–266.